Couvre Oracle 12c

Christian Soutou

SQL pour Oracle

7e édition

Applications avec Java, PHP et XML
Optimisation des requêtes et schémas

Avec 50 exercices corrigés

EYROLLES

Christian Soutou est maître de conférences à l'université Toulouse Jean-Jaurès et consultant indépendant. Rattaché au département Réseaux et Télécoms de l'IUT de Blagnac, il intervient autour des technologies de l'information en DUT, licence et master professionnels, ainsi que pour le compte de la société Orsys. Il est également l'auteur d'ouvrages sur SQL Server, MySQL, UML et les bases de données, tous parus aux éditions Eyrolles.

Apprendre SQL par l'exemple

Tout particulièrement destiné aux débutants et aux étudiants, cet ouvrage permet d'acquérir les notions essentielles d'Oracle, leader des systèmes de gestion de bases de données. Concis et de difficulté progressive, il est émaillé de nombreux exemples et de 50 exercices corrigés qui illustrent tous les aspects fondamentaux de SQL. Couvrant les versions 9*i* à 12*c* d'Oracle, il permet de se familiariser avec ses principales fonctionnalités, ainsi qu'avec les API les plus utilisées (JBDC, PHP et XML DB). Ce livre consacre également un chapitre entier à l'optimisation des requêtes et des schémas relationnels, en étudiant l'optimiseur, les statistiques, la mesure des performances et l'emploi de la boite à outils : contraintes, index, tables organisées en index, partitionnement, vues matérialisées et dénormalisation. Mise à jour et augmentée, cette septième édition actualise la partie XML DB et présente l'architecture multitenant de la version 12*c*.

À qui s'adresse cet ouvrage ?

- À tous ceux qui souhaitent s'initier à SQL, à Oracle ou à la gestion de bases de données
- Aux développeurs C, C++, Java, PHP et XML qui souhaitent stocker leurs données

Installez vous-même Oracle !

Les compléments web de cet ouvrage décrivent en détail les procédures d'installation des différentes versions d'Oracle, de la 9*i* à la 12*c* (éditions *Express* et *Enterprise*). Ces versions peuvent être téléchargées gratuitement sur le site d'Oracle : destinées à des fins non commerciales, elles sont complètes et sans limitation de durée.

Au sommaire

Sur le site www.editions-eyrolles.com

- Téléchargez le code source des exemples et le corrigé des exercices
- Consultez les mises à jour et les compléments
- Dialoguez avec l'auteur

Code éditeur : G14156
ISBN : 978-2-212-14156-6

Groupe Eyrolles | Diffusion Geodif

SQL pour Oracle

DU MÊME AUTEUR

C. Soutou, F. Brouard, N. Souquet et D. Barbarin. – **SQL Server 2014.**
N°13592, 2015, 890 pages.

C. Soutou. – **Programmer avec MySQL (3e édition).**
N°13719, 2013, 520 pages.

C. Soutou. – **Modélisation de bases de données (3e édition).**
N°14206, 2015, 352 pages. *À paraître.*

AUTOUR D'ORACLE ET DE SQL

R. Bizoï – **Oracle 12*c* – Administration.**
N°14056, 2014, 564 pages.

R. Bizoï – **Oracle 12*c* – Sauvegarde et restauration.**
N°14057, 2014, 336 pages.

R. Bizoï – **SQL pour Oracle 12*c*.**
N°14054, 2014, 416 pages.

R. Bizoï – **PL/SQL pour Oracle 12*c*.**
N°14055, 2014, 340 pages.

C. Pierre de Geyer et G. Ponçon – **Mémento PHP et SQL (3e édition).**
N°13602, 2014, 14 pages.

R. Bizoï – **Oracle 11g – Administration.**
N°12899, 2011, 600 pages.

R. Bizoï – **Oracle 11g – Sauvegarde et restauration.**
N°12899, 2011, 432 pages.

G. Briard – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

R. Bizoï – **SQL pour Oracle 10*g*.**
N°12055, 2006, 650 pages.

G. Briard – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

G. Briard – **Oracle9*i* sous Linux.**
N°11337, 2003, 894 pages.

Christian Soutou

SQL pour Oracle

7e édition

Applications avec Java, PHP et XML
Optimisation des requêtes et schémas

Avec 50 exercices corrigés

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Si Oracle était doué d'écriture, il penserait certainement aux journalistes et aux autres victimes qui ont perdu la vie au cours des attentats de Paris en janvier 2015.

Avant-propos

Nombre d'ouvrages traitent de SQL et d'Oracle ; certains résultent d'une traduction hasardeuse et sans vocation pédagogique, d'autres ressemblent à des annuaires téléphoniques. Les survivants, bien qu'intéressants, ne sont quant à eux plus vraiment à jour.

Ce livre a été rédigé avec une volonté de concision et de progression dans sa démarche ; il est illustré par ailleurs de nombreux exemples et figures. Bien que notre source principale d'informations fût la documentation en ligne d'Oracle, l'ouvrage ne constitue pas, à mon sens, un simple condensé de commandes SQL. Chaque notion importante est introduite par un exemple facile et démonstratif (du moins je l'espère). À la fin de chaque chapitre, des exercices vous permettront de tester vos connaissances.

Depuis quelques années, la documentation d'Oracle représente des centaines d'ouvrages au format HTML ou PDF (soit plusieurs dizaines de milliers de pages) ! Ainsi, il est vain de vouloir expliquer tous les concepts, même si cet ouvrage ressemblait à un annuaire. J'ai tenté d'extraire les aspects fondamentaux sous la forme d'une synthèse. Ce livre résulte de mon expérience d'enseignement dans des cursus d'informatique à vocation professionnelle (IUT, master professionnel et interentreprise).

Cet ouvrage s'adresse principalement aux novices désireux de découvrir SQL et de programmer sous Oracle.

- Les étudiants trouveront des exemples pédagogiques pour chaque concept abordé, ainsi que des exercices thématiques.
- Les développeurs C, C++, PHP ou Java découvriront des moyens de stocker leurs données.
- Les professionnels connaissant déjà Oracle seront peut-être intéressés par certaines nouveautés décrites dans cet ouvrage.

Les fonctionnalités de la version 11*g* ont été prises en compte lors de la troisième édition de cet ouvrage. Certains mécanismes d'optimisation (index, *clusters*, partitionnement, tables organisées en index, vues matérialisées et dénormalisation) sont apparus lors de la quatrième édition en même temps que quelques nouveautés SQL (pivots, transpositions, requêtes *pipe line*, CTE et récursivité). La cinquième édition enrichissait l'intégration avec Java (connexion à une base MySQL, *Data Sources* et *RowSets*) et PHP (API PDO : *PHP Data Objects*). La sixième édition présentait l'outil *SQL Data Modeler*. Celle-ci inclut des nouveautés de la version 12*c* et actualise principalement la technologie XML DB.

Par ailleurs, plusieurs compléments qui concernent des usages d'Oracle moins courants sont disponibles en téléchargement sur la fiche de l'ouvrage (à l'adresse *www.editions-eyrolles.com*) :

- l'installation de différentes versions (complément 1 : Installation des versions 9*i* à 12*c*) ;
- la technologie SQLJ (complément 2 : L'approche SQLJ) ;
- les procédures externes (complément 3 : Procédures stockées et externes) ;
- les fonctions PL/SQL pour construire des pages HTML (complément 4 : PL/SQL Web Toolkit et PL/SQL Server Pages).

Guide de lecture

Ce livre s'organise autour de trois parties distinctes mais complémentaires. La première intéressera le lecteur novice en la matière, car elle concerne les instructions SQL et les notions de base d'Oracle. La deuxième partie décrit la programmation avec le langage procédural d'Oracle PL/SQL. La troisième partie attirera l'attention des programmeurs qui envisagent d'utiliser Oracle tout en programmant avec des langages évolués (C, C++, PHP ou Java) ou via des interfaces Web.

Première partie : SQL de base

Cette partie présente les différents aspects du langage SQL d'Oracle en étudiant en détail les instructions élémentaires. À partir d'exemples simples et progressifs, nous expliquons notamment comment déclarer, manipuler, faire évoluer et interroger des tables avec leurs différentes caractéristiques et éléments associés (contraintes, index, vues, séquences). Nous étudions aussi SQL dans un contexte multi-utilisateur (droits d'accès), et au niveau du dictionnaire de données.

Deuxième partie : PL/SQL

Cette partie décrit les caractéristiques du langage procédural PL/SQL d'Oracle. Le chapitre 6 aborde des éléments de base (structure d'un programme, variables, structures de contrôle, interactions avec la base, transactions). Le chapitre 7 traite des sous-programmes, des curseurs, de la gestion des exceptions, des déclencheurs et de l'utilisation du SQL dynamique.

Troisième partie : SQL avancé

Cette partie intéressera les programmeurs qui envisagent d'exploiter une base Oracle en utilisant un langage de troisième ou quatrième génération (C, C++ ou Java), ou en employant une interface Web. Le chapitre 8 est consacré à l'étude des mécanismes de base du précompilateur d'Oracle Pro*C/C++. Le chapitre 9 présente les principales fonctionnalités de l'API JDBC.

Le chapitre 10 traite des deux principales API disponibles avec le langage PHP (OCI8 et PDO). Le chapitre 11 présente les fonctionnalités de XML DB et l'environnement *XML DB Repository*. Enfin, le chapitre 12 est dédié à l'optimisation des requêtes et des schémas relationnels.

Conventions d'écriture et pictogrammes

La police `courrier` est utilisée pour souligner les instructions SQL, noms de types, tables, contraintes, etc. (exemple : `SELECT nom FROM Pilote`).

Les majuscules sont employées pour les directives SQL, et les minuscules pour les autres éléments. Les noms des tables, index, vues, fonctions, procédures, etc., sont précédés d'une majuscule (exemple : la table `CompagnieAerienne` contient la colonne `nomComp`).

Les termes d'Oracle (bien souvent traduits littéralement de l'anglais) sont notés en italique (exemple : *row*, *trigger*, *table*, *column*, etc.).

Dans une instruction SQL, les symboles { et } désignent une liste d'éléments, et le symbole | un choix (par exemple, `CREATE {TABLE | VIEW}` exprime deux instructions possibles : `CREATE TABLE` ou `CREATE VIEW`). Les signes [et] désignent le caractère facultatif d'une option (par exemple, `CREATE TABLE Avion(...) [TABLESPACE USERS]` exprime deux écritures possibles : `CREATE TABLE Avion(...) TABLESPACE USERS` et `CREATE TABLE Avion(...)`).

Ce pictogramme introduit une définition, un concept ou une remarque importante. Il apparaît soit dans une partie théorique, soit dans une partie technique, pour souligner des instructions importantes ou la marche à suivre avec SQL.

Ce pictogramme annonce soit une impossibilité de mise en œuvre d'un concept, soit une mise en garde. Il est principalement utilisé dans la partie consacrée à SQL.

Ce pictogramme indique une astuce ou un conseil personnel.

Ce pictogramme indique une commande ou option disponible uniquement à partir de la version 12*c*.

Contact avec l'auteur et site Web

Si vous avez des remarques à formuler sur le contenu de cet ouvrage, n'hésitez pas à m'écrire (*christian.soutou@gmail.com*). Vous trouverez sur le site d'accompagnement, accessible par *www.editions-eyrolles.com*, les compléments et errata, ainsi que le code de tous les exemples et les exercices corrigés.

Table des matières

Introduction

Cette introduction présente tout d'abord le cadre général dans lequel cet ouvrage se positionne (SQL, le modèle de données et l'offre d'Oracle). Vient ensuite l'utilisation des principales interfaces de commandes pour que vous puissiez programmer avec SQL dès le chapitre 1. Vous trouverez dans les compléments (sur la fiche de l'ouvrage disponible à l'adresse *www.editions-eyrolles.com*) les procédures d'installation de différentes versions d'Oracle pour Windows (édition *Express* ou *Enterprise*, de 9*i* à 12*c*).

SQL, une norme, un succès

C'est IBM, à tout seigneur tout honneur, qui, avec System-R, a implanté le modèle relationnel au travers du langage SEQUEL (*Structured English as QUEry Language*) rebaptisé par la suite SQL (*Structured Query Language*).

La première norme (SQL1) date de 1987. Elle était le résultat de compromis entre constructeurs, mais fortement influencée par le dialecte d'IBM. SQL2 a été normalisée en 1992. Elle définit quatre niveaux de conformité : le niveau d'entrée (*entry level*), les niveaux intermédiaires (*transitional* et *intermediate levels*) et le niveau supérieur (*full level*). Les langages SQL des principaux éditeurs sont tous conformes au premier niveau et ont beaucoup de caractéristiques relevant des niveaux supérieurs. La norme SQL3 (intitulée initialement SQL:1999) comporte de nombreuses parties : concepts objets, entrepôts de données, séries temporelles, accès à des sources non SQL, réplication des données, etc. (chaque partie étant nommée ISO/IEC 9075-*i*:*année*, *i* allant de 1 à 14 et *année* étant la date de sortie de la dernière spécification). Une partie récente de la norme concerne la programmation côté serveur (ISO/IEC 9075-4:2011, partie 4 : *Persistent Stored Modules*).

Le succès que connaissent les grands éditeurs de SGBD relationnels (IBM, Oracle, Microsoft, Sybase et Computer Associates) a plusieurs origines et repose notamment sur SQL :

- Le langage est une norme depuis 1986 qui s'enrichit au fil du temps.
- SQL peut s'interfacer avec des langages de troisième génération comme C ou Cobol, mais aussi avec des langages plus évolués comme C++ et Java. Certains considèrent ainsi que le langage SQL n'est pas assez complet (le dialogue entre la base et l'interface n'est pas direct) et la littérature parle de « défaut d'impédance » (*impedance mismatch*).
- Les SGBD rendent indépendants programmes et données (la modification d'une structure de données n'entraîne pas forcément une importante refonte des programmes d'application).
- Ces systèmes sont bien adaptés aux grandes applications informatiques de gestion (architectures type client-serveur et Internet) et ont acquis une maturité sur le plan de la fiabilité et des performances.

- Ils intègrent des outils de développement comme les précompilateurs, les générateurs de code, d'états et de formulaires.
- Ils offrent la possibilité de stocker des informations non structurées (comme le texte, l'image, etc.) dans des champs appelés LOB (*Large Object Binary*).

Les principaux SGBD Open Source (MySQL, Firebird, Berkeley DB, PostgreSQL) ont adoptés depuis longtemps SQL pour ne pas rester en marge.

Nous étudierons les principales instructions SQL d'Oracle qui sont classifiées dans le tableau suivant.

Tableau I-1 Classification des ordres SQL

Ordres SQL	Aspect du langage
`CREATE - ALTER - DROP - COMMENT - RENAME - TRUNCATE - GRANT - REVOKE`	Définition des données (DDL : *Data Definition Language*).
`SELECT - INSERT - UPDATE - DELETE - MERGE - LOCK TABLE`	Manipulation des données (DML : *Data Manipulation Language*).
`COMMIT - ROLLBACK - SAVEPOINT - SET TRANSACTION`	Contrôle des transactions (TCL : *Transaction Control Statements*).

Modèle de données

Le modèle de données relationnel repose sur une théorie rigoureuse bien qu'adoptant des principes simples. La table relationnelle (*relational table*) est la structure de données de base qui contient des enregistrements, également appelés « lignes » (*rows*). Une table est composée de colonnes (*columns*) qui décrivent les enregistrements.

Tables et données

Considérons la figure suivante qui présente deux tables relationnelles permettant de stocker des compagnies, des pilotes et le fait qu'un pilote soit embauché par une compagnie :

***Figure I-1** Deux tables*

`compagnie`

comp	nrue	rue	ville	nom_comp
AB	1	Georges Brassens	Blagnac	Air Bus
ACTMP	24	René Lagasse	Balma	AC Toulouse

`pilote`

id_pil	brevet	nom_pil	nb_h_vol	*compa*
250	PL-1	Sarda	8500	AB
25	PL-2	Benech	5900	ACTMP
12	PL-3	Soutou	2000	ACTMP

Les clés

La clé primaire (*primary key*) d'une table est l'ensemble minimal de colonnes qui permet d'identifier de manière unique chaque enregistrement.

Dans la figure précédente, les colonnes « clés primaires » sont notées en gras. La colonne `comp` identifie chaque compagnie, tandis que la colonne `id_pil` permet d'identifier chaque pilote.

Une clé est dite « candidate » (*candidate key*) si elle peut se substituer à la clé primaire à tout instant. Une table peut contenir plusieurs clés candidates ou aucune.

Dans l'exemple, la colonne `brevet` pourrait jouer le rôle d'une clé candidate, car il est probable que chaque numéro de brevet soit unique. Pour les compagnies, le nom (`nom_comp`) s'il est supposé unique peut également jouer le rôle de clé candidate.

Une clé étrangère (*foreign key*) référence dans la majorité des cas une clé primaire d'une autre table (sinon une clé candidate sur laquelle un index unique aura été défini). Une clé étrangère est composée d'une ou plusieurs colonnes. Une table peut contenir plusieurs clés étrangères ou aucune.

La colonne `compa` (notée en italique dans la figure) est une clé étrangère, car elle permet de référencer un enregistrement unique de la table `compagnie` via la clé primaire `comp`.

Le modèle relationnel est ainsi fondamentalement basé sur les valeurs. Les associations entre tables sont toujours binaires et assurées par les clés étrangères. Les théoriciens considèrent celles-ci comme des pointeurs logiques. Les clés primaires et étrangères seront définies dans les tables en SQL à l'aide de contraintes.

Oracle

Il sera très difficile, pour ne pas dire impossible, à un autre éditeur de logiciels de trouver un nom mieux adapté à la gestion des données que celui d'« Oracle ». Ce nom semble prédestiné à cet usage ; citons *Le Petit Larousse* :

> **ORACLE** *n.m. (lat. oraculum)* **ANTIQ.** *Réponse d'une divinité au fidèle qui la consultait ; divinité qui rendait cette réponse ; sanctuaire où cette réponse était rendue.* **LITT.** *Décision jugée infaillible et émanant d'une personne de grande autorité ; personne considérée comme infaillible.*

Oracle représenterait ainsi à la fois une réponse infaillible, un lieu où serait rendue cette réponse et une divinité. Rien que ça ! Tout cela peut être en partie vérifié si votre conception

est bien faite, vos données insérées cohérentes, vos requêtes et programmes bien écrits. Ajoutons aussi le fait que les ordinateurs fonctionnent bien et qu'une personne compétente se trouve au support. C'est tout le mal que nous vous souhaitons.

Oracle Corporation, société américaine située en Californie, développe et commercialise un SGBD et un ensemble de produits de développement. Oracle a des filiales dans un grand nombre de pays. Initialement composée de cinq départements (marketing, commercial, avant-vente, conseil et formation), la filiale française (Oracle France) a été créée en 1986. Le département formation a été dissous en 2010, donnant naissance à la société EASYTEAM (premier partenaire Platinum en France), composée des ex-formateurs d'Oracle France.

Un peu d'histoire

En 1977, Larry Ellison, Bob Miner et Ed Oates fondent la société *Software Development Laboratories* (SDL). L'article de Edgar Frank Codd (1923-2003), « A Relational Model of Data for Large Shared Data Banks », *Communications of the ACM* paru en 1970, fait devenir le mathématicien et ancien pilote de la RAF durant la Seconde Guerre mondiale, inventeur du modèle relationnel et de SQL. Les associés de SDL devinent le potentiel des concepts de Codd et se lancent dans l'aventure en baptisant leur logiciel « Oracle ». En 1979, SDL devient *Relational Software Inc.* (RSI) qui donnera naissance à la société *Oracle Corp.* en 1983. La première version du SGBD s'appelle RSI-1 et utilise SQL. Le tableau suivant résume la chronologie des versions.

Tableau I-2 Chronologie des versions d'Oracle

1979 Oracle 2	Première version commerciale écrite en C/assembleur pour Digital – pas de mode transactionnel.
1983 Oracle 3	Réécrit en C – verrous.
1984 Oracle 4	Portage sur IBM/VM, MVS, PC – transaction (lecture consistante).
1986 Oracle 5	Architecture client-serveur avec SQL*Net – version pour Apple.
1988 Oracle 6	Verrouillage niveau ligne – sauvegarde/restauration – AGL – PL/SQL.
1991 Oracle 6.1	*Parallel Server* sur DEC.
1992 Oracle 7	Contraintes référentielles – procédures cataloguées – déclencheurs – version Windows en 1995.
1994	Serveur de données vidéo.
1995	Connexions sur le Web.
1997 Oracle 8	Objet-relationnel – partitionnement – LOB – Java.
1998 Oracle8*i*	*i* comme Internet, SQLJ – Linux – XML.
2001 Oracle9*i*	Services Web – serveur d'applications – architectures sans fil.
2004 Oracle 10*g*	*g* comme *Grid computing* (ressources en *clusters*).
2007 Oracle 11*g*	Auto-configuration.
2013 Oracle 12*c*	Architecture multitenant, Cloud et Big Data.

Avec IBM, Oracle a fait un pas vers l'objet en 1997, mais cette approche ne compte toujours pas parmi les priorités des clients d'Oracle. L'éditeur met plus en avant ses aspects transactionnels, décisionnels, de partitionnement et de réplication. Les technologies liées à Java, bien qu'elles soient largement présentes sous Oracle9*i*, ne constituent pas non plus la majeure partie des applicatifs exploités par les clients d'Oracle.

La version 10*g* renforce le partage et la coordination des serveurs en équilibrant les charges afin de mettre à disposition des ressources réparties (répond au concept de l'informatique à la demande). Cette idée est déjà connue sous le nom de « mise en grappe » des serveurs (*clustering*). Une partie des fonctions majeures de la version 10*g* est présente dans la version 9*i* RAC (*Real Application Cluster*).

La version 11*g* Oracle insiste sur les capacités d'auto-diagnostic, d'auto-administration et d'auto-configuration pour optimiser la gestion de la mémoire et pour pouvoir faire remonter des alertes de dysfonctionnement. En raison des exigences en matière de traçabilité et du désir de capacité de décision (*datamining*), la quantité de données gérées par les SGBD triplant tous les deux ans, 11*g* met aussi l'accent sur la capacité à optimiser le stockage.

La version 12*c* bouleverse l'architecture d'une instance assimilée à une base unique en introduisant l'architecture multitenant capable d'héberger plusieurs bases de données enfichables (*pluggable database*) dans une base de données de conteneur multipropriétaire (*container database*).

Rachat de Sun (et de MySQL)

Contrairement à la rumeur du début de 2007, MySQL n'entre pas en Bourse, il est racheté pour un milliard de dollars en janvier 2008 par Sun Microsystems déjà propriétaire de Java. Sun arrive ainsi sur un segment où il était absent, aux côtés d'Oracle, d'IBM et de Microsoft.

Craignant l'achat de Sun par IBM et redoutant HP dans le haut de gamme Unix, Oracle se repositionne dans le hardware et sur le marché des services pour *datacenters* en avril 2009, en achetant Sun. Ce sont aussi les langages Java et le système d'exploitation Solaris qui ont pesé dans la balance. En effet, c'est sur Solaris, et non sur Linux, que sont déployés le plus grand nombre de serveurs Oracle.

Il faudra attendre novembre 2009 pour que la Commission européenne confirme son refus de la fusion entre Oracle et Sun, suspectant que le rachat de MySQL aboutisse à une situation de quasi monopole sur le marché des SGDB. En décembre 2009, avec le soutien de quelque 59 sénateurs américains, Oracle publie 10 engagements concernant toutes les zones géographiques et pour une durée de 5 ans.

1. Assurer aux utilisateurs le choix de leur moteur (*MySQL's Pluggable Storage Engine Architecture*).
2. Ne pas changer les clauses d'utilisation d'une manière préjudiciable à un fournisseur tiers.
3. Poursuivre les accords commerciaux contractés par Sun.

4. Garder MySQL sous licence GPL.
5. Ne pas imposer un support des services d'Oracle aux clients du SGBD.
6. Augmenter les ressources allouées à la R&D de MySQL.
7. Créer un comité d'utilisateurs pour, dans les 6 mois, étudier les retours et priorités de développement de MySQL.
8. Créer ce même comité pour les fournisseurs de solutions incluant MySQL.
9. Continuer d'éditer, mettre à jour et distribuer gratuitement le manuel d'utilisation du SGBD.
10. Laisser aux utilisateurs le choix de la société qui assurera le support de MySQL.

Considérant d'une part ces engagements, et d'autre part l'existence de concurrents (notamment IBM, Microsoft et PostgreSQL dans le monde du libre), la Commission européenne avalise la fusion fin janvier 2010 pour un montant de 7,4 milliards de dollars. Cinq ans après (en 2015), MySQL est toujours dans le giron d'Oracle et, selon un dirigeant d'Oracle, les effectifs dédiés au développement et au support de MySQL ont doublé en cinq ans, et ceux de l'assurance qualité ont triplé.

Offre du moment

La page d'accueil d'Oracle (*www.oracle.com*) focalise sur les technologies à la mode (pour le moment le cloud). Sans parler des matériels, services, support, progiciels, etc., la base de données semble n'être qu'une brique à l'offre tentaculaire…

Figure I-2 *Offre Oracle*

Depuis la version 10*g*, les principales éditions du produit Oracle Database ont pour nom *Entreprise*, *Standard* et *Standard One*. Le produit monoposte est qualifié de *Personal* et la version gratuite de *Express*.

Figure I-3 *Éditions d'Oracle Database ; extrait du site*

	Oracle Database Express Edition	Oracle Database Standard Edition One	Oracle Database Standard Edition	Oracle Database Enterprise Edition
	Download Now	Price Now	Price Now	Price Now
Maximum	1 CPU	2 Sockets	4 Sockets	No Limit
RAM	1GB	OS Max	OS Max	OS Max
Database Size	11GB	No Limit	No Limit	No Limit
Oracle Multitenant				Option

Un grand nombre d'options (payantes en sus de la base et en fonction de l'édition) permettent de renforcer, notamment, les performances, la sécurité, le traitement transactionnel et le *datawarehouse*. Citons les plus connues : *Data Guard*, *Real Application Clusters*, *Partitioning*, *Advanced Security*, *Advanced Compression*, *Diagnostics Pack*, *Tuning Pack*, *OLAP*, *Data Mining* et *Spatial*.

Les prix des licences, clairement affichés sur le site d'Oracle, permettent deux modes de calcul : en fonction du nombre d'utilisateurs nommés (*named user plus*) ou du nombre de processeurs (*processors*). Le premier calcul convient généralement à des applications en mode client-serveur, le second serait davantage adapté aux architectures multitiers et Web.

Figure I-4 *Prix d'Oracle 2015 ; extrait du site*

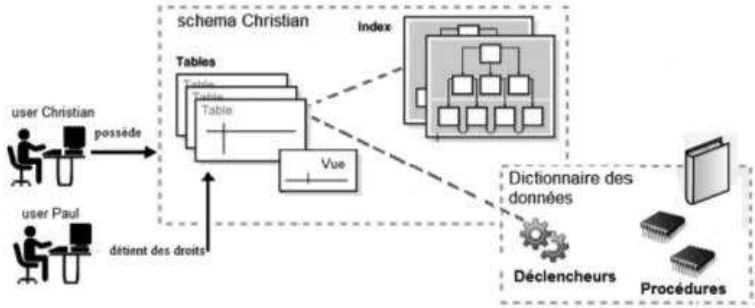
Prices in USA (Dollar)

	Oracle Database			
	Named User Plus	Software Update License & Support	Processor License	Software Update License & Support
Database Products				
Oracle Database				
Standard Edition One	180	39.60	5,800	1,276.00
Standard Edition	350	77.00	17,500	3,850.00
Enterprise Edition	950	209.00	47,500	10,450.00
Personal Edition	460	101.20	-	-
Mobile Server	-	-	23,000	5,060.00
NoSQL Database Enterprise Edition	200	44	10,000	2,200.00
Enterprise Edition Options:				
Multitenant	350	77.00	17,500	3,850.00
Real Application Clusters	460	101.20	23,000	5,060.00
Real Application Clusters One Node	200	44.00	10,000	2,200.00
Active Data Guard	230	50.60	11,500	2,530.00
Partitioning	230	50.60	11,500	2,530.00
Real Application Testing	230	50.60	11,500	2,530.00
Advanced Compression	230	50.60	11,500	2,530.00
Advanced Security	300	66.00	15,000	3,300.00
Label Security	230	50.60	11,500	2,530.00

Notion de schéma

Au niveau d'une base de données, qu'elle soit conventionnelle, enfichable ou conteneur (avec l'option multitenant de la version 12*c*), un schéma ne se distingue d'un utilisateur (*user*) que parce qu'il contient des objets (table, index, vue, etc.). Ainsi, chaque objet d'une base est associé à son propriétaire (l'utilisateur qui l'a créé ; c'est le cas de l'utilisateur Christian de la figure suivante). S'il ne détient aucun objet, un user peut être perçu simplement comme un identificateur de connexion (c'est le cas de Paul). Tout utilisateur peut toutefois accéder à des objets ne lui appartenant pas, sous réserve d'avoir reçu des droits accordés par le propriétaire ou un administrateur.

Figure I-5 *Schéma et utilisateur*



Tous les éléments d'un schéma ne seront pas étudiés car certains sont très spécifiques et sortent du cadre traditionnel de l'apprentissage. Le tableau suivant indique dans quel chapitre du livre vous trouverez les principaux éléments d'un schéma :

Tableau I-3 Éléments d'un schéma Oracle

Éléments étudiés – Chapitre	Aspects non étudiés
Déclencheurs (*triggers*) – 7 Fonctions et procédures cataloguées, paquetages – 7 Librairies de procédures externes – site d'accompagnement Index – 1, 12 Java – 9, site d'accompagnement Séquences et synonymes – 2, 5 Tables et tables en index – 1 Vues (*views*) – 5 XML – 11 Clusters – 12 Partitions – 12 Vues matérialisées – 12	Dimensions (cubes) Liens de bases de données (*database links*) Opérateurs Tables, types et vues objets Spatial

Accès à Oracle depuis Windows

Après avoir installé Oracle sur votre ordinateur, vous serez libre de choisir l'accès qui vous convient (le client SQL comme on dit). Ce livre utilise principalement l'interface SQL*Plus (livrée avec la base et dans toutes les versions clientes d'Oracle). Vous pouvez opter pour SQL Developer (produit Java gratuit sur le site d'Oracle qui ne nécessite aucune installation, simplement une décompression dans un de vos répertoires), ou pour un programme Java (via JDBC) ou PHP.

Il existe d'autres clients SQL qui sont payants ou gratuits ; citons SQL Developer, SQLTools, SQL Navigator et TOAD le plus renommé et probablement le plus performant.

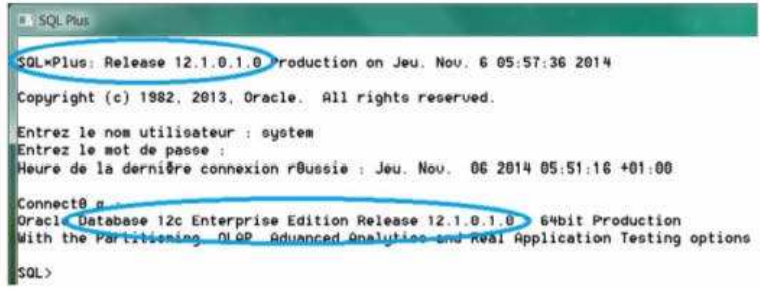
Détail d'un numéro de version

Détaillons la signification du numéro de la version 11.1.0.6.0 (première *release* de la 11*g* disponible sous Windows) :

- 11 désigne le numéro de la version majeure de la base ;
- 1 désigne le numéro de version de la maintenance ;
- 0 désigne le numéro de version du serveur d'applications ;
- 6 désigne le numéro de version du composant (*patch*) ;
- 0 est le numéro de la version de la plate-forme.

Vous pourrez contrôler la version de l'interface SQL*Plus et celle du serveur à l'issue de votre première connexion comme le montre la figure I-6 (ici, les versions de l'outil client et du serveur sont identiques car l'installation du serveur inclut l'installation de l'interface de même version).

Figure I-6 *Version du serveur et du client*



Les clients SQL

Les clients SQL permettent de dialoguer avec la base de différentes manières :

- exécution de commandes SQL, SQL*Plus et de blocs PL/SQL ;
- échanges de messages avec d'autres utilisateurs ;
- création de rapports d'impression en incluant des calculs ;
- réalisation des tâches d'administration en ligne.

SQL*Plus

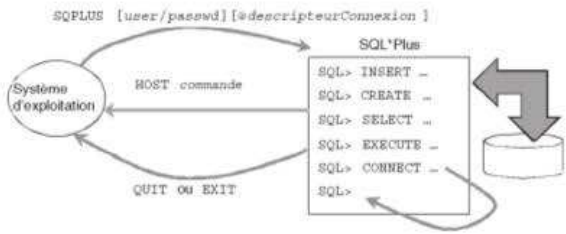
En fonction de la version d'Oracle dont vous disposez, plusieurs interfaces SQL*Plus peuvent être disponibles sous Windows :

- en mode ligne de commande (qui ressemble à une fenêtre DOS ou `telnet`) ;
- avec l'interface graphique (qui est la plus proche du monde Windows) ;
- avec l'interface graphique SQL*Plus Worksheet de l'outil *Enterprise Manager* (plus évoluée que la précédente) ;
- avec le navigateur via l'interface web *i*SQL*Plus (*i* comme « Internet » ; cette interface s'apparente assez à celle de EasyPHP en étant très intuitive).

Du fait que les interfaces graphiques et web aient été abandonnées depuis la version 11*g*, utilisez toujours l'interface en ligne de commande qui restera nécessairement disponible pour les versions à venir.

Le principe général de ces interfaces est le suivant : après une connexion locale ou distante, des instructions sont saisies et envoyées à la base qui retourne des résultats affichés dans la même fenêtre de commandes. La commande SQL*Plus `HOST` permet d'exécuter une commande du système d'exploitation qui héberge le client Oracle (exemple : `DIR` sous Window ou `ls` sous Unix).

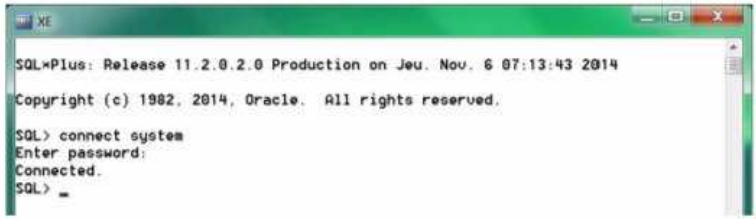
Figure I-7 *Principe général des interfaces SQL*Plus*



Connexion à Oracle

Quel que soit le mode de connexion que vous allez choisir, vous devrez toujours renseigner le nom de l'utilisateur et son mot de passe. D'autres paramètres peuvent être nécessaires comme le nom ou l'adresse du serveur, un numéro de port, un nom d'instance ou de service. Commençons par faire simple et utilisons l'interface SQL*Plus pour connecter l'utilisateur `system` à l'aide du mot de passe donné lors de l'installation, vous devez obtenir un résultat analogue (si vous disposez d'une version *Express* ; sinon, vous visualiserez davantage d'informations concernant la version du serveur).

Figure I-8 *Connexion à Oracle*



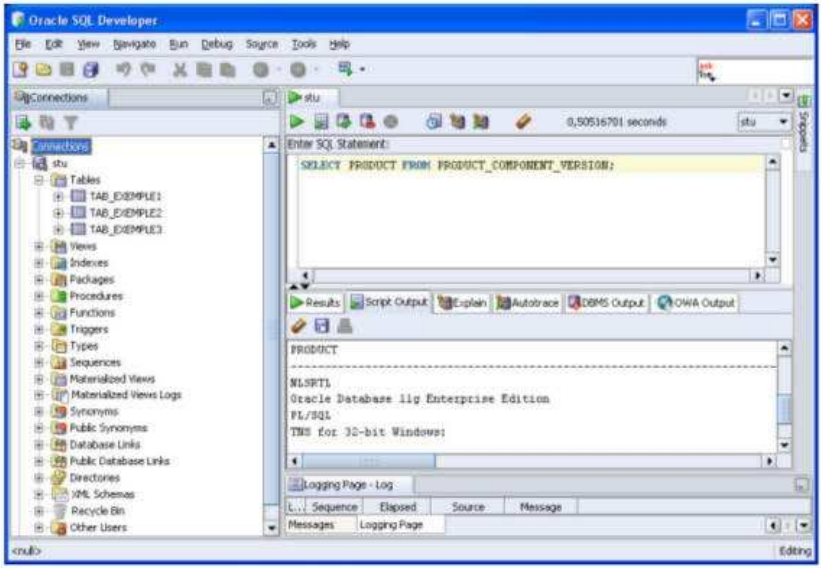
SQL Developer

SQL Developer est un outil gratuit de développement (écrit en Java) disponible sur le site d'Oracle (*www.oracle.com/technetwork/developer-tools/sql-developer*). Différentes versions sont disponibles (Windows 32 et 64 bits, Mac et Linux RPM et autres)…

Depuis la version 3, cet outil inclut un générateur de requêtes (*query builder*) et un gestionnaire de jobs (*schedule builder*). De plus, il est possible d'analyser sommairement des performances de requêtes (*Explain, Autotrace* et *SQL Tuning Advisor*). Des assistants d'exportation (par exemple, au format CSV) et d'importation (par exemple, de données Excel) sont également disponibles. Il permet même de visualiser et de manipuler des données spatiales et décisionnelles (*Spatial* et *Data Miner*).

Depuis la première version 4, une vue DBA permet d'administrer en partie une base (paramètres de configuration, backup et recovery avec RMAN, exportation et importation, comptes utilisateur, profils, rôles, privilèges, etc.). Utilisé conjointement avec le pack Tuning et Diagnostic, il est possible de visualiser l'activité (avec ADDM, AWR et ASH).

Figure I-9 SQL Developer



Après avoir téléchargé SQL Developer, vous n'aurez qu'à décompresser l'archive dans le répertoire `Programmes` et à exécuter `sqldeveloper.exe`. Mises à part les éditions *Express* d'Oracle, SQL Developer est inclus dans les éditions *Standard* et *Enterprise*, et se trouve dans le menu `Démarrer Oracle.../Développement d'applications`. À la première exécution, le chemin du JDK vous sera probablement demandé.

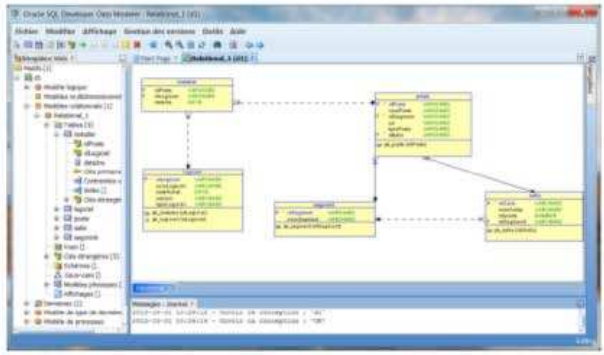
Bien que l'outil permette un grand nombre de fonctionnalités, certaines commandes SQL*Plus (`GET, START, COL, ACCEPT…`) ne sont pas opérantes.

SQL Data Modeler

Un outil de développement SQL n'est pas forcément un outil de conception. Ce dernier vise à construire ou « cartographier » des tables alors que le premier les manipule. Les outils de conception sont nombreux mais le plus souvent payants (TOAD Data Modeler, PowerAMC, DeZign for Databases, ERwin Data Modeler, Enterprise Architect, ER/Studio, Navicat, etc.). Oracle fournit gratuitement SQL Data Modeler (*www.oracle.com/technetwork/developer-tools/datamodeler*) qui est construit sur une interface analogue à son homologue SQL Developer. Le

niveau conceptuel des données n'est pas le plus abouti mais si vous travaillez uniquement au niveau des tables, contraintes et index, il vous conviendra sans doute. Il vous permettra de générer des scripts de création de tables ou de visualiser graphiquement des tables d'un schéma, ce qui est très intéressant pour la compréhension et pour écrire des requêtes réalisant des jointures cohérentes (voir le chapitre 4).

Figure I-10 *SQL Developer Data Modeler*



Premiers pas

Débutez votre apprentissage avec l'interface SQL*Plus afin de vous familiariser avec les manipulations basiques qui vous serviront fréquemment par la suite, car il y a de grandes chances pour que cette interface soit présente dans les différents environnements que vous fréquenterez. Si vous commencez par *SQL Developer*, vous n'utiliserez plus SQL*Plus et le jour où vous ne disposerez que de cette interface, vous risquez d'être bloqué et de ne pas pouvoir fournir les résultats attendus…

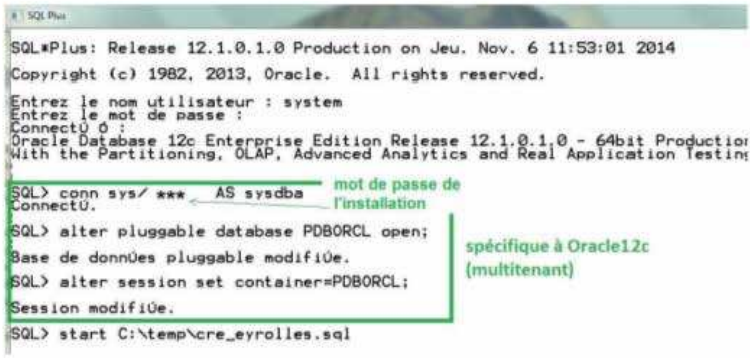
Création d'un utilisateur

Pour créer un utilisateur, utilisez le script `CreaUtilisateur.sql` situé dans le répertoire `Introduction`. Choisissez-lui ensuite un nom (supprimer les caractères < et >) ainsi qu'un mot de passe. Si vous enregistrez ce fichier avec un autre nom dans un autre répertoire, il est préférable de ne pas utiliser de caractères spéciaux (ni d'espaces) dans le nom de vos répertoires.

Une fois connecté, exécutez votre script dans l'interface SQL*Plus grâce à la commande `start` *`chemin/nom_script`* (par exemple, `start C:\temp\cre_eyrolles.sql`). L'écran suivant concerne la création d'un utilisateur dans la base enfichable (`PDBORCL` par

défaut). Pour des éditions antérieures à la version 12*c*, les trois instructions encadrées ne sont pas à exécuter.

Figure I-11 *Création d'un utilisateur*



Vous devez obtenir les deux messages suivants (aux accents près) :

```
Utilisateur créé.
Autorisation de privilèges (GRANT) acceptée.
```

Voilà, votre utilisateur est créé, il peut se connecter et possède les prérogatives minimales pour exécuter la plupart des commandes décrites dans cet ouvrage.

Si vous voulez afficher vos instructions avant qu'elles ne s'exécutent sous SQL*Plus (utile pour tracer l'exécution de plusieurs commandes), lancez la commande `set echo on` qui restera valable pour toute la session.

À l'instar de la syntaxe du langage SQL d'Oracle, les commandes SQL*Plus sont insensibles à la casse. Dans cet ouvrage, elles sont en général mentionnées en majuscules.

*Commandes basiques de SQL*Plus*

Le tableau I-4 récapitule les commandes qui permettent de manipuler le buffer de l'interface SQL*Plus. Une fois écrite dans l'interface, la commande (qui peut constituer une instruction SQL ou un bloc PL/SQL) pourra être manipulée, avant ou après son exécution.

Tableau I-4 Commandes du buffer d'entrée (pas pour /SQL*Plus)

Commande	Commentaires
`R`	Exécute (*run*).
`L`	Liste le contenu du buffer.
`L*`	Liste la ligne courante.
`Ln`	Liste la *nième* ligne du buffer qui devient la ligne courante.
`I`	Insère une ligne après la ligne courante.
`A` *texte*	Ajoute *texte* à la fin de la ligne courante.
`DEL`	Supprime la ligne courante.
`C`/*texte1*/*texte2*/	Substitution de la première occurrence de *texte1* par *texte2* dans la ligne courante.
`CLEAR`	Efface le contenu du buffer.
`QUIT` ou `EXIT`	Quitte SQL*Plus.
`CONNECT` *user*/*password*@*descripteur*	Autre connexion (sans sortir de l'interface).
`GET` *fichier*	Charge dans le buffer le contenu du *fichier.sql* qui se trouve dans le répertoire courant.
`SAVE` *fichier*	Écrit le contenu du buffer dans *fichier.sql* qui se trouve dans le répertoire courant.
`START` *fichier* ou *@fichier*	Charge dans le buffer et exécute *fichier.sql*.
`SPOOL` *fichier*	Crée *fichier.lst* dans le répertoire courant qui va contenir la trace des entrées/sorties jusqu'à la commande `SPOOL OFF`.

Variables d'environnement

Les variables d'environnement (voir le tableau I-5) vous permettront de paramétrer votre session SQL*Plus. L'affectation d'une variable s'opère avec `SET` (ou par un menu si vous utilisez encore l'interface Windows graphique). À tout moment, la commande `SHOW` *`nom_variable`* vous renseignera à propos d'une variable d'environnement (voir le tableau I-6).

Tableau I-5 Variables d'environnement

Commande	Commentaires
`SET AUTOCOMMIT {ON | OFF | IMMEDIATE | n}`	Validation automatique après une ou *n* commandes.
`SET ECHO {ON | OFF}`	Affichage des commandes avant exécution.
`SET LINESIZE {80 | n}`	Taille en caractères d'une ligne de résultats.
`SET PAGESIZE {24 | n}`	Taille en lignes d'une page de résultats.
`SET SERVEROUT [PUT]{ON | OFF}`	Activation de l'affichage pour tracer des exécutions.
`SET TERMOUT {ON|OFF}`	Affichage des résultats.
`SET TIME {ON|OFF}`	Affichage de l'heure dans le prompt.

Tableau I-6 Paramètres de la commande SHOW

Paramètre	Commentaires
`variableEnvironnement`	Variable d'environnement (`AUTOCOMMIT`, `ECHO`, etc.).
`ALL`	Toutes les variables d'environnement.
`ERRORS`	Erreurs de compilation d'un bloc ou d'un sous-programme.
`RELEASE`	Version du SGBD utilisé.
`USER`	Nom de l'utilisateur connecté.

À propos des accents et jeux de caractères

Il est possible de paramétrer sous Oracle certains formats, tels que la date, l'heure, les jours de la semaine, la monnaie, le jeu de caractères, etc. La principale difficulté étant que ces paramètres peuvent être différents entre le serveur Oracle, les systèmes d'exploitation hébergeant la base et le programme client (client Oracle natif comme l'interface SQL*Plus ou client utilisant un pilote Oracle JDBC par exemple).

En soit cette différence n'est pas dangereuse car Oracle opère les conversions automatiquement, mais il est important de savoir quel format de données le client attend. Une base de données peut stocker des prix en dollars car son jeu de caractères est américain et les restituer en euros car le client est européen. Bien sûr le chiffre stocké en base est en valeur d'euros et s'il est affiché par un client local il apparaîtra sous la forme de dollar. Ce raisonnement vaut pour les dates et accents.

Configuration côté serveur

Pour connaître la configuration côté serveur (instance sur laquelle vous êtes connecté), il faut interroger la vue NLS_DATABASE_PARAMETERS qui renseigne, entre autres, la langue, au territoire (pour le format des dates, des monnaies) et au jeu de caractères. Dans cet exemple, la base installée est Oracle 10*g Express Edition*.

```
SQL> SELECT PARAMETER, VALUE FROM NLS_DATABASE_PARAMETERS
       WHERE parameter IN ('NLS_LANGUAGE','NLS_TERRITORY',
       'NLS_CHARACTERSET','NLS_CURRENCY');

PARAMETER            VALUE
-------------------- --------------------
NLS_LANGUAGE         AMERICAN
NLS_TERRITORY        AMERICA
NLS_CURRENCY         $
NLS_CHARACTERSET     AL32UTF8
PARAMETER            VALUE
```

Il apparaît la langue anglaise (AMERICAN) de l'instance, les codes américains pour le format des dates, des monnaies (AMERICA) et le jeu de caractères par défaut (AL32UTF8) qui est une extension (pour les plates-formes ASCII) du classique Unicode UTF-8 codé sur 4 octets.

Configuration côté client

Pour connaître la configuration côté client (ici une session SQL*Plus), il faut interroger la vue NLS_SESSION_PARAMETERS qui renseigne un certain nombre de paramètres mais pas le jeu de caractères.

```
SQL> SELECT PARAMETER, VALUE FROM NLS_SESSION_PARAMETERS
     WHERE parameter IN
    ('NLS_LANGUAGE','NLS_TERRITORY','NLS_CURRENCY');

PARAMETER            VALUE
-------------------- --------------------
NLS_LANGUAGE         FRENCH
NLS_TERRITORY        FRANCE
NLS_CURRENCY         Ç
```

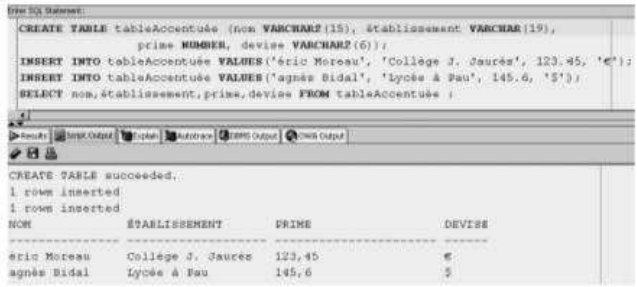
Il apparaît que le client a été installé en choisissant la langue française avec ses conventions (notamment pour le format de dates et de la monnaie). Le jeu de caractères n'est pas ici accessible. Le symbole € n'est pas restitué car certaines interfaces SQL*Plus utilisent une police de caractère de type Courrier qui n'inclut pas ce symbole.

Afin de pouvoir restituer des caractères accentués :

- Concernant le client SQL*Plus en mode ligne de commande, il faut affecter la variable `NLS_LANG` (sous Windows `set NLS_LANG=FRENCH_FRANCE.WE8PC850`, par exemple avec Unix `export NLS_LANG=...`).
- Pour les autres clients graphiques Windows tels que *SQL Developer*, vous devrez vous assurer que la base de registres contient la valeur `FRENCH_FRANCE.WE8MSWIN1252` pour la clé `NLS_LANG` (choix `Edition/Rechercher...` en lançant `regedit`).

Une fois ceci fait, vous devriez pouvoir gérer les accents au niveau des données, tables, colonnes, etc. L'exemple suivant illustre cette possibilité (ici le test est réalisé dans la console *SQL Developer*).

Figure I-12 *Restitution de caractères accentués*



Partie I

SQL de base

Chapitre 1

Définition des données

Ce chapitre décrit les instructions SQL qui constituent l'aspect LDD (langage de définition des données) de SQL. À cet effet, nous verrons notamment comment déclarer une table, ses éventuels contraintes et index.

Tables relationnelles

Une table est créée en SQL par l'instruction `CREATE TABLE`, modifiée au niveau de sa structure par l'instruction `ALTER TABLE` et supprimée par la commande `DROP TABLE`.

Création d'une table (CREATE TABLE)

Pour pouvoir créer une table dans votre schéma, il faut que vous ayez reçu le privilège `CREATE TABLE`. Si vous avez le privilège `CREATE ANY TABLE`, vous pouvez créer des tables dans tout schéma. Le mécanisme des privilèges est décrit au chapitre « Contrôle des données ».

La syntaxe SQL simplifiée est la suivante :

```
CREATE TABLE [schéma.]nomTable
   ( colonne1 type1 [DEFAULT valeur1] [NOT NULL]
   [, colonne2 type2 [DEFAULT valeur2] [NOT NULL] ]
   [CONSTRAINT nomContrainte1 typeContrainte1]…) ;
```

- *schéma* : s'il est omis, il sera assimilé au nom de l'utilisateur connecté. S'il est précisé, il désigne soit l'utilisateur courant soit un autre utilisateur de la base (dans ce cas, il faut que l'utilisateur courant ait le droit de créer une table dans un autre schéma). Nous aborderons ces points dans le chapitre 5 et nous considérerons jusque-là que nous travaillons dans le schéma de l'utilisateur couramment connecté (ce sera votre configuration la plupart du temps).
- *nomTable* : peut comporter au maximum 30 caractères (lettres, chiffres et caractères _, $ et #). Si l'identificateur n'est pas encadré par des guillemets (*quoted identifier*), le nom est insensible à la casse et sera converti en majuscules dans le dictionnaire de données (il en va de même pour le nom des colonnes).

- *colonnei typei* : nom de colonne et son type (NUMBER, VARCHAR2, DATE...). L'option DEFAULT fixe une valeur en cas de non-renseignement (NULL). L'option NOT NULL interdit que la valeur de la colonne ne soit pas renseignée.

Le marqueur NULL ne désigne pas une valeur mais une absence de valeur qu'on peut traduire comme non disponible, non affectée, inconnue ou inapplicable. NULL est différent d'une chaîne vide, d'un zéro ou des espaces. Ce marqueur est à étudier de près car, dans bien des cas, deux NULL ne sont pas identiques. Les requêtes d'extraction peuvent renvoyer des résultats aberrants si les NULL sont mal interprétés. En positionnant le plus possible de NOT NULL dans vos colonnes, vous diminuerez les traitements additionnels à opérer par la suite.

- *nomContrainte1 typeContrainte1* : noms de la contrainte et son type (clé primaire, clé étrangère, etc.). Nous allons détailler dans le paragraphe suivant les différentes contraintes possibles.
- ; : symbole qui termine une instruction SQL d'Oracle. Le slash (/) peut également terminer une instruction à condition de le placer à la première colonne de la dernière ligne.

Casse et commentaires

Dans toute instruction SQL (déclaration, manipulation, interrogation et contrôle des données), il est possible d'inclure des retours chariots, des tabulations, espaces et commentaires (sur une ligne précédée de deux tirets --, sur plusieurs lignes entre /* et */). De même, la casse n'a pas d'importance au niveau des mots-clés de SQL, des noms de tables, colonnes, index, etc. Les scripts suivants décrivent la déclaration d'une même table en utilisant différentes conventions :

Tableau 1-1 Différentes écritures SQL

Sans commentaire	Avec commentaires
`CREATE TABLE MêmesévénementsàNoël (colonne CHAR);`	`CREATE TABLE -- nom de la table TEST( -- description COLONNE NUMBER(38,8) ) -- fin, ne pas oublier le point-virgule. ;`
`CREATE TABLE Test (colonne NUMBER(38,8));`	
`CREATE table test (Colonne NUMBER(38,8));`	`CREATE TABLE Test ( /* une plus grande description des colonnes */ COLONNE NUMBER(38,8));`

La casse a une incidence majeure dans les expressions de comparaison entre colonnes et valeurs, que ce soit dans une instruction SQL ou un test dans un programme. Ainsi, l'expression « `nomComp='Air France'` » n'aura pas la même signification que l'expression « `nomComp ='AIR France'` ».

Premier exemple

Le tableau 1-2 décrit l'instruction SQL qui permet de créer, dans le schéma *soutou*, la table `vol_jour` illustrée par la figure suivante. L'absence du préfixe `soutou.` aurait conduit au même résultat si la connexion était établie par l'utilisateur *soutou* lors de la création de la table. L'utilisateur *soutou* devient propriétaire (*owner*) de l'objet table *vol_jour* (on dit aussi que le schéma *soutou* contient la table `vol_jour`).

Figure 1-1 *Table à créer*

Tableau 1-2 Création d'une table et de ses contraintes

Instruction SQL	Commentaires

`CREATE TABLE vol_jour`
`(num_vol VARCHAR2(6) NOT NULL,`
`aero_dep VARCHAR2(3) NOT NULL,`
`aero_arr VARCHAR2(3) NOT NULL,`
`comp VARCHAR2(4) DEFAULT 'AF',`
`jour_vol DATE NOT NULL,`
`nb_passagers NUMBER(3));`
	La table contient six colonnes (quatre chaînes de caractères variables, une date et un entier relatif de trois chiffres).
	La table inclut cinq contraintes en ligne.
	• 4 `NOT NULL` qui imposent de renseigner quatre colonnes.
	• 1 `DEFAULT` qui fixe un code compagnie à défaut d'être renseigné.

Contraintes de colonnes

Les contraintes de colonnes ont pour but de programmer des règles de gestion au niveau des colonnes des tables. Elles peuvent alléger un développement côté client (si on déclare qu'une note doit être comprise entre 0 et 20, les programmes de saisie n'ont plus à tester les valeurs en entrée mais seulement le code retour après connexion à la base ; on déporte les contraintes côté serveur).

Les contraintes de colonnes peuvent être déclarées de deux manières :

- En même temps que la colonne (valable pour les contraintes monocolonnes), ces contraintes sont dites « en ligne » (*inline constraints*). L'exemple précédent en déclare deux.

- Une fois la colonne déclarée, ces contraintes ne sont pas limitées à une colonne et peuvent être personnalisées par un nom (*out-of-line constraints*).

En nommant chacune de vos contraintes de colonnes, vous disposez de quatre types possibles.

```
CONSTRAINT nomContrainte
  • UNIQUE (colonne1 [,colonne2]…)
  • PRIMARY KEY (colonne1 [,colonne2]…)
  • FOREIGN KEY (colonne1 [,colonne2]…)
    REFERENCES [schéma.]nomTablePere (colonne1 [,colonne2]…)
    [ON DELETE { CASCADE | SET NULL }]
  • CHECK (condition)
```

- La contrainte `UNIQUE` impose une valeur distincte sur les colonnes concernées (les `NULL` font exception à moins que `NOT NULL` soit aussi appliqué sur chaque colonne).
- La contrainte `PRIMARY KEY` déclare la clé primaire, qui impose une valeur distincte sur les colonnes concernées (`NOT NULL` est aussi appliqué sur chaque colonne).
- La contrainte `FOREIGN KEY` déclare une clé étrangère pour relier cette table à une autre table père (voir la section « Intégrité référentielle » du chapitre 2).
- La contrainte `CHECK` impose une condition simple ou complexe entre les colonnes de la table. Par exemple, `CHECK(nb_passagers>0)` interdira toute valeur négative tandis que `CHECK(aero_dep!=aero_arr)` interdira la saisie d'un trajet qui part et revient du même aéroport.

Dans le cas de `UNIQUE` et `PRIMARY KEY`, un index unique est généré sur les colonnes concernées. Vous pouvez disposer de plusieurs contraintes `UNIQUE` mais seule une clé primaire est autorisée.

Si vous ne nommez pas une de vos contraintes, un nom sera généré sous la forme suivante (figure 1-2 ci-contre).

Nous verrons au chapitre 3 comment ajouter, supprimer, désactiver, réactiver et différer des contraintes (options de la commande `ALTER TABLE`).

Conventions recommandées

Adoptez les conventions d'écriture suivantes pour vos contraintes :

- Préfixez par `pk_` le nom d'une contrainte clé primaire, `fk_` une clé étrangère, `ck_` une vérification, `un_` une unicité.
- Pour une contrainte clé primaire, suffixez du nom de la table la contrainte (exemple `pk_Pilote`).
- Pour une contrainte clé étrangère, renseignez (ou abrégez) les noms de la table source, de la clé, et de la table cible (exemple `fk_Pil_compa_Comp`).

Le script d'écriture des tables suivantes respecte ces conventions. La clé étrangère concrétise une association *un-à-plusieurs* entre les deux tables. Ici, il s'agit de relier chaque vol à sa compagnie (pour plus de détails concernant la modélisation, consultez la bibliographie « UML 2 pour les bases de données »).

Tableau 1-3 Contraintes en ligne et nommées

Tables	Contraintes

`CREATE TABLE compagnie`
`(comp VARCHAR2(4),`
`nom_comp VARCHAR2(15),`
`date_creation DATE CONSTRAINT nn_date_crea NOT NULL,`
`CONSTRAINT pk_compagnie PRIMARY KEY(comp),`
`CONSTRAINT un_nom_comp UNIQUE(nom_comp));`
	Une contrainte en ligne et deux contraintes hors ligne.

`CREATE TABLE vol_jour`
`(num_vol VARCHAR2(6) NOT NULL,`
`aero_dep VARCHAR2(3) CONSTRAINT nn_depart NOT NULL,`
`aero_arr VARCHAR2(3) CONSTRAINT nn_arrivee NOT NULL,`
`comp VARCHAR2(4) DEFAULT 'AF',`
`jour_vol DATE NOT NULL,`
`nb_passagers NUMBER(3),`
`CONSTRAINT pk_vol_jour PRIMARY KEY(num_vol, jour_vol),`
`CONSTRAINT fk_vol_jour_comp_compagnie`
`FOREIGN KEY(comp)`
`REFERENCES compagnie(comp),`
`CONSTRAINT ck_nb_pax CHECK (nb_passagers>0),`
`CONSTRAINT ck_trajet CHECK (aero_dep != aero_arr));`
	Une contrainte en ligne nommée (`NOT NULL`) et quatre contraintes hors ligne nommées :
	• Clé primaire.
	• `CHECK` (nombre d'heures de vol compris entre 0 et 20000).
	• `UNIQUE` (homonymes interdits).
	• Clé étrangère.

La figure suivante présente le détail des contraintes (capture d'écran de l'outil SQL Developer).

Figure 1-2 Contraintes d'une table

VOL_JOUR

Colonnes | Données | Contraintes | Droits | Statistiques | Déclencheurs | Flashback | Dépendances | Détails

	CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION
1	CK_NB_PAX	Check	nb_passagers>0
2	CK_TRAJET	Check	aero_dep != aero_arr
3	FK_VOL_JOUR_COMP_COMPAGNIE	Foreign_Key	(null)
4	NN_ARRIVEE	Check	"AERO_ARR" IS NOT NULL
5	NN_DEPART	Check	"AERO_DEP" IS NOT NULL
6	PK_VOL_JOUR	Primary_Key	(null)
7	SYS_C0010521	Check	"NUM_VOL" IS NOT NULL
8	SYS_C0010524	Check	"JOUR_VOL" IS NOT NULL

L'ordre de création des contraintes hors ligne n'est pas important au sein d'une table.

En revanche, l'ordre de création des tables est imposé, si les contraintes sont créées en même temps que les tables. En effet, il existe une certaine hiérarchie à respecter pour les clés étrangères : il faut d'abord créer les tables référentes, puis les tables qui en dépendent (la destruction des tables se fera dans l'ordre inverse).

Il est possible de créer les contraintes après avoir créé les tables via la commande `ALTER TABLE` (voir le chapitre 3).

Types des colonnes

Pour décrire les colonnes d'une table, Oracle fournit les types prédéfinis suivants (*built-in datatypes*) :

- caractères (`CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `CLOB`, `NCLOB`, `LONG`) ;
- valeurs numériques `NUMBER` ;
- date/heure (`DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`) ;
- données binaires (`BLOB`, `BFILE`, `RAW`, `LONG RAW`) ;
- adressage des enregistrements `ROWID`.

Détaillons à présent ces types. Nous verrons comment utiliser les plus courants au chapitre 2 et les autres au fil de l'ouvrage.

Caractères

Le tableau 1-4 décrit les types convenant aux données textuelles. `NCHAR`, `NVARCHAR2` et `NCLOB` permettent de stocker des caractères Unicode (*multibyte*). Cette méthode de codage fournit une valeur unique pour chaque caractère quels que soient la plate-forme, le programme ou la langue.

Réservez le type `CHAR` aux données textuelles de taille fixe et constante.

Depuis Oracle 9, le type `VARCHAR` est remplacé par `VARCHAR2`. Le premier gérait des chaînes maximales de 2 000 caractères et utilisait des `NULL` pour compléter chaque donnée. Le second est plus puissant en termes de stockage ; il n'occupe pas d'espace supplémentaire et n'utilise pas de `NULL` en interne.

Depuis la version 12*c*, la taille maximale d'un `VARCHAR2` ou `NVARCHAR2` peut être étendue à 32 767 octets (32 Ko) si le paramètre d'initialisation `MAX_STRING_SIZE` est positionné à `EXTENDED` (`STANDARD` par défaut). Une fois positionné, il ne vous sera plus possible de revenir à un comportement standard (limitation à 4 000 caractères).

Tableau 1-4 Types de données caractères

Type	Description	Commentaires pour une colonne
`CHAR(n [BYTE | CHAR])`	Chaîne fixe de *n* caractères ou octets.	Taille fixe (complétée par des blancs si nécessaire). Maximum de 2 000 octets ou caractères.
`VARCHAR2(n [BYTE|CHAR])`	Chaîne variable de *n* caractères ou octets.	Taille variable. Maximum de 4 000 octets ou caractères.
`NCHAR(n)`	Chaîne fixe de *n* caractères Unicode.	Taille fixe (complétée par des blancs si nécessaire). Taille double pour le jeu `AL16UTF16` et triple pour le jeu `UTF8`. Maximum de 2 000 caractères.
`NVARCHAR2(n)`	Chaîne variable de *n* caractères Unicode.	Taille variable. Mêmes caractéristiques que `NCHAR` sauf pour la taille maximale qui est ici de 4 000 octets.
`CLOB`	Flot de caractères (`CHAR`).	Jusqu'à 4 gigaoctets.
`NCLOB`	Flot de caractères Unicode (`NCHAR`).	Idem `CLOB`.
`LONG`	Flot variable de caractères.	Jusqu'à 2 gigaoctets. Toujours fourni pour assurer la compatibilité, mais à remplacer par le type `CLOB`.
`XMLTYPE`	Stockage de documents XML	Jusqu'à 4 gigaoctets.

Valeurs numériques

Le type `NUMBER` sert à stocker des entiers positifs ou négatifs, des réels à virgule fixe ou flottante. La plage de valeurs possibles va de $\pm 1 \times 10^{-130}$ à $\pm 9.9\ldots99 \times 10^{125}$ (trente-huit 9 suivis de quatre-vingt-huit 0).

Tableau 1-5 Type de données numériques

Type	Description	Commentaires pour une colonne
`NUMBER[(t,d)]`	Flottant de *t* chiffres dont *d* décimales.	Maximum pour *t* : 38. Plage pour *d* : [–84, +127]. Espace maximum utilisé : 21 octets.

Lorsque la valeur de *d* est négative, l'arrondi se réalise à gauche de la décimale comme le montre le tableau suivant.

Tableau 1-6 Représentation du nombre 7456123.89

Type	Description
`NUMBER`	7456123.89
`NUMBER(9)`	7456124
`NUMBER(9,2)`	7456123.89
`NUMBER(9,1)`	7456123.9
`NUMBER(6)`	Précision inférieure à la taille du nombre.
`NUMBER(7,-2)`	7456100
`NUMBER(-7,2)`	Précision inférieure à la taille du nombre.

Déterminez toujours un nombre de décimales fixe de sorte à ne pas subir des arrondis, et donc des approximations, lors de calculs importants (sur des montants de facture ou des soldes de comptes bancaires, par exemple).

Pour définir des colonnes clé primaire, fixez toujours un nombre de décimales à zéro (par exemple, `NUMBER(3,0)` qui est identique à `NUMBER(3)`).

Enfin, n'utilisez pas toujours des entiers pour définir des clés primaire numériques, (par exemple, un numéro de Sécurité sociale `CHAR(13)` est préférable à `NUMBER(13)` car vous n'opérerez jamais de calculs sur ces données, juste des tris ou des extractions de parties). De plus, si la taille de ce type de donnée n'est pas fixe, vous pourrez compléter avec des 0 devant (ce qui n'est pas possible pour les numériques).

Depuis la version 12*c*, il est possible d'utiliser un type numérique (entier) pour définir une colonne auto-incrémentée (voir le chapitre 2). Le mot-clé qui est utilisé dans les instructions `CREATE TABLE` et `ALTER TABLE` pour désigner un tel mécanisme est `GENERATED… AS IDENTITY…`

Flottants

Depuis Oracle 10*g*, deux types numériques apparaissent : `BINARY_FLOAT` et `BINARY_DOUBLE` qui permettent de représenter des grands nombres (plus importants que ceux définis par `NUMBER`) sous la forme de flottants. Les nombres flottants peuvent disposer d'une décimale située à tout endroit (de la première position à la dernière) ou ne pas avoir de décimale du tout. Un exposant peut éventuellement être utilisé (exemple : $1.777\ e^{-20}$). Une échelle de valeurs ne peut être imposée à un flottant puisque le nombre de chiffres apparaissant après la décimale n'est pas restreint.

Tableau 1-7 Types de flottants

Type	Description	Commentaire pour une colonne
`BINARY_FLOAT`	Flottant simple précision.	Sur 5 octets (un représentant la longueur). Valeur entière maximale $3,4 \times 10^{+38}$, valeur entière minimale $-3,4 \times 10^{+38}$. Plus petite valeur positive $1,2 \times 10^{-38}$, plus petite valeur négative $-1,2 \times 10^{-38}$.
`BINARY_DOUBLE`	Flottant double précision.	Sur 9 octets (un représentant la longueur). Valeur entière maximale $1,79 \times 10^{+308}$, valeur entière minimale $-1,79 \times 10^{+308}$. Plus petite valeur positive $2,3 \times 10^{-308}$, plus petite valeur négative $-2,3 \times 10^{-308}$.

Le stockage des flottants diffère de celui des NUMBER en ce sens que le mécanisme de représentation interne est propre à Oracle. Pour une colonne NUMBER, les nombres à virgule ont une précision décimale. Pour les types BINARY_FLOAT et BINARY_DOUBLE, les nombres à virgule ont une précision exprimée en binaire.

Oracle fournit également le type ANSI FLOAT qui peut aussi s'écrire FLOAT(*n*). L'entier *n* (de 1 à 126) indique la précision binaire. Afin de convertir une précision binaire en précision décimale, il convient de multiplier l'entier par 0.30103. La conversion inverse nécessite de multiplier *n* par 3.32193. Le maximum de 126 bits est à peu près équivalent à une précision de 38 décimales.

L'écriture d'un flottant est la suivante :

```
[+|-] {chiffre [chiffre]...[.] [chiffre [chiffre]...].chiffre [chiffre]...}
        [e[+|-] chiffre [chiffre]...] [f|d]
```

- e (ou E) indique la notation scientifique (mantisse et exposant) ;
- f (ou F) indique que le nombre est de type BINARY_FLOAT ;
- d (ou D) indique que le nombre est de type BINARY_DOUBLE.

Si le type n'est pas explicitement précisé, l'expression est considérée comme de type NUMBER.

Date/heure

- Le type DATE permet de stocker des moments ponctuels, la précision est composée du siècle, de l'année, du mois, du jour, de l'heure, des minutes et des secondes.
- Le type TIMESTAMP est plus précis dans la définition d'un moment (fraction de seconde).
- Le type TIMESTAMP WITH TIME ZONE prend en compte les fuseaux horaires.
- Le type TIMESTAMP WITH LOCAL TIME ZONE permet de faire la dichotomie entre une heure côté serveur et une heure côté client.
- Le type INTERVAL YEAR TO MONTH permet d'extraire une différence entre deux moments avec une précision mois/année.
- Le type INTERVAL DAY TO SECOND permet d'extraire une différence plus précise entre deux moments (précision de l'ordre de la fraction de seconde).

Tableau 1-8 Types de données date/heure

Type	Description	Commentaires pour une colonne
`DATE`	Date et heure du 1er janvier 4712 avant J.-C. au 31 décembre 4712 après J.-C.	Sur 7 octets. Le format par défaut est spécifié par le paramètre `NLS_DATE_FORMAT`.
`INTERVAL YEAR (an) TO MONTH`	Période représentée en années et mois.	Sur 5 octets. La précision de *an* va de 0 à 9 (par défaut 2).
`INTERVAL DAY (jo) TO SECOND (fsec)`	Période représentée en jours, heures, minutes et secondes.	Sur 11 octets. Les précisions *jo* et *fsec* vont de 0 à 9 (par défaut 2 pour le jour et 6 pour les fractions de secondes).
`TIMESTAMP (fsec)`	Date et heure incluant des fractions de secondes (précision qui dépend du système d'exploitation).	De 7 à 11 octets. La valeur par défaut du paramètre d'initialisation est située dans `NLS_TIMESTAMP_FORMAT`. La précision des fractions de secondes va de 0 à 9 (par défaut 6).
`TIMESTAMP (fsec) WITH TIME ZONE`	Date et heure avec le décalage de Greenwich (UTC) au format '*h:m*' (*heures:minutes* par rapport au méridien, exemple : '-5:0').	Sur 13 octets. La valeur par défaut du paramètre de l'heure du serveur est située dans `NLS_TIMESTAMP_TZ_FORMAT`.
`TIMESTAMP (fsec) WITH LOCAL TIME ZONE`	Comme le précédent mais cadré sur l'heure locale (client) qui peut être différente de celle du serveur.	De 7 à 11 octets.

N'utilisez jamais un format textuel pour stocker des dates ou des heures (par exemple, `CHAR(10)` pour un format *jj/mm/aaaaa*) car vous ne pourrez pas bénéficier de contrôles et de calculs, toujours nécessaires à un moment donné dans ces cas-là.

Données binaires

Le tableau 1-9 présente les types permettant de stocker des données non structurées (images, sons, etc.).

Tableau 1-9 Types de données binaires

Type	Description	Commentaires pour une colonne
`BLOB`	Données binaires non structurées.	Jusqu'à 4 gigaoctets.
`BFILE`	Données binaires stockées dans un fichier externe à la base.	idem.
`RAW(size)`	Données binaires.	Jusqu'à 2 000 octets.
`LONG RAW`	Données binaires.	Jusqu'à 2 gigaoctets, toujours fourni pour assurer la compatibilité, mais à remplacer par le type `BLOB`.

Structure d'une table (DESC)

DESC (raccourci de DESCRIBE) est une commande SQL*Plus, car elle n'est comprise que dans l'interface de commandes d'Oracle. Elle permet d'extraire la structure brute d'une table. Elle peut aussi s'appliquer à une vue ou un synonyme. Enfin, elle révèle également les paramètres d'une fonction ou procédure cataloguée.

```
DESC[RIBE] [schéma.]élément
```

Si le schéma n'est pas indiqué, il s'agit de celui de l'utilisateur connecté. L'élément désigne le nom d'une table, vue, procédure, fonction ou synonyme.

La structure brute des tables précédemment créées est présentée à l'aide des commandes suivantes. Il n'y a que le nom, le type et la non-nullité de la colonne qui apparaissent. Le nom des contraintes n'est pas indiqué ici (comme peut le produire l'outil SQL Developer, voir figure 1-2).

Figure 1-3 *Structure brute des tables*

```
SQL> desc vol_jour
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- ------------
 NUM_VOL                                   NOT NULL VARCHAR2(6)
 AERO_DEP                                  NOT NULL VARCHAR2(3)
 AERO_ARR                                  NOT NULL VARCHAR2(3)
 COMP                                               VARCHAR2(4)
 JOUR_VOL                                  NOT NULL DATE
 NB_PASSAGERS                                       NUMBER(3)

SQL> desc compagnie
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- ------------
 COMP                                      NOT NULL VARCHAR2(4)
 NOM_COMP                                           VARCHAR2(15)
 DATE_CREATION                             NOT NULL DATE
```

Commentaires stockés (COMMENT)

Les commentaires stockés permettent de documenter une table, une colonne ou une vue. L'instruction SQL pour créer un commentaire est COMMENT.

```
COMMENT ON { TABLE [schéma.]nomTable |
COLUMN [schéma.]nomTable.nomColonne }
IS 'Texte décrivant le commentaire';
```

Pour supprimer un commentaire, il suffit de le redéfinir en inscrivant une chaîne vide (' ') dans la clause IS. Une fois définis, nous verrons à la section « Dictionnaire des données » du chapitre 5 comment retrouver ces commentaires.

Le premier commentaire du script ci-après documente la table Compagnie, les trois suivants renseignent trois colonnes de cette table. La dernière instruction supprime le commentaire à propos de la colonne nomComp.

```
COMMENT ON TABLE Compagnie IS 'Table des compagnies aériennes françaises';
COMMENT ON COLUMN Compagnie.comp    IS 'Code abréviation de la compagnie';
COMMENT ON COLUMN Compagnie.nomComp IS 'Un mauvais commentaire';
COMMENT ON COLUMN Compagnie.ville   IS 'Ville de la compagnie, défaut :
Paris';
COMMENT ON COLUMN Compagnie.nomComp IS '';
```

Noms des objets

Chaque objet ou constituant de la base (table, index, contrainte, colonne, variable, etc.) est nommé à l'aide d'un identifiant de 1 à 30 caractères (composé de lettres, de chiffres ou des caractères _, $ et #).

Le nom peut être écrit entre guillemets – la casse doit alors être impérativement respectée de même que l'utilisation des guillemets (on parle de *quoted identifier*). Le seul exemple présenté dans cet ouvrage qui adopte ce style de notation est le suivant ; vous y découvrirez la possibilité de décrire des identifiants sous la forme de mots séparés par des espaces (ce qui n'est pas conseillé).

Figure 1-4 *Nommage de type « quoted identifier »*

```
SQL> CREATE TABLE "vols du jour"
  2   ("numVol"     VARCHAR2(6)  NOT NULL,
  3    "comp"       VARCHAR2(4)  DEFAULT 'AF',
  4    "jour vol"   DATE         NOT NULL,
  5    CONSTRAINT   "pk vols du jour"  PRIMARY KEY("numVol", "jour vol"));

Table créée.
SQL> DESC "vols du jour"
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- ----------------------
 numVol                                    NOT NULL VARCHAR2(6)
 comp                                               VARCHAR2(4)
 jour vol                                  NOT NULL DATE

SQL> SELECT numVol FROM "vols du jour";
       *
ERREUR à la ligne 1 :
ORA-00904: "NUMVOL" : identificateur non valide
```

Seuls les noms de base (8 caractères) et les noms de *database link* (128 caractères) sont toujours stockés en majuscules et sont insensibles à la casse.

Le nom de chaque colonne doit être unique pour une table (en revanche, le même nom d'une colonne peut être utilisé dans différentes tables). Les noms des objets (tables, colonnes, contraintes, vues, etc.) doivent être uniques au niveau du schéma (en revanche, plusieurs tables peuvent porter le même nom dans différents schémas).

Avec une notation sans guillemets, vous ne devez pas emprunter les mots réservés (`TABLE`, `SELECT`, `INSERT`, `IF`, etc.). Vous trouverez la liste de ces mots réservés dans la documentation officielle à l'annexe D du livre *SQL Reference*.

Il n'est pas recommandé d'utiliser les caractères `$` et `#` (très employés en interne par Oracle).

Les identifiants sans guillemets (*nonquoted identifiers*) ne sont donc pas sensibles à la casse et sont traduits en majuscules dans le dictionnaire des données (voir le chapitre 5). Ainsi, les trois écritures désignent le même identifiant : `aeroport`, `AEROPORT` et `"AEROPORT"`.

Utilisation de SQL Developer Data Modeler

Une des utilisations de l'outil d'Oracle SQL Developer Data Modeler consiste à générer des scripts de création de tables (DDL scripts, DDL pour *Data Definition Language*) après avoir saisi les caractéristiques de chaque table sous une forme graphique (modèle relationnel des données). Ce procédé est appelé *forward engineering* car il chemine dans le sens d'une conception classique pour laquelle l'étape finale est concrétisée par la création des tables.

Dans l'arborescence de gauche, par un clic droit sur l'élément `Modèles relationnels`, choisir `Nouveau modèle relationnel`. Une fois dans le modèle relationnel, les icônes indiquées vous permettront de créer vos tables et toutes les liaisons entre elles (clés étrangères). À titre d'exemple, créons deux tables reliées par une clé étrangère (ici, un pilote qui est rattaché à sa compagnie).

Pour créer une table, vous devez la nommer dans la fenêtre de saisie (le choix `Appliquer` modifiera le nom complet), puis définir ses colonnes. En choisissant l'entrée `Colonnes`, le symbole « + » vous permettra de saisir le nom et le type de chaque colonne de la table. N'ajoutez aucune contrainte pour l'instant (clé primaire et clé étrangère), contentez-vous de saisir les colonnes sans ajouter de colonnes de nature clé étrangère.

Figure 1-5 *Création d'un modèle relationnel avec Data Modeler*

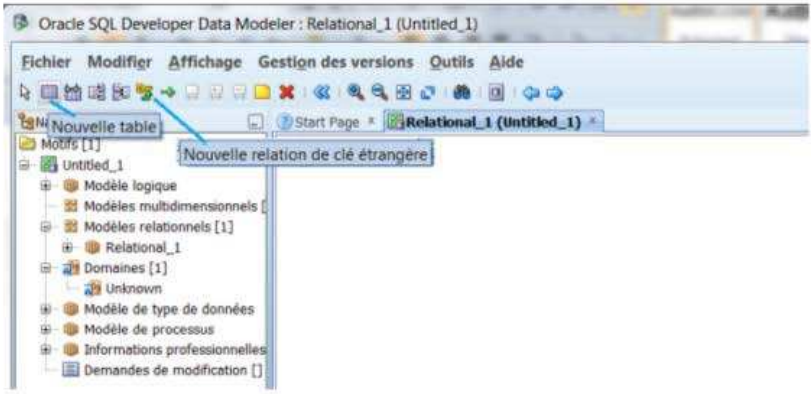
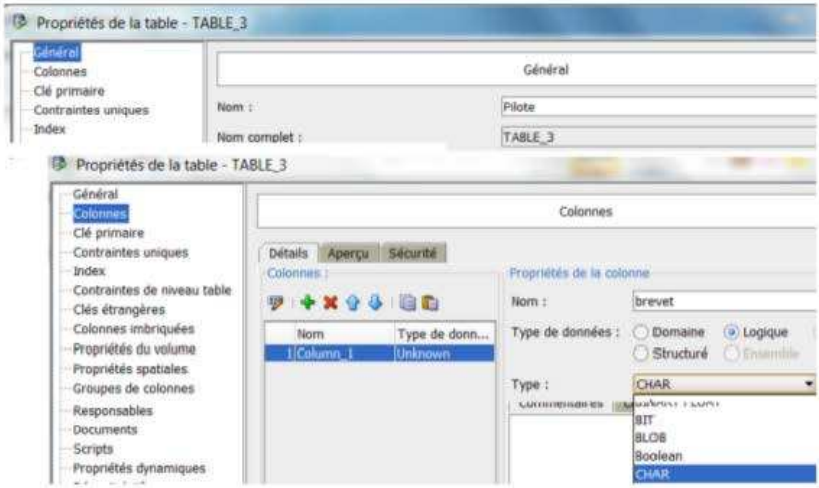
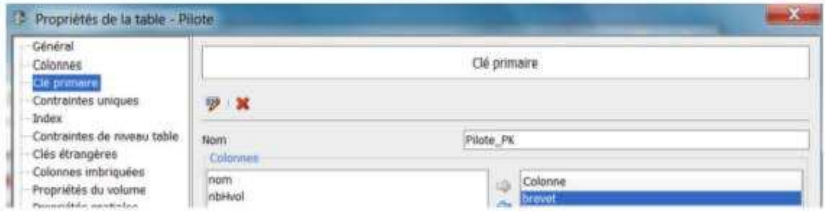


Figure 1-6 *Colonnes d'une table avec Data Modeler*



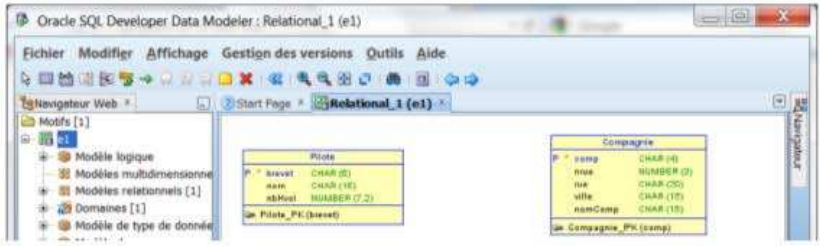
Définissez ensuite la clé primaire de chaque table (colonne `brevet` de la table `Pilote` et colonne `comp` de la table `Compagnie`).

Figure 1-7 *Clé primaire avec Data Modeler*



À l'issue de cette étape, le diagramme se modifie pour faire apparaître les deux nouvelles contraintes.

Figure 1-8 *Tables dotées d'une clé primaire avec Data Modeler*



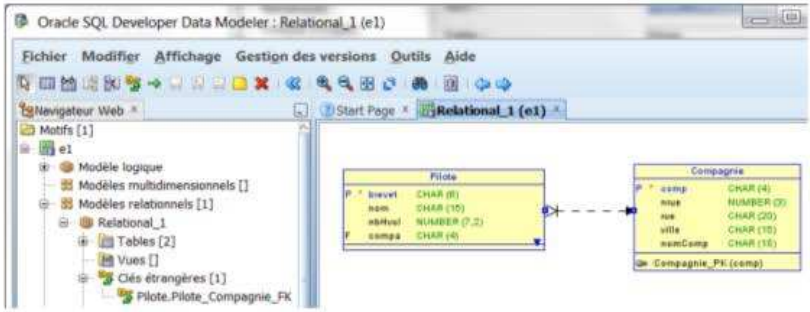
Reliez la table `Compagnie` à la table `Pilote` en sélectionnant l'icône de clé étrangère. Une boîte de dialogue s'affiche alors et décrit les caractéristiques de la nouvelle contrainte référentielle (voir le chapitre 3).

Assurez-vous que la table source corresponde à la table de référence (ici, les compagnies) et n'interprétez pas le terme « source » comme source de la flèche qui décrit le sens du lien, et « cible » comme la table cible de ce lien. C'est tout l'inverse.

Vous remarquerez que la colonne de type clé étrangère générée est automatiquement nommée à l'aide de la table et de la colonne de référence : *tablesource_cleprimaire* (ici, `Compagnie_comp`, que nous choisissons de renommer `compa`).

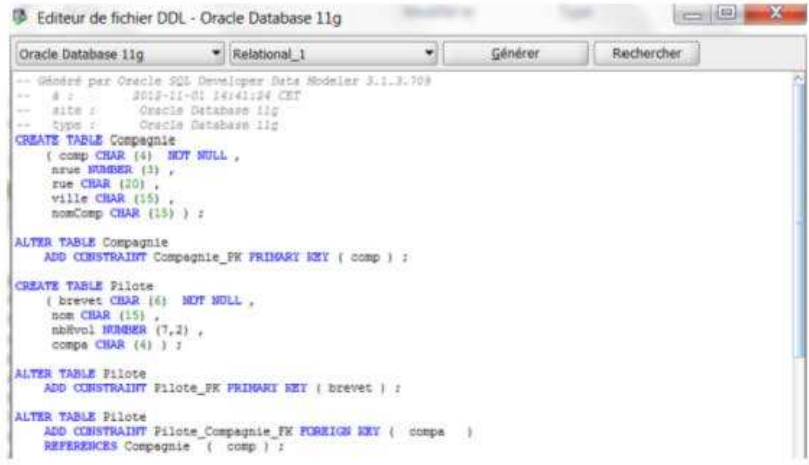
Figure 1-9 *Définition d'une clé étrangère avec Data Modeler*

Figure 1-10 *Modèle relationnel final avec Data Modeler*



La génération du script SQL s'opère par le menu `Fichier/Exporter/Fichier DDL`. Sélectionner la version du SGBD cible, puis choisir `Générer`. Tous les éléments du modèle relationnel sont sélectionnés par défaut, mais vous pouvez volontairement écarter certaines tables du script. Une fois votre sélection faite, le script SQL se génère automatiquement. Vous noterez que les contraintes sont déclarées après les tables (voir la commande `ALTER TABLE` au chapitre 3). Ce procédé est bien adapté à la majorité des outils de conception, qui l'adoptent pour leur processus de *reverse engineering*.

Figure 1-11 *Script de génération des tables avec Data Modeler*



Suppression des tables

Il vous sera sans doute utile d'écrire un script qui supprime tout ou partie des tables de votre schéma. Ainsi, vous pourrez recréer un ensemble homogène de tables (comme une sorte de base « vierge ») à la demande. Bien entendu, si des données sont présentes dans vos tables, vous devrez opter pour une stratégie d'exportation ou de sauvegarde avant de réinjecter vos données dans les nouvelles tables. À ce stade de la lecture de l'ouvrage, vous n'en êtes pas là, et le script de suppression vous permettra de corriger les erreurs de syntaxe que vous risquez fort de faire lors de l'écriture du script de création des tables.

Si vous définissez des contraintes en même temps que les tables (dans l'ordre `CREATE TABLE...`), vous devrez respecter l'ordre suivant : tables « pères » (de référence), puis les tables « fils » (dépendantes). L'ordre de suppression des tables, pour des raisons de cohérence, est totalement inverse : vous devez supprimer les tables « fils » d'abord, puis les tables de référence. Dans l'exemple présenté à la section « Conventions recommandées », il serait malvenu de vouloir supprimer la table `Compagnie` avant de supprimer la table `Pilote`. En effet, la clé étrangère `compa` n'aurait plus de sens. Cela n'est d'ailleurs pas possible sans forcer l'option `CASCADE CONSTRAINTS` (voir plus loin).

```
DROP TABLE [schéma.]nomTable [CASCADE CONSTRAINTS] [PURGE];
```

- Pour pouvoir supprimer une table dans son schéma, il faut que la table appartienne à l'utilisateur. Si l'utilisateur a le privilège `DROP ANY TABLE`, il peut supprimer une table dans tout schéma.
- L'instruction `DROP TABLE` entraîne la suppression des données, de la structure, de la description dans le dictionnaire des données, des index, des déclencheurs associés (*triggers*) et la récupération de la place dans l'espace de stockage.

- `CASCADE CONSTRAINTS` permet de s'affranchir des clés étrangères actives contenues dans d'autres tables et qui référencent la table à supprimer. Cette option détruit les contraintes des tables « fils » associées sans rien modifier aux données qui y sont stockées (voir section « Intégrité référentielle » du prochain chapitre).
- `PURGE` permet de récupérer instantanément l'espace alloué aux données de la table (les blocs de données) sans les disposer dans la poubelle d'Oracle (*recycle bin*) .

Certains éléments qui utilisaient la table (vues, synonymes, fonctions ou procédures) ne sont pas supprimés mais sont temporairement inopérants. En revanche, les éventuels index et déclencheurs sont supprimés.

- Une suppression (avec `PURGE`) ne peut pas être annulée par la suite.
- La suppression d'une table sans `PURGE` peut être récupérée via l'espace `recycle bin` par la technologie *flashback* (ce mécanisme, qui relève davantage de l'administration, sort du cadre de cet ouvrage).

Si les contraintes sont déclarées au sein des tables (dans chaque instruction `CREATE TABLE`), il vous suffit de relire à l'envers le script de création des tables pour en déduire l'ordre de suppression.

Utilisez avec parcimonie l'option `CASCADE CONSTRAINTS` qui fera fi, sans vous le dire, du mécanisme de l'intégrité référentielle assuré par les clés étrangères (voir le chapitre 3).

Exercices

L'objectif de ces exercices est de créer des tables, leur clé primaire et des contraintes de vérification (`NOT NULL` et `CHECK`). La première partie des exercices (de 1.1 à 1.4) concerne la base *Parc Informatique*). Le dernier exercice traite d'une autre base (*Chantiers*) que vous pouvez appliquer à la version d'Oracle à partir de la 11*g*.

Exercice 1.1 Présentation de la base de données

Une entreprise désire gérer son parc informatique à l'aide d'une base de données. Le bâtiment est composé de trois étages. Chaque étage possède son réseau (ou segment distinct) Ethernet. Ces réseaux traversent des salles équipées de postes de travail. Un poste de travail est une machine sur laquelle sont installés certains logiciels. Quatre catégories de postes de travail sont recensées (stations Unix, terminaux X, PC Windows et PC NT). La base de données devra aussi décrire les installations de logiciels.

Les noms et types des colonnes sont les suivants :

Tableau 1-10 Caractéristiques des colonnes

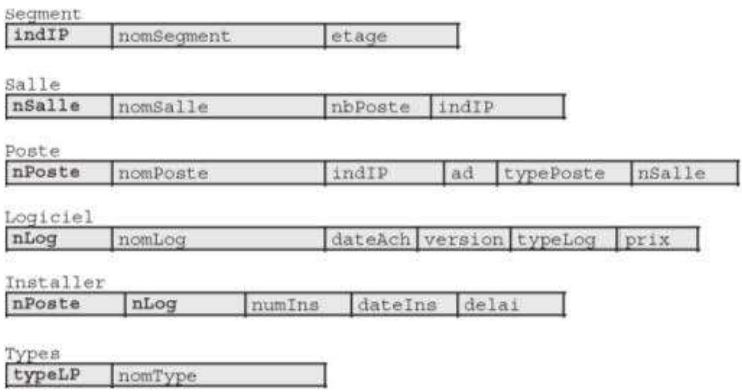
Colonne	Commentaires	Types
indIP	Trois premiers groupes IP (exemple : 130.120.80).	VARCHAR2(11)
nomSegment	Nom du segment.	VARCHAR2(20)
etage	Étage du segment.	NUMBER(2)
nSalle	Numéro de la salle.	VARCHAR2(7)
nomSalle	Nom de la salle.	VARCHAR2(20)
nbPoste	Nombre de postes de travail dans la salle.	NUMBER(2)
nPoste	Code du poste de travail.	VARCHAR2(7)
nomPoste	Nom du poste de travail.	VARCHAR2(20)
ad	Dernier groupe de chiffres IP (exemple : 11).	VARCHAR2(3)
typePoste	Type du poste (Unix, TX, PCWS, PCNT).	VARCHAR2(9)
dateIns	Date d'installation du logiciel sur le poste.	DATE
nLog	Code du logiciel.	VARCHAR2(5)
nomLog	Nom du logiciel.	VARCHAR2(20)
dateAch	Date d'achat du logiciel.	DATE
version	Version du logiciel.	VARCHAR2(7)
typeLog	Type du logiciel (Unix, TX, PCWS, PCNT).	VARCHAR2(9)
prix	Prix du logiciel.	NUMBER(6,2)
numIns	Numéro séquentiel des installations.	NUMBER(5)
dateIns	Date d'installation du logiciel.	DATE
delai	Intervalle entre achat et installation.	INTERVAL DAY(5) TO SECOND(2),
typeLP	Types des logiciels et des postes.	VARCHAR2(9)
nomType	Noms des types (Terminaux X, PC Windows...).	VARCHAR2(20)

Exercice 1.2 Création des tables

Écrivez puis exécutez le script SQL (que vous appellerez `creParc.sql`) de création des tables avec leur clé primaire (en gras dans le schéma suivant) et les contraintes suivantes :

- Les noms des segments, des salles et des postes sont non nuls.
- Le domaine de valeurs de la colonne `ad` s'étend de 0 à 255.
- La colonne `prix` est supérieure ou égale à 0.
- La colonne `dateIns` est égale à la date du jour par défaut.

Figure 1-12 *Schéma des tables*



Exercice 1.3 Structure des tables

Écrivez puis exécutez le script SQL (que vous appellerez `descParc.sql`) qui affiche la description de toutes ces tables (en utilisant des commandes `DESC`). Comparez avec le schéma.

Exercice 1.4 Destruction des tables

Écrivez puis exécutez le script SQL de destruction des tables (que vous appellerez `dropParc.sql`). Lancez ce script puis à nouveau celui de la création des tables.

Exercice 1.5 Schéma de la base *Chantiers* (Oracle 11*g*)

Une société désire informatiser les visites des chantiers de ses employés. Pour définir cette base de données, une première étude fait apparaître les informations suivantes :

- Chaque employé est modélisé par un numéro, un nom et une qualification.
- Un chantier est caractérisé par un numéro, un nom et une adresse.
- L'entreprise dispose de véhicules pour lesquels il est important de stocker pour le numéro d'immatriculation, le type (un code valant par exemple 0 pour une camionnette, 1 pour une moto et 2 pour une voiture) ainsi que le kilométrage en fin d'année.
- Le gestionnaire a besoin de connaître les distances parcourues par un véhicule pour chaque visite d'un chantier.
- Chaque jour, un seul employé sera désigné conducteur des visites d'un véhicule.
- Pour chaque visite, il est important de pouvoir connaître les employés transportés.

Les colonnes à utiliser sont les suivantes :

Tableau 1-11 Caractéristiques des colonnes à ajouter

Colonne	Commentaires	Types
kilometres	Kilométrage d'un véhicule lors d'une sortie.	NUMBER
n_conducteur	Numéro de l'employé conducteur.	VARCHAR2(4)
n_transporte	Numéro de l'employé transporté.	VARCHAR2(4)

L'exercice consiste à compléter le schéma relationnel ci-après (ajout de colonnes et définition des contraintes de clé primaire et étrangère).

```
CREATE TABLE Employe (n_emp VARCHAR(4),nom_emp VARCHAR(20),
 qualif_emp VARCHAR(12), CONSTRAINT pk_emp PRIMARY KEY(n_emp));

CREATE TABLE Chantier (n_chantier VARCHAR(10), nom_ch VARCHAR(10),
  adresse_ch VARCHAR(15), CONSTRAINT pk_chan PRIMARY KEY(n_chantier));

CREATE TABLE Vehicule (n_vehicule VARCHAR(10), type_vehicule VARCHAR(1),
  kilometrage NUMBER, CONSTRAINT pk_vehi PRIMARY KEY(n_vehicule));

CREATE TABLE Visite(n_chantier VARCHAR(10), n_vehicule VARCHAR(10),
 date_jour DATE,  ...
 CONSTRAINT pk_visite PRIMARY KEY(...),
 CONSTRAINT fk_depl_chantier FOREIGN KEY(n_chantier) ...,
 CONSTRAINT fk_depl_vehicule FOREIGN KEY(n_vehicule)  ...,
 CONSTRAINT fk_depl_employe FOREIGN KEY(n_conducteur) ... );

CREATE TABLE Transporter (...
 CONSTRAINT pk_transporter PRIMARY KEY (...),
 CONSTRAINT fk_transp_visite FOREIGN KEY ... ,
 CONSTRAINT fk_transp_employe FOREIGN KEY ...);
```

Chapitre 2

Manipulation des données

Ce chapitre décrit une partie de l'aspect DML (*Data Manipulation Language*) du langage SQL d'Oracle. Bien qu'il existe d'autres possibilités d'insérer des données (techniques d'importation ou de chargement), SQL propose trois instructions de base pour manipuler des données :

- l'insertion d'enregistrements : `INSERT` ;
- la modification de données : `UPDATE` ;
- la suppression d'enregistrements : `DELETE`.

Insertions d'enregistrements (INSERT)

Pour pouvoir insérer des enregistrements dans une table, il faut que cette dernière soit dans votre schéma ou que vous ayez reçu le privilège `INSERT` sur la table. Si vous avez le privilège `INSERT ANY TABLE`, vous pouvez ajouter des données dans n'importe quelle table de tout schéma.

Il existe plusieurs possibilités d'insertion : l'insertion monoligne qui ajoute un enregistrement par instruction (que nous allons détailler maintenant) et l'insertion multiligne qui insère plusieurs valeurs (que nous détaillerons au chapitre 4).

Syntaxe

La syntaxe simplifiée de l'instruction `INSERT` monoligne est la suivante :

```
INSERT INTO [schéma.] { nomTable | nomVue | requêteSELECT }
   [(colonne1, colonne2…)]
VALUES (valeur1 | DEFAULT, valeur2 | DEFAULT…);
```

À l'aide d'exemples, nous allons détailler les possibilités de cette instruction en considérant la majeure partie des types de données proposés par Oracle.

Renseigner ou pas toutes les colonnes

Le script suivant insère trois compagnies et quatre vols en utilisant différentes options de l'instruction INSERT. Il s'agit de renseigner ou pas les colonnes d'une table par une liste de valeurs de type adéquat. Le mot-clé DEFAULT utilisé en tant que valeur permet d'affecter explicitement une valeur par défaut à la colonne associée.

Tableau 2-1 Insertions de lignes

Instructions SQL	Commentaires

`INSERT INTO compagnie`
`  VALUES ('SING', 'Singapore AL', TO_`
`  DATE('19470101','YYYYMMDD'));`
	Les valeurs sont renseignées dans l'ordre de la structure de la table.

`INSERT INTO compagnie`
`  (nom_comp, comp, date_creation)`
`  VALUES ('Air France', 'AF', TO_DATE('19330101',`
`  'YYYYMMDD'));`
	Les valeurs sont renseignées dans l'ordre de la liste.

`INSERT INTO compagnie`
`  (nom_comp, comp, date_creation)`
`  VALUES (NULL, 'GO',`
`  TO_DATE('20141231','YYYYMMDD'));`	

`INSERT INTO vol_jour`
`  (num_vol,aero_dep,aero_arr,jour_vol,nb_passagers)`
`  VALUES ('AF6143', 'TLS', 'ORY', TO_DATE('20141120`
`  15:30','YYYYMMDD HH24:MI'),120);`
	La compagnie est omise, donc la valeur par défaut s'appliquera (AF).

`INSERT INTO vol_jour`
`  (num_vol,aero_dep,aero_arr,jour_vol)`
`  VALUES ('AF6145', 'ORY', 'TLS', TO_DATE('20141120`
`  18:45','YYYYMMDD HH24:MI'));`
	Le nombre de passagers est omis, donc NULL s'appliquera.

`INSERT INTO vol_jour`
`  (num_vol,aero_dep,aero_arr,comp,jour_vol,`
`  nb_passagers)`
`  VALUES ('SQ747', 'CDG', 'SIN', 'SING',`
`  TO_DATE('20141120 19:30',`
`  'YYYYMMDD HH24:MI'),NULL);`
	Le nombre de passagers est explicitement valué à NULL.

`INSERT INTO vol_jour`
`  (num_vol,aero_dep,aero_arr,comp,jour_vol,nb_`
`  passagers)`
`  VALUES ('AF6550', 'CDG', 'TLS', DEFAULT,`
`  TO_DATE('20141120 20:00',`
`  'YYYYMMDDHH24:MI'),195);`
	La compagnie est explicitement valuée à la valeur par défaut.

Bannissez le premier style d'écriture et renseignez toujours le plus de colonnes possible dans vos instructions INSERT, vous subirez ainsi le moins de comportements par défaut.

Une fois la validation effectuée (par `commit`, voir chapitre 6), le résultat est présenté avec SQL Developer de la manière suivante.

Figure 2-1 *Tables après les insertions*

COMPAGNIE

COMP	NOM_COMP	DATE_CREATION
SING	Singapore AL	01/01/47
AF	Air France	01/01/33
GO	(null)	31/12/14

VOL_JOUR

NUM_VOL	AERO_DEP	AERO_ARR	COMP	JOUR_VOL	NB_PASSAGERS
AF6143	TLS	ORY	AF	20/11/14	120
AF6550	CDG	TLS	AF	20/11/14	195
SQ747	CDG	SIN	SING	20/11/14	(null)
AF6145	ORY	TLS	AF	20/11/14	(null)

Depuis la version 12*c*, toute colonne peut être définie à `DEFAULT ON NULL` *`valeur_defaut`*. Ainsi, l'insertion d'un `NULL`, qu'elle soit explicite ou implicite, sera remplacée par la valeur par défaut.

Ne pas respecter des contraintes

Insérons des vols qui ne respectent pas des contraintes. Les messages renvoyés pour chaque erreur font apparaître le nom de la contrainte. Les valeurs qui sont les facteurs déclenchant sont notées en gras. La première erreur est un doublon de clé primaire, la deuxième un `NULL` interdit et la troisième une condition de vérification. La dernière erreur signifie que la clé étrangère référence un absent (pour plus de détails à ce sujet, consultez la section « Intégrité référentielle »).

Tableau 2-2 Erreur typique de contraintes

Instruction	Message d'erreur
`SQL> INSERT INTO vol_jour (num_vol,aero_dep,aero_arr,comp,jour_vol, nb_passagers) VALUES ('AF6550', 'AGN', 'TLS', 'AF', TO_DATE('20141120 20:00', 'YYYYMMDD HH24:MI'),95);`	ERREUR à la ligne 1 : ORA-00001: violation de contrainte unique (SOUTOU.PK_VOL_JOUR)
`SQL> INSERT INTO vol_jour (num_vol,aero_dep,aero_arr,comp,jour_vol, nb_passagers) VALUES ('AF6530', 'AGN', NULL, 'AF', TO_DATE('20141120 10:00', 'YYYYMMDD HH24:MI'),95);`	ERREUR à la ligne 3 : ORA-01400: impossible d'insérer NULL dans ("SOUTOU"."VOL_JOUR"."AERO_ARR")
`SQL> INSERT INTO vol_jour (num_vol,aero_dep,aero_arr,comp,jour_vol, nb_passagers) VALUES ('AF6530', 'AGN', 'AGN', 'AF', TO_DATE('20141120 10:00', 'YYYYMMDD HH24:MI'),95);`	ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOUTOU.CK_TRAJET) de vérification
`SQL> INSERT INTO vol_jour (num_vol,aero_dep,aero_arr,comp,jour_vol, nb_passagers) VALUES ('AF6530', 'AGN', 'TLS', 'BA', TO_DATE('20141120 10:00', 'YYYYMMDD HH24:MI'),95);`	ERREUR à la ligne 1 : ORA-02291: violation de contrainte d'intégrité (SOUTOU.FK_VOL_JOUR_COMP_COMPAGNIE) - clé parent introuvable

Dates/heures

Nous avons décrit au chapitre 1 les caractéristiques générales des types Oracle pour stocker des éléments de type date/heure.

Type DATE

Déclarons la table `Pilote` qui contient deux colonnes de type `DATE`.

```
CREATE TABLE pilote
(brevet    VARCHAR2(6),          prenom    VARCHAR2(20) NOT NULL,
 nom       VARCHAR2(20) NOT NULL, date_nais DATE NOT NULL,
 embauche  DATE NOT NULL,
 CONSTRAINT pk_pilote PRIMARY KEY(brevet));
```

La première insertion initialise la date de naissance au 5 février 1965 (à zéro heure, zéro minute et zéro seconde), tandis que la date d'embauche inclura les heures, minutes et secondes

par la fonction SYSDATE. La seconde insertion utilise un autre format en entrée et initialisera l'embauche au jour présent (à zéro heure, zéro minute et zéro seconde) par la fonction TRUNC.

```
INSERT INTO Pilote(brevet, prenom, nom, date_nais, embauche)
  VALUES ('B1', 'Christian', 'Mermoz',
          TO_DATE('05/02/1965','DD/MM/YYYY'),SYSDATE);

INSERT INTO Pilote(brevet, prenom, nom, date_nais, embauche)
  VALUES ('B2', 'Christian', 'Mermoz',
          TO_DATE('19650205','YYYYMMDD'), TRUNC(SYSDATE));
```

La fonction TO_DATE doit toujours être utilisée pour appliquer un format à la date qui peut être précis à la seconde. Par exemple, le 5 février 1965 à 6 h 30 sera codé TO_DATE('05-02-1965:06:30','DD-MM-YYYY:HH24:MI').

Nous verrons au chapitre 4 comment afficher les heures, minutes et secondes d'une colonne de type DATE. Nous verrons aussi qu'il est possible d'ajouter ou de soustraire des dates entre elles.

Types TIMESTAMP

La table Evenements contient la colonne arrive (TIMESTAMP) pour stocker des fractions de secondes et la colonne arriveLocalement (TIMESTAMP WITH TIME ZONE) pour considérer aussi le fuseau horaire.

```
CREATE TABLE Evenements
(arrive TIMESTAMP, arriveLocalement TIMESTAMP WITH TIME ZONE);
```

L'insertion suivante initialise :

- la colonne arrive au 5 février 1965 à 9 heures, 30 minutes, 2 secondes et 123 centièmes dans le fuseau défini au niveau de la base ;
- la colonne arriveLocalement au 16 janvier 1965 à 12 heures, 30 minutes, 5 secondes et 98 centièmes dans le fuseau décalé vers l'est de 4 h 30 par rapport au méridien de Greenwich.

```
INSERT INTO Evenements(arrive, arriveLocalement)
  VALUES (TIMESTAMP '1965-02-05 09:30:02.123',
       TIMESTAMP '1965-01-16 12:30:05.98 + 4:30');
```

Le format par défaut de ces types est décrit dans les variables NLS_TIMESTAMP_FORMAT ('YYYY-MM-DD HH:MM:SS.d' d : décimales) et NLS_TIMESTAMP_TZ_FORMAT ('YYYY-MM-DD HH:MM:SS.d ± hh:mn', avec hh:mn en heures-minutes par rapport à Greenwich).

Types INTERVAL

Les types INTERVAL permettent de déclarer des durées et non pas des moments.

La table Durees contient la colonne dureeAnneesMois (INTERVAL YEAR TO MONTH) pour stocker des intervalles en années et en jours, et la colonne dureeJourSecondes (INTERVAL DAY TO SECOND) pour stocker des intervalles en jours, heures, minutes, secondes et fractions de secondes.

```
CREATE TABLE Durees
  (dureeAnneesMois    INTERVAL YEAR TO MONTH,
   dureeJourSecondes INTERVAL DAY TO SECOND);
```

L'insertion suivante initialise :

- la colonne dureeAnneesMois à la valeur d'1 an et 7 mois ;
- la colonne dureeJourSecondes à la valeur de 5 jours, 15 heures, 13 minutes, 56 secondes et 97 centièmes.

```
INSERT INTO Durees(dureeAnneesMois, dureeJourSecondes)
            VALUES('1-7',           '5 15:13:56.97');
```

Nous verrons comment ajouter ou soustraire un intervalle à une date ou à un autre intervalle.

Variables utiles

Les variables suivantes permettent de retrouver le moment de la session et le fuseau du serveur (si tant est qu'il soit déporté par rapport au client).

- CURRENT_DATE : date et heure de la session (format DATE) ;
- LOCALTIMESTAMP : date et heure de la session (format TIMESTAMP) ;
- SYSTIMESTAMP : date et heure du serveur (format TIMESTAMP WITH TIME ZONE) ;
- DBTIMEZONE : fuseau horaire du serveur (format VARCHAR2) ;
- SESSIONTIMEZONE : fuseau horaire de la session client (format VARCHAR2).

Il faut utiliser la pseudo-table DUAL, que nous détaillerons au chapitre 4, qui permet d'afficher une expression dans l'interface SQL*Plus.

L'exemple suivant montre que le script a été exécuté le 23 avril 2003 à 19 h 33, 8 secondes et 729 centièmes. Le client est sur le fuseau GMT+2h, le serveur quelque part aux États-Unis (GMT-7), option par défaut à l'installation d'Oracle. Ce dernier sait pertinemment qu'on a choisi la langue française mais a quand même laissé sa situation géographique. Il faudra la

modifier (dans le fichier de configuration) si on désire positionner le fuseau du serveur dans le même fuseau que le client.

```
SELECT CURRENT_DATE, LOCALTIMESTAMP, SYSTIMESTAMP, DBTIMEZONE,
SESSIONTIMEZONE FROM DUAL;
CURRENT_DATE   LOCALTIMESTAMP              SYSTIMESTAMP
------------   -------------------------   -------------------------
23/04/03       23/04/03 19:33:08,729000    23/04/03 19:33:08,729000 +02:00
DBTIMEZONE                                 SESSIONTIMEZONE
----------                                 -------------------------
-07:00                                     +02:00
```

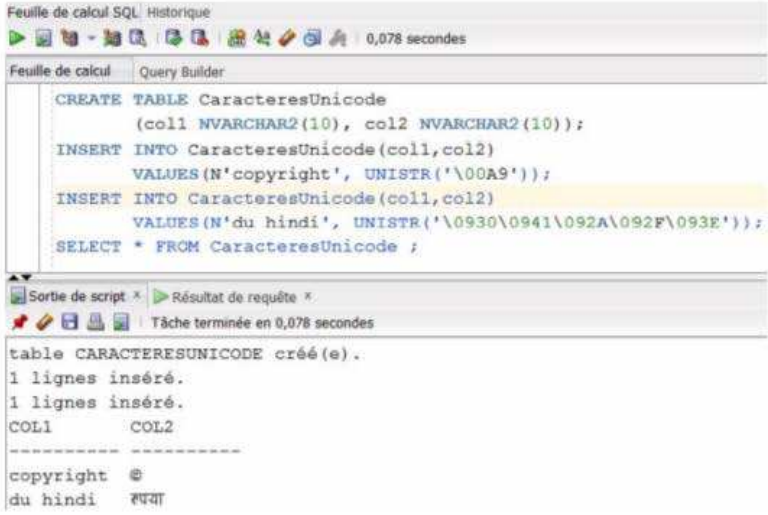
Caractères Unicode

Si vous envisagez de stocker des données qui ne sont ni des lettres, ni des chiffres, ni les symboles courants : *espace, tabulation,* % ` ' ( ) * + – , . / \ : ; < > = ! _ & ~ { } | ^ ? $ # @ " [ ], vous devrez utiliser, pour manipuler des caractères Unicode, les types NCHAR, NCHAR2 et NCLOB.

Le jeu de caractères d'Oracle pour une installation française est WE8ISO8859P1 (condensé de *Western Europe 8-bit ISO 8859 Part 1*). Le jeu de caractères national utilisé par défaut pour les types NCHAR est AL16UTF16.

Pour plus de détails, consultez le livre consacré au support du multilangue (*Database Globalization Support Guide*). Vous y découvrirez que, comme pour les caractères accentués (voir l'introduction), c'est au niveau du client (SQL Developer, Toad, SQL*Plus…) que vous devrez paramétrer des variables d'environnement ou des fichiers de configuration afin de pouvoir visualiser les données Unicode stockées. Par ailleurs, il est recommandé de préférer l'interface SQL Developer (plus apte à gérer des informations UTF-8) à SQL*Plus qui n'est pas adaptée et dédiée aux caractères ANSI ou ASCII. La fonction UNISTR sert à transformer un paramètre Unicode pour retourner une information codée dans le jeu de caractères de la base (AL16UTF16 ou UTF8).

Figure 2-2 Insertion de caractères Unicode



Données LOB

Les types LOB (*Large Object Binary*) d'Oracle sont `BLOB`, `CLOB`, `NCLOB` et `BFILE`. Ils servent à stocker de grandes quantités de données non structurées (textes, images, vidéos, sons). Ils succèdent aux types `LONG`. Les LOB sont étudiés plus en détail dans la partie consacrée à la programmation PL/SQL.

Considérons la table suivante.

```
CREATE TABLE Trombinoscope (nomEtudiant VARCHAR(30), photo BFILE);
```

Le stockage de l'image `photoCS.jpg`, qui se trouve à l'extérieur de la base (dans le répertoire `D:\PhotosEtudiant`), est réalisé par l'insertion dans la colonne `BFILE` d'un pointeur (*locator*) qui adresse le fichier externe via la fonction `BFILENAME` du paquetage `DBMS_LOB`. L'utilisateur doit avoir reçu au préalable le privilège `CREATE ANY DIRECTORY`.

```
CREATE DIRECTORY repertoire_etudiants AS 'D:\PhotosEtudiant';
INSERT INTO Trombinoscope
   VALUES ('Soutou', BFILENAME('repertoire_etudiants', 'photoCS.jpg'));
```

L'interface en mode texte SQL*Plus n'est pas capable d'afficher cette image. Il faudra pour cela utiliser un logiciel approprié (une interface Web ou Java par exemple, après avoir chargé cette image par la fonction LOADFROMFILE du paquetage DBMS_LOB).

Séquences

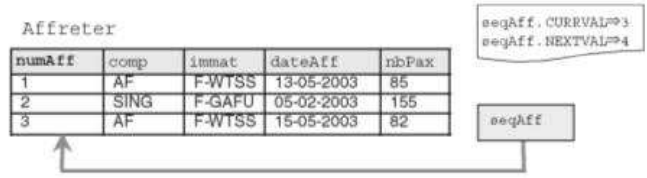
Une séquence est un objet de schéma (appartenant à l'utilisateur qui l'a créé) qui a pour objectif de générer automatiquement des valeurs (de type NUMBER). Bien qu'elles soient majoritairement utilisées pour composer des valeurs auto-incrémentées pour les clés primaires, il est possible de les employer au sein de différentes tables.

Gérée indépendamment d'une table, une séquence peut être partagée par plusieurs utilisateurs.

Depuis la version 12*c*, il est possible de définir une colonne auto-incrémentée à l'aide de la directive GENERATED... AS IDENTITY... avec les mêmes options dédiées initialement aux séquences (voir l'instruction CREATE SEQUENCE). La directive d'auto-incrémentation peut être utilisée dans une instruction CREATE TABLE ou ALTER TABLE.

La figure suivante illustre la séquence seqAff utilisée pour initialiser les valeurs de la clé primaire numAff de la table Affreter. Seules deux fonctions (aussi appelées pseudo-colonnes ou directives) peuvent être appliquées à une séquence : CURRVAL retourne la valeur courante, NEXTVAL incrémente la séquence et retourne la valeur obtenue (ici le pas est de 1, nous verrons qu'il peut être différent de cette valeur).

Figure 2-3 *Séquence appliquée à une clé primaire*



Création d'une séquence (CREATE SEQUENCE)

Vous devez avoir le privilège CREATE SEQUENCE pour pouvoir créer une séquence dans votre schéma. Pour en créer une dans un schéma différent du vôtre, le privilège CREATE ANY SEQUENCE est requis.

La syntaxe de création d'une séquence est la suivante :

```
CREATE SEQUENCE [schéma.]nomSéquence
   [INCREMENT BY entier ]
   [START WITH entier ]
   [ { MAXVALUE entier | NOMAXVALUE } ]
   [ { MINVALUE entier | NOMINVALUE } ]
   [ { CYCLE | NOCYCLE } ]
   [ { CACHE entier | NOCACHE } ]
   [ { ORDER | NOORDER } ] ;
```

Si aucun nom de schéma n'est spécifié la séquence créée vous appartient. Si aucune option n'est précisée, la séquence créée commencera à 1 et augmentera sans fin (la limite réelle d'une séquence est de 10^{29}-1). En spécifiant seulement « `INCREMENT BY -1` » la séquence créée commencera à -1 et sa valeur diminuera sans limites (la borne inférieure réelle d'une séquence est de -10^{27}-1).

- `INCREMENT BY` : donne l'intervalle entre deux valeurs de la séquence (entier positif ou négatif mais pas nul). La valeur absolue de cet intervalle doit être plus petite que (`MAXVALUE-MINVALUE`). L'intervalle par défaut est 1.
- `START WITH` : précise la première valeur de la séquence à générer. Pour les séquences ascendantes (d'un incrément positif), la valeur par défaut est égale à la valeur minimale de la séquence. Pour les séquences descendantes, la valeur par défaut est égale à la valeur maximale de la séquence (entier jusqu'à 10^{28}-1, pour les négatifs : -10^{27}+1).
- `MAXVALUE` : donne la valeur maximale de la séquence. Cette limite doit être supérieure ou égale à l'entier défini dans `START WITH` et supérieure à `MINVALUE`.
- `NOMAXVALUE` (par défaut) fixe le maximum à 10^{28}-1 pour une séquence ascendante et à -10^{27}+1 pour une séquence descendante.
- `MINVALUE` précise la valeur minimale de la séquence. Cette limite doit être inférieure ou égale à l'entier défini dans `START WITH` et inférieure à `MAXVALUE`.
- `NOMINVALUE` (par défaut) fixe le minimum à 1 pour une séquence ascendante et à la valeur -10^{27}-1 pour une séquence descendante.
- `CYCLE` indique que la séquence doit continuer de générer des valeurs même après avoir atteint sa limite. Au-delà de la valeur maximale, la séquence générera la valeur minimale et incrémentera comme cela est défini dans la clause concernée. Après la valeur minimale, la séquence produira la valeur maximale et décrémentera comme cela est défini dans la clause concernée.
- `NOCYCLE` (par défaut) indique que la séquence ne doit plus générer de valeurs une fois la limite atteinte.
- `CACHE` fixe le nombre de valeurs de la séquence que le cache va contenir (et qui évite la sollicitation du compteur en temps réel). Le minimum est 2 et le maximum théorique est fonction d'une formule analogue à *maxi_sequence-mini_sequence/increment_sequence* (par exemple, pour une séquence de valeur maximale 50 000 et de valeur mininale 1 avec

un pas de 2, le nombre maximal de valeurs en cache serait de 25 000). Par défaut, le cache contient 20 valeurs.

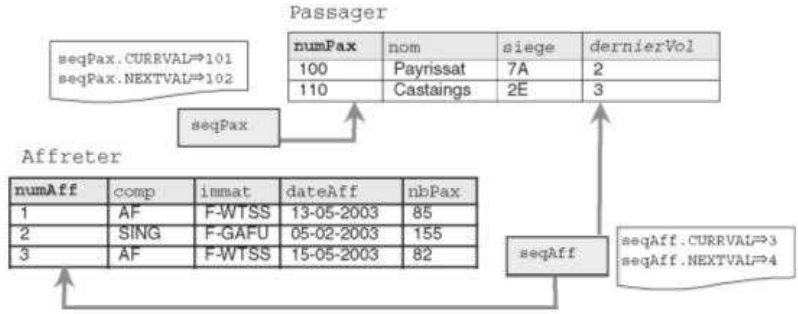
En fonction du nombre de séquences mises en cache, et selon l'utilisation (même si d'autres utilisateurs ou d'autres connexions emploient cette séquence) voire l'interruption du serveur, il est possible que des valeurs retournées ne se succèdent pas d'une seule valeur du pas de l'incrément du fait du cache perdu. En revanche, le contrat qu'Oracle remplit consiste à délivrer toujours une nouvelle valeur après l'appel de `NEXTVAL`.

- `ORDER` garantit que les valeurs de la séquence sont générées dans l'ordre des requêtes. Si vos séquences jouent le rôle d'horodatage (*timestamp*), vous devrez utiliser cette option. Pour la génération de clés primaires, cette option n'est pas importante.

Créons les deux séquences (`seqAff` et `seqPax`) qui vont permettre de donner leur valeur aux clés primaires des deux tables illustrées à la figure suivante. On suppose qu'on ne stockera pas plus de 100 000 passagers et pas plus de 10 000 affrètements.

Servons-nous aussi de la séquence `seqAff` dans la table `Passager` pour indiquer le dernier vol de chaque passager. `seqAff` sert à donner leur valeur à la clé primaire de `Affreter` et à la clé étrangère de `Passager`. La section « Intégrité référentielle » détaille les mécanismes relatifs aux clés étrangères.

Figure 2-4 *Séquences*



Le script SQL de définition des données est indiqué ci-après. Notez que les déclarations sont indépendantes, ce n'est qu'au moment des insertions qu'on affectera aux colonnes concernées les valeurs des séquences.

Tableau 2-3 Tables et séquences

Tables	Séquences

`CREATE TABLE Affreter`
`(numAff NUMBER(5),comp VARCHAR2(4),`
`immat VARCHAR2(6), dateAff DATE, nbPax NUMBER(3),`
`CONSTRAINT pk_Affreter PRIMARY KEY (numAff));`
`CREATE TABLE Passager`
`(numPax NUMBER(6), nom VARCHAR2(15),`
`siege VARCHAR2(4), dernierVol NUMBER(5),`
`CONSTRAINT pk_Passager PRIMARY KEY(numPax),`
`CONSTRAINT fk_Pax_vol_Affreter`
`FOREIGN KEY(dernierVol)REFERENCES Affreter(numAff));`
	`CREATE SEQUENCE seqAff`
	`MAXVALUE 10000`
	`NOMINVALUE;`
	
	`CREATE SEQUENCE seqPax`
	`INCREMENT BY 10`
	`START WITH 100`
	`MAXVALUE 100000`
	`NOMINVALUE;`

Manipulation d'une séquence

Vous devez avoir le privilège `SELECT` sur une séquence (privilège donné par `GRANT SELECT ON` *seq* `TO` *utilisateur*) pour pouvoir en utiliser une. Pour manipuler une séquence dans un schéma différent du vôtre, le privilège `SELECT ANY SEQUENCE` est requis. Dans ce cas il faudra toujours préfixer le nom de la séquence par celui du schéma (par exemple *jean.seq*).

Une fois créée, une séquence *seq* ne peut se manipuler que via deux directives (qu'Oracle appelle aussi pseudo-colonnes) :

- *seq*.`CURRVAL` qui retourne la valeur courante de la séquence (lecture seule) ;
- *seq*.`NEXTVAL` qui incrémente la séquence et retourne la nouvelle valeur de celle-ci (écriture et lecture).

Le premier appel à `NEXTVAL` retourne la valeur initiale de la séquence (définie dans `START WITH`). Les appels suivants augmentent la séquence de la valeur définie dans `INCREMENT WITH`.

Chaque appel à `CURRVAL` retourne la valeur courante de la séquence. Il faut utiliser au moins une fois `NEXTVAL` avant d'appeler `CURRVAL` dans une même session (SQL*Plus, bloc PL/SQL ou programme). Ces directives peuvent s'utiliser :

- au premier niveau d'une requête `SELECT` (voir le chapitre 4) ;
- dans la clause `SELECT` d'une instruction `INSERT` (voir la section « Insertion multilignes » du chapitre 4) ;
- dans la clause `VALUES` d'une instruction `INSERT` (voir l'exemple suivant) ;
- dans la clause `SET` d'une instruction `UPDATE` (voir la section ci-après).

Les principales restrictions d'utilisation de NEXTVAL et CURRVAL sont :

- sous-interrogation dans une instruction DELETE, SELECT, ou UPDATE (voir le chapitre 4) ;
- dans un SELECT d'une vue (voir le chapitre 5) ;
- dans un SELECT utilisant DISTINCT, GROUP BY, ORDER BY ou des opérateurs ensemblistes (voir le chapitre 4) ;
- en tant que valeur par défaut (DEFAULT) d'une colonne d'un CREATE TABLE ou ALTER TABLE ;
- dans la condition d'une contrainte CHECK d'un CREATE TABLE ou ALTER TABLE.

Le tableau suivant illustre l'évolution de nos deux séquences en fonction de l'insertion des enregistrements décrits dans la figure précédente. Nous utilisons NEXTVAL pour les clés primaires et CURRVAL pour la clé étrangère (de manière à récupérer la dernière valeur de la séquence utilisée pour la clé primaire).

Tableau 2-4 Manipulation de séquences

Instructions SQL	Séquences	seqAff		seqPax	
		CURRVAL	NEXTVAL	CURRVAL	NEXTVAL
--Aucune insertion encore		Pas définies			
INSERT INTO Affreter VALUES (seqAff.**NEXTVAL**,'AF','F-WTSS','13-05-2003',85);		1	2	Pas définies	
INSERT INTO Affreter VALUES (seqAff.**NEXTVAL**,'SING','F-GAFU','05-02-2003',155);					
INSERT INTO Passager VALUES (seqPax.**NEXTVAL**,'Payrissat','7A', seqAff.**CURRVAL**);		2	3		
INSERT INTO Affreter VALUES (seqAff.**NEXTVAL**,'AF','F-WTSS','15-05-2003',82);				100	110
INSERT INTO Passager VALUES (seqPax.**NEXTVAL**,'Castaings','2E', seqAff.**CURRVAL**);		3	4	110	120

Utilisation d'une séquence dans un DEFAULT

Depuis la version 12*c*, la valeur par défaut d'une colonne numérique entière peut être définie avec `NEXTVAL` ou `CURRVAL`. Ainsi, lors d'insertions, la génération automatique de valeurs se produira en utilisant la séquence précisée dans la clause `DEFAULT`.

Le code suivant présente l'utilisation de cette option pour deux séquences dédiées à une clé primaire et à une clé étrangère, sous réserve que les transactions s'exécutent toujours dans l'ordre : ajout du vol, puis des passagers. La génération des clés étrangères ne perturbe pas la valeur de la clé primaire qui est gérée par la séquence principale.

Tableau 2-5 Séquences par défaut sur une colonne

Création des séquences et mise en place	Insertions

`CREATE SEQUENCE master_seq;`
`CREATE SEQUENCE detail_seq;`
`CREATE TABLE vol (`
`  id  NUMBER DEFAULT master_seq.`
`  NEXTVAL,`
`  description VARCHAR2(6),`
`  jour_vol  DATE);`
`CREATE TABLE places (`
`  id  NUMBER DEFAULT detail_seq.NEXTVAL,`
`  vol_id  NUMBER DEFAULT master_seq.`
`  CURRVAL,`
`  pax_nom  VARCHAR2(30));`
	`INSERT INTO vol (descrip-`
	`tion,jour_vol)`
	`  VALUES ('AF6143',SYSDATE-1);`
	
	`INSERT INTO places (pax_nom)`
	`  VALUES ('Joppé');`
	`INSERT INTO places (pax_nom)`
	`  VALUES ('Rienna');`
	`INSERT INTO places (pax_nom)`
	`  VALUES ('Guilbaud');`

Figure 2-5 *Séquences par défaut*

Modification d'une séquence (ALTER SEQUENCE)

Vous devez avoir le privilège `ALTER SEQUENCE` pour pouvoir modifier une séquence de votre schéma. Pour modifier une séquence dans un schéma différent du vôtre, le privilège `ALTER ANY SEQUENCE` est requis.

Les modifications les plus courantes sont celles qui consistent à augmenter les limites d'une séquence ou à changer le pas de son incrémentation. Dans tous les cas, seules les valeurs à venir de la séquence modifiée seront changées (heureusement pour les données existantes des tables).

La syntaxe de modification d'une séquence reprend la plupart des éléments de sa création.

```
ALTER SEQUENCE [schéma.]nomSéquence
   [INCREMENT BY entier ]
   [ { MAXVALUE entier | NOMAXVALUE } ]
   [ { MINVALUE entier | NOMINVALUE } ]
   [ { CYCLE | NOCYCLE } ]
   [ { CACHE entier | NOCACHE } ]
   [ { ORDER | NOORDER } ] ;
```

La clause `START WITH` ne peut être modifiée sans supprimer et recréer la séquence. Des contrôles sont opérés sur les limites, par exemple `MAXVALUE` ne peut pas être affectée à une valeur plus petite que la valeur courante de la séquence.

Supposons qu'on ne stockera pas plus de 95 000 passagers et pas plus de 850 affrètements. De plus les incréments des séquences doivent être égaux à 5. Les instructions SQL à appliquer sont les suivantes : chaque invocation des méthodes `NEXTVAL` prendra en compte désormais le nouvel incrément tout en laissant intactes les données existantes des tables.

```
ALTER SEQUENCE seqAff INCREMENT BY 5 MAXVALUE 850;
ALTER SEQUENCE seqPax INCREMENT BY 5 MAXVALUE 95000;
```

Visualisation d'une séquence

La pseudo-table `DUAL` peut être utilisée pour visualiser le contenu d'une séquence. En appliquant la directive `CURRVAL` on extrait le contenu actuel de la séquence (la dernière valeur générée).

En appliquant la directive `NEXTVAL` dans un `SELECT` la séquence s'incrémente avant de s'afficher. Vous réalisez alors un effet de bord car la valeur qui apparaît à l'écran est désormais perdue pour une éventuelle utilisation dans une clé primaire.

Le tableau suivant illustre l'utilisation de la pseudo-table `DUAL` pour visualiser les séquences créées auparavant.

Tableau 2-6 Visualisation de séquences

Besoin	Requête SQL et résultat sous SQL*Plus

Quelles sont les dernières valeurs générées par mes séquences ?	`SELECT seqAff.CURRVAL "seqaff (CURRVAL)" , seqPax.CURRVAL "seqPax (CURRVAL)" FROM DUAL;`
	`seqAff (CURRVAL) seqPax (CURRVAL)`
	`----------`
	`3 110`

Quelles sont les prochaines valeurs produites par mes séquences ? (qui sont perdues car les incréments s'opèrent lors de la requête)	`SELECT seqAff.NEXTVAL "seqAff (NEXTVAL)", seqAff.NEXTVAL "seqPax (NEXTVAL)" FROM DUAL;`
	`seqAff (NEXTVAL) seqPax (NEXTVAL)`
	`----------`
	`4 120`

Suppression d'une séquence (DROP SEQUENCE)

L'instruction `DROP SEQUENCE` supprime une séquence. Celle-ci doit se trouver dans votre schéma (vous en êtes propriétaire) ou vous devez avoir le privilège `DROP ANY SEQUENCE`.

La suppression d'une séquence peut être utilisée pour refaire partir une séquence donnée à un chiffre nouveau (clause `START WITH`). En ce cas, il faut bien sûr recréer la séquence après l'avoir supprimée.

La syntaxe de suppression d'une séquence est la suivante.

```
DROP SEQUENCE [schéma.]nomSéquence ;
```

Supprimons les deux séquences de notre schéma par les instructions suivantes :

```
DROP SEQUENCE seqAff;
DROP SEQUENCE seqPax;
```

Colonnes auto-incrémentées

Depuis la version 12*c*, il est possible d'utiliser un type numérique (entier) pour définir une colonne auto-incrémentée avec la clause `GENERATED… AS IDENTITY…` disponible dans les instructions `CREATE TABLE` et `ALTER TABLE`.

Avant de détailler les options d'auto-incrémentation, vous devez savoir qu'une colonne auto-incrémentée dispose en interne d'une séquence générée automatiquement et qui lui est dédiée. Ainsi, vous retrouverez les options `INCREMENT`, `START`… lorsque vous définirez un auto-incrément.

```
GENERATED [ALWAYS | BY DEFAULT [ON NULL]]
          AS IDENTITY [(options_sequence)]
```

- ALWAYS (par défaut) utilise le générateur de séquences et interdit qu'une valeur soit explicitement imposée lors d'un INSERT ou UPDATE (erreur ORA-32795 impossible d'insérer la valeur dans une colonne d'identité...).
- BY DEFAULT utilise le générateur de séquences mais n'interdit pas qu'une valeur soit explicitement imposée d'un INSERT ou UPDATE. Avec l'option ON NULL, la séquence est capable d'affecter implicitement une valeur à chaque INSERT ou si un quelconque NULL arrive en lieu et place de la colonne concernée.
- *options_sequence* sont identiques à celles du CREATE SEQUENCE.

Le code suivant présente l'utilisation de cette option pour une clé primaire. Il est à noter que le NOT NULL est implicite sur une telle colonne et que vous ne pouvez disposer que d'un seul auto-incrément par table.

Tableau 2-7 Colonne auto-incrémentée

Création de la table et de son auto-incrémentation	Insertions

`CREATE TABLE billets (`
`  id  NUMBER(6)`
`  GENERATED ALWAYS`
`  AS IDENTITY,`
`  vol_id  VARCHAR2(6) NOT NULL,`
`  jour_vol  DATE DEFAULT SYSDATE NOT NULL,`
`  pax_nom  VARCHAR2(30) NOT NULL,`
`  siege_pax CHAR(3) NOT NULL ,`
`  CONSTRAINT pk_billets PRIMARY KEY(id));`
	`INSERT INTO billets(vol_id,pax_nom,siege_pax)`
	`  VALUES  ('AF6143','Guilbaud','03F');`
	`INSERT INTO billets(vol_id,pax_nom,siege_pax)`
	`  VALUES  ('AF6145','Blanchet','23B');`
	`INSERT INTO billets(vol_id,pax_nom,siege_pax)`
	`  VALUES  ('AF6145','Bruchez','02A');`

Figure 2-6 Colonne auto-incrémentée

```
SQL> SELECT * FROM billets;

        ID VOL_ID JOUR_VOL PAX_NOM                        SIEGE_PAX
---------- ------ -------- ------------------------------ ----------
         1 AF6143 20/11/14 Guilbaud                       03F
         2 AF6145 20/11/14 Blanchet                       23B
         3 AF6145 20/11/14 Bruchez                        02A
```

Modifications de valeurs

L'instruction UPDATE permet la mise à jour des colonnes d'une table. Pour pouvoir modifier des enregistrements d'une table, il faut que cette dernière soit dans votre schéma ou que vous ayez reçu le privilège UPDATE sur la table. Si vous avez le privilège UPDATE ANY TABLE, vous pouvez modifier des enregistrements de tout schéma.

Syntaxe (UPDATE)

La syntaxe simplifiée de l'instruction UPDATE est la suivante.

```
UPDATE [schéma.] nomTable
SET  colonne1 = { expression | (requête_SELECT) | DEFAULT }
    [colonne2 ...]
```

La première écriture de la clause SET met à jour une colonne en lui affectant une expression (valeur, valeur par défaut, calcul, résultat d'une requête). La deuxième écriture rafraîchit plusieurs colonnes à l'aide du résultat d'une requête.

La condition filtre les lignes à mettre à jour dans la table. Si aucune condition n'est précisée, tous les enregistrements seront mis à jour. Si la condition ne filtre aucune ligne, aucune mise à jour ne sera réalisée.

Modification d'une ligne

Affectons un nom à la compagnie de code 'GO' et modifions sa date de création.

```
UPDATE compagnie
SET    nom_comp      = 'Go Airways',
       date_creation = TO_DATE('30/12/2014','DD/MM/YYYY')
WHERE  comp = 'GO';
```

Modification de plusieurs lignes

Pour remplacer le marqueur NULL par le nombre 10, il suffit de conditionner la mise à jour à l'aide de la fonction IS NULL (voir le chapitre 4). Attention, si vous utilisez la condition WHERE nb_passagers = NULL, vous ne sélectionnerez aucune ligne car deux NULL sont différents entre eux.

```
UPDATE vol_jour
SET    nb_passagers = 10
WHERE  nb_passagers IS NULL;
```

Les modifications sont présentées dans la figure suivante.

Figure 2-7 Table après les modifications

Ne pas respecter des contraintes

Il faut, comme pour les insertions, respecter les contraintes qui existent au niveau des colonnes. Dans le cas inverse, une erreur est renvoyée (le nom de la contrainte apparaît) et la mise à jour n'est pas effectuée.

Le tableau suivant décrit une tentative de modification pour chaque type de contrainte que vous pourrez être amené à rencontrer. Les problèmes présentés ici sont respectivement une clé primaire en doublon, une colonne obligatoire, un aéroport de départ semblable à celui d'arrivée, un libellé dupliqué et une compagnie inexistante.

Tableau 2-8 Modifications impossibles

Type de contrainte	Instructions SQL et résultats

Clé primaire	`SQL> UPDATE compagnie SET comp = 'AF' WHERE comp = 'GO';`
	`ERREUR : ORA-00001: violation de contrainte unique`
	`(SOUTOU.PK_COMPAGNIE)`

Non-nullité	`SQL> UPDATE vol_jour SET aero_dep = NULL WHERE num_vol = 'AF6143';`
	`ERREUR : ORA-01407: impossible de mettre à jour`
	`("SOUTOU"."VOL_JOUR"."AERO_DEP") avec NULL`

Vérification	`SQL> UPDATE vol_jour SET aero_arr = 'TLS' WHERE num_vol = 'AF6143';`
	`ERREUR : ORA-02290: violation de contraintes`
	`(SOUTOU.CK_TRAJET) de vérification`

Unicité	`SQL> UPDATE compagnie SET nom_comp = 'Go Airways' WHERE comp = 'AF';`
	`ERREUR : ORA-00001: violation de contrainte unique`
	`(SOUTOU.UN_NOM_COMP)`

Clé étrangère	`SQL> UPDATE vol_jour SET comp = 'EJET' WHERE num_vol = 'AF6143';`
	`ERREUR : ORA-02291: violation de contrainte d'intégrité`
	`(SOUTOU.FK_VOL_JOUR_COMP_COMPAGNIE) - clé parent introuvable`

La mise à jour d'une clé étrangère est possible si la nouvelle valeur est bien référencée. La mise à jour d'une clé primaire est possible si aucune ligne d'aucune table ne la référence déjà (voir la section « Intégrité référentielle »).

Dates et intervalles

Le tableau suivant résume les opérations possibles entre des colonnes de type `DATE` et `Interval`.

Tableau 2-9 Opérations entre dates et intervalles

Opérande 1	Opérateur	Opérande 2	Résultat
DATE	+ ou -	INTERVAL	DATE
DATE	+ ou -	NUMBER	DATE
Interval	+	DATE	DATE
DATE	-	DATE	NUMBER
Interval	+ ou -	INTERVAL	INTERVAL
Interval	* ou /	NUMBER	INTERVAL

Considérons la table suivante :

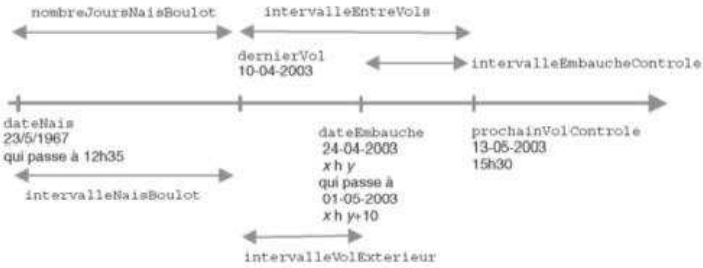
```
CREATE TABLE Pilote
   (brevet VARCHAR(6), nom VARCHAR(20), dateNaiss DATE, dernierVol DATE,
    dateEmbauche DATE, prochainVolControle DATE,
     nombreJoursNaisBoulot      NUMBER,
    intervalleNaisBoulot        INTERVAL DAY(7) TO SECOND(3),
    intervalleVolExterieur      INTERVAL DAY(2) TO SECOND(0),
    intervalleEntreVols         INTERVAL DAY(2) TO SECOND(2),
    intervalleEmbaucheControle INTERVAL DAY(2) TO SECOND(1),
    compa VARCHAR(4), CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

À l'insertion du pilote, nous initialisons sa date de naissance, la date de son dernier vol, sa date d'embauche (à celle du jour via `SYSDATE`) et la date de son prochain contrôle en vol au 13 mai 2003, 15 h 30 (heures et minutes évaluées à l'aide de la fonction `TO_DATE` qui convertit une chaîne en date).

```
INSERT INTO Pilote
   VALUES ('PL-1', 'Thierry Albaric', '25-03-1967', '10-04-2003', SYSDATE,
      TO_DATE('13-05-2003 15:30:00','DD:MM:YYYY HH24:MI:SS'), NULL, NULL,
      NULL, NULL, NULL, 'AF');
```

Les mises à jour par `UPDATE` sur cet enregistrement vont consister, sur la base de ces quatre dates, à calculer les intervalles illustrés à la figure suivante :

Figure 2-8 Intervalles à calculer



Modification d'une heure

On modifie une date en précisant une heure via la fonction TO_DATE.

```
UPDATE Pilote
   SET dateNaiss = TO_DATE('25-03-1967 12:35:00',
   'DD:MM:YYYY HH24:MI:SS')
   WHERE brevet = 'PL-1';
```

Ajout d'un délai

On modifie la date d'embauche de 10 minutes après la semaine prochaine. L'ajout d'une semaine se fait par l'opération +7 à une date. L'addition de 10 minutes se fait par l'ajout de la fraction de jour correspondante (10/(24*60)).

```
UPDATE Pilote
   SET dateEmbauche = dateEmbauche + 7 + (10/(24*60)) WHERE brevet =
   'PL-1';
```

Différence entre deux dates

La différence entre deux dates renvoie un entier correspondant au nombre de jours.

```
UPDATE Pilote
   SET nombreJoursNaisBoulot = dateEmbauche-dateNaiss WHERE brevet =
   'PL-1';
```

Cette même différence au format INTERVAL en nombre de jours requiert l'utilisation de la fonction NUMTODSINTERVAL.

```
UPDATE Pilote
SET intervalleNaisBoulot =
NUMTODSINTERVAL(dateEmbauche-dateNaiss,'DAY'),
```

```
    intervalleEntreVols =
    NUMTODSINTERVAL(prochainVolControle-dernierVol,'DAY'),
    intervalleVolExterieur =
    NUMTODSINTERVAL(dateEmbauche-dernierVol,'DAY')
WHERE brevet = 'PL-1';
```

Différence entre deux intervalles

La différence entre deux intervalles homogènes renvoie un intervalle.

```
UPDATE  Pilote
SET intervalleEmbaucheControle =
intervalleEntreVols-intervalleVolExterieur
WHERE brevet = 'PL-1';
```

La ligne contient désormais les informations suivantes. Les données en gras correspondent aux mises à jour. On trouve qu'il a fallu 13 186 jours, 3 heures, 49 minutes et 53 secondes pour que ce pilote soit embauché. 21 jours, 16 heures, 24 minutes et 53 secondes séparent le dernier vol du pilote au moment de son embauche. 33 jours, 15 heures et 30 minutes séparent son dernier vol de son prochain contrôle en vol. La différence entre ces deux délais est de 11 jours, 23 heures, 5 minutes et 7 secondes.

Figure 2-9 *Ligne modifiée par des calculs de dates*

Pilote

brevet	nom	dateNaiss	dernierVol	dateEmbauche	prochainVolControle
PL-1	Thierry Albanc	*25-03-1967* 25-03-1987 **12:35:00**	10-04-2003	*24-04-2003 04:14:53* **01:05:2003 04:24:53**	13-05-2003 15:30:00

nombreJoursNaisBoulot	intervalleNaisBoulot	intervalleVolExterieur
13186,1596	+0013186 03:49:53.000	+21 16:24:53

intervalleEntreVols	intervalleEmbaucheControle	compa
+33 15:30:00.00	+11 23:05:07.0	AF

Fonctions utiles

Les fonctions suivantes vous seront d'un grand secours pour manipuler des dates et des intervalles.

- `TO_CHAR(colonneDate [, format [, 'NLS_DATE_LANGUAGE=Langue']])` convertit une date en chaîne suivant un certain format dans un certain langage ;
- `TO_DATE(chaîneCaractères [, format [, 'NLS_DATE_LANGUAGE=Langue']])` convertit une chaîne en date suivant un certain format dans un certain langage ;

- `EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM {expressionDATE | expressionINTERVAL})` extrait une partie donnée d'une date ou d'un intervalle ;
- `NUMTOYMINTERVAL(expressionNumérique, {'YEAR' | 'MONTH'})` convertit un nombre dans un type `INTERVAL YEAR TO MONTH` ;
- `NUMTODSINTERVAL(expressionNumérique, {'DAY' | 'HOUR' | 'MINUTE' | 'SECOND'})` convertit un nombre dans un type `INTERVAL DAY TO SECOND`.

Les tableaux suivants présentent quelques exemples d'utilisation de ces fonctions.

Tableau 2-10 Quelques formats pour TO_CHAR

Expression	Résultats	Commentaires
`TO_CHAR(dateNaiss ,'J')`	2439575	Le calendrier julien est utilisé ici (comptage du nombre de jours depuis le 1er janvier, 4712 av. J.-C. jusqu'au 25 mars 1967).
`TO_CHAR(dateNaiss, 'DAY - MONTH - YEAR')`	SAMEDI - MARS - NINETEEN SIXTY-SEVEN	Affichage des libellés des jours, mois et années. Oracle ne traduit pas encore notre année.
`TO_CHAR(dateEmbauche ,'DDD')`	121	Affichage du numéro du jour de l'année (ici il s'agit 1er mai 2003).

Tableau 2-11 Quelques formats pour TO_DATE

Expression	Commentaires
`TO_DATE('May 13, 1995, 12:30 A.M.','MONTH DD, YYYY, HH:MI A.M.','NLS_DATE_LANGUAGE = American')`	Définition d'une date à partir d'un libellé au format américain.
`TO_DATE('13 Mai, 1995, 12:30','DD MONTH, YYYY, HH24:MI','NLS_DATE_LANGUAGE = French')`	Définition de la même date pour les francophones (l'option NLS par défaut est la langue à l'installation).

Tableau 2-12 Utilisation de EXTRACT

Expression	Résultats	Commentaires
`EXTRACT(DAY FROM intervalleVolExterieur)`	21	Extraction du nombre de jours dans l'intervalle contenu dans la colonne.
`EXTRACT(MONTH FROM dateNaiss)`	3	Extraction du mois de la date contenue dans la colonne.

Tableau 2-13 Conversion en intervalles

Expression	Résultats	Commentaires
`NUMTOYMINTERVAL(1.54,'YEAR')`	`+000000001-06`	1 an et 54 centièmes d'année est converti en 1 an et 6 mois.
`NUMTOYMINTERVAL(1.54,'MONTH')`	`+000000000-01`	1 mois et 54 centièmes de mois est converti en 1 mois à l'arrondi.
`NUMTODSINTERVAL(1.54,'DAY')`	`+000000001 12:57:36.00`	1 jour et 54 centièmes de jour est converti en 1 jour, 12 heures, 57 minutes et 36 secondes.
`NUMTODSINTERVAL(1.54,'HOUR')`	`+000000000 01:32:24.00`	1 heure et 54 centièmes est converti en 1 heure, 32 minutes et 24 secondes.

Suppressions d'enregistrements

Les instructions DELETE et TRUNCATE permettent de supprimer un ou plusieurs enregistrements d'une table. Pour pouvoir supprimer des données dans une table, il faut que cette dernière soit dans votre schéma ou que vous ayez reçu le privilège DELETE sur la table. Si vous avez le privilège DELETE ANY TABLE, vous pouvez détruire des enregistrements dans n'importe quelle table de tout schéma.

Instruction DELETE

La syntaxe simplifiée de l'instruction DELETE est la suivante :

```
DELETE FROM [schéma.]nomTable [WHERE condition];
```

La condition sélectionne les lignes à supprimer. Si aucune condition n'est précisée, toutes les lignes sont supprimées. Si l'expression ne sélectionne aucune ligne, rien ne sera supprimé et aucune erreur n'est retournée. Supprimons une compagnie et un vol journalier (notez qu'il faut préciser l'heure dans le format de date si une heure a été incluse dans la date) à l'aide de cette instruction :

```
SQL> DELETE FROM compagnie WHERE nom_comp = 'Go Airways';
1 ligne supprimée.
SQL> DELETE FROM vol_jour
     WHERE  jour_vol = TO_DATE('20/11/2014 15:30','DD/MM/YYYY HH24:MI')
     AND    num_vol  = 'AF6143';
1 ligne supprimée.
```

Les suppressions sont présentées dans la figure suivante.

Figure 2-10 *Table après les suppressions*

COMPAGNIE

COMP	NOM_COMP	DATE_CREATION
SING	Singapore AL	01/01/47
AF	Air France	01/01/33
~~GO~~	~~Go Airways~~	~~30/12/14~~

VOL_JOUR

NUM_VOL	AERO_DEP	AERO_ARR	COMP	JOUR_VOL	NB_PASSAGERS
AF6550	CDG	TLS	AF	20/11/14	195
~~AF6143~~	~~TLS~~	~~ORY~~	~~AF~~	~~20/11/14~~	~~120~~
SQ747	CDG	SIN	SING	20/11/14	10
AF6145	ORY	TLS	AF	20/11/14	10

La suppression d'une ligne contenant une clé étrangère est possible si cette même ligne ne joue pas le rôle de référent (cible d'une clé étrangère d'une autre table). La suppression d'une ligne pour laquelle la clé primaire est utilisée dans une autre table en tant que clé étrangère n'est pas possible sans un mécanisme de cascade (voir la section « Intégrité référentielle »).

Tentons de supprimer une compagnie qui est référencée par un pilote à l'aide d'une clé étrangère. Une erreur s'affiche, laquelle sera expliquée dans la section « Intégrité référentielle ».

```
DELETE FROM Compagnie WHERE comp = 'SING';
ORA-02292: violation de contrainte (SCOTTOU.FK_PIL_COMPA_COMP) d'intégrité - enregistrement fils existant
```

Instruction TRUNCATE

La commande TRUNCATE supprime tous les enregistrements d'une table et libère éventuellement l'espace de stockage utilisé par la table (chose que ne peut pas faire DELETE) :

```
TRUNCATE TABLE [schéma.]nomTable [{ DROP | REUSE } STORAGE];
```

Il n'est pas possible de tronquer une table qui est référencée par des clés étrangères actives (sauf si la clé étrangère est elle-même dans la table à supprimer). La solution consiste à désactiver les contraintes puis à tronquer la table.

La récupération de l'espace est réalisée à l'aide de l'option DROP STORAGE (option par défaut). Dans le cas inverse (REUSE STORAGE), l'espace est utilisable par les nouvelles données de la table.

Intégrité référentielle

Les contraintes référentielles forment le cœur de la cohérence d'une base de données relationnelle. Ces contraintes sont fondées sur une relation entre clés étrangères et clés primaires et permettent de programmer des règles de gestion (exemple : l'affrètement d'un avion doit se faire par une compagnie existant dans la base de données). Ce faisant, les contrôles côté client (interface) sont ainsi déportés côté serveur.

C'est seulement dans sa version 7 en 1992, qu'Oracle a inclus dans son offre les contraintes référentielles.

Pour les règles de gestion trop complexes (exemple : l'affrètement d'un avion doit se faire par une compagnie qui a embauché au moins quinze pilotes dans les six derniers mois), il faudra programmer un déclencheur (voir le chapitre 7). Il faut savoir que les déclencheurs sont plus pénalisants que des contraintes dans un mode transactionnel (lectures consistantes).

La contrainte référentielle concerne toujours deux tables – une table « père » aussi dite « maître » (*parent/referenced*) et une table « fils » (*child/dependent*) – possédant une ou plusieurs colonnes en commun. Pour la table « père », ces colonnes composent la clé primaire (ou candidate avec un index unique). Pour la table « fils », ces colonnes composent une clé étrangère.

Il est recommandé de créer un index par clé étrangère (Oracle ne le fait pas comme pour les clés primaires). La seule exception concerne les tables « pères » possédant des clés primaires (ou candidates) jamais modifiées ni supprimées dans le temps.

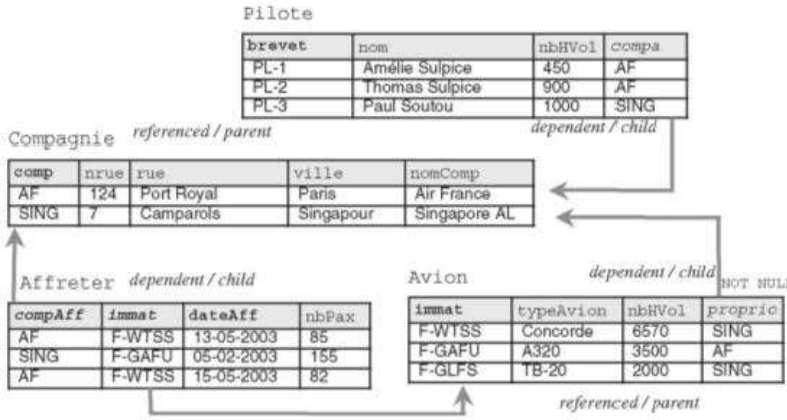
Cohérences

L'exemple suivant illustre quatre contraintes référentielles. Une table peut être « père » pour une contrainte et « fils » pour une autre (c'est le cas de la table `Avion`).

Deux types de problèmes sont automatiquement résolus par Oracle pour assurer l'intégrité référentielle :

- La cohérence du « fils » vers le « père » : on ne doit pas pouvoir insérer un enregistrement « fils » (ou modifier sa clé étrangère) rattaché à un enregistrement « père » inexistant. Il est cependant possible d'insérer un « fils » (ou de modifier sa clé étrangère) sans rattacher d'enregistrement « père » à la condition qu'il n'existe pas de contrainte `NOT NULL` au niveau de la clé étrangère.
- La cohérence du « père » vers le « fils » : on ne doit pas pouvoir supprimer un enregistrement « père » (ou modifier sa clé primaire) si un enregistrement « fils » y est encore rattaché. Il est possible de supprimer les « fils » associés (`DELETE CASCADE`) ou d'affecter la valeur nulle aux clés étrangères des « fils » associés (`DELETE SET NULL`). Oracle ne permet pas de propager une valeur par défaut (*set to default*) comme la norme SQL2 le propose.

Figure 2-11 Tables et contraintes référentielles



Déclarons à présent ces contraintes sous SQL.

Contraintes côté « père »

La table « père » contient soit une contrainte de clé primaire soit une contrainte de clé candidate qui s'exprime par un index unique. Le tableau suivant illustre ces deux possibilités dans le cas de la table `Compagnie`. Notons que la table possédant une clé candidate aurait pu aussi contenir une clé primaire.

Tableau 2-14 Écritures des contraintes de la table « père »

Clé primaire	Clé candidate

`CREATE TABLE Compagnie`
`(comp VARCHAR2(4), nrue NUMBER(3),`
`Rue VARCHAR2(20), ville VARCHAR2(15),`
`nomComp VARCHAR2(15),`
`CONSTRAINT pk_Compagnie`
`PRIMARY KEY(comp));`
	`CREATE TABLE Compagnie`
	`(comp VARCHAR2(4), nrue NUMBER(3),`
	`rue VARCHAR2(20), ville VARCHAR2(15),`
	`nomComp VARCHAR2(15),`
	`CONSTRAINT un_Compagnie UNIQUE(comp));`

Contraintes côté « fils »

Indépendamment de l'écriture de la table « père », deux écritures sont possibles au niveau de la table « fils ». La première définit la contrainte en même temps que la colonne. Ainsi elle ne

convient qu'aux clés composées d'une seule colonne. La deuxième écriture détermine la contrainte après la définition de la colonne. Cette écriture est préférable car elle convient aussi aux clés composées de plusieurs colonnes de par sa lisibilité.

Tableau 2-15 Écritures des contraintes de la table « fils »

Colonne et contrainte	Contrainte et colonne

`CREATE TABLE Pilote`
`(brevet VARCHAR2(6) CONSTRAINT`
`pk_Pilote PRIMARY KEY,`
`nom VARCHAR2(15), nbHVol NUMBER(7,2),`
`compa VARCHAR2(4) CONSTRAINT fk_Pil_compa_Comp`
`REFERENCES Compagnie(comp));`
	`CREATE TABLE Pilote`
	`(brevet VARCHAR2(6), nom VARCHAR2(15),`
	`nbHVol NUMBER(7,2), compa VARCHAR2(4),`
	`CONSTRAINT pk_Pilote PRIMARY KEY(brevet),`
	`CONSTRAINT fk_Pil_compa_Comp`
	`FOREIGN KEY(compa)`
	`REFERENCES Compagnie(comp));`

Clés composites et nulles

- Les clés étrangères ou primaires peuvent être définies sur trente-deux colonnes au maximum (*composite keys*).
- Les clés étrangères peuvent être nulles si aucune contrainte `NOT NULL` n'est déclarée.

Décrivons à présent le script SQL qui crée notre exemple (la syntaxe de création des deux premières tables a été discutée plus haut) et étudions ensuite les mécanismes programmés par ces contraintes.

```
CREATE TABLE Compagnie …

CREATE TABLE Pilote …

CREATE TABLE Avion
   (immat VARCHAR2(6), typeAvion VARCHAR2(15), nbhVol NUMBER(10,2),
   proprio VARCHAR2(4),
   CONSTRAINT pk_Avion PRIMARY KEY(immat),
   CONSTRAINT nn_proprio CHECK (proprio IS NOT NULL),
   CONSTRAINT fk_Avion_comp_Compag FOREIGN KEY(proprio)
       REFERENCES Compagnie(comp));
CREATE TABLE Affreter
   (compAff VARCHAR2(4), immat VARCHAR2(6), dateAff DATE,
        nbPax NUMBER(3),
   CONSTRAINT pk_Affreter PRIMARY KEY (compAff, immat, dateAff),
   CONSTRAINT fk_Aff_na_Avion FOREIGN KEY(immat)
       REFERENCES Avion(immat),
   CONSTRAINT fk_Aff_comp_Compag FOREIGN KEY(compAff)
       REFERENCES Compagnie(comp));
```

Cohérence du fils vers le père

Si la clé étrangère est déclarée `NOT NULL`, l'insertion d'un enregistrement « fils » n'est possible que s'il est rattaché à un enregistrement « père » existant. Dans le cas inverse, l'insertion d'un enregistrement « fils » rattaché à aucun « père » est possible.

Le tableau suivant décrit des insertions correctes et une insertion incorrecte. Le message d'erreur est ici en anglais (en français : `violation de contrainte d'intégrité - touche parent introuvable`).

Tableau 2-16 Insertions correctes et incorrectes

Insertions correctes	Insertion incorrecte

`-- fils avec père`
`INSERT INTO Pilote VALUES`
`('PL-3', 'Paul Soutou', 1000, 'SING');`
`-- fils sans père`
`INSERT INTO Pilote VALUES`
`('PL-4', 'Un Connu', 0, NULL);`
`-- fils avec pères`
`INSERT INTO Avion VALUES`
`('F-WTSS', 'Concorde', 6570, 'SING');`
`INSERT INTO Affreter VALUES`
`('AF', 'F-WTSS', '15-05-2003', 82)`
	`-- avec père inconnu`
	`INSERT INTO Pilote VALUES`
	`('PL-5', 'Pb de Compagnie', 0, '?');`
	
	`ORA-02291: integrity constraint`
	`(SOUTOU.FK_PIL_COMPA_COMP) violated -`
	`parent key not found`

Pour insérer un affrètement, il faut donc avoir ajouté au préalable au moins une compagnie et un avion.

Le chargement de la base de données est conditionné par la hiérarchie des contraintes référentielles. Ici, il faut insérer d'abord les compagnies, puis les pilotes (ou les avions), enfin les affrètements.

Il suffit de relire le script de création de vos tables pour en déduire l'ordre d'insertion des enregistrements.

Cohérence du père vers le fils

Trois alternatives sont possibles pour assurer la cohérence de la table « père » vers la table « fils » via une clé étrangère :

- Prévenir la modification ou la suppression d'une clé primaire (ou candidate) de la table « père ». Cette alternative est celle par défaut. Dans notre exemple, toutes les clés étrangères sont ainsi composées. La suppression d'un avion n'est donc pas possible si ce dernier est référencé dans un affrètement.

- Propager la suppression des enregistrements « fils » associés à l'enregistrement « père » supprimé. Ce mécanisme est réalisé par la directive `ON DELETE CASCADE`. Dans notre exemple, nous pourrions ainsi décider de supprimer tous les affrètements dès qu'on retire un avion.
- Propager l'affectation de la valeur nulle aux clés étrangères des enregistrements « fils » associés à l'enregistrement « père » supprimé. Ce mécanisme est réalisé par la directive `ON DELETE SET NULL`. Il ne faut pas de contrainte `NOT NULL` sur la clé étrangère. Dans notre exemple, nous pourrions ainsi décider de mettre `NULL` dans la colonne `compa` de la table `Pilote` pour chaque pilote d'une compagnie supprimée. Nous ne pourrions pas appliquer ce mécanisme à la table `Affreter` qui dispose de contraintes `NOT NULL` sur ses clés étrangères (car composant la clé primaire).

Tableau 2-17 Cohérence du « père » vers le « fils »

Alternative	Exemple de syntaxe
Prévenir la modification ou la suppression d'une clé primaire	`CONSTRAINT fk_Aff_na_Avion FOREIGN KEY(immat) REFERENCES Avion(immat)`
Propager la suppression des enregistrements	`CONSTRAINT fk_Aff_na_Avion FOREIGN KEY(immat) REFERENCES Avion(immat) ON DELETE CASCADE`
Propager l'affectation de la valeur nulle aux clés étrangères	`CONSTRAINT fk_Pil_compa_Comp FOREIGN KEY(compa) REFERENCES Compagnie(comp) ON DELETE SET NULL`

L'extension de la modification d'une clé primaire vers les tables référencées n'est pas automatique (il faut la programmer si nécessaire par un déclencheur).

En résumé

Le tableau suivant résume les conditions requises pour modifier l'état de la base de données en respectant l'intégrité référentielle.

Tableau 2-18 Instructions SQL sur les clés

Instruction	Table « parent »	Table « fils»
`INSERT`	Correcte si la clé primaire (ou candidate) est unique.	Correcte si la clé étrangère est référencée dans la table « père » ou est nulle (partiellement ou en totalité).
`UPDATE`	Correcte si l'instruction ne laisse pas d'enregistrements dans la table « fils » ayant une clé étrangère non référencée.	Correcte si la nouvelle clé étrangère référence un enregistrement « père » existant.
`DELETE`	Correcte si aucun enregistrement de la table « fils » ne référence le ou les enregistrements détruits.	Correcte sans condition.
`DELETE CASCADE`	Correcte sans condition.	Correcte sans condition.
`DELETE SET NULL`	Correcte sans condition.	Correcte sans condition.

Exercices

Les objectifs des premiers exercices sont :

- d'insérer des données dans les tables du schéma *Parc Informatique* et du schéma des chantiers ;
- de créer une séquence et d'insérer des données en utilisant une séquence ;
- de modifier des données.

Exercice 2.1 Insertion de données

Écrivez puis exécutez le script SQL (que vous appellerez `insParc.sql`) afin d'insérer les données dans les tables suivantes :

Tableau 2-19 Données des tables

Table Segment

INDIP	NOMSEGMENT	ETAGE
130.120.80	Brin RDC	
130.120.81	Brin 1er étage	
130.120.82	Brin 2ème étage	

Table Salle

NSALLE	NOMSALLE	NBPOSTE	INDIP
s01	Salle 1	3	130.120.80
s02	Salle 2	2	130.120.80
s03	Salle 3	2	130.120.80
s11	Salle 11	2	130.120.81
s12	Salle 12	1	130.120.81
s21	Salle 21	2	130.120.82
s22	Salle 22	0	130.120.83
s23	Salle 23	0	130.120.83

Table Poste

NPOSTE	NOMPOSTE	INDIP	AD	TYPEPOSTE	NSALLE
p1	Poste 1	130.120.80	01	TX	s01
p2	Poste 2	130.120.80	02	UNIX	s01
p3	Poste 3	130.120.80	03	TX	s01
p4	Poste 4	130.120.80	04	PCWS	s02
p5	Poste 5	130.120.80	05	PCWS	s02
p6	Poste 6	130.120.80	06	UNIX	s03
p7	Poste 7	130.120.80	07	TX	s03
p8	Poste 8	130.120.81	01	UNIX	s11
p9	Poste 9	130.120.81	02	TX	s11
p10	Poste 10	130.120.81	03	UNIX	s12
p11	Poste 11	130.120.82	01	PCNT	s21
p12	Poste 12	130.120.82	02	PCWS	s21

Tableau 2-19 Données des tables *(suite)*

Table	Données

Logiciel

NLOG	NOMLOG	DATEACH	VERSION	TYPELOG	PRIX
log1	Oracle 6	13/05/95	6.2	UNIX	3000
log2	Oracle 8	15/09/99	8i	UNIX	5600
log3	SQL Server	12/04/98	7	PCNT	2700
log4	Front Page	03/06/97	5	PCWS	500
log5	WinDev	12/05/97	5	PCWS	750
log6	SQL*Net		2.0	UNIX	500
log7	I. I. S.	12/04/02	2	PCNT	810
log8	DreamWeaver	21/09/03	2.0	BeOS	1400

Types

TYPELP	NOMTYPE
TX	Terminal X-Window
UNIX	Système Unix
PCNT	PC Windows NT
PCWS	PC Windows
NC	Network Computer

Exercice 2.2 Gestion d'une séquence

Dans ce même script, créez la séquence `sequenceIns` commençant à la valeur 1, d'incrément 1, de valeur maximale 10 000 et sans cycle. Utilisez cette séquence pour estimer la colonne `numIns` de la table `Installer`. Insérez les enregistrements suivants :

Tableau 2-20 Données de la table Installer

Table	Données

Installer

NPOSTE	NLOG	NUMINS	DATEINS	DELAI
p2	log1	1	15/05/03	
p2	log2	2	17/09/03	
p4	log5	3		
p6	log6	4	20/05/03	
p6	log1	5	20/05/03	
p8	log2	6	19/05/03	
p8	log6	7	20/05/03	
p11	log3	8	20/04/03	
p12	log4	9	20/04/03	
p11	log7	10	20/04/03	
p7	log7	11	01/04/02	

Exercice 2.3 **Modification de données**

Écrivez le script `modification.sql`, qui permet de modifier (avec `UPDATE`) la colonne `etage` (pour l'instant nulle) de la table `Segment` afin d'affecter un numéro d'étage correct (0 pour le segment 130.120.80, 1 pour le segment 130.120.81, 2 pour le segment 130.120.82).

Diminuez de 10 % le prix des logiciels de type 'PCNT'.

Vérifiez :

```
SELECT * FROM Segment;
SELECT nLog, typeLog, prix FROM Logiciel;
```

Exercice 2.4 **Insertion dans la base *Chantiers***

Écrivez puis exécutez le script SQL (que vous appellerez `insChantier.sql`) afin d'insérer les données suivantes :

- une dizaine d'employés (numéros E1 à E10) en considérant diverses qualifications (OS, Assistant, Ingénieur et Architecte) ;
- quatre chantiers et cinq véhicules ;
- deux ou trois visites de différents chantiers durant trois jours ;
- la composition (de un à trois employés transportés) de chaque visite.

Chapitre 3

Évolution d'un schéma

L'évolution d'un schéma est un aspect très important à prendre en compte, car il répond aux besoins de maintenance des applicatifs qui utilisent la base de données. Nous verrons qu'il est possible de modifier une base de données d'un point de vue structurel (colonnes et index) mais aussi comportemental (contraintes).

L'instruction principalement utilisée est `ALTER TABLE` (commande du LDD) qui permet d'ajouter, de renommer, de modifier et de supprimer des colonnes d'une table. Elle permet aussi d'ajouter, de supprimer, d'activer, de désactiver et de différer des contraintes. Avant de détailler ces mécanismes, étudions la commande qui permet de renommer une table.

Renommer une table (RENAME)

L'instruction `RENAME` renomme une table. Cette commande convient aussi aux séquences, synonymes et vues. Il faut être propriétaire de l'objet que l'on renomme.

```
RENAME ancienNom TO nouveauNom;
```

Les contraintes d'intégrité, index et prérogatives associés à l'ancienne table sont automatiquement transférés sur la nouvelle. En revanche, les vues, synonymes et procédures catalogués sont invalidés et doivent être recréés.

Il est aussi possible d'utiliser la directive `RENAME TO` de l'instruction `ALTER TABLE` pour renommer une table existante. Le tableau suivant décrit comment renommer la table `Pilote` sans perturber l'intégrité référentielle :

Tableau 3-1 Renommer une table

Commande `RENAME`	Commande `ALTER TABLE`
`RENAME Pilote TO Navigant;`	`ALTER TABLE Pilote RENAME TO Navigant;`

Modifications structurelles (ALTER TABLE)

Considérons la table suivante que nous allons faire évoluer :

```
SQL> CREATE TABLE Pilote
     (brevet VARCHAR2(4), prenom VARCHAR2(20), nom VARCHAR2(20));
Table créée.
SQL> INSERT INTO Pilote (brevet, prenom, nom) VALUES ('PL-1', 'JP',
                         'Ferrage');
1 ligne créée.
```

Figure 3-1 *Table avant les modifications*

PILOTE

BREVET	PRENOM	NOM
PL-1	JP	Ferrage

Ajouter des colonnes

La directive `ADD` de l'instruction `ALTER TABLE` permet d'ajouter une nouvelle colonne à une table. Cette colonne est initialisée à `NULL` pour tous les enregistrements (à moins de spécifier une contrainte `DEFAULT`, auquel cas tous les enregistrements de la table sont mis à jour avec une valeur non nulle).

Il est possible d'ajouter une colonne en ligne `NOT NULL` seulement si la table est vide ou si une contrainte `DEFAULT` est définie sur la nouvelle colonne (dans le cas inverse, il faudra utiliser `MODIFY` à la place de `ADD`).

Le script suivant ajoute trois colonnes à la table `Pilote`. La première instruction insère la colonne `nbHVol` en l'initialisant à `NULL` pour tous les pilotes (ici il n'en existe qu'une seule). La deuxième commande ajoute deux colonnes initialisées à une valeur non nulle. La colonne `ville` ne sera jamais nulle.

```
ALTER TABLE Pilote ADD (nbHVol NUMBER(7,2));
ALTER TABLE Pilote
      ADD (compa VARCHAR2(4)  DEFAULT 'AF',
           ville VARCHAR2(30) DEFAULT 'Paris' NOT NULL);
```

La table est désormais la suivante :

Figure 3-2 *Table après l'ajout de colonnes*

PILOTE

BREVET	PRENOM	NOM	NBHVOL	COMPA	VILLE
PL-1	JP	Ferrage	(null)	AF	Paris

Renommer des colonnes

Il faut utiliser la directive RENAME COLUMN de l'instruction ALTER TABLE pour renommer une colonne existante. Le nom de la nouvelle colonne ne doit pas être déjà utilisé par une colonne de la table.

L'instruction suivante permet de renommer la colonne ville en adresse :

```
ALTER TABLE Pilote RENAME COLUMN ville TO adresse;
```

Modifier le type des colonnes

La directive MODIFY de l'instruction ALTER TABLE modifie le type d'une colonne existante.

Il est possible d'augmenter la taille d'une colonne numérique (largeur ou précision) – ou d'une chaîne de caractères (CHAR et VARCHAR2) – ou de la diminuer si toutes les données présentes dans la colonne peuvent s'adapter à la nouvelle taille.

Les contraintes en ligne peuvent être aussi modifiées par cette instruction (DEFAULT, NOT NULL, UNIQUE, PRIMARY KEY et FOREIGN KEY). Une fois la colonne changée, les nouvelles contraintes s'appliqueront aux mises à jour ultérieures de la base.

Le tableau suivant présente différentes modifications de colonnes.

Tableau 3-2 Modifications de colonnes

Instructions SQL	Commentaires

`ALTER TABLE Pilote`
`  MODIFY compa VARCHAR(6)`
`  DEFAULT 'SING';`
`INSERT INTO Pilote (brevet, prenom, nom)`
`VALUES ('PL-2', 'Arnaud', 'Sayag');`
	Augmente la taille de la colonne compa et change la contrainte de valeur par défaut.

`ALTER TABLE Pilote` `MODIFY compa CHAR(4) NOT NULL;`	Diminue la colonne et modifie également son type de VARCHAR2 en CHAR tout en le déclarant NOT NULL (possible car les données contenues dans la colonne ne dépassent pas quatre caractères).
`ALTER TABLE Pilote MODIFY compa NULL;`	Rend possible l'insertion de valeur nulle dans la colonne compa.

La table est désormais la suivante :

Figure 3-3 Après modification des colonnes

PILOTE

BREVET	PRENOM	NOM	NBHVOL	COMPA	ADRESSE
PL-1	JP	Ferrage	(null)	AF	Paris
PL-2	Arnaud	Sayag	(null)	SING	Paris

Supprimer des colonnes

Longtemps absente, la possibilité de supprimer une colonne permet à présent de récupérer rapidement de l'espace disque et évite aux administrateurs d'exporter, d'ajouter, d'importer des tables et de recréer les index et les contraintes.

La directive `DROP COLUMN` de l'instruction `ALTER TABLE` permet de supprimer une colonne.

Il n'est pas possible de supprimer avec cette instruction :

- des clés primaires (ou candidates par `UNIQUE`) référencées par des clés étrangères ;
- des colonnes à partir desquelles un index a été construit ;
- des pseudo-colonnes (`ROWID` et `LEVEL`) ou des colonnes de tables objets ;
- toutes les colonnes d'une table.

La suppression de la colonne `adresse` de la table `Pilote` est programmée par l'instruction suivante :

```
ALTER TABLE Pilote DROP COLUMN adresse;
```

Colonnes UNUSED

Si vous désirez marquer des colonnes à l'effacement (sans les enlever de la table), il faut utiliser la directive `SET UNUSED COLUMN` de l'instruction `ALTER TABLE`.

Les colonnes n'apparaîtront plus dans la description de la table et ne seront plus accessibles tout en restant toujours présentes dans la table. Les contraintes, index associés à ces colonnes, sont supprimées.

Cette option est intéressante dans la mesure où le résultat est immédiat, car aucune répercussion d'ordre physique sur la base n'est opérée. Le temps d'exécution de suppression de colonnes sur des bases de taille importante peut être très pénalisant.

Marquons à l'effacement la colonne compa :

```
ALTER TABLE Pilote SET UNUSED COLUMN compa;
```

Il n'est plus possible de récupérer les colonnes marquées à l'effacement d'une table pour les rendre à nouveau opérationnelles. Seule la directive DROP UNUSED COLUMNS est permise pour manipuler de telles colonnes. Elle détruit toutes les colonnes d'une table qui sont marquées à l'effacement.

Détruisons les colonnes marquées à l'effacement de la table Pilote :

```
ALTER TABLE Pilote DROP UNUSED COLUMNS;
```

Colonne virtuelle

La particularité d'une colonne virtuelle réside dans le fait qu'elle n'est pas stockée sur le disque mais évaluée automatiquement à la demande (au sein d'une requête, ou d'une instruction de mise à jour). Par analogie, les vues (étudiées au chapitre 5) sont des tables virtuelles.

Création d'une table

Au niveau de la création d'une table, la syntaxe à adopter est la suivante :

```
colonne [typeSQL] [GENERATED ALWAYS] AS (expression)
   [VIRTUAL] [ contrainteLigne [contrainteLigne2]...]
```

- Le type de la colonne (si vous ne voulez pas qu'il soit automatiquement déduit de l'expression) suit éventuellement son nom.
- Les directives GENERATED ALWAYS et VIRTUAL sont fournies pour rendre le code plus clair (considérées comme des commentaires).
- L'expression qui suit la directive AS détermine la valeur de la colonne (valeur scalaire).

Le script suivant déclare la table Avion comportant une colonne virtuelle (qui permet ici de calculer le nombre d'heures de vol par mois). Deux lignes sont ensuite ajoutées.

```
CREATE TABLE Avion(immat VARCHAR2(6), typeAvion VARCHAR2(15),
  nbhVol NUMBER(10,2), age NUMBER(4,1),
  freqVolMois GENERATED ALWAYS AS (nbhVol/age/12) VIRTUAL,
  nbPax NUMBER(3), CONSTRAINT pk_Avion PRIMARY KEY(immat));

INSERT INTO Avion (immat,typeAvion,nbhVol,age,nbPax)
   VALUES ('F-WTSS', 'Concorde', 20000, 18, 90);

INSERT INTO Avion (immat,typeAvion,nbhVol,age,nbPax)
   VALUES ('F-GHTY', 'A380', 450, 0.5, 460);
```

La description de cette table (DESC) fait apparaître la colonne virtuelle. Pour obtenir les valeurs de la colonne, il suffit, par exemple, d'évaluer son expression à partir d'une requête.

```
SELECT immat,freqVolMois FROM Avion;

IMMAT  FREQVOLMOIS
------ --------------
F-WTSS 92,59259592...
F-GHTY 75
```

Une colonne virtuelle peut être indexée mais pas directement modifiable. Seule une modification des valeurs qui interviennent dans l'expression fera évoluer une colonne virtuelle.

Ajout d'une colonne

Au niveau de la création d'une table, la syntaxe à adopter est la suivante :

```
ALTER TABLE nomTable ADD
 colonne [typeSQL] [GENERATED ALWAYS] AS (expression)
   [VIRTUAL] [ contrainteLigne [contrainteLigne2]...];
```

Ajoutons à la table `Avion` la colonne virtuelle qui détermine le ratio du nombre d'heures de vol par passager en y ajoutant deux contraintes en ligne.

```
ALTER TABLE Avion
  ADD heurePax NUMBER(10,2) AS (nbhVol/age)
  CHECK (heurePax BETWEEN 0 AND 2000) NOT NULL;
```

La figure suivante illustre comment la table se comporte lorsqu'elle est sollicitée en `INSERT`, `UPDATE` ou `DELETE`.

Figure 3-4 *Colonne virtuelles*

`Avion`

immat	typeAvion	nbHVol	age	nbPax	freqVolMois	HeurePax
F-WTSS	Concorde	20 000	18	90	92,5925...	111,11
F-GHTY	A380	450	0,5	460	75	900

Restrictions

- Seules les tables relationnelles de type *heap* (par défaut) peuvent héberger des colonnes virtuelles (interdites dans les tables organisées en index, externes, objet-relationnelles, cluster, et temporaires).
- L'expression de définition d'une colonne virtuelle ne peut pas faire référence à une autre colonne virtuelle et ne peut être construite qu'avec des colonnes d'une même table.
- Le type d'une colonne virtuelle ne peut être XML, *any*, *spatial*, *media*, personnalisé (*user-defined*), *LOB* ou `LONG RAW`.

Colonnes invisibles

Depuis la version 12*c*, il est possible de masquer des colonnes au niveau des applications. Par défaut, toute colonne est visible mais peut être masquée lors de la création de la table en ajoutant l'option `INVISIBLE` après avoir déclaré son type. L'instruction `ALTER TABLE` *`nom_table`* `MODIFY` *`nom_colonne`* `[VISIBLE | INVISIBLE]` permet de démasquer (ou de masquer) des colonnes existantes.

Le code suivant présente l'utilisation de cette option dans une table. La première insertion est incorrecte car elle sous-entend que les quatre colonnes de la table sont visibles.

Tableau 3-3 Colonne invisible

Déclaration d'une colonne invisible

```
CREATE TABLE passagers
 (id_pax     NUMBER(3),
  nom_pax    VARCHAR2(30),
  bonus      NUMBER(7,2) INVISIBLE,
  date_nais DATE);
```

Insertions

```
SQL> INSERT INTO passagers VALUES
  (1,'Belfont',20.5,TO_DATE('05/03/1974','DD/MM/YYYY'));
ERREUR à la ligne 1 : ORA-00913: trop de valeurs

SQL> INSERT INTO passagers(id_pax,nom_pax,date_nais)
       VALUES
        (1,'Belfont',TO_DATE('05/03/1974','DD/MM/YYYY'));
1 ligne créée.

SQL> INSERT INTO passagers
    (id_pax,nom_pax,bonus,date_nais) VALUES
(2,'Fontbel',45.6,TO_DATE('05/03/1990','DD/MM/YYYY'));
1 ligne créée.
```

Une fois la colonne invisible insérée, ses données sont accessibles d'une manière explicite. Ce comportement sera respecté jusqu'à ce que la colonne redevienne visible par `ALTER TABLE passagers MODIFY bonus VISIBLE`.

Tableau 3-4 Extraction d'une colonne invisible

Requête implicite

```
SQL> SELECT * FROM passagers ;

    ID_PAX NOM_PAX              DATE_NAI
---------- -------------------- --------
         1 Belfont              05/03/74
         2 Fontbel              05/03/90
```

Requête explicite

```
SQL> SELECT id_pax, nom_pax, bonus FROM passagers;

    ID_PAX NOM_PAX                BONUS
---------- -------------------- -------
         1 Belfont
         2 Fontbel                 45,6
```

Dans SQL*Plus, positionnez la variable d'environnement `SET COLINVISIBLE ON` afin de constater l'existence de colonnes virtuelles à la suite d'une commande `DESCRIBE` *`nom_table`*.

Modifications comportementales

Nous étudions dans cette section les mécanismes d'ajout, de suppression, d'activation et de désactivation des contraintes.

Faisons évoluer le schéma suivant. Les clés primaires sont nommées `pk_Compagnie` pour la table `Compagnie` et `pk_Avion` pour la table `Avion`.

Figure 3-5 *Schéma à faire évoluer*

`Compagnie`

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

`Affreter`

compAff	immat	dateAff	nbPax
AF	F-WTSS	13-05-2003	85
SING	F-GAFU	05-02-2003	155
AF	F-WTSS	15-05-2003	82

`Avion`

immat	typeAvion	nbHVol	proprio
F-WTSS	Concorde	6570	SING
F-GAFU	A320	3500	AF
F-GLFS	TB-20	2000	SING

Ajout de contraintes

Jusqu'à présent, nous avons créé des tables en même temps que les contraintes. Il est possible de créer des tables seules (dans ce cas l'ordre de création n'est pas important et on peut même les créer par ordre alphabétique), puis d'ajouter les contraintes. Les outils de conception (*Win'Design*, *Designer* ou *PowerAMC*) adoptent cette démarche lors de la génération automatique de scripts SQL.

La directive `ADD CONSTRAINT` de l'instruction `ALTER TABLE` permet d'ajouter une contrainte à une table. La syntaxe générale est la suivante :

```
ALTER TABLE [schéma.]nomTable
ADD [CONSTRAINT nomContrainte] typeContrainte;
```

Comme pour l'instruction `CREATE TABLE`, quatre types de contraintes sont possibles :

- `UNIQUE (colonne1 [,colonne2]…)`
- `PRIMARY KEY (colonne1 [,colonne2]…)`

  `FOREIGN KEY (colonne1 [,colonne2]…)`

  `REFERENCES [schéma.]nomTablePère (colonne1 [,colonne2]…)`

  `[ON DELETE { CASCADE | SET NULL }]`
- `CHECK (condition)`

Clé étrangère

Ajoutons la clé étrangère à la table `Avion` au niveau de la colonne `proprio` en lui assignant une contrainte `NOT NULL` :

```
ALTER TABLE Avion
  ADD (CONSTRAINT nn_proprio CHECK (proprio IS NOT NULL),
       CONSTRAINT fk_Avion_comp_Compag FOREIGN KEY(proprio)
                        REFERENCES Compagnie(comp));
```

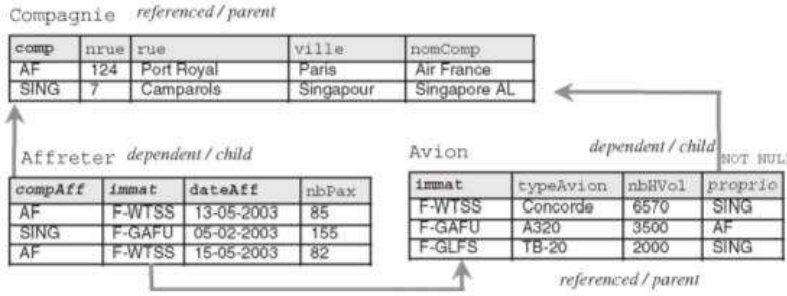
Clé primaire

Ajoutons la clé primaire de la table `Affreter` et deux clés étrangères (vers les tables `Avion` et `Compagnie`) :

```
ALTER TABLE Affrete ADD (
  CONSTRAINT pk_Affreter          PRIMARY KEY (compAff, immat, dateAff),
  CONSTRAINT fk_Aff_na_Avion      FOREIGN KEY(immat) REFERENCES
                                  Avion(immat),
  CONSTRAINT fk_Aff_comp_Compag FOREIGN KEY(compAff)
                  REFERENCES Compagnie(comp));
```

Pour que l'ajout d'une contrainte soit possible, il faut que les données présentes dans la table respectent la nouvelle contrainte (nous étudierons plus tard les moyens de pallier ce problème). Les tables contiennent les contraintes suivantes :

Figure 3-6 *Après ajout de contraintes*



Suppression de contraintes

La directive `DROP CONSTRAINT` de l'instruction `ALTER TABLE` permet d'enlever une contrainte d'une table. La syntaxe générale est la suivante :

```
ALTER TABLE [schéma.]nomTable DROP CONSTRAINT nomContrainte [CASCADE];
```

La directive CASCADE supprime les contraintes référentielles des tables « pères ». On comprend mieux maintenant pourquoi il est si intéressant de nommer les contraintes plutôt que d'utiliser les noms automatiquement générés.

Supprimons la contrainte NOT NULL qui porte sur la colonne proprio de la table Avion :

```
ALTER TABLE Avion DROP CONSTRAINT nn_proprio;
```

Clé étrangère

Supprimons la clé étrangère de la colonne proprio. Il n'est pas besoin de spécifier CASCADE, car il s'agit d'une table « fils » pour cette contrainte d'intégrité référentielle.

```
ALTER TABLE Avion DROP CONSTRAINT fk_Avion_comp_Compag;
```

Clé primaire (ou candidate)

Supprimons la clé primaire de la table Avion. Il faut préciser CASCADE, car cette table est référencée par une clé étrangère dans la table Affreter. Cette commande supprime à la fois la clé primaire de la table Avion mais aussi les contraintes clés étrangères des tables dépendantes (ici seule la clé étrangère de la table Affreter est supprimée).

```
ALTER TABLE Avion DROP CONSTRAINT pk_Avion CASCADE;
```

Si l'option CASCADE n'avait pas été spécifiée, Oracle aurait renvoyé l'erreur « ORA-02273: cette clé unique/primaire est référencée par des clés étrangères ».

La figure suivante illustre les trois contraintes qui restent : les clés primaires des tables Compagnie et Affreter et la clé étrangère de la table Affreter.

Figure 3-7 *Après suppression de contraintes*

Compagnie *referenced / parent*

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Affreter *dependent / child*

compAff	immat	dateAff	nbPax
AF	F-WTSS	13-05-2003	85
SING	F-GAFU	05-02-2003	155
AF	F-WTSS	15-05-2003	82

Avion

immat	typeAvion	nbHVol	proprio
F-WTSS	Concorde	6570	SING
F-GAFU	A320	3500	AF
F-GLFS	TB-20	2000	SING

Les deux possibilités pour supprimer ces trois contraintes sont décrites dans le tableau suivant. La deuxième écriture est plus rigoureuse car elle prévient des effets de bord. Il suffit, pour les

éviter, de détruire les contraintes dans l'ordre inverse d'apparition dans le script de création (tables « fils » puis « pères »).

Tableau 3-5 Suppression de contraintes

Avec CASCADE	Sans CASCADE

`ALTER TABLE Compagnie`
`  DROP CONSTRAINT pk_Compagnie CASCADE;`
`ALTER TABLE Affreter`
`  DROP CONSTRAINT pk_Affreter;`
	`ALTER TABLE Affreter`
	`  DROP CONSTRAINT fk_Aff_comp_Compag;`
	`ALTER TABLE Compagnie`
	`  DROP CONSTRAINT pk_Compagnie;`
	`ALTER TABLE Affreter`
	`  DROP CONSTRAINT pk_Affreter;`

Désactivation de contraintes

La désactivation de contraintes peut être intéressante pour accélérer des procédures de chargement (importation par SQL*Loader) et d'exportation massive de données. Ce mécanisme améliore aussi les performances de programmes *batchs* qui ne modifient pas des données concernées par l'intégrité référentielle ou pour lesquelles on vérifie la cohérence de la base à la fin.

La directive `DISABLE CONSTRAINT` de l'instruction `ALTER TABLE` permet de désactiver temporairement (jusqu'à la réactivation) une contrainte existante.

Syntaxe

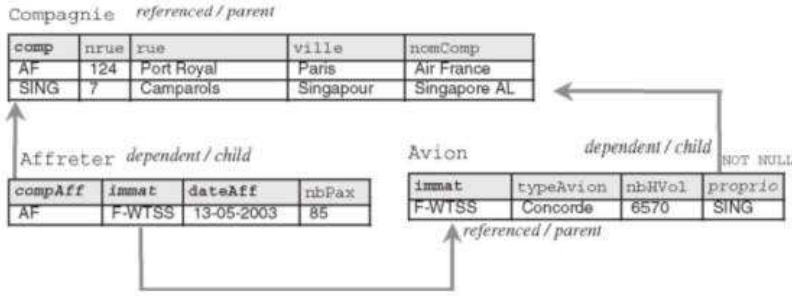
La syntaxe générale est la suivante :

```
ALTER TABLE [schéma.]nomTable
   DISABLE [ VALIDATE | NOVALIDATE ] CONSTRAINT nomContrainte
   [CASCADE] [ { KEEP | DROP } INDEX ] ;
```

- `CASCADE` répercute la désactivation des clés étrangères des tables « fils » dépendantes. Si vous voulez désactiver une clé primaire référencée par une clé étrangère sans cette option, le message d'Oracle renvoyé est : « `ORA-02297: impossible désactiver contrainte... - les dépendences existent` ».
- Les options `KEEP INDEX` et `DROP INDEX` permettent de préserver ou de détruire l'index dans le cas de la désactivation d'une clé primaire.
- Nous verrons plus loin l'explication des options `VALIDATE` et `NOVALIDATE`.

En considérant l'exemple suivant, désactivons quelques contraintes et insérons des enregistrements ne respectant pas les contraintes désactivées.

Figure 3-8 Avant la désactivation de contraintes



Contrainte de vérification

Désactivons la contrainte `NOT NULL` qui porte sur la colonne `proprio` de la table `Avion` et insérons un avion qui n'est rattaché à aucune compagnie :

```
ALTER TABLE Avion DISABLE CONSTRAINT nn_proprio;
INSERT INTO Avion VALUES ('Bidon1', 'TB-20', 2000, NULL);
```

Clé étrangère

Désactivons la contrainte de clé étrangère qui porte sur la colonne `proprio` de la table `Avion` et insérons un avion rattaché à une compagnie inexistante :

```
ALTER TABLE Avion DISABLE CONSTRAINT fk_Avion_comp_Compag;
INSERT INTO Avion VALUES ('F-GLFS', 'TB-22', 500, 'Toto');
```

Clé primaire

Désactivons la contrainte de clé primaire de la table `Avion`, en supprimant en même temps l'index, et insérons un avion ne respectant plus la clé primaire :

```
ALTER TABLE Avion DISABLE CONSTRAINT pk_Avion CASCADE DROP INDEX;
INSERT INTO Avion VALUES ('Bidon1', 'TB-21', 1000, 'AF');
```

La désactivation de cette contrainte par `CASCADE` supprime aussi une des clés étrangères de la table `Affreter`. Insérons un affrètement qui référence un avion inexistant :

```
INSERT INTO Affreter VALUES ('AF', 'Toto', '13-05-2003', 0);
```

L'état de la base est désormais comme suit. Les *rowids* sont précisés pour illustrer les options de réactivation.

Bien qu'il semble incohérent de réactiver les contraintes sans modifier les valeurs ne respectant pas les contraintes (notées en gras), nous verrons que plusieurs alternatives sont possibles.

Figure 3-9 Après désactivation de contraintes

Réactivation de contraintes

La directive `ENABLE CONSTRAINT` de l'instruction `ALTER TABLE` permet de réactiver une contrainte.

Syntaxe

La syntaxe générale est la suivante :

```
ALTER TABLE [schéma.]nomTable
   ENABLE [ VALIDATE | NOVALIDATE ] CONSTRAINT nomContrainte
   [USING INDEX ClauseIndex] [EXCEPTIONS INTO tableErreurs];
```

- La clause d'index permet, dans le cas des clés primaires ou candidates (`UNIQUE`), de pouvoir recréer l'index associé.
- La clause d'exceptions permet de retrouver les enregistrements ne vérifiant pas la nouvelle contrainte (cas étudié au paragraphe suivant).

Il n'est pas possible de réactiver une clé étrangère tant que la contrainte de clé primaire référencée n'est pas active.

En supposant que les tables contiennent des données qui respectent les contraintes à réutiliser, la réactivation de la clé primaire (en recréant l'index) et d'une contrainte `NOT NULL` de la table `Avion` se programmerait ainsi :

```
ALTER TABLE Avion ENABLE CONSTRAINT pk_Avion
   USING INDEX (CREATE UNIQUE INDEX pk_Avion ON Avion (immat));

ALTER TABLE Avion ENABLE CONSTRAINT nn_proprio;
```

Récupération de données erronées

L'option `EXCEPTIONS INTO` de l'instruction `ALTER TABLE` permet de récupérer automatiquement les enregistrements qui ne respectent pas des contraintes afin de les traiter (modifier, supprimer ou déplacer) avant de réactiver les contraintes en question sur une table saine.

Il faut créer une table composée de quatre colonnes :

- La première, de type `ROWID`, contiendra les adresses des enregistrements ne respectant pas la contrainte ;
- la deuxième colonne de type `varchar2(30)` contiendra le nom du propriétaire de la table ;
- la troisième colonne de type `varchar2(30)` contiendra le nom de la table ;
- la quatrième, de type `varchar2(30)`, contiendra le nom de la contrainte.

Le tableau suivant décrit deux tables permettant de stocker les enregistrements erronés après réactivation de contraintes.

Il est permis d'utiliser des noms de table ou de colonne différents mais il n'est pas possible d'utiliser une structure de table différente.

Tableau 3-6 Tables des rejets

Tables conventionnelles (*heap*)	Toutes tables (*heap*, *index-organized*)

`CREATE TABLE Problemes`
`  (adresse ROWID,`
`  utilisateur VARCHAR2(30),`
`  nomTable VARCHAR2(30),`
`  nomContrainte VARCHAR2(30));`
	`CREATE TABLE ProblemesBis`
	`  (adresse UROWID,`
	`  utilisateur VARCHAR2(30),`
	`  nomTable VARCHAR2(30),`
	`  nomContrainte VARCHAR2(30));`

La commande de réactivation d'une contrainte avec l'option met automatiquement à jour la table des rejets et renvoie une erreur s'il existe un enregistrement ne respectant pas la contrainte.

Réactivons la contrainte `NOT NULL` concernant la colonne `proprio` de la table `Avion` (enregistrement incohérent de `ROWID` R4) :

```
ALTER TABLE Avion ENABLE CONSTRAINT nn_proprio EXCEPTIONS INTO
Problemes;

ORA-02293: impossible de valider (SOUTOU.NN_PROPRIO) - violation d'une
contrainte de contrôle
```

Réactivons la contrainte de clé étrangère sur cette même colonne (enregistrement incohérent : `ROWID` R6 n'a pas de compagnie référencée).

```
ALTER TABLE Avion ENABLE CONSTRAINT fk_Avion_comp_Compag
      EXCEPTIONS INTO Problemes;

ORA-02298: impossible de valider (SOUTOU.FK_AVION_COMP_COMPAG) - clés
parents introuvables
```

Réactivons la contrainte de clé primaire de la table Avion (enregistrements incohérents : ROWID R5 et R6 ont la même immatriculation) :

```
ALTER TABLE Avion ENABLE CONSTRAINT pk_Avion EXCEPTIONS INTO
Problemes;

ORA-02437: impossible de valider (SOUTOU.PK_AVION) - violation de la clé
primaire
```

La table Problemes contient à présent les enregistrements suivants :

Figure 3-10 *Table des rejets*

Problemes

adresse	utilisateur	nomTable	nomContrainte
R4	*nomUserOracle*	AVION	NN_PROPRIO
R6	*nomUserOracle*	AVION	FK_AVION_COMP_COMPAG
R5	*nomUserOracle*	AVION	PK_AVION
R4	*nomUserOracle*	AVION	PK_AVION

Il apparaît que les trois enregistrements (R4, R5 et R6) ne respectent pas des contraintes dans la table Avion. Il convient de les traiter au cas par cas et par type de contrainte. Il est possible d'automatiser l'extraction des enregistrements qui ne respectent pas les contraintes en faisant une jointure (voir le chapitre suivant) entre la table des exceptions et la table des données (on testera la valeur des *rowids*).

Dans notre exemple, choisissons :

- de modifier l'immatriculation de l'avion 'Bidon1' (*rowid* R4) en 'F-TB20' dans la table Avion :

```
UPDATE Avion SET immat = 'F-TB20'
       WHERE immat = 'Bidon1' AND typeAvion = 'TB-20';
```

- d'affecter la compagnie 'AF' aux avions n'appartenant pas à la compagnie 'SING' dans la table Avion (mettre à jour les enregistrements de *rowid* R4 et R6) :

```
UPDATE Avion SET proprio = 'AF' WHERE NOT(proprio = 'SING');
```

- de modifier l'immatriculation de l'avion 'Toto' en 'F-TB20' dans la table Affreter :

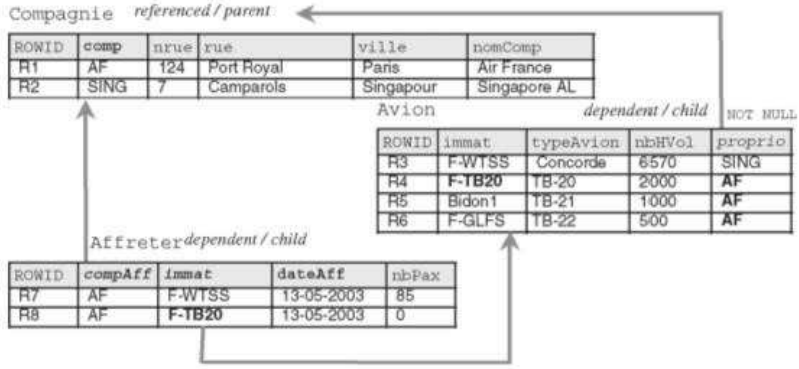
```
UPDATE Affreter SET immat = 'F-TB20' WHERE immat = 'Toto';
```

Avant de réactiver à nouveau les contraintes, il convient de supprimer les lignes de la table d'exceptions (ici Problemes). La réactivation de toutes les contraintes avec l'option EXCEPTIONS INTO ne génère plus aucune erreur et la table d'exceptions est encore vide.

```
DELETE FROM Problemes ;
ALTER TABLE Avion ENABLE CONSTRAINT nn_proprio EXCEPTIONS INTO
Problemes;
ALTER TABLE Avion ENABLE CONSTRAINT fk_Avion_comp_Compag
EXCEPTIONS INTO Problemes;
ALTER TABLE Avion ENABLE CONSTRAINT pk_Avion EXCEPTIONS INTO Problemes;
ALTER TABLE Affreter ENABLE CONSTRAINT fk_Aff_na_Avion
EXCEPTIONS INTO Problemes;
```

L'état de la base avec les contraintes réactivées est le suivant (les mises à jour sont en gras) :

Figure 3-11 *Tables après modification et réactivation des contraintes*



Contraintes différées

Une contrainte est dite « différée » (*deferred*) si elle déclenche sa vérification seulement en atteignant le premier `COMMIT` rencontré. Si la contrainte n'existe pas, aucune commande de la transaction (suite d'instructions terminées par `COMMIT`) n'est réalisée. Les contraintes que nous avons étudiées jusqu'à maintenant étaient des contraintes immédiates (*immediate*) qui sont contrôlées après chaque instruction.

Directives DEFERRABLE et INITIALLY

Depuis la version 8*i*, il est possible de différer à la fin d'un traitement la vérification des contraintes par les directives `DEFERRABLE` et `INITIALLY`.

Chaque contrainte peut être reportée ou pas et est initialement définie différée ou immédiate. En l'absence de directives particulières, le comportement par défaut de toute contrainte est NOT DEFERRABLE INITIALLY IMMEDIATE.

Les contraintes NOT DEFERRABLE ne pourront jamais être différées (à moins de les détruire et de les recréer). Pour différer une ou plusieurs contraintes DEFERRABLE INITIALLY IMMEDIATE dans une transaction, il faut utiliser les instructions SQL SET CONSTRAINT(S). Pour reporter une ou plusieurs contraintes DEFERRABLE INITIALLY IMMEDIATE dans une session (suite de transactions), il faut employer la commande ALTER SESSION SET CONSTRAINTS.

Les instructions SET CONSTRAINT(S) caractérisent une ou plusieurs contraintes DEFERRABLE en mode différé (DEFERRED) ou en mode immédiat (IMMEDIATE). Il n'est pas possible d'utiliser l'instruction SET CONSTRAINT dans le corps d'un déclencheur.

Le tableau suivant illustre l'utilisation des deux modes en différant une clé étrangère :

Tableau 3-7 Contrainte DEFERRABLE

Mode différé	Mode immédiat
CREATE TABLE Compagnie (comp VARCHAR2(4), nrue NUMBER(3), rue VARCHAR2(20), ville VARCHAR2(15), nomComp VARCHAR2(15), CONSTRAINT pk_Compagnie PRIMARY KEY(comp) **NOT DEFERRABLE INITIALLY IMMEDIATE**);	
CREATE TABLE Avion (immat VARCHAR2(6), typeAvion VARCHAR2(15), nbhVol NUMBER(10,2), proprio VARCHAR2(4), CONSTRAINT fk_Avion_comp_Compag FOREIGN KEY(proprio) REFERENCES Compagnie(comp) **DEFERRABLE INITIALLY DEFERRED**, CONSTRAINT pk_Avion PRIMARY KEY(immat));	CREATE TABLE Avion (immat VARCHAR2(6), typeAvion VARCHAR2(15), nbhVol NUMBER(10,2), proprio VARCHAR2(4), CONSTRAINT fk_Avion_comp_Compag FOREIGN KEY(proprio) REFERENCES Compagnie(comp) **DEFERRABLE INITIALLY IMMEDIATE**, CONSTRAINT pk_Avion PRIMARY KEY(immat));
-- fils sans père INSERT INTO Avion VALUES ('F-WTSS', 'Concorde', 6570, **'SING'**); 1 ligne créée. -- Problème à la validation **COMMIT**; ORA-02091: transaction annulée ORA-02291: violation de contrainte (SOUTOU.FK_AVION_COMP_COMPAG) d'intégrité - touche parent introuvable	-- fils sans père INSERT INTO Avion VALUES ('F-WTSS', 'Concorde', 6570, **'SING'**); ORA-02091: transaction annulée ORA-02291: violation de contrainte (SOUTOU.FK_AVION_COMP_COMPAG) d'intégrité - touche parent introuvable
	-- Modification du mode **SET CONSTRAINT** fk_Avion_comp_Compag **DEFERRED**; -- fils sans père INSERT INTO Avion VALUES ('F-WTSS', 'Concorde', 6570, **'SING'**); 1 ligne créée. -- Même problème au COMMIT

Instructions SET CONSTRAINT

Pour modifier une ou toutes les contraintes DEFERRABLE dans une transaction, il faut utiliser une des instructions de type SET CONSTRAINT(S). La syntaxe générale de cette instruction est la suivante :

```
SET { CONSTRAINT | CONSTRAINTS }
    { nomContrainte1 [,nomContrainte2]… | ALL }
    { IMMEDIATE | DEFERRED };
```

- L'option ALL place toutes les contraintes DEFERRABLE du schéma courant dans le mode spécifié dans la suite de l'instruction.
- L'option IMMEDIATE place la ou les contraintes du schéma courant en mode immédiat.
- L'option DEFERRED place la ou les contraintes du schéma courant en mode différé.

Instruction ALTER SESSION SET CONSTRAINTS

Pour modifier une ou plusieurs contraintes DEFERRABLE dans une session (suite de transactions), il faut utiliser l'instruction ALTER SESSION SET CONSTRAINTS. La syntaxe de cette instruction est la suivante :

```
ALTER SESSION SET CONSTRAINTS = { IMMEDIATE | DEFERRED | DEFAULT }
```

- L'option IMMEDIATE place toutes les contraintes du schéma courant en mode immédiat.
- L'option DEFERRED place toutes les contraintes du schéma courant en mode différé.
- DEFAULT remet les contraintes du schéma dans le mode qu'elles avaient lors de leur définition (DEFERRED ou IMMEDIATE) dans les instructions CREATE TABLE ou ALTER TABLE.

Directives VALIDATE et NOVALIDATE

Depuis la version 8*i*, les contraintes peuvent être actives alors que certaines données contenues dans les tables ne les vérifient pas. Ce mécanisme est rendu possible par l'utilisation des directives VALIDATE et NOVALIDATE.

VALIDATE et NOVALIDATE peuvent se combiner aux directives ENABLE et DISABLE précédemment étudiées dans les instructions CREATE TABLE et ALTER TABLE.

Les directives de validation ont la signification suivante :

- ENABLE vérifie les mises à jour à venir (insertions et nouvelles modifications de la table) ;
- DISABLE autorise toute mise à jour ;
- VALIDATE vérifie que les données courantes de la table respectent la contrainte ;

- NOVALIDATE permet que certaines données présentes dans la table ne respectent pas la contrainte.

Quelques remarques :

- ENABLE VALIDATE est semblable à ENABLE, la contrainte est vérifiée et certifie qu'elle sera respectée pour les enregistrements présents.
- DISABLE NOVALIDATE est semblable à DISABLE, la contrainte n'est plus vérifiée et ne garantit pas les enregistrements présents.
- ENABLE NOVALIDATE signifie que la contrainte est vérifiée, mais elle peut ne pas assurer tous les enregistrements. Cela permet de conserver des données anciennes qui ne vérifient plus la contrainte tout en la respectant pour les mises à jour ultérieures.
- DISABLE VALIDATE désactive la contrainte, supprime les index éventuels tout en préservant le respect de la contrainte pour les enregistrements présents.

Étudions dans le tableau suivant ces deux derniers cas :

- L'exemple avec ENABLE NOVALIDATE souligne le fait qu'on peut avoir une contrainte active tout en ayant des données ne la respectant plus.
- L'exemple avec DISABLE VALIDATE illustre la situation où on ne peut pas désactiver la contrainte (des données ne la respectant pas sont encore présentes dans la table). Pour résoudre ce problème, il faut extraire les enregistrements en réactivant la contrainte avec l'option EXCEPTIONS INTO… et les traiter au cas par cas.

Tableau 3-8 VALIDATE et NOVALIDATE

ENABLE NOVALIDATE	DISABLE VALIDATE
CREATE TABLE Compagnie (comp CHAR(4), nrue NUMBER(3), rue CHAR(20), ville CHAR(15), nomComp CHAR(15), CONSTRAINT pk_Compagnie PRIMARY KEY(comp));	
CREATE TABLE Avion (immat CHAR(6), typeAvion CHAR(15), proprio CHAR(4), CONSTRAINT fk_Avion_proprio_Compag FOREIGN KEY(proprio) REFERENCES Compagnie(comp)**DISABLE**, CONSTRAINT pk_Avion PRIMARY KEY(immat));	CREATE TABLE Avion (immat CHAR(6), typeAvion CHAR(15), proprio CHAR(4), CONSTRAINT fk_Avion_proprio_Compag FOREIGN KEY(proprio) REFERENCES Compagnie(comp), CONSTRAINT pk_Avion PRIMARY KEY(immat));
INSERT INTO Compagnie VALUES ('SING', 7, 'Camparols', 'Singapour', 'Singapore AL');	
INSERT INTO Avion VALUES ('F-WTSS', 'Concorde', 'Toto'); ALTER TABLE Avion **ENABLE NOVALIDATE** CONSTRAINT fk_Avion_proprio_Compag;	INSERT INTO Avion VALUES ('F-WTSS', 'Concorde', 'SING'); ALTER TABLE Avion DISABLE CONSTRAINT fk_Avion_proprio_Compag;

Tableau 3-8 VALIDATE et NOVALIDATE *(suite)*

ENABLE NOVALIDATE	DISABLE VALIDATE
--interdit INSERT INTO Avion VALUES ('F-TB20', 'Concorde', **'Toto'**); ORA-02291: violation de contrainte (SOU- TOU.FK_AVION_PROPRIO_COMPAG) d'intégrité - touche parent introuvable --possible INSERT INTO Avion VALUES ('F-ABCD', 'Concorde', **'SING'**); --Table Avion Immat TypeAvion Proprio ------ --------------- ------- F-WTSS Concorde Toto F-ABCD Concorde SING	--permis INSERT INTO Avion VALUES ('F-TB20', 'Concorde', **'Toto'**); --interdit (dernier avion ne vérifie pas) ALTER TABLE Avion **DISABLE VALIDATE** CONSTRAINT fk_Avion_proprio_Compag; ORA-02298: impossible de valider (SOU- TOU.FK_AVION_PROPRIO_COMPAG) - clés parents introuvables --Table Avion Immat TypeAvion Proprio ------ --------------- ------- F-WTSS Concorde SING F-TB20 Concorde Toto

Directive MODIFY CONSTRAINT

Il est possible de modifier le mode d'une contrainte en utilisant la directive `MODIFY CONSTRAINT` de la commande `ALTER TABLE`. La modification concerne les options suivantes :

- `DEFERRABLE` ou `NOT DEFERRABLE` ;
- `INITIALLY DEFERRED` ou `INITIALLY IMMEDIATE` ;
- `ENABLE` ou `DISABLE` ;
- `VALIDATE` ou `NOVALIDATE`.

L'exemple suivant déclare la table `Pilote` possédant trois contraintes. La troisième contrainte (clé primaire) adopte le mode par défaut (`NOT DEFERRABLE INITIALLY IMMEDIATE ENABLE VALIDATE`).

```
CREATE TABLE Pilote
   (brevet CHAR(6), nbHVol NUMBER(7,2), nom CHAR(30), CONSTRAINT
   nn_nom NOT NULL DEFERRABLE INITIALLY DEFERRED DISABLE VALIDATE,
   CONSTRAINT ck_nbHVol CHECK (nbHVol BETWEEN 0 AND 20000)
      DEFERRABLE INITIALLY IMMEDIATE ENABLE NOVALIDATE,
   CONSTRAINT pk_Pilote PRIMARY KEY (brevet));
```

Les instructions suivantes modifient tous les paramètres des deux premières contraintes :

```
ALTER TABLE Pilote
   MODIFY CONSTRAINT nn_nom INITIALLY IMMEDIATE ENABLE NOVALIDATE;
ALTER TABLE Pilote
   MODIFY CONSTRAINT ck_nbHVol INITIALLY DEFERRED DISABLE VALIDATE;
```

Exercices

Les objectifs de ces exercices sont :

- d'ajouter et de modifier des colonnes ;
- d'ajouter des contraintes ;
- de traiter les rejets.

Exercice 3.1 Ajout de colonnes

Écrivez le script `évolution.sql` qui contient les instructions nécessaires pour ajouter les colonnes suivantes (avec `ALTER TABLE`). Le contenu de ces colonnes sera modifié ultérieurement.

Tableau 3-9 Données de la table Installer

Table	Nom, type et signification des nouvelles colonnes
`Segment`	`nbSalle NUMBER(2)` : nombre de salles `nbPoste NUMBER(2)` : nombre de postes
`Logiciel`	`nbInstall NUMBER(2)` : nombre d'installations
`Poste`	`nbLog NUMBER(2)` : nombre de logiciels installés

Vérifier la structure et le contenu de chaque table avec `DESC` et `SELECT`.

Exercice 3.2 Modification de colonnes

Dans ce même script, ajoutez les instructions nécessaires pour :

- augmenter la taille dans la table `Salle` de la colonne `nomSalle` (passer à `VARCHAR2(30)`) ;
- diminuer la taille dans la table `Segment` de la colonne `nomSegment` à `VARCHAR2(15)` ;
- tenter de diminuer la taille dans la table `Segment` de la colonne `nomSegment` à `VARCHAR2(14)`. Pourquoi la commande n'est-elle pas possible ?

Vérifiez par `DESC` la nouvelle structure des deux tables.

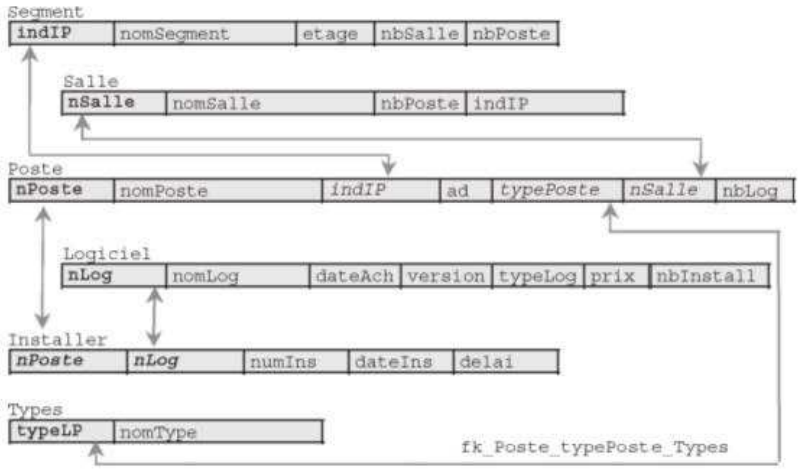
Vérifiez le contenu des tables :

```
SELECT * FROM Salle;
SELECT * FROM Segment;
```

Exercice 3.3 Ajout de contraintes

Ajoutez les contraintes de clés étrangères pour assurer l'intégrité référentielle entre les tables suivantes (avec `ALTER TABLE... ADD CONSTRAINT...`). Adoptez les conventions recommandées dans le chapitre 1 (comme indiqué pour la contrainte entre `Poste` et `Types`).

Figure 3-12 Contraintes référentielles à créer



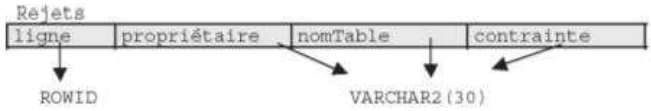
Si l'ajout d'une contrainte référentielle renvoie une erreur, vérifier les enregistrements des tables « pères » et « fils » (notamment au niveau de la casse des chaînes de caractères, 'Tx' est différent de 'TX' par exemple).

Modifiez le script SQL de destruction des tables (`dropParc.sql`) en fonction des nouvelles contraintes. Lancer ce script puis tous ceux écrits jusqu'ici.

Exercice 3.4 Traitements des rejets

Créez la table `Rejets` avec la structure suivante (ne pas mettre de clé primaire) :

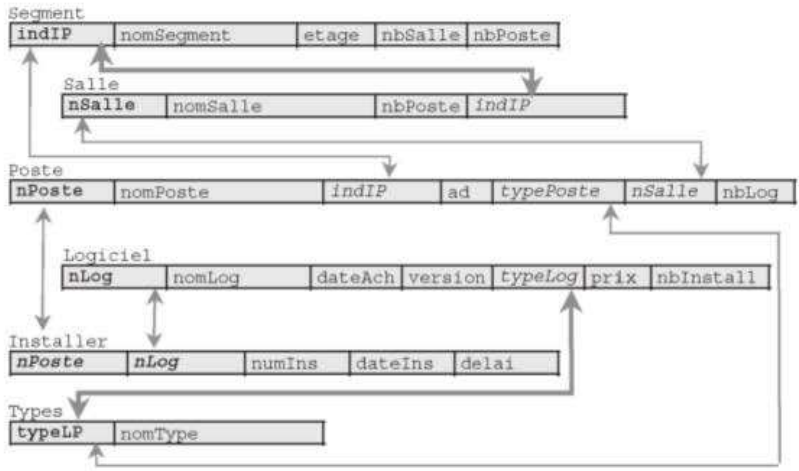
Figure 3-13 Table des rejets (exceptions)



Cette table permettra de retrouver les enregistrements qui ne vérifient pas de contraintes lors de la réactivation.

Ajoutez les contraintes de clés étrangères entre les tables `Salle` et `Segment` et entre `Logiciel` et `Types` (en gras dans le schéma suivant). Utilisez la directive `EXCEPTIONS INTO` pour récupérer des informations sur les erreurs.

Figure 3-14 *Contraintes référentielles à créer*



La création de ces contraintes doit renvoyer une erreur car :

- il existe des salles ('s22' et 's23') ayant un numéro de segment qui n'est pas référencé dans la table `Segment` ;
- il existe un logiciel ('log8') dont le type n'est pas référencé dans la table `Types`.

Vérifiez dans la table `Rejets` les enregistrements qui posent problème. Vérifier la correspondance avec les `ROWID` des tables `Salle` et `Logiciel` :

```
SELECT * FROM Rejets;
SELECT ROWID,s.* FROM Salle s
       WHERE ROWID IN (SELECT ligne FROM Rejets);
SELECT ROWID,l.* FROM Logiciel l
       WHERE ROWID IN (SELECT ligne FROM Rejets);
```

Supprimez les enregistrements de la table `Rejets`.

Supprimez les enregistrements de la table `Salle` qui posent problème. Ajouter le type de logiciel ('`BeOS`', '`Système Be`') dans la table `Types`.

Exécutez à nouveau l'ajout des deux contraintes de clé étrangère. Vérifier que les instructions ne renvoient plus d'erreur et que la table `Rejets` reste vide.

Exercice 3.5 **Ajout de colonnes dans la base *Chantiers***

Écrivez le script `évolChantier.sql` qui modifie la base *Chantiers* afin de pouvoir stocker :

- la capacité en nombre de places de chaque véhicule ;
- la liste des types de véhicule interdits de visite concernant certains chantiers ;
- la liste des employés autorisés à conduire certains types de véhicule ;
- le temps de trajet pour chaque visite (basé sur une vitesse moyenne de 40 kilomètres par heure). Vous utiliserez une colonne virtuelle.

Vérifiez la structure de chaque table avec `DESC`.

Exercice 3.6 **Mise à jour de la base *Chantiers***

Écrivez le script `majChantier.sql` qui met à jour les nouvelles colonnes de la base *Chantiers* de la manière suivante :

- affectation automatique du nombre de places disponibles pour chaque véhicule (1 pour les motos, 3 pour les voitures et 6 pour les camionnettes) ;
- déclaration d'un chantier inaccessible pour une camionnette et d'un autre inaccessible aux motos ;
- déclaration de diverses autorisations pour chaque conducteur (affecter toutes les autorisations à un seul conducteur).

Vérifiez le contenu de chaque table (et de la colonne virtuelle) avec `SELECT`.

Chapitre 4

Interrogation des données

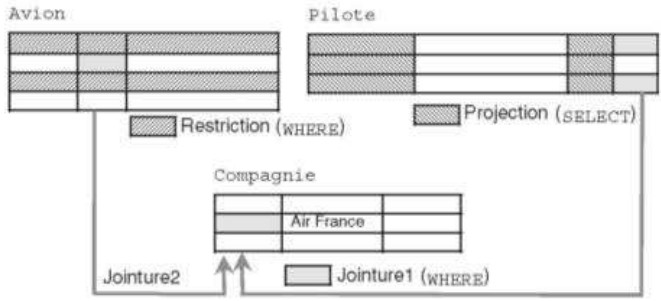
Ce chapitre traite de l'aspect le plus connu du langage SQL à savoir l'extraction des données par requêtes (nom donné aux instructions `SELECT`). Une requête permet de rechercher des données dans une ou plusieurs tables ou vues à partir de critères simples ou complexes. Les instructions `SELECT` peuvent être exécutées dans l'interface SQL*Plus (voir les exemples de ce chapitre) ou au sein d'un programme PL/SQL, Java, C, etc.

Généralités

L'instruction `SELECT` est une commande déclarative (décrit ce que l'on cherche sans décrire le moyen de le réaliser). À l'inverse, une instruction procédurale (comme un programme) développerait le moyen de réaliser l'extraction de données (comme le chemin à emprunter entre tables ou une itération pour parcourir un ensemble d'enregistrements).

La figure suivante schématise les principales fonctionnalités de l'instruction `SELECT`. Celle-ci est composée d'une directive `FROM` qui précise la (les) table(s) interrogée(s) et d'une directive `WHERE` qui contient les critères.

Figure 4-1 *Possibilités de l'instruction SELECT*



- La restriction qui est programmée dans le WHERE de la requête permet de restreindre la recherche à une ou plusieurs lignes. Dans notre exemple, une restriction répond à la question « Quels sont les avions de type 'A320' ? » ;
- La projection qui est programmée dans le SELECT de la requête permet d'extraire une ou plusieurs colonnes. Dans notre exemple, elle répond à la question « Quels sont les numéros de brevet et nombres d'heures de vol de tous les pilotes ? » ;
- La jointure qui est programmée dans le WHERE de la requête permet d'extraire des données de différentes tables en les reliant deux à deux (le plus souvent à partir de contraintes référentielles). Dans notre exemple, la première jointure répond à la question « Quels sont les numéros de brevet et nombres d'heures de vol des pilotes de la compagnie de nom Air France ? » La deuxième jointure répond à la question « Quels sont les avions de la compagnie de nom Air France ? »

En combinant ces trois fonctionnalités, toute question logique devrait trouver en théorie une réponse par une ou plusieurs requêtes. Les questions trop complexes peuvent être programmées à l'aide des vues (chapitre 5) ou par traitement (PL/SQL mélangeant requêtes et instructions procédurales).

Syntaxe (SELECT)

Pour pouvoir extraire des enregistrements d'une table, il faut avoir reçu le privilège SELECT sur la table. Le privilège SELECT ANY TABLE permet d'extraire des données dans toute table de tout schéma.

La syntaxe SQL simplifiée de l'instruction SELECT est la suivante :

```
SELECT [ { DISTINCT | UNIQUE } | ALL ] {listeColonnes | expression}
   FROM nomTable1 [,nomTable2]…
   [ WHERE condition ]
   [ clauseHiérarchique ]
   [ clauseRegroupement ]
   [ HAVING condition ]
   [ { UNION | UNION ALL | INTERSECT | MINUS } ( sousRequête )]
   [ clauseOrdonnancement ] ;
```

Au cours de ce chapitre, nous détaillerons chaque option à l'aide d'exemples.

Pseudo-table DUAL

La table DUAL est une table utilisable par tous (en lecture seulement) et qui appartient à l'utilisateur SYS. Le paradoxe de DUAL réside dans le fait qu'elle est couramment sollicitée, mais les interrogations ne portent jamais sur sa seule colonne (DUMMY définie en VARCHAR2 et contenant un seul enregistrement avec la valeur X). En conséquence, DUAL est qualifiée de pseudo-table (c'est la seule qui soit ainsi composée).

L'interrogation de DUAL est utile pour évaluer une *expression* de la manière suivante : « SELECT expression FROM DUAL » (seule l'instruction SELECT est permise sur DUAL). Comme DUAL n'a qu'un seul enregistrement, les résultats fournis seront uniques (si aucune jointure ou opérateur ensembliste ne sont utilisés dans l'interrogation).

Tableau 4-1 Utilisations de DUAL

Besoin	Requête et résultat sous SQL*Plus

Aucun, utilisation probablement la plus superflue de DUAL.	`SELECT 'Il reste encore beaucoup de pages?'`
	`  FROM DUAL;`
	
	`'ILRESTEENCOREBEAUCOUPDEPAGES?'`
	`----------`
	`Il reste encore beaucoup de pages?`

J'ai oublié ma montre !	`SELECT TO_CHAR(SYSDATE, 'DD, MONTH YYYY, HH24:MI:SS')`
	`  "Maintenant : " FROM DUAL;`
	
	`Maintenant :`
	`----------`
	`12, MAI  2003, 00:13:39`

Pour les matheux qui voudraient retrouver le résultat de 2^{14}, le carré du cosinus de $3\pi/2$ et e^1.	`SELECT POWER(2,14), POWER(COS(135*3.14159265359/180),2),`
	`EXP(1) FROM DUAL;`
	
	`POWER(2,14) POWER(COS(135*3.14159265359/180),2)  EXP(1)`
	`---------- ---------- -------`
	`16384  ,5  2,71828183`

Projection (éléments du SELECT)

Étudions la partie de l'instruction SELECT qui permet de programmer l'opérateur de projection (en surligné dans la syntaxe suivante) :

```
SELECT [ { DISTINCT | UNIQUE } | ALL ] listeColonnes
 FROM nomTable [aliasTable]
[ clauseOrdonnancement ] ;
```

- DISTINCT et UNIQUE jouent le même rôle : ne pas prendre en compte les duplicatas.
- ALL : prend en compte les duplicatas (option par défaut).
- *ListeColonnes* : { * | *expression1* [[AS] *alias1*] [, *expression2* [[AS] *alias2*]…}.
- * : extrait toutes les colonnes de la table.
- *expression* : nom de colonne, fonction, constante ou calcul.
- *alias* : renomme l'expression (nom valable pendant la durée de la requête).

- `FROM` : désigne la table (qui porte un alias ou non) à interroger.
- *`clauseOrdonnancement`* : tri sur une ou plusieurs colonnes ou expressions.

Interrogeons la table suivante en utilisant les principales options à l'aide d'exemples divers.

Figure 4-2 *Table Pilote*

PILOTE

BREVET	PRENOM	NOM	NBHVOL	COMPA
PL-1	Benoit	Sarda	4500	AF
PL-2	Aime	Giaconne	2000	AF
PL-3	Pierre	Calac	1500	SING
PL-4	Jean Phi	Ferrage	2450	CAST
PL-5	Jean	Gazagnes	(null)	SING
PL-6	Arnaud	Sayag	2450	AF

Extraction de toutes les colonnes

L'extraction de toutes les colonnes d'une table est rendue possible par l'utilisation du caractère *. L'ordre des lignes retournées, dans le cas des tables classiques (en tas : *heap*), suit le séquencement des blocs de données dans lesquels se trouvent les lignes (l'ordre chronologique des insertions est en premier lieu suivi).

Tableau 4-2 Utilisation de *

Requête SQL	Résultat sous SQL*Plus
`SELECT * FROM Pilote ;`	BREVET PRENOM NOM NBHVOL COMPA ------ ---------- ---------- ------- ----- PL-1 Benoit Sarda 4500 AF PL-2 Aime Giaconne 2000 AF PL-3 Pierre Calac 1500 SING PL-4 Jean Phi Ferrage 2450 CAST PL-5 Jean Gazagnes SING PL-6 Arnaud Sayag 2450 AF

Pratique au premier abord, limitez au maximum l'utilisation du caractère *, car vous risquez de solliciter le réseau en interrogeant un nombre inconnu de colonnes et le SGBD qui devra dynamiquement extraire la liste des colonnes de la table interrogée. De plus, vous considérez la structure du jeu de résultats comme étant constante alors que des colonnes de la table peuvent apparaître ou disparaître entre deux exécutions.

Extraction de certaines colonnes

La liste des colonnes à extraire se trouve dans la clause SELECT.

Tableau 4-3 Liste de colonnes

Requête SQL	Résultat sous SQL*Plus

```
SELECT compa, brevet FROM Pilote;
```

```
COMPA BREVET
----- ------
AF    PL-1
AF    PL-2
SING  PL-3
CAST  PL-4
SING  PL-5
AF    PL-6
```

Alias

Les alias permettent de renommer des colonnes à l'affichage ou des tables dans la requête. Les alias de colonnes sont utiles pour les calculs.

- Oracle traduit les noms des alias en majuscules (valable aussi pour les expressions, colonnes, vues, tables, etc.).
- L'utilisation de la directive AS est facultative (pour se rendre conforme à SQL2).
- Il faut préfixer les colonnes par l'alias de la table lorsqu'il existe.

Tableau 4-4 Alias (colonnes et tables)

Alias de colonnes

```
SELECT compa AS compagnie, nom AS name,
       brevet num_pilote
FROM   Pilote;
```

```
COMP NAME                 NUM_PILOTE
---- -------------------- ----------
AF   Sarda                PL-1
AF   Giaconne             PL-2
SING Calac                PL-3
CAST Ferrage              PL-4
SING Gazagnes             PL-5
AF   Sayag                PL-6
```

Alias de table

```
SELECT a_pilotes.compa AS compagnie,
a_pilotes.nom
FROM   Pilote a_pilotes;
```

```
COMP NOM
---- ---------------
AF   Sarda
AF   Giaconne
SING Calac
CAST Ferrage
SING Gazagnes
AF   Sayag
```

Duplicatas

La directive DISTINCT (UNIQUE est un synonyme) élimine tout doublon de valeur. La première requête filtre les doublons au niveau du code compagnie. La deuxième filtre les doublons au niveau d'un couple de valeurs. Ici, toutes les lignes sont retournées car aucun pilote n'est identique au niveau de ces deux colonnes.

Tableau 4-5 Gestion des duplicatas

Utilisation	Requêtes SQL et résultats

Liste des compagnies (sans doublon)	`SQL> SELECT DISTINCT compa FROM Pilote;`
	
	`COMPA`
	`-----`
	`AF`
	`CAST`
	`SING`

Élimination de doublon de couples	`SQL> SELECT UNIQUE nbhvol,compa FROM Pilote;`
	
	`NBHVOL COMPA`
	`---------- -----`
	`2450 CAST`
	`4500 AF`
	`2000 AF`
	`1500 SING`
	`2450 AF`
	`SING`

La gestion des NULL par la directive DISTINCT ou UNIQUE est identique à celle des traitements ensemblistes (voir la section « Les opérateurs ensemblistes ») à savoir que deux NULL sont considérés identiques.

Expressions

Il est possible d'évaluer des expressions numériques ou alphanumériques (incluant des fonctions de dates) dans la clause SELECT.

Le résultat d'une expression numérique incluant un NULL retourne un NULL. Si vous désirez modifier ce comportement par défaut, vous devrez transformer un NULL en une valeur choisie (0 ou 1 par exemple) avec la fonction NVL.

Le résultat d'une expression alphanumérique incluant un NULL ne retourne pas systématiquement un NULL.

L'exemple suivant présente deux expressions incluant un NULL ; l'expression numérique est impactée alors que la concaténation ne l'est pas. L'opérateur de concaténation d'Oracle (||) accepte différentes expressions (colonnes, calculs, fonctions, etc.) et retourne une chaîne de caractères.

Tableau 4-6 Expressions numériques

Requête	Résultat sous SQL*Plus

`SELECT brevet,`
`nbHVol*1.75 AS prime,`
`nbHVol||nom AS heure_nom`
`FROM Pilote;`
	`BREVET PRIME HEURE_NOM`
	`------ ---------- --------------`
	`PL-1 7875 4500Sarda`
	`PL-2 3500 2000Giaconne`
	`PL-3 2625 1500Calac`
	`PL-4 4287,5 2450Ferrage`
	`PL-5 Gazagnes`
	`PL-6 4287,5 2450Sayag`

Ordonnancement

Pour trier le résultat d'une requête, il faut spécifier la clause d'ordonnancement par ORDER BY de la manière suivante :

```
ORDER [SIBLINGS] BY
{ expression1 | position1 | alias1 } [ASC | DESC] [ NULLS FIRST |
NULLS LAST ]
[,{expression2 | position2 | alias2} [ASC | DESC] [NULLS FIRST | NULLS
LAST]]…
```

- SIBLINGS : relatif aux requêtes hiérarchiques, couplé au CONNECT BY (étudié en fin de chapitre).
- *expression* : nom de colonne, fonction, constante, calcul.
- *position* : entier qui désigne l'expression (au lieu de la nommer) dans son ordre d'apparition dans la clause SELECT.
- ASC ou DESC : tri ascendant ou descendant (par défaut ASC).
- NULLS FIRST ou NULLS LAST : position des valeurs nulles (au début ou à la fin du résultat). NULLS LAST par défaut pour l'option ASC, NULLS FIRST par défaut pour l'option DESC.

Tableau 4-7 Ordonnancement

Options par défaut	Option sur les valeurs NULL
`SELECT brevet, nom FROM Pilote` `  ORDER BY nom;`	`SELECT brevet,nbHVol FROM Pilote` `  ORDER BY nbHvol ASC NULLS FIRST;`

`BREVET NOM`
`------ ----------------`
`PL-5  Daniel Vielle`
`PL-2  Didier Donsez`
`PL-1  Gratien Viel`
`PL-4  Placide Fresnais`
`PL-3  Richard Grin`
	`BREVET  NBHVOL`
	`------ ----------`
	`PL-5`
	`PL-2  0`
	`PL-1  450`
	`PL-3  1000`
	`PL-4  2450`

Substitutions conditionnelles

Deux structures permettent de conditionner une expression : CASE (propre à Oracle et étudiée au chapitre 6) et DECODE (la fonction SQL normative). À l'aide de ce mécanisme, si la valeur de l'expression est identique à la valeur testée, un résultat prévu peut être retourné. Si aucune correspondance n'est trouvée, une éventuelle valeur par défaut peut être retournée (sinon NULL). La requête suivante permet de substituer des libellés à des codes.

Tableau 4-8 Structures conditionnelles

Requête	Résultat

`SELECT nom, DECODE(compa,`
`  'AF', 'Air France',`
`  'SING', 'Singapore Air',`
`  'CAST', 'Trans Casta',`
`  'Autre ou aucune')`
`  AS compagnie`
`FROM Pilote`
`ORDER BY nom;`
	`NOM  COMPAGNIE`
	`-------------------- -------------`
	`Calac  Singapore Air`
	`Ferrage  Trans Casta`
	`Gazagnes  Singapore Air`
	`Giaconne  Air France`
	`Sarda  Air France`
	`Sayag  Air France`

Pseudo-colonne ROWID

Le format du *rowid* de chaque enregistrement inclut le numéro de l'objet, le numéro relatif du fichier, le numéro du bloc dans le fichier et le déplacement dans le bloc. Le mot-clé qui désigne cette pseudo-colonne non modifiable (mais accessible) est ROWID.

Tableau 4-9 Affichage de plusieurs ROWID

Requête	Résultat sous SQL*Plus
SELECT **ROWID**, brevet, nom FROM Pilote;	**ROWID** BREVET NOM ------------------ ------ ---------- AAAIaVAAJAAAAAOAAA PL-1 Gratien Viel AAAIaVAAJAAAAAOAAB PL-2 Didier Donsez AAAIaVAAJAAAAAOAAC PL-3 Richard Grin AAAIaVAAJAAAAAOAAD PL-4 Placide Fresnais AAAIaVAAJAAAAAOAAE PL-5 Daniel Vielle

Pseudo-colonne ROWNUM

La pseudo-colonne ROWNUM retourne un entier indiquant l'ordre séquentiel de chaque enregistrement extrait par la requête. Le premier possède implicitement une colonne ROWNUM évaluée à 1, pour le deuxième elle l'est à 2, etc.

Tableau 4-10 Affichage de ROWNUM

Requête	Résultat sous SQL*Plus
SELECT **ROWNUM**, brevet, nom FROM Pilote;	**ROWNUM** BREVET NOM ---------- ------ ---------- 1 PL-1 Gratien Viel 2 PL-2 Didier Donsez 3 PL-3 Richard Grin 4 PL-4 Placide Fresnais 5 PL-5 Daniel Vielle

Vous pouvez facilement utiliser cette pseudo-colonne pour limiter le nombre de lignes extraites (WHERE ROWNUM<=n) si aucun tri (ORDER BY) n'est associé à votre requête. En revanche, si vous triez également le résultat tout en désirant extraire les *n* premières lignes (ou les *n* dernières si le tri est descendant), vous devrez utiliser une sous-requête (voir la section « Sous-interrogations dans la clause FROM »). Depuis la version 12*c*, la clause FETCH permet de limiter le nombre de lignes plus facilement (voir plus loin).

Insertion multiligne

Nous pouvons maintenant décrire l'insertion multiligne évoquée au chapitre précédent. Dans l'exemple suivant, il s'agit d'insérer tous les pilotes de la table Pilote (en considérant le nom, le nombre d'heures de vol et la compagnie) dans la table NomsetHVoldesPilotes. La requête extrait des nouveaux *rowids* car il s'agit d'enregistrements différents de ceux contenus dans la table source.

Notez que les instructions (CREATE TABLE et INSERT…) peuvent être remplacées par une unique instruction (option AS SELECT de la commande CREATE TABLE) comme le montre la ligne suivante :

```
CREATE TABLE NomsetHVoldesPilotes AS SELECT nom, nbHVol, compa FROM
Pilote;
```

Tableau 4-11 Insertion multiligne

Création et insertion	Requête sous SQL*Plus

`CREATE TABLE NomsetHVoldesPilotes (nom VARCHAR(16), nbHVol NUMBER(7,2), compa CHAR(4));`
`INSERT INTO NomsetHVoldesPilotes SELECT nom, nbHVol, compa FROM Pilote;`
	`SELECT ROWID, p.* FROM NomsetHVoldesPilotes p;`
	
	`ROWID NOM NBHVOL COMPA`
	`AAAIaaAAJAAAAAmAAA Gratien Viel 450 AF`
	`AAAIaaAAJAAAAAmAAB Didier Donsez 0 AF`
	`AAAIaaAAJAAAAAmAAC Richard Grin 1000 SING`
	`AAAIaaAAJAAAAAmAAD Placide Fresnais 2450 CAST`
	`AAAIaaAAJAAAAAmAAE Daniel Vielle AF`

Limitation du nombre de lignes

Depuis la version 12c, la clause FETCH (éventuellement précédée de OFFSET) vous permettra de limiter le nombre de lignes d'un résultat (avec ou sans ex æquo si vous appliquez aussi un tri). On trouve l'équivalent de cette fonctionnalité dans SQL Server (avec TOP) et dans MySQL (avec LIMIT), dans les instructions CREATE TABLE et ALTER TABLE.

Dans la syntaxe suivante que vous devez utiliser à la fin de votre requête (après le ORDER BY), les termes ROW ou ROWS sont équivalents, de même que FIRST ou NEXT (encore une querelle de langage SQL entre Oracle et la norme).

```
[ OFFSET nb_lignes_a_sauter { ROW | ROWS } ]
[ FETCH { FIRST | NEXT }
        [ { nb_lignes_a_inclure | pourcentage PERCENT } ]
    { ROW | ROWS } { ONLY | WITH TIES } ]
```

- OFFSET permet d'ignorer un certain nombre de lignes en amont.
- PERCENT permet de raisonner en pourcentage de lignes plutôt qu'en nombre.
- WITH TIES inclut les éventuels ex æquo après un tri.

Le tableau 4-12 présente trois extractions avec ces différentes options.

Tableau 4-12 Limitation de lignes

Requêtes	Résultats

Les deux pilotes les mieux payés (triés par ordre alphabétique), en se limitant à deux même s'il existe des salaires identiques...	`SELECT brevet, prenom, nom, nbhvol`
	`FROM Pilote`
	`WHERE nbhvol IS NOT NULL`
	`ORDER BY nbhvol DESC`
	`FETCH FIRST 2 ROWS ONLY;`
	
	`BREVET PRENOM NOM NBHVOL`
	`------ ---------- --------------- -------`
	`PL-1 Benoit Sarda 4500`
	`PL-4 Jean Phi Ferrage 2450`

Même besoin en considérant les ex aequo.	`...`
	`FETCH NEXT 2 ROWS WITH TIES;`
	
	`BREVET PRENOM NOM NBHVOL`
	`------ ---------- --------------- -------`
	`PL-1 Benoit Sarda 4500`
	`PL-4 Jean Phi Ferrage 2450`
	`PL-6 Arnaud Sayag 2450`

La moitié de la population des pilotes, triés par ordre alphabétique.	`SELECT brevet, prenom, nom, compa`
	`FROM Pilote`
	`ORDER BY nom ASC`
	`FETCH FIRST 50 PERCENT ROWS ONLY;`
	
	`BREVET PRENOM NOM COMPA`
	`------ ---------- --------------- -----`
	`PL-3 Pierre Calac SING`
	`PL-4 Jean Phi Ferrage CAST`
	`PL-5 Jean Gazagnes SING`

Vous ne pouvez pas bénéficier de cette fonctionnalité dans un `SELECT ... FOR UPDATE` ou une requête définissant une vue matérialisée (voir le chapitre 12). De même, aucune séquence avec les pseudo-colonnes `CURRVAL` et `NEXTVAL` ne peut être utilisée dans une requête incluant la clause `FETCH`.

Restriction (WHERE)

Les éléments de la clause `WHERE` d'une requête permettent de programmer l'opérateur de restriction. Cette clause limite la recherche aux enregistrements qui respectent une condition simple ou complexe. Cette section s'intéresse à la partie surlignée de l'instruction `SELECT` suivante :

```
SELECT [ { DISTINCT | UNIQUE } | ALL ] { listeColonnes | expression }
 FROM nomTable [aliasTable]
 [ WHERE condition ] ;
```

- *condition* : est composée de colonnes, d'expressions, de constantes liées deux à deux entre des opérateurs :
  - de comparaison (>, =, <, >=, <=, <>) ;
  - logiques (NOT, AND ou OR) ;
  - intégrés (BETWEEN, IN, LIKE, IS NULL).

Interrogeons la table suivante en utilisant chaque type d'opérateur :

Figure 4-3 *Table Pilote*

Pilote

brevet	nom	nbHVol	prime	compa
PL-1	Gratien Viel	450	500	AF
PL-2	Didier Donsez	0		AF
PL-3	Richard Grin	1000	90	SING
PL-4	Placide Fresnais	2450	500	CAST
PL-5	Daniel Vielle	400	600	SING
PL-6	Francoise Tort		0	CAST

Opérateurs de comparaison

Le tableau suivant décrit des requêtes pour lesquelles la clause WHERE contient des opérateurs de comparaison.

Les écritures « prime=500 » et « (prime=500) » sont équivalentes. Les écritures « prime<>500 » et « NOT (prime=500) » sont équivalentes. Les parenthèses sont utiles pour composer des conditions.

Notez l'utilisation du simple guillemet pour comparer des chaînes de caractères.

Tableau 4-13 Égalité, inégalité et comparaison

Égalité	Comparaison et inégalité

`SELECT brevet, nom AS "Prime 500"`
`FROM Pilote WHERE prime = 500;`
`BREVET Prime 500`
`------ ----------------`
`PL-1 Gratien Viel`
`PL-4 Placide Fresnais`
	`SELECT brevet, nom, prime`
	`FROM Pilote WHERE prime <= 400;`
	`BREVET NOM PRIME`
	`------ ---------- ----------`
	`PL-3 Richard Grin 90`
	`PL-6 Francoise Tort 0`

`SELECT brevet, nom "de Air-France"`
`FROM Pilote WHERE compa = 'AF';`
`BREVET de Air-France`
`------ ----------------`
`PL-1 Gratien Viel`
`PL-2 Didier Donsez`
	`SELECT brevet, nom, prime FROM Pilote`
	`WHERE prime <> 500;`
	`BREVET NOM PRIME`
	`------ ---------- ----------`
	`PL-3 Richard Grin 90`
	`PL-5 Daniel Vielle 600`
	`PL-6 Francoise Tort 0`

Opérateurs logiques

- L'ordre de priorité des opérateurs logiques est NOT, AND et OR.
- Les opérateurs de comparaison (>, =, <, >=, <=, <> et !=) sont prioritaires par rapport à NOT.
- Les parenthèses permettent de modifier les règles de priorité.

La première requête de l'exemple suivant contient une condition composée de trois prédicats qui sont évalués par ordre de priorité (d'abord AND puis OR). La conséquence est l'affichage des pilotes de la compagnie 'SING' avec les pilotes de 'AF' ayant moins de 500 heures de vol.

La deuxième requête force la priorité avec les parenthèses (AND et OR sur le même pied d'égalité). La conséquence est l'affichage des pilotes ayant moins de 500 heures de vol des compagnies 'SING' et 'AF'.

Tableau 4-14 Opérateurs logiques

Requête	Résultat sous SQL*Plus

`SELECT brevet, nom, compa`
`FROM Pilote WHERE`
`(compa = 'SING' OR`
`compa = 'AF' AND nbHVol < 500 );`
	`BREVET NOM COMPA`
	`------ ---------- -----`
	`PL-1 Gratien Viel AF`
	`PL-2 Didier Donsez AF`
	`PL-3 Richard Grin SING`
	`PL-5 Daniel Vielle SING`

`SELECT brevet, nom, compa`
`FROM Pilote WHERE`
`((compa = 'SING' OR compa = 'AF')`
`AND nbHVol < 500);`
	`BREVET NOM COMPA`
	`------ ---------- -----`
	`PL-1 Gratien Viel AF`
	`PL-2 Didier Donsez AF`
	`PL-5 Daniel Vielle SING`

Opérateurs intégrés

Les opérateurs intégrés sont BETWEEN, IN, LIKE et IS NULL.

Tableau 4-15 Opérateurs intégrés

Opérateur	Exemple

BETWEEN *limiteInf* AND *limiteSup* teste l'appartenance à un intervalle de valeurs.	`SELECT brevet, nom, nbHVol FROM Pilote`
	`WHERE nbHVol BETWEEN 399 AND 1000;`
	
	`BREVET NOM NBHVOL`
	`------ --------------- ----------`
	`PL-1 Gratien Viel 450`
	`PL-3 Richard Grin 1000`
	`PL-5 Daniel Vielle 400`

IN *(listeValeurs)* compare une expression avec une liste de valeurs.	`SELECT brevet, nom, compa FROM Pilote`
	`WHERE compa IN ('CAST', 'SING');`
	
	`BREVET NOM COMPA`
	`------ --------------- -----`
	`PL-3 Richard Grin SING`
	`PL-4 Placide Fresnais CAST`
	`PL-5 Daniel Vielle SING`
	`PL-6 Francoise Tort CAST`

LIKE *(expression)* compare de manière générique des chaînes de caractères à une expression.
Le symbole % remplace un ou plusieurs caractères.
Le symbole _ remplace un caractère.
Ces symboles peuvent se combiner.
Utilisez de préférence des colonnes VARCHAR ou complétez si nécessaire par des blancs jusqu'à la taille maximale pour des CHAR.
	`SELECT brevet, nom, compa FROM Pilote`
	`WHERE compa LIKE ('%A%');`
	
	`BREVET NOM COMPA`
	`------ --------------- -----`
	`PL-1 Gratien Viel AF`
	`PL-2 Didier Donsez AF`
	`PL-4 Placide Fresnais CAST`
	`PL-6 Francoise Tort CAST`
	
	`SELECT brevet, nom, compa FROM Pilote`
	`WHERE compa LIKE ('A_');`
	
	`BREVET NOM COMPA`
	`------ --------------- -----`
	`PL-1 Gratien Viel AF`
	`PL-2 Didier Donsez AF`

IS NULL compare une expression (colonne, calcul, constante) à la valeur NULL.
La négation s'écrit soit « *expression* IS NOT NULL » soit « NOT (*expression* IS NULL) ».
	`SELECT nom, prime, nbHVol, compa`
	`FROM Pilote`
	`WHERE prime IS NULL OR nbHVol IS NULL;`
	
	`NOM PRIME NBHVOL COMPA`
	`--------------- --------- -------- -----`
	`Didier Donsez 0 AF`
	`Francoise Tort 0 CAST`

Fonctions

Oracle propose un grand nombre de fonctions qui s'appliquent dans les clauses SELECT ou WHERE d'une requête. La syntaxe générale d'une fonction est la suivante :

```
nomFonction(colonne1 | expression1 [,colonne2 | expression2 …])
```

- Une fonction monoligne agit sur une ligne à la fois et ramène un résultat par ligne. On distingue quatre familles de fonctions monolignes : caractères, numériques, dates et conversions de types de données. Ces fonctions peuvent se combiner entre elles (exemple : MAX(COS(ABS(n))) désigne le maximum des cosinus de la valeur absolue de la colonne n).
- Une fonction multiligne (fonction d'agrégat) agit sur un ensemble de lignes pour ramener un résultat (voir la section « Regroupements »).

Caractères

Interrogeons la table suivante en utilisant des fonctions pour les caractères :

Figure 4-4 *Table Pilote*

Pilote

brevet	prenom	nom	surnom	compa
PL-1	Gratien	viel	dba	AF
PL-2	Didier	donsez	smith	AF
PL-3	richard	Grin	Faucon	SING
PL-4	placide	Fresnais	cool	CAST
PL-5	Daniel	vielle	jones	SING
PL-6	Francoise	tort	NormaleSup	CAST

La plupart des fonctions pour les caractères acceptent une chaîne de caractères en paramètre de nature CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, ou NCLOB. Le tableau suivant décrit les principales fonctions :

Tableau 4-16 Fonctions pour les caractères

Fonction	Objectif	Exemple
`ASCII(c)`	Retourne le caractère ASCII équivalent.	`ASCII('A')` donne 65
`CHR(n)`	Retourne le caractère équivalent dans le jeu de caractères de la base ou du jeu national (NLS).	`CHR(161) || CHR(162)` donne ¡¢

`CONCAT(c1,c2)`	Concatène (équivalent à ||), est opérationnel pour les LOB.	`SELECT CONCAT( CONCAT(nom,' vole pour '), compa) "Personnel" FROM Pilote;`
		
		`Personnel`
		`----------`
		`viel travaille pour AF`
		`Grin travaille pour SING`
		…

`INITCAP(c)`	Première lettre de chaque mot en majuscule.	`SELECT INITCAP(prenom) "Prénom", INITCAP(nom) "Nom" FROM Pilote WHERE compa = 'SING';`
		
		`Prénom Nom`
		`---------- ----------`
		`Richard Grin`
		`Daniel Vielle`

`INSTR(c1, c2 [,p [,o]])`	Premier indice d'une sous-chaîne *c2* dans une chaîne *c1*.
	Exemple : indice du 2e 'Air' après le 9e caractère.
		`SELECT INSTR('Infos-Air : AirBus pour Air-France','Air', 9, 2) "Indice" FROM DUAL;`
		
		`Indice`
		`----------`
		`25`

`LOWER(c)`	Tout en minuscules.	`SELECT LOWER(prenom) || ' ' || LOWER(nom) "Etat civil" FROM Pilote WHERE compa = 'SING';`
		
		`Etat civil`
		`--------------------`
		`richard grin`
		`daniel vielle`

`LENGTH(c)`	Longueur de la chaîne.	`SELECT LENGTH('Infos-Air : AirBus pour Air-France') "Taille" FROM DUAL;`
		
		`Taille`
		`----------`
		`34`

Tableau 4-16 Fonctions pour les caractères *(suite)*

Fonction	Objectif	Exemple

LPAD(c1,n,c2)	Insertion à gauche de *c2* dans *c1* sur *n* caractères.	`SELECT LPAD('Rien',20,'-.-') "sur 20" FROM DUAL;`
		
		`sur 20`
		`--------------------`
		`-.--.--.--.--.--Rien`

LTRIM(c1,c2)	Enlève *c2* à *c1* en examinant la gauche de *c1*.	`SELECT LTRIM('B747B747A380 à Blagnac','B747') "Bye les Jumbo" FROM DUAL;`
		
		`Bye les Jumbo`
		`--------------`
		`A380 à Blagnac`

REPLACE(c1,c2,c3)	Recherche les *c2* présentes dans *c1* et les remplace par *c3*.	`SELECT REPLACE('Matra et Aerospatiale', 'Matra','EADS') "Changement" FROM DUAL;`
		
		`Changement`
		`--------------------`
		`EADS et Aerospatiale`

RPAD(c1,n,c2)	Insertion à droite de *c2* dans *c1* sur *n* caractères.	`SELECT RPAD('Rien',19,'-.-') "sur 19" FROM DUAL;`
		
		`sur 19`
		`-------------------`
		`Rien-.--.--.--.--.-`

RTRIM(c1,c2)	Enlève *c2* à *c1* en examinant la droite de *c1*.	`SELECT RTRIM('A380 à BlagnacB747B747','B747') "Bye les Jumbo" FROM DUAL;`
		
		`Bye les Jumbo`
		`--------------`
		`A380 à Blagnac`

SOUNDEX(c)	Extrait la phonétique d'une expression (*in english only !*).	`SELECT nom, surnom, compa FROM Pilote WHERE SOUNDEX(surnom) IN (SOUNDEX('SMYTHE'),SOUNDEX('John'));`
		
		`NOM SURNOM COMPA`
		`---------- ---------- -----`
		`donsez **smith** AF`
		`vielle **jones** SING`

Tableau 4-16 Fonctions pour les caractères *(suite)*

Fonction	Objectif	Exemple

`SUBSTR(c,n,[t])`	Extraction de la sous-chaîne *c* commençant à la position *n* sur *t* caractères.	`SELECT SUBSTR('Air France à Blagnac Con!',12,9) "Où ça?" FROM DUAL;`
		
		`Où ça?`
		`---------`
		`à Blagnac`

`TRANSLATE('c1', 'de', 'vers')`	Transforme chaque caractère de *c1* existant dans *de* ayant un correspondant dans *vers*.	`SELECT TRANSLATE('ORACLE9i', '0123456789ABCDEFGHIJKLMNOPQR', 'ChaineVers-Codage-0123456789') "Codage" FROM DUAL;`
		
		`Codage`
		`--------`
		`69-o3asi`

`TRIM(c1 FROM c2)`	Enlève les caractères *c1* à la chaîne *c2* (options `LEADING` et `TRAILING` pour préciser le sens du découpage).	`SELECT TRIM('B' FROM 'BA380 à BlagnacBBBBB') "Bye les Jumbo" FROM DUAL;`
		
		`Bye les Jumbo`
		`--------------`
		`A380 à Blagnac`

`UPPER`	Tout en majuscules.	`SELECT UPPER(prenom) || ' ' || UPPER(nom) "Pilotes de CAST" FROM Pilote WHERE compa = 'CAST';`
		
		`Pilotes de CAST`
		`--------------------`
		`PLACIDE FRESNAIS`
		`FRANCOISE TORT`

Numériques

La plupart des fonctions numériques acceptent en paramètre une ou plusieurs expressions de type NUMBER.

Tableau 4-17 Fonctions numériques

Fonction	Objectif	Exemple
ABS(*n*)	Valeur absolue de *n*.	
ACOS(*n*)	Arc cosinus (*n* de -1 à 1), retour exprimé en radians (de 0 à pi).	
ATAN(*n*)	Arc tangente (∀ *n*), retour exprimé en radians (de –pi/2 à pi/2).	
CEIL(*n*)	Plus petit entier ≥ à *n*.	`CEIL(15.7)` donne `16`.
COS(*n*)	Cosinus de *n* exprimé en radians de 0 à 2 pi (conversion en degrés : *d*°3.14159265359/180).	`COS(60*3.14159265359/180)` donne `0.5`.
COSH(*n*)	Cosinus hyperbolique de *n*.	
EXP(*n*)	*e* (2.71828183) à la puissance *n*.	
FLOOR(*n*)	Plus grand entier ≤ à *n*.	`FLOOR(15.7)` donne `15`.
LN(*n*)	Logarithme népérien de *n*.	
LOG(*n*) (*m*,*n*)	Logarithme de *n* dans une base *m*.	
MOD(*m*,*n*)	Division entière de *m* par *n*.	
POWER(*m*,*n*)	*m* puissance *n*.	
ROUND(*m*,*n*)	Arrondi à une ou plusieurs décimales.	`ROUND(17.567,2)` donne `17,57`.
SIGN(*n*)	Retourne le signe d'un nombre.	
SIN(*n*)	Sinus de *n* exprimé en radians de 0 à 2 pi (conversion en degrés : *d*°3.14159265359/180).	`SIN(30*3.14159265359/180)` donne `0.5`.
SINH(*n*)	Sinus hyperbolique de *n*.	
SQRT(*n*)	Racine carrée de *n*.	
TAN(*n*)	Tangente de *n* exprimée en radians de 0 à 2 pi.	
TANH(*n*)	Tangente hyperbolique de *n*.	

Tableau 4-17 Fonctions numériques *(suite)*

Fonction	Objectif	Exemple
`TRUNC(n,m)`	Coupure de *n* à *m* décimales.	`TRUNC(15.79,1)` donne 15.7.
`WIDTH_BUCKET (expression, min, max,num)`	Construction d'histogrammes, *expression* nombre ou date, *min* limite inférieure, *max* limite supérieure, *num* nombre d'intervalles à construire.	

Valeurs spéciales pour les flottants

La recommandation IEEE 754 définit des valeurs spéciales pour les flottants : l'infini positif (+INF), l'infini néfatif (-INF) et NaN (*Not a Number*) qui est utilisé pour représenter les résultats des opérations indéfinies. L'obtention de ces valeurs se réalise par les opérations suivantes : dépassement de capacité (*overflow*) pour obtenir -INF, +INF, opération invalide retourne NaN, la division par zéro peut retourner -INF, +INF ou NaN. Les opérateurs SQL NAN et INFINITE permettent de tester ces valeurs spéciales sur des flottants.

Le script suivant crée une table, insère deux flottants, modifie les chiffres pour insérer des valeurs infinies (la première résultant d'une division par zéro, la seconde d'un dépassement de capacité).

```
CREATE TABLE  Flottants (bfloat BINARY_FLOAT, bdouble BINARY_DOUBLE);
INSERT INTO   Flottants VALUES (+3.4e+38f, +1.77e+308d);
SELECT * FROM Flottants;
    BFLOAT    BDOUBLE
---------- ----------
  3,4E+038  1,77E+308

UPDATE Flottants SET bfloat = bfloat/0, bdouble= 2*bdouble;
SELECT * FROM Flottants WHERE bfloat IS INFINITE OR bdouble IS INFINITE;
    BFLOAT    BDOUBLE
---------- ----------
       Inf        Inf
```

Fonctions pour les flottants

Plusieurs fonctions sont disponibles pour manipuler des flottants.

TO_BINARY_DOUBLE

Comme son nom l'indique, cette fonction transforme une expression en flottant de type BINARY_DOUBLE. La syntaxe est la suivante :

TO_BINARY_DOUBLE(*expression* [,*'format'* [, *'nlsparam'*]])

- *format* et *nlsparam* ont la même signification que dans TO_CHAR ;
- *expression* représente une valeur numérique ou 'INF', '-INF', 'NaN'.

Le script suivant présente l'utilisation de cette fonction.

```
SELECT TO_BINARY_DOUBLE(13.56767) FROM DUAL;
TO_BINARY_DOUBLE(13.56767)
--------------------------
                1,357E+001

SELECT TO_BINARY_DOUBLE('-INF') FROM DUAL;
TO_BINARY_DOUBLE('-INF')
------------------------
                     -Inf
```

TO_BINARY_FLOAT

Cette fonction transforme une expression en flottant de type BINARY_FLOAT. La syntaxe est la suivante :

TO_BINARY_FLOAT(*expression*[,*'format*'[,*'nlsparam*']])

La signification des paramètres est identique à la fonction précédente.

DUMP

La fonction DUMP n'est pas dédiée aux flottants mais elle peut être utile pour mieux visualiser leur représentation. Cette fonction décrit la représentation interne de toute information sous la forme d'une chaîne de caractères incluant le code du type de données, la taille en octets et la valeur de chaque octet. Sa syntaxe est la suivante :

DUMP(*expression*[, *FormatRetour* [, *position* [, *longueur*]]])

- *FormatRetour* :
  - 8 pour retourner une notation octale.
  - 10 pour retourner une notation décimale.
  - 16 pour retourner une notation hexadécimale.
  - 17 pour retourner des caractères distincts.
- *position* et *longueur* combinent la portion de la représentation interne à retourner (par défaut, toute l'expression est décodée).

Voici deux exemples d'utilisation de cette fonction. La confirmation qu'un flottant de type BINARY_DOUBLE est représenté sur 9 octets (dont 8 visibles) apparaît ici clairement. La valeur de chaque octet en décimale est précisée dans la liste de valeurs retournées.

```
SELECT DUMP(TO_BINARY_DOUBLE(13.56767),10) FROM DUAL;
DUMP(TO_BINARY_DOUBLE(13.56767),10)
----------------------------------------
Typ=101 Len=8: 192,43,34,165,164,105,215,52

SELECT DUMP('C.Soutou', 10) "C.Soutou en ASCII" FROM DUAL;
C.Soutou en ASCII
---------------------------------------
Typ=96 Len=8: 67,46,83,111,117,116,111,117
```

NANVL

La fonction NANVL permet de substituer la valeur NaN (*Not a Number*) contenue dans un flottant par une autre valeur donnée et compréhensible (exemple : zéro ou NULL). La syntaxe de cette fonction est la suivante :

```
NANVL(expression, substitution)
```

- *expression* désigne la valeur à substituer (tout type numérique ou non numérique pouvant être implicitement converti en numérique). Si l'expression n'est pas NaN, la valeur de l'expression est retournée. Sinon la valeur *substitution* est retournée.

Le code suivant décrit l'utilisation de cette fonction appliquée à deux flottants. L'opérateur IS NAN est utilisé dans la deuxième requête. Dans la troisième requête, l'opérateur NANVL permet de substituer la valeur 0 au premier flottant et -1 au second quand ces deux valeurs sont indéterminées.

```
INSERT INTO Flottants VALUES (+3.4e+38f,+1.77e+308d) ;
INSERT INTO Flottants VALUES ('NaN','NaN') ;
SELECT * FROM Flottants;
    BFLOAT    BDOUBLE
---------- ----------
  3,4E+038  1,77E+308
       Nan        Nan
SELECT * FROM Flottants WHERE bfloat IS NOT NAN AND bdouble IS NOT NAN;
    BFLOAT    BDOUBLE
---------- ----------
  3,4E+038  1,77E+308
SELECT NANVL(bfloat,0), NANVL(bdouble,-1) FROM Flottants;
```

```
NANVL(BFLOAT,0) NANVL(BDOUBLE,-1)
--------------- -----------------
       3,4E+038          1,77E+308
              0          -1,0E+000
```

REMAINDER

La fonction REMAINDER retourne le reste de la division de *m* par *n*. La fonction MOD étudiée au chapitre 4 est quelque peu similaire à REMAINDER (MOD utilise l'opérateur FLOOR alors que REMAINDER utilise ROUND). La syntaxe de cette fonction est la suivante :

```
REMAINDER(m, n)
```

- *m* désigne la valeur à diviser (tout type numérique ou non numérique pouvant être implicitement converti en numérique). *n* désigne de la même manière le diviseur.
- Si *n* = 0 ou si *m* est infini, et si les arguments sont de type NUMBER, la valeur retournée est une erreur. Dans le cas de flottants (BINARY_FLOAT or BINARY_DOUBLE), la valeur retournée est NaN (*Not a Number*).
- Si *n* est différent de zéro, la fonction retourne la valeur *m* - (*n***N*) avec *N* plus grand entier plus proche du résultat *m*/*n*.
- Si *m* est un flottant et si le résultat vaut zéro, alors le signe du résultat est du signe de *m*. Si *m* est un NUMBER et si le résultat vaut zéro, alors le résultat n'est pas signé.

Le code suivant décrit l'utilisation de cette fonction appliquée à deux flottants de différents types également valués (1234,56). La valeur retournée n'est pas zéro du fait da la différence des types.

```
INSERT INTO Flottants VALUES (1234.56,1234.56);

SELECT * FROM Flottants;
    BFLOAT    BDOUBLE

---------- ----------
1,235E+003 1,235E+003


SELECT bfloat, bdouble, REMAINDER(bfloat,bdouble) FROM Flottants;
    BFLOAT    BDOUBLE REMAINDER(BFLOAT,BDOUBLE)
---------- ---------- ------------------------
1,235E+003 1,235E+003                5,859E-005
```

Dates

Le tableau suivant décrit les principales fonctions pour les dates.

Tableau 4-18 Fonctions pour les dates

Fonction	Objectif	Retour
`ADD_MONTHS`	Ajoute des mois à une date.	`DATE`
`CURRENT_DATE`	Retourne la date courante (calendrier grégorien) dans la session et le fuseau de la base.	`DATE`
`EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM {d | i})`	Extrait une partie donnée d'une date ou d'un intervalle.	`NUMBER`
`LAST_DAY(d)`	Retourne le dernier jour du mois.	`DATE`
`MONTHS_BETWEEN(d1,d2)`	Retourne le nombre de mois entre deux dates (*d1* et *d2* avec *d1>d2*).	`NUMBER`
`NEW_TIME (d,z1,z2)`	Retourne la date *d* exprimée en zone *z1* dans la zone z2.	`DATE`
`NEXT_DAY(d,jour)`	Retourne la date du prochain jour ouvrable (exemple jour `'LUNDI'`) à partir de *d*.	`DATE`
`ROUND(d,format)`	Arrondit une date *d* selon un format (exemple : `'YEAR'`).	`DATE`
`SYSDATE`	Date courante (du système).	`DATE`
`TRUNC(d,format)`	Tronque une date *d* selon un format (exemple : `'YEAR'`).	`DATE`

Quelques exemples d'utilisation (`SYSDATE` est ici mercredi 14 mai 2003) sont donnés dans le tableau suivant.

Tableau 4-19 Exemples de fonctions pour les dates

Besoin et fonction	Résultat

Mercredi en 7 ?
`SELECT NEXT_DAY(SYSDATE,'MERCREDI') "Mercr/7" FROM DUAL;`
	`Mercr/7`
	`--------`
	`21/05/03`

Rendez-vous dans 4 mois.
`SELECT ADD_MONTHS(SYSDATE,4) "RDV" FROM DUAL;`
	`RDV`
	`--------`
	`14/09/03`

Numéro du mois d'il y a 65 jours ?
`SELECT EXTRACT(MONTH FROM (SYSDATE-65)) "Mois" FROM DUAL;`
	`Mois`
	`----------`
	`3`

Tableau 4-19 Exemples de fonctions pour les dates *(suite)*

Besoin et fonction	Résultat

Arrondi du 28 octobre 2005 au niveau du mois.
`SELECT ROUND(TO_DATE('28-OCT-2005'),'MONTH') "Arrondi" FROM DUAL;`
	`Arrondi`
	`--------`
	`01/11/05`

Coupe du 28 octobre 2005 au niveau du mois.
`SELECT TRUNC(TO_DATE('28-OCT-2005'),'MONTH') "Tronque" FROM DUAL;`
	`Tronque`
	`--------`
	`01/10/05`

Conversions

Oracle autorise des conversions de types implicites ou explicites.

Implicites

Il est possible d'affecter dans une expression ou dans une instruction SQL (`INSERT`, `UPDATE`...), une donnée de type `NUMBER` (ou `DATE`) à une donnée de type `VARCHAR2` (ou `CHAR`). Il en va de même pour l'affectation d'une colonne `VARCHAR2` par une donnée de type `DATE` (ou `NUMBER`). On parle ainsi de conversions implicites.

Pour preuve, le script suivant ne renvoie aucune erreur :

```
CREATE TABLE Test          (c1 NUMBER, c2 DATE, c3 VARCHAR2(1), c4 CHAR);
INSERT INTO Test VALUES    ('548,45', '13-05-2003', 3, 5);
```

Explicites

Une conversion est dite explicite quand on utilise une fonction à cet effet. Les fonctions de conversion les plus connues sont `TO_NUMBER`, `TO_CHAR` et `TO_DATE`.

Les fonctions de conversion sont décrites dans le tableau suivant.

Tableau 4-20 Fonctions de conversion

Fonction	Conversion	Exemple
`BIN_TO_NUM(`*`b1,b2…`*`)`	Les bits en `NUMBER`.	`BIN_TO_NUM(1,0,1,0)` donne `10`.
`CAST(`*`expression AS typeOracle`*`)`	L'expression dans le type en paramètre.	`CAST(2 AS CHAR)` donne `'2'`.

Tableau 4-20 Fonctions de conversion *(suite)*

Fonction	Conversion	Exemple
CHARTOROWID(*c*)	La chaîne *c* en ROWID.	
COMPOSE('*c*')	La chaîne *c* en Unicode.	
CONVERT(*c*, jeudest [,jeusource])	La chaîne *c* du jeu de caractères source en jeu de destination.	CONVERT('Ä Ê Î Ø', 'US7ASCII', 'WE8ISO8859P1') donne "A E I ?".
NUMTODSINTERVAL	Un nombre dans un type INTERVAL DAY TO SECOND.	Déjà étudié.
NUMTOYMINTERVAL	Un nombre dans un type INTERVAL YEAR TO MONTH.	Déjà étudié.
ROWIDTOCHAR(*r*)	Le ROWID *r* en VARCHAR2.	
TO_CHAR(*c*)	La chaîne en VARCHAR2.	
TO_CHAR(*d*[,*format*])	La date en VARCHAR2.	Déjà étudié.
TO_CHAR(*n*[,*format*])	Le nombre en VARCHAR2.	TO_NUMBER('1234.567','9.9EEEE') donne 1.3E+02.
TO_DSINTERVAL(*c* ['*paramNLS*'])	Une chaîne *c* dans un type INTERVAL DAY TO SECOND.	
TO_NUMBER(*c* [,*format* [,'*paramNLS*']])	Une chaîne *c* contenant un nombre dans un type NUMBER selon un format et une langue.	TO_NUMBER('100,9678') donne 100,9678.
TO_YMINTERVAL(*c*)	Une chaîne *c* dans un type INTERVAL YEAR TO MONTH.	SYSDATE + TO_YMINTERVAL('01-02') donne la date du jour + 1 an et 2 mois.
UNISTR('*c*')	La chaîne *c* en Unicode.	UNISTR('\00D6') donne Ö.

Autres fonctions

D'autres fonctions n'appartenant pas à la classification précédente sont présentées dans le tableau suivant :

Tableau 4-21 Autres fonctions

Fonction	Objectif	Exemple
DECODE(*colonne, cherche, resultat* [,*cherche,resultat*]…)	Programme un *case*.	**DECODE**(grade, 1, 'Copilote', 2,' Instructeur') affiche 'Copilote' si la colonne grade=1.
GREATEST(*expression*[, *expression*]…)	Retourne la plus grande des expressions.	**GREATEST**('Raffarin','Chirac', 'X-Men') retourne 'X-Men'.
LEAST(*expression*[, *expression*]…)	Retourne la plus petite des expressions.	**LEAST**('Raffarin','Chirac', 'X-Men') retourne 'Chirac'.

Tableau 4-21 Autres fonctions *(suite)*

Fonction	Objectif	Exemple
`NULLIF(expr1,expr2)`	Si *expr1* = *expr2* retourne `NULL`, sinon retourne *expr1*.	`NULLIF('Raffarine','Parafine')` retourne `'Raffarine'`.
`NVL(expr1,expr2)`	Convertit *expr1* susceptible d'être nulle en une valeur réelle (*expr2*).	`NVL(grade,'Aucun !')` retourne `'Aucun !'` si `grade` est `NULL`.

Regroupements

Cette section traite à la fois des regroupements de lignes (agrégats) et des fonctions de groupe (multiligne). Nous étudierons la partie surlignée de l'instruction `SELECT` suivante :

```
SELECT [ { DISTINCT | UNIQUE } | ALL ] listeColonnes
  FROM nomTable
  [ WHERE condition ]
  [ clauseRegroupement ]
  [ HAVING condition ]
  [ clauseOrdonnancement ] ;
```

- *`listeColonnes`* : peut inclure des expressions (présentes dans la clause de regroupement) ou des fonctions de groupe.
- *`clauseRegroupement`* : `GROUP BY (expression1[,expression2]...)` permet de regrouper des lignes selon la valeur des expressions (colonnes, fonction, constante, calcul).
- `HAVING` *`condition`* : pour inclure ou exclure des lignes aux groupes (la condition ne peut faire intervenir que des expressions du `GROUP BY`).

Interrogeons la table suivante en composant des regroupements et en appliquant des fonctions de groupe :

Figure 4-5 *Table Pilote*

Pilote

brevet	nom	nbHVol	prime	embauche	typeAvion	compa
PL-1	Gratien Viel	450	500	05/02/1965	A320	AF
PL-2	Didier Donsez	0		13/05/1995	A320	AF
PL-3	Richard Grin	1000		11/09/2001	A320	SING
PL-4	Placide Fresnais	2450	500	21/09/2001	A330	SING
PL-5	Daniel Vielle	400	600	16/01/1965	A340	AF
PL-6	Francoise Tort		0	24/12/2000	A340	CAST

Fonctions de groupe

Nous étudions dans cette section les fonctions usuelles. D'autres sont proposées pour manipuler des cubes (*datawarehouse*).

Le tableau suivant présente les principales fonctions. L'option DISTINCT évite les duplicatas alors que ALL les prend en compte (par défaut). À l'exception de COUNT, toutes les fonctions ignorent les valeurs NULL (il faudra utiliser NVL pour contrer cet effet).

Tableau 4-22 Fonctions de groupe

Fonction	Objectif
AVG([DISTINCT | ALL] *expr*)	Moyenne de *expr* (nombre).
COUNT({* | [DISTINCT | ALL] *expr*})	Nombre de lignes (* toutes les lignes, *expr* pour les colonnes non nulles).
MAX([DISTINCT | ALL] *expr*)	Maximum de *expr* (nombre, date, chaîne).
MIN([DISTINCT | ALL] *expr*)	Minimum de *expr* (nombre, date, chaîne).
STDDEV([DISTINCT | ALL] *expr*)	Écart type de *expr* (nombre).
SUM([DISTINCT | ALL] *expr*)	Somme de *expr* (nombre).
VARIANCE([DISTINCT | ALL] *expr*)	Variance de *expr* (nombre).

Utilisées sans GROUP BY, ces fonctions s'appliquent à la totalité ou à une seule partie d'une table comme le montrent les exemples suivants.

Tableau 4-23 Exemples de fonctions de groupe

Fonction	Exemples

AVG	Moyenne des heures de vol et des primes des pilotes de la compagnie 'AF'.
	`SELECT AVG(nbHVol), AVG(prime) FROM Pilote WHERE compa = 'AF';`
	
	`AVG(NBHVOL) AVG(PRIME)`
	`----------- ----------`
	`283,333333 550`

COUNT	Nombre de pilotes, d'heures de vol et de primes (toutes et distinctes) recensées dans la table.
	`SELECT COUNT(*), COUNT(nbHVol), COUNT(prime), COUNT(DISTINCT prime) FROM Pilote;`
	
	`COUNT(*) COUNT(NBHVOL) COUNT(PRIME) COUNT(DISTINCTPRIME)`
	`---------- ------------ ------------ --------------------`
	`6 5 4 3`

Tableau 4-23 Exemples de fonctions de groupe *(suite)*

Fonction	Exemples

MAX - MIN	Nombre d'heures de vol le plus élevé, date d'embauche la plus récente. Nombre d'heures de vol le moins élevé, date d'embauche la plus ancienne.
	`SELECT MAX(nbHVol), MAX(embauche) "Date+", MIN(prime), MIN(embauche) "Date-" FROM Pilote;`
	`MAX(NBHVOL) Date+ MIN(PRIME) Date-`
	`---------- -------- ---------- --------`
	`2450 21/09/01 0 16/01/65`

STDEV – SUM – VARIANCE	Écart type des primes, somme des heures de vol, variance des primes des pilotes de la compagnie 'AF'.
	`SELECT STDDEV(prime), SUM(nbHVol), VARIANCE(prime) FROM Pilote WHERE compa = 'AF';`
	`STDDEV(PRIME) SUM(NBHVOL) VARIANCE(PRIME)`
	`------------- ----------- ---------------`
	`70,7106781 850 5000`

Étudions à présent ces fonctions dans le cadre de regroupements de lignes.

Étude du GROUP BY et HAVING

Le groupement de lignes dans une requête se programme au niveau surligné de l'instruction SQL suivante :

```
SELECT col1[, col2…], fonction1Groupe(…)[,fonction2Groupe(…)…]
   FROM nomTable
   [ WHERE condition ]
   GROUP BY col1[, col2]…)
   [ HAVING condition ]
   [ ORDER BY… ] ;
```

- la clause `WHERE` de la requête permet d'exclure des lignes pour chaque groupement, ou de rejeter des groupements entiers. Elle s'applique donc à la totalité de la table ;
- la clause `GROUP BY` liste les colonnes du groupement ;
- la clause `HAVING` permet de poser des conditions sur chaque groupement.

Les colonnes présentes dans le `SELECT` doivent apparaître dans le `GROUP BY`. Seules des fonctions ou expressions peuvent exister en plus dans le `SELECT`.

Les alias de colonnes ne peuvent pas être utilisés dans la clause `GROUP BY`.

Dans l'exemple suivant, en groupant sur la colonne `compa`, trois ensembles de lignes (groupements) sont composés. Il est alors possible d'appliquer des fonctions de groupe à chacun de

ces ensembles (dont le nombre n'est pas précisé dans la requête ni limité par le système qui parcourt toute la table).

Figure 4-6 *Groupement sur la colonne compa*

Pilote

brevet	nom	nbHVol	prime	embauche	typeAvion	compa
PL-1	Gratien Viel	450	500	05/02/1965	A320	AF
PL-2	Didier Donsez	0		13/05/1995	A320	AF
PL-5	Daniel Vielle	400	600	16/01/1965	A340	AF
PL-6	Francoise Tort		0	24/12/2000	A340	CAST
PL-3	Richard Grin	1000		11/09/2001	A320	SING
PL-4	Placide Fresnais	2450	500	21/09/2001	A330	SING

Il est aussi possible de grouper sur plusieurs colonnes (par exemple ici sur les colonnes `compa` et `typeAvion` pour classifier les pilotes selon ces deux critères).

Utilisées avec `GROUP BY`, les fonctions s'appliquent désormais à chaque regroupement comme le montrent les exemples suivants :

Tableau 4-24 Exemple de fonctions de groupe avec GROUP BY

Fonction	Exemples

AVG	Moyenne des heures de vol et des primes pour chaque compagnie.
	`SELECT compa, AVG(nbHVol), AVG(prime) FROM Pilote`
	`GROUP BY(compa);`
	
	`COMP AVG(NBHVOL) AVG(PRIME)`
	`---- ----------- ----------`
	`AF 283,333333 550`
	`CAST 0`
	`SING 1725 500`

COUNT	Nombre de pilotes (et ceux qui ont de l'expérience du vol) par compagnie.
	`SELECT compa, COUNT(*), COUNT(nbHVol) FROM Pilote`
	`GROUP BY(compa);`
	
	`COMP COUNT(*) COUNT(NBHVOL)`
	`---- ---------- -------------`
	`AF 3 3`
	`CAST 1 0`
	`SING 2 2`

Tableau 4-24 Exemple de fonctions de groupe avec GROUP BY *(suite)*

Fonction	Exemples

MAX	Nombre d'heures de vol le plus élevé, date d'embauche la plus récente pour chaque compagnie.
	`SELECT compa, MAX(nbHVol), MAX(embauche) "Date+"`
	`FROM Pilote GROUP BY(compa);`
	
	`COMP MAX(NBHVOL) Date+`
	`---- ----------- --------`
	`AF 450 13/05/95`
	`CAST 24/12/00`
	`SING 2450 21/09/01`

STDEV - SUM (avec WHERE)	Écarts type des primes et sommes des heures de vol des pilotes volant sur 'A320' de chaque compagnie.
	`SELECT compa, STDDEV(prime), SUM(nbHVol) FROM Pilote`
	`WHERE typeAvion = 'A320' GROUP BY(compa);`
	
	`COMP STDDEV(PRIME) SUM(NBHVOL)`
	`---- ------------- -----------`
	`AF 0 450`
	`SING 1000`

Plusieurs colonnes dans le GROUP BY	Nombre de pilotes qualifiés par type d'appareil et par compagnie.
	`SELECT compa,typeAvion, COUNT(brevet) FROM Pilote`
	`GROUP BY(compa,typeAvion);`
	
	`COMP TYPE COUNT(BREVET)`
	`---- ---- -------------`
	`AF A320 2`
	`AF A340 1`
	`CAST A340 1`
	`SING A320 1`
	`SING A330 1`

GROUP BY et HAVING	Compagnies (et nombre de leurs pilotes) ayant plus d'un pilote.
	`SELECT compa, COUNT(brevet) FROM Pilote`
	`GROUP BY(compa)`
	`HAVING COUNT(brevet)>=2;`
	
	`COMP COUNT(BREVET)`
	`---- -------------`
	`AF 3`
	`SING 2`

Opérateurs ensemblistes

Une des forces du modèle relationnel repose sur le fait qu'il est fondé sur une base mathématique (théorie des ensembles). Le langage SQL programme les opérations binaires (entre deux tables) suivantes :

- **intersection** par l'opérateur `INTERSECT` qui extrait des données présentes simultanément dans les deux tables ;
- **union** par les opérateurs `UNION` et `UNION ALL` qui fusionnent des données des deux tables ;
- **différence** par l'opérateur `MINUS` qui extrait des données présentes dans une table sans être présentes dans la deuxième table ;
- **produit cartésien** par le fait de disposer de deux tables dans la clause `FROM`, ce qui permet de composer des combinaisons à partir des données des deux tables.

Un opérateur ensembliste se place entre deux requêtes comme le montre la syntaxe simplifiée suivante :

```
SELECT … FROM nomTable [WHERE …] opérateur SELECT … FROM nomTable [WHERE …];
```

Les opérateurs ensemblistes ont pour l'instant tous la même priorité. Cependant, pour être conformes aux nouvelles directives de la norme, les versions ultérieures d'Oracle privilégieront l'opérateur `INTERSECT` par rapport aux autres.

Si une requête contient plusieurs de ces opérateurs, ils sont évalués de la gauche vers la droite, quand aucune parenthèse ne spécifie un autre ordre. Ainsi, les deux écritures suivantes produisent des résultats différents :

```
SELECT … INTERSECT SELECT … UNION SELECT … MINUS SELECT…
SELECT … INTERSECT SELECT … UNION (SELECT … MINUS SELECT …)
```

Restrictions

Seules des colonnes de même type (`CHAR`, `VARCHAR2`, `DATE` ou `NUMBER`) doivent être comparées avec des opérateurs ensemblistes.

Il n'est pas possible d'utiliser les opérateurs ensemblistes sur des colonnes `BLOB`, `CLOB`, `BFILE`, ou `LONG`. Les collections *varrays* et *nested tables* (extensions objets) sont également exclues.

Attention, pour les colonnes `CHAR`, à veiller à ce que la taille soit identique entre les deux tables pour que la comparaison fonctionne. Le nom des colonnes n'a pas d'importance. Il est possible de comparer plusieurs colonnes de deux tables.

Exemple

Étudions à présent chaque opérateur à partir de l'exemple composé des deux tables suivantes. Il est visible que seules les deux premières colonnes peuvent être comparées. Il ne serait pas logique de tenter de faire une intersection ou une union entre l'ensemble des prix d'achat et des heures de vol par exemple.

Bien que permise par Oracle, l'union des prix et des heures de vol (deux colonnes NUMBER) ne serait pas non plus valide d'un point de vue sémantique.

Figure 4-7 *Tables*

AviondeAF

immat	typeAvion	nbHVol
F-WTSS	Concorde	6570
F-GLFS	A320	3500
F-GTMP	A340	

AviondeSING

immatriculation	typeAv	PrixAchat
S-ANSI	A320	104 500
S-AVEZ	A320	156 000
S-SMILE	A330	198 000
F-GTMP	A340	204 500

Opérateur INTERSECT

L'opérateur INTERSECT est commutatif (*requête1* INTERSECT *requête2* est identique à *requête2* INTERSECT *requête1*). Cet opérateur élimine les duplicatas entre les deux tables avant d'opérer l'intersection.

Notez qu'à l'affichage, le nom des colonnes est donné par la première requête. La deuxième fait apparaître deux colonnes dans le SELECT.

Tableau 4-25 Exemples avec INTERSECT

Besoin	Requête

Quels sont les types d'avions que les deux compagnies exploitent en commun ?	`SELECT typeAvion FROM AvionsdeAF`
	`INTERSECT`
	`SELECT typeAv FROM AvionsdeSING;`
	
	`TYPEAVION`
	`----------`
	`A320`
	`A340`

Quels sont les avions qui sont exploités par les deux compagnies en commun ?	`SELECT immat,typeAvion FROM AvionsdeAF`
	`INTERSECT`
	`SELECT immatriculation,typeAv FROM AvionsdeSING;`
	
	`IMMAT TYPEAVION`
	`------ ----------`
	`F-GTMP A340`

Si vous voulez continuer ce raisonnement en vous basant sur trois compagnies, il suffit d'ajouter une clause INTERSECT et de la faire suivre d'une requête concernant la troisième compagnie. Ce principe se généralise, et, pour *n* compagnies, il faudra *n* requêtes reliées entre elles par *n-1* clauses INTERSECT.

Opérateurs UNION et UNION ALL

Les opérateurs UNION et UNION ALL sont commutatifs. L'opérateur UNION permet d'éviter les duplicatas (comme DISTINCT ou UNIQUE dans un SELECT). UNION ALL ne les élimine pas.

Tableau 4-26 Exemples avec les opérateurs UNION

Besoin	Requête

Quels sont tous les types d'avions que les deux compagnies exploitent ?	`SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING;`
	
	`TYPEAVION`
	`----------`
	`A320`
	`A330`
	`A340`
	`Concorde`

Même requête avec les duplicatas. On extrait les types de la compagnie 'AF' suivis des types de la compagnie 'SING'.	`SELECT typeAvion FROM AvionsdeAF`
	`UNION ALL`
	`SELECT typeAv FROM AvionsdeSING;`
	
	`TYPEAVION`
	`----------`
	`Concorde`
	`A320`
	`A340`
	`A320`
	`A320`
	`A330`
	`A340`

Ce principe se généralise à l'union de *n* ensembles par *n* requêtes reliées avec *n-1* clauses UNION ou UNION ALL.

Opérateur MINUS

L'opérateur MINUS est le seul opérateur ensembliste qui ne soit pas commutatif. Il élimine les duplicatas avant d'opérer la soustraction.

Tableau 4-27 Exemples avec l'opérateur MINUS

Besoin	Requête

Quels sont les types d'avions exploités par la compagnie 'AF' mais pas par 'SING' ?	`SELECT typeAvion FROM AvionsdeAF`
	`MINUS`
	`SELECT typeAv FROM AvionsdeSING;`
	
	`TYPEAVION`
	`----------`
	`Concorde`

Quels sont les types d'avions exploités par la compagnie 'SING' mais pas par 'AF' ?	`SELECT typeAv FROM AvionsdeSING`
	`MINUS`
	`SELECT typeAvion FROM AvionsdeAF ;`
	
	`TYPEAV`
	`----------`
	`A330`

Ce principe se généralise à la différence entre *n* ensembles par *n* requêtes reliées (dans le bon ordre) par *n-1* clauses `MINUS`.

Ordonner les résultats

Pour un faible volume de données, le résultat d'une requête ensembliste semble trié par défaut par ordre croissant selon la première colonne extraite mais il n'en est rien dès que le nombre de lignes retourné devient important. Pour trier un jeu de résultats issu d'une requête ensembliste, vous devez utiliser explicitement la clause `ORDER BY`. Elle doit se placer une seule fois à la fin de la requête, et accepte soit des noms (ou alias) de colonne de la première requête, soit la position des colonnes (mais cela n'est pas recommandé).

Le tableau suivant présente la même requête (types d'avions que les deux compagnies exploitent) dont le résultat est trié par ordre décroissant. La première écriture utilise la clause `ORDER BY` à l'aide du nom de la colonne de la première requête, la deuxième utilise un alias de colonne, et la dernière indique seulement la position de la colonne.

Tableau 4-28 Exemples avec les opérateurs UNION

Technique	Requête

Nom de la colonne	`SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING`
	`ORDER BY typeAvion;`

Tableau 4-28 Exemples avec les opérateurs UNION *(suite)*

Technique	Requête

Alias de colonne	`SELECT typeAvion AS type FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING`
	`ORDER BY type DESC;`

Position de colonne	`SELECT typeAvion AS type FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING`
	`ORDER BY 1 DESC;`

	`TYPE` (`TYPEAVION` pour la 1re requête)
	`---------`
	`Concorde`
	`A340`
	`A330`
	`A320`

Pour illustrer une autre utilisation d'un alias, tentons d'extraire les avions et leur prix d'achat augmenté de 20 % – liste triée en fonction de cette dernière hausse. Le problème est que la table `AvionsdeAF` ne possède pas une telle colonne. Il suffit d'ajouter au `SELECT` de cette table l'expression `0` pour rendre homogène les deux jeux de résultats pour l'opérateur `UNION`.

Tableau 4-29 Alias pour ORDER BY

Requête	Résultat

`SELECT immatriculation,`
`1.2*prixAchat px FROM AvionsdeSING`
`UNION`
`SELECT immat, 0 FROM AvionsdeAF`
`ORDER BY px DESC;`
	`IMMATR PX`
	`------ ----------`
	`F-GTMP 245400`
	`S-MILE 227600`
	`S-AVEZ 187200`
	`S-ANSI 125400`
	`F-GLFS 0`
	`F-GTMP 0`
	`F-WTSS 0`

Produit cartésien

En mathématiques, le produit cartésien de deux ensembles E et F est l'ensemble des couples (x, y) où $x \in E$ et $y \in F$. En transposant au modèle relationnel, le produit cartésien de deux tables *T1* et *T2* est l'ensemble des enregistrements (x, y) où $x \in T1$ et $y \in T2$.

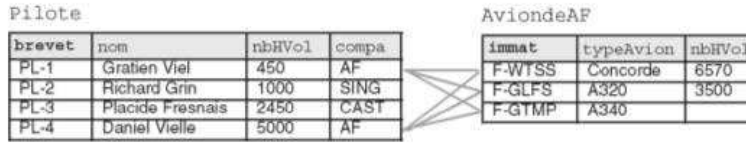
Le produit cartésien total entre deux tables *T1* et *T2* se programme sous SQL en positionnant les deux tables dans la clause `FROM` sans ajouter de conditions dans la clause `WHERE`.

Si les conditions sont de la forme « *c1* `opérateur` *c2* » avec $c1 \in T1$ et $c2 \in T2$, on parlera de jointure.

Si les conditions sont de la forme « *c1* `opérateur` *valeur1* » ou « *c2* `opérateur` *valeur2* », on parlera de produit cartésien restreint.

Le produit cartésien restreint, illustré par l'exemple suivant, exprime les combinaisons d'équipage qu'il est possible de réaliser en considérant les pilotes de la compagnie 'AF' et les avions de la table `AviondeAF`.

Figure 4-8 *Produit cartésien d'enregistrements de tables*



Le nombre d'enregistrements résultant d'un produit cartésien est égal au produit du nombre d'enregistrements des deux tables mises en relation.

Dans le cadre de notre exemple, le nombre d'enregistrements du produit cartésien sera de 2 pilotes × 3 avions = 6 enregistrements. Le tableau suivant décrit la requête SQL permettant de construire le produit cartésien restreint de notre exemple. Les alias distinguent les colonnes s'il advenait qu'il en existe de même nom entre les deux tables.

Tableau 4-30 Produit cartésien

Besoin	Requête

Quels sont les couples possibles (*avion*, *pilote*) en considérant les avions et les pilotes de la compagnie 'AF' ?	`SELECT p.brevet, avAF.immat`
	`  FROM Pilote p, AvionsdeAF avAF`
	`  WHERE p.compa = 'AF';`

6 lignes extraites	`BREVET IMMAT`
	`------ ------`
	`PL-1 F-WTSS`
	`PL-4 F-WTSS`
	`PL-1 F-GLFS`
	`PL-4 F-GLFS`
	`PL-1 F-GTMP`
	`PL-4 F-GTMP`

Bilan

Seules les colonnes de même type et représentant la même sémantique peuvent être comparées à l'aide de termes ensemblistes. Il est possible d'ajouter des expressions (constantes ou

calculs) à une requête pour rendre homogènes les deux requêtes et permettre ainsi l'utilisation d'un opérateur ensembliste (voir l'exemple décrit au tableau 4-27).

Sous-interrogations dans la clause FROM

Introduite dans la norme SQL2, la possibilité de disposer une requête au sein de la clause `FROM` d'une requête principale permet d'évaluer dynamiquement un jeu de résultats en construisant une table avant de l'interroger. Vous découvrirez à la fin de ce chapitre que la directive `WITH`, plus récente dans le langage, permet de généraliser ce mécanisme.

La construction d'une table par sous-interrogation pour alimenter une requête principale suit la syntaxe suivante :

```
SELECT colonnes_ou_expressions
   FROM   [table1 alias_1, ...]
          (SELECT... FROM table2 WHERE...) alias_2
          [, (SELECT...) alias_n]
[ WHERE (conditions_table1_table2…) ];
```

Considérons deux exemples afin d'illustrer cette fonctionnalité.

Calcul d'un pourcentage partiel

Le premier exemple consiste à extraire le pourcentage partiel de pilotes par compagnie. Dans l'exemple suivant, 5 pilotes sont représentés (dont 3 associés à la compagnie 'AF'). En considérant cette compagnie, 60 % (soit 3/5) des pilotes en dépendent.

Figure 4-9 *Table Pilote*

Pilote

brevet	prenom	nom	nbHVol	compa
PL-1	Pierre	Lamothe	450	AF
PL-2	Didier	Linxe	900	AF
PL-3	Christian	Soutou	1000	SING
PL-4	Henri	Alquié	3400	AF
PL-5	Michel	Castaings		

La requête utilise deux sous-interrogations pour construire deux tables (respectivement d'alias a et b) dans la clause `FROM`. Ces sous-interrogations sont illustrées dans le tableau ci-après ; ces jeux de résultats permettent de calculer les pourcentages pour chaque compagnie.

Tableau 4-31 Sous-interrogations pour des pourcentages partiels

Requête et tables évaluées dans le FROM	Résultat

```
SELECT a.compa "Comp",
       a.nbpil/b.total*100 "%Pilote"
 FROM (SELECT compa, COUNT(*) nbpil-
              FROM Pilote  GROUP BY compa) a,
       (SELECT COUNT(*) total
              FROM Pilote) b;
```

```
Comp    %Pilote
---- ----------
AF            60
SING          20
              20
```

a

compa	nbpil
AF	3
SING	1
	1

b

total
5

Afin d'isoler les pilotes qui ne sont associés à aucune compagnie, vous devrez ajouter le prédicat `WHERE compa IS NOT NULL` aux deux sous-interrogations pour que les pilotes de la compagnie `'AF'` représentent 75 % (soit 3/4) de la population totale.

Extraire les n premières/dernières lignes d'un jeu de résultats

Le deuxième exemple consiste à extraire les 5 premiers pilotes d'une compagnie donnée.

Présentée en début de chapitre, la pseudo-colonne `ROWNUM` s'applique avant un tri. Pour remédier à ce mécanisme problématique, utilisez une sous-interrogation qui sera triée (par ordre croissant ou décroissant). Elle sera ensuite exploitée par une requête principale qui limitera le nombre de lignes retournées à l'aide de `ROWNUM` :

```
SELECT ...
FROM  (requête qui ordonne avec ORDER BY ASC ou DESC
       et sans utiliser ROWNUM)
WHERE  ROWNUM < (ou >) x;
```

Supposons que le tableau des pilotes contiennent davantage de lignes incluant plusieurs compagnies et appliquons ce principe à la requête désirée, voir tableau suivant.

Tableau 4-32 Sous-interrogation pour limiter un jeu de résultats

Requête	Résultat

```
SELECT ROWNUM, prenom, nom
FROM  (SELECT prenom, nom
       FROM   Pilote
       WHERE  compa = 'AF'
       ORDER BY nom ASC)
WHERE ROWNUM < 6;
```

```
 ROWNUM PRENOM          NOM
------- --------------- ---------
      1 Henri           Alquié
      2 Agnes           Bidal
      3 Fabienne        Bonnet
      4 Fred            Brouard
      5 Rudy            Bruchez
```

Jointures

Les jointures permettent d'extraire des données issues de plusieurs tables. Le processus de normalisation du modèle relationnel est basé sur la décomposition et a pour conséquence d'augmenter le nombre de tables d'un schéma. Ainsi, la majorité des requêtes utilisent des jointures nécessaires pour pouvoir extraire des données de tables distinctes.

Une jointure met en relation deux tables sur la base d'une clause de jointure (comparaison de colonnes). Généralement, cette comparaison fait intervenir une clé étrangère d'une table avec une clé primaire d'une autre table (le modèle relationnel est basé sur les valeurs).

En considérant les tables suivantes, les seules jointures logiques doivent se faire sur l'égalité soit des colonnes `comp` et `compa` soit des colonnes `brevet` et `chefPil`. Ces jointures permettront d'afficher des données d'une table (ou des deux tables) tout en posant des conditions sur une table (ou les deux). Par exemple, l'affichage du nom des compagnies (colonne de la table `Compagnie`) qui ont embauché un pilote ayant moins de 500 heures de vol (condition sur la table `Pilote`).

Figure 4-10 *Deux tables à mettre en jointure*

`Compagnie`

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL
CAST	1	G. Brassens	Blagnac	Castanet AL

`Pilote`

brevet	nom	nbHVol	compa	chefPil
PL-1	Pierre Lamothe	450	AF	PL-4
PL-2	Didier Linxe	900	AF	PL-4
PL-3	Christian Soutou	1000	SING	
PL-4	Henri Alquié	3400	AF	

Classification

Une jointure peut s'écrire, dans une requête SQL, de différentes manières :

- « relationnelle » (aussi appelée « SQL89 » pour rappeler la version de la norme SQL) ;
- « SQL2 » (aussi appelée « SQL92 ») ;
- « procédurale » (qui qualifie la structure de la requête) ;
- « mixte » (combinaison des trois approches précédentes).

Nous allons principalement étudier les deux premières écritures qui sont les plus utilisées. Nous parlerons en fin de section des deux dernières.

Jointure relationnelle

La forme la plus courante de la jointure est la jointure dite « relationnelle » (aussi appelée SQL89 [MAR 94]), caractérisée par une seule clause FROM contenant les tables et alias à mettre en jointure deux à deux. La syntaxe générale suivante décrit une jointure relationnelle :

```
SELECT [alias1.]col1, [alias2.]col2…
   FROM nomTable1 [alias1], nomTable2 [alias2]…
   WHERE (conditionsDeJointure);
```

Cette forme est la plus utilisée car elle est la plus simple à écrire. Un autre avantage de ce type de jointure est qu'elle laisse le soin au SGBD d'établir la meilleure stratégie d'accès (choix du premier index à utiliser, puis du deuxième, etc.) pour optimiser les performances.

Afin d'éviter les ambiguïtés concernant le nom des colonnes, on utilise en général des alias de tables pour suffixer les tables dans la clause FROM et préfixer les colonnes dans les clauses SELECT et WHERE.

Jointures SQL2

Afin de se rendre conforme à la norme SQL2, Oracle propose aussi des directives qui permettent de programmer d'une manière plus verbale les différents types de jointures :

```
SELECT [ { DISTINCT | UNIQUE } | ALL ] listeColonnes
 FROM nomTable1 [{ INNER | { LEFT | RIGHT | FULL } [OUTER] }]
      JOIN nomTable2{ ON condition | USING ( colonne1 [, colonne2]… ) }
  | { CROSS JOIN | NATURAL [{ INNER | { LEFT | RIGHT | FULL } [OUTER] }]
      JOIN nomTable2 } …
[ WHERE condition ];
```

Cette écriture est moins utilisée que la syntaxe relationnelle. Bien que plus concise pour des jointures à deux tables, elle se complique pour des jointures plus complexes.

Types de jointures

Bien que dans le vocabulaire courant, on ne parle que de « jointures » en fonction de la nature de l'opérateur utilisé dans la requête, de la clause de jointure et des tables concernées, on distingue :

- les jointures internes (*inner joins*).
- l'équijointure (*equi join*) est la plus connue, elle utilise l'opérateur d'égalité dans la clause de jointure. La jointure naturelle est conditionnée en plus par le nom des colonnes. La non équijointure utilise l'opérateur d'inégalité dans la clause de jointure.
- l'autojointure (*self join*) est un cas particulier de l'équijointure qui met en œuvre deux fois la même table (des alias de tables permettront de distinguer les enregistrements entre eux).
- la jointure externe (*outer join*), la plus compliquée, qui favorise une table (dite « dominante ») par rapport à l'autre (dite « subordonnée »). Les lignes de la table dominante sont retournées même si elles ne satisfont pas aux conditions de jointure.

Le tableau suivant illustre cette classification sous la forme de quelques conditions appliquées à notre exemple :

Tableau 4-33 Exemples de conditions

Type de jointure	Syntaxe de la condition
Équijointure	`WHERE comp = compa;`
Autojointure	`WHERE alias1.chefPil = alias2.brevet;`
Jointure externe	`WHERE comp= compa (+) ;`

Pour mettre trois tables *T1*, *T2* et *T3* en jointure, il faut utiliser deux clauses de jointures (une entre *T1* et *T2* et l'autre entre *T2* et *T3*). Pour *n* tables, il faut *n*–1 clauses de jointures. Si vous oubliez une clause de jointure, un produit cartésien restreint est composé.

Étudions à présent chaque type de jointure avec les syntaxes « relationnelle » et « SQL2 ».

Équijointure

Une équijointure utilise l'opérateur d'égalité dans la clause de jointure et compare généralement des clés primaires avec des clés étrangères.

En considérant les tables suivantes, les équijointures se programment soit sur les colonnes `comp` et `compa` soit sur les colonnes `brevet` et `chefPil`. Extrayons par exemple :

- l'identité des pilotes de la compagnie de nom 'Air France' ayant plus de 500 heures de vol (requête *R1*) ;
- les coordonnées des compagnies qui embauchent des pilotes de plus de 950 heures de vol (requête *R2*).

La jointure qui résoudra la première requête est illustrée dans la figure par les données grisées, tandis que la deuxième jointure est représentée par les données en gras.

Figure 4-11 *Équijointures*

Compagnie

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL
CAST	1	G. Brassens	Blagnac	Castanet AL

Pilote

brevet	nom	nbHVol	compa	chefPil
PL-1	Pierre Lamothe	450	AF	PL-4
PL-2	Didier Linxe	900	AF	PL-4
PL-3	Christian Soutou	1000	SING	
PL-4	Henri Alquié	3400	AF	

Écriture « relationnelle »

- Oracle recommande d'utiliser des alias de tables pour améliorer les performances.
- Les alias sont obligatoires pour des colonnes qui portent le même nom ou pour les autojointures.

Écriture « SQL2 »

- La clause `JOIN ... ON condition` permet de programmer une équijointure.
- L'utilisation de la directive `INNER` devant `JOIN...` est optionnelle et est appliquée par défaut.

Le tableau suivant détaille ces requêtes avec les deux syntaxes. Les clauses de jointures sont grisées.

Tableau 4-34 Exemples d'équijointures

Requête	Jointure relationnelle	Jointure SQL2

R1	`SELECT brevet, nom`
	`FROM Pilote, Compagnie`
	`WHERE comp = compa`
	`AND nomComp = 'Air France'`
	`AND nbHVol > 500;`
		`SELECT brevet, nom`
		`FROM Compagnie`
		`JOIN Pilote ON comp = compa`
		`WHERE nomComp = 'Air France'`
		`AND nbHVol > 500;`

	`BREVET NOM`
	`------ --------------------`
	`PL-4 Henri Alquié`
	`PL-2 Didier Linxe`	

R2	`SELECT cpg.nomComp, cpg.nrue,`
	`cpg.rue, cpg.ville`
	`FROM Pilote pil, Compagnie cpg`
	`WHERE cpg.comp = pil.compa`
	`AND pil.nbHVol > 950;`
		`SELECT nomComp, nrue, rue, ville`
		`FROM Compagnie`
		`JOIN Pilote ON comp = compa`
		`WHERE nbHVol > 950;`

	`NOMCOMP NRUE RUE VILLE`
	`-------------- ---------- -------------------- ---------------`
	`Air France 124 Port Royal Paris`
	`Singapore AL 7 Camparols Singapour`	

Autojointure

Cas particulier de l'équijointure, l'autojointure relie une table à elle-même.

Extrayons par exemple :

- l'identité des pilotes placés sous la responsabilité des pilotes de nom 'Alquié' (requête *R3*) ;
- la somme des heures de vol des pilotes placés sous la responsabilité des chefs pilotes de la compagnie de nom 'Air France' (requête *R4*).

Ces requêtes doivent être programmées à l'aide d'une autojointure car elles imposent de parcourir deux fois la table `Pilote` (examen de chaque pilote en le comparant à un autre). Les autojointures sont réalisées entre les colonnes `brevet` et `chefPil`.

La jointure de la première requête est illustrée dans la figure par les données surlignées en clair, tandis que la deuxième jointure est mise en valeur par les données surlignées en foncé.

Figure 4-12 Autojointures

Le tableau suivant détaille ces requêtes, les clauses d'autojointures sont surlignées. Dans les deux syntaxes, il est impératif d'utiliser des alias. Concernant l'écriture « SQL2 », les clauses `JOIN` peuvent s'imbriquer pour joindre plus de deux tables.

Tableau 4-35 Exemples d'autojointures

Requête	Jointure relationnelle	Jointure SQL2

R3	`SELECT p1.brevet, p1.nom`
	`FROM Pilote p1, Pilote p2`
	`WHERE p1.chefPil = p2.brevet`
	`AND p2.nom LIKE '%Alquié%';`
		`SELECT p1.brevet, p1.nom`
		`FROM Pilote p1`
		`JOIN Pilote p2`
		`ON p1.chefPil = p2.brevet`
		`WHERE p2.nom LIKE '%Alquié%';`

	`BREVET NOM`
	`------ --------------------`
	`PL-1 Pierre Lamothe`
	`PL-2 Didier Linxe`	

R4	`SELECT SUM(p1.nbHVol)`
	`FROM Pilote p1, Pilote p2,`
	`Compagnie cpg`
	`WHERE p1.chefPil = p2.brevet`
	`AND cpg.comp = p2.compa`
	`AND cpg.nomComp = 'Air France';`
		`SELECT SUM(p1.nbHVol)`
		`FROM Pilote p1`
		`JOIN Pilote p2`
		`ON p1.chefPil = p2.brevet`
		`JOIN Compagnie`
		`ON comp = p2.compa`
		`WHERE nomComp = 'Air France';`

	`SUM(P1.NBHVOL)`
	`---------------`
	`1350`	

Inéquijointure

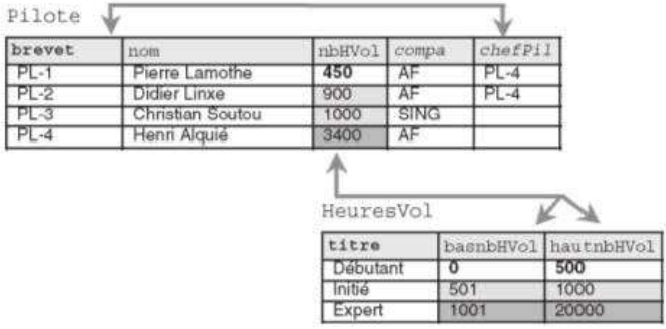
Les requêtes d'inéquijointures font intervenir tout type d'opérateur (<>, >, <, >=, <=, `BETWEEN`, `LIKE`, `IN`). À l'inverse des équijointures, la clause d'une inéquijointure n'est pas basée sur l'égalité de clés primaires (ou candidates) et de clés étrangères.

En considérant les tables suivantes, extrayons par exemple :

- les pilotes ayant plus d'expérience que le pilote de numéro de brevet 'PL-2' (requête *R5*).
- le titre de qualification des pilotes en raisonnant sur la comparaison des heures de vol avec un ensemble de références, ici la table `HeuresVol` (requête *R6*). Dans notre exemple, il s'agit par exemple de retrouver le fait que le premier pilote est débutant.

La jointure qui résoudra la deuxième requête est illustrée par les niveaux de gris.

Figure 4-13 *Inéquijointures*



Le tableau suivant détaille ces requêtes, les clauses d'inéquijointures sont surlignées :

Tableau 4-36 Exemples d'inéquijointures

Requête	Jointure relationnelle	Jointure SQL2
R5	`SELECT p1.brevet, p1.nom, p1.nbHVol, p2.nbHVol "Référence" FROM Pilote p1, Pilote p2 WHERE p1.nbHVol > p2.nbHVol AND p2.brevet = 'PL-2';`	`SELECT p1.brevet, p1.nom, p1.nbHVol, p2.nbHVol "Référence" FROM Pilote p1 JOIN Pilote p2 ON p1.nbHVol>p2.nbHVol WHERE p2.brevet = 'PL-2';`

	`BREVET NOM NBHVOL Référence`
	`PL-4 Henri Alquié 3400 900`
	`PL-3 Christian Soutou 1000 900`	

R6	`SELECT pil.brevet, pil.nom, pil.nbHVol, hv.titre FROM Pilote pil, HeuresVol hv WHERE pil.nbHVol BETWEEN hv.basnbHVol AND hv.hautnbHVol;`	`SELECT brevet, nom, nbHVol, titre FROM Pilote JOIN HeuresVol ON nbHVol BETWEEN basnbHVol AND hautnbHVol;`

	`BREVET NOM NBHVOL TITRE`
	`PL-1 Pierre Lamothe 450 Débutant`
	`PL-2 Didier Linxe 900 Initié`
	`PL-3 Christian Soutou 1000 Initié`
	`PL-4 Henri Alquié 3400 Expert`	

Jointures externes

Les jointures externes permettent d'extraire des enregistrements qui ne répondent pas aux critères de jointure. Lorsque deux tables sont en jointure externe, une table est « dominante »

par rapport à l'autre (qui est dite « subordonnée »). Ce sont les enregistrements de la table dominante qui sont retournés (même si les valeurs des colonnes des tables subordonnées ne satisfont pas aux conditions de jointure ou sont nulles).

Comme les jointures internes, les jointures externes sont généralement basées sur les clés primaires et étrangères. On distingue les jointures unilatérales qui considèrent une table dominante et une table subordonnée, et les jointures bilatérales pour lesquelles les tables jouent un rôle symétrique (pas de dominant).

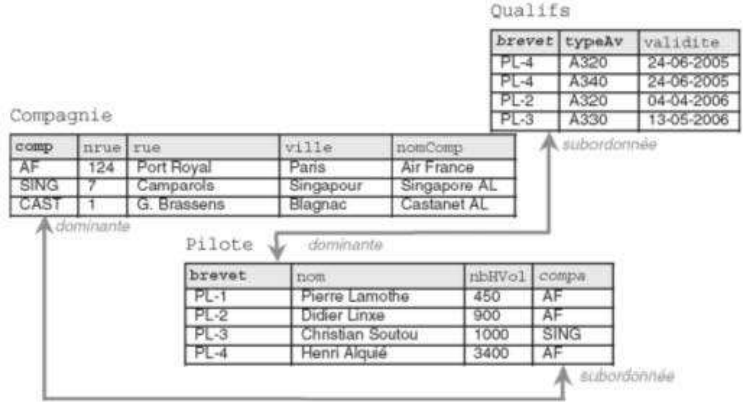
Jointures unilatérales

En considérant les tables suivantes, une jointure externe unilatérale permet d'extraire :

- la liste des compagnies et leurs pilotes, même les compagnies n'ayant pas de pilote (requête *R7*). Sans une jointure externe, la compagnie 'CAST' ne peut être extraite ;
- la liste des pilotes et leurs qualifications, même les pilotes n'ayant pas encore de qualification (requête *R8*).

La figure illustre les tables dominantes et subordonnées :

Figure 4-14 *Jointures externes unilatérales*



Écriture « relationnelle »

- La directive de jointure externe « `(+)` » se place du côté de la table subordonnée.
- Cette directive peut se placer à gauche ou à droite d'une clause de jointure, pas des deux côtés.
- Une clause de jointure externe ne peut ni utiliser l'opérateur `IN` ni être associée à une autre condition par l'opérateur `OR`.

Écriture « SQL2 »

Le sens de la directive de jointure externe `LEFT` ou `RIGHT` de la clause `OUTER JOIN` désigne la table dominante.

Le tableau suivant détaille les requêtes de notre exemple, les clauses de jointures externes unilatérales sont grisées. Les tables dominantes sont notées en gras (`Compagnie` pour la première requête et `Pilote` pour la deuxième).

Tableau 4-37 Écritures équivalentes de jointures externes unilatérales

Requête	Jointures relationnelles	Jointures SQL2

R7	`SELECT cpg.nomComp, pil.brevet, pil.nom`
	` FROM Pilote pil, **Compagnie** cpg`
	` WHERE cpg.comp = pil.compa(+);`
	`--équivalent à`
	` WHERE pil.compa(+) = cpg.comp;`
		`SELECT nomComp, brevet, nom`
		` FROM **Compagnie** LEFT OUTER JOIN`
		`  Pilote ON comp = compa;`
		`--équivalent à`
		`SELECT nomComp, brevet, nom`
		` FROM Pilote RIGHT OUTER JOIN`
		`  **Compagnie** ON comp = compa;`

	`NOMCOMP BREVET NOM`
	`--------------- ------ --------------------`
	`Air France PL-4 Henri Alquié`
	`Air France PL-1 Pierre Lamothe`
	`Air France PL-2 Didier Linxe`
	`Singapore AL PL-3 Christian Soutou`
	`Castanet AL`	

R8	`SELECT qua.typeAv, pil.brevet, pil.nom`
	` FROM **Pilote** pil, Qualifs qua`
	` WHERE qua.brevet(+)=pil.brevet;`
	`--équivalent à`
	` WHERE pil.brevet=qua.brevet(+);`
		`SELECT qua.typeAv, pil.brevet, pil.nom`
		` FROM Qualifs qua`
		`  RIGHT OUTER JOIN **Pilote** pil`
		`  ON pil.brevet = qua.brevet;`
		`--équivalent à`
		`SELECT qua.typeAv, pil.brevet, pil.nom`
		` FROM **Pilote** pil`
		`  LEFT OUTER JOIN Qualifs qua`
		`  ON pil.brevet = qua.brevet;`

	`TYPE BREVET NOM`
	`---- ------ --------------------`
	`A320 PL-4 Henri Alquié`
	`A340 PL-4 Henri Alquié`
	`A320 PL-2 Didier Linxe`
	`A330 PL-3 Christian Soutou`
	`PL-1 Pierre Lamothe`	

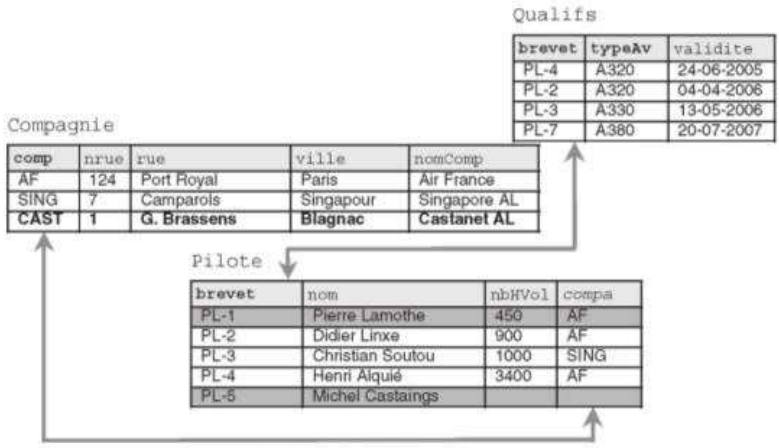
Jointures bilatérales

Les deux tables jouent un rôle symétrique, il n'y a pas de table dominante. Ce type de jointure permet d'extraire des enregistrements qui ne répondent pas aux critères de jointure des deux côtés de la clause de jointure.

En considérant les tables suivantes, une jointure externe bilatérale permet d'extraire par exemple :

- la liste des compagnies et leurs pilotes, incluant les compagnies n'ayant pas de pilote et les pilotes rattachés à aucune compagnie (requête *R9*) ;
- la liste des pilotes et leurs qualifications, incluant les pilotes n'ayant pas encore d'expérience et les qualifications associées à des pilotes inconnus (requête *R10*).

Figure 4-15 *Jointures externes bilatérales*



Écriture « relationnelle »

La jointure externe bilatérale se programme en faisant l'union de deux jointures externes unilatérales, en plaçant alternativement le symbole « `(+)` ».

Écriture « SQL2 »

La directive `FULL OUTER JOIN` permet d'ignorer l'ordre (et donc le sens de la jointure) des tables dans la requête.

Le tableau suivant détaille les requêtes de notre exemple, les clauses de jointures externes bilatérales sont surlignées. Les enregistrements qui ne respectent pas la condition de jointure sont surlignés.

Tableau 4-38 Jointures externes bilatérales

Requête	Jointures relationnelles	Jointures SQL2
R9	`SELECT cpg.nomComp, pil.brevet, pil.nom FROM Pilote pil, Compagnie cpg WHERE cpg.comp(+) = pil.compa UNION SELECT cpg.nomComp, pil.brevet, pil.nom FROM Pilote pil, Compagnie cpg WHERE cpg.comp = pil.compa(+);`	`SELECT nomComp, brevet, nom FROM Pilote FULL OUTER JOIN Compagnie ON comp = compa; --équivalent à SELECT nomComp, brevet, nom FROM Compagnie FULL OUTER JOIN Pilote ON comp = compa;`

	`NOMCOMP BREVET NOM`
	`Air France PL-4 Henri Alquié`
	`Air France PL-1 Pierre Lamothe`
	`Air France PL-2 Didier Linxe`
	`Singapore AL PL-3 Christian Soutou`
	`Castanet AL`
	`PL-5 Michel Castaings`	

R10	`SELECT qua.typeAv, pil.brevet, pil.nom FROM Pilote pil, Qualifs qua WHERE qua.brevet(+)=pil.brevet UNION SELECT qua.typeAv, pil.brevet, pil.nom FROM Pilote pil, Qualifs qua WHERE qua.brevet=pil.brevet(+);`	`SELECT qua.typeAv, pil.brevet, pil.nom FROM Pilote pil FULL OUTER JOIN Qualifs qua ON pil.brevet = qua.brevet; --équivalent à SELECT qua.typeAv, pil.brevet, pil.nom FROM Qualifs qua FULL OUTER JOIN Pilote pil ON pil.brevet = qua.brevet;`

	`TYPE BREVET NOM`
	`A320 PL-4 Henri Alquié`
	`A320 PL-2 Didier Linxe`
	`A330 PL-3 Christian Soutou`
	`A380`
	`PL-1 Pierre Lamothe`
	`PL-5 Michel Castaings`	

Jointures procédurales

Les jointures procédurales sont écrites par des requêtes qui contiennent des sous-interrogations (SELECT imbriqué). Chaque clause FROM ne contient qu'une seule table.

```
SELECT colonnesTable1
    FROM nomTable1
    WHERE colonne(s) | expression(s) { IN | = | opérateur }
       (SELECT colonne(s)delaTable2 FROM nomTable2
         WHERE colonne(s) | expression(s) { IN | = | opérateur }
           (SELECT …)
         [AND (conditionsTable2)]
       )
         [AND (conditionsTable1)];
```

Cette forme d'écriture n'est pas la plus utilisée mais elle permet de mieux visualiser certaines jointures. Elle est plus complexe à écrire, car l'ordre d'apparition des tables dans les clauses FROM a son importance.

Seules les colonnes de la table qui se trouve au niveau du premier SELECT peuvent être extraites.

La sous-interrogation doit être placée entre parenthèses. Elle ne doit pas comporter de clause ORDER BY mais peut inclure GROUP BY et HAVING.

Le résultat d'une sous-interrogation est utilisé par la requête de niveau supérieur. Une sous-interrogation est exécutée avant la requête de niveau supérieur.

Une sous-interrogation peut ramener une ou plusieurs lignes. Les opérateurs =, >, <, >=, <= permettent d'en extraire une, les opérateurs IN, ANY et ALL permettent d'en ramener plusieurs.

Sous-interrogations monolignes

Le tableau suivant détaille quelques sous-interrogations monolignes. Nous nous basons sur certaines requêtes déjà étudiées (forme relationnelle et SQL2).

Tableau 4-39 Sous-interrogations monolignes

Opérateur	Besoin	Requête
= pour les équijointures ou autojointures (= teste une ligne)	*R1* (Pilotes de la compagnie de nom 'Air France' ayant plus de 500 heures de vol.)	`SELECT brevet, nom FROM Pilote WHERE compa = (SELECT comp FROM Compagnie WHERE nomComp = 'Air France') AND nbHVol>500;`
	R3 (Pilotes sous la responsabilité du pilote de nom 'Alquié'.)	`SELECT brevet, nom FROM Pilote WHERE chefPil = (SELECT brevet FROM Pilote WHERE nom LIKE '%Alquié%');`
> pour les inéquijointures	*R5* (Pilotes ayant plus d'expérience que le pilote de brevet 'PL-2'.)	`SELECT brevet, nom, nbHVol FROM Pilote WHERE nbHVol > (SELECT nbHVol FROM Pilote WHERE brevet = 'PL-2');`

Sous-interrogations multilignes (IN, ALL et ANY)

Les opérateurs multilignes sont les suivants :

- `IN` compare un élément à une donnée quelconque d'une liste ramenée par la sous-interrogation. Cet opérateur est utilisé pour les équijointures ou autojointures. L'opérateur `NOT IN` sera employé pour les jointures externes.
- `ANY` compare l'élément à chaque donnée ramenée par la sous-interrogation. L'opérateur « `=ANY` » équivaut à `IN`. L'opérateur « `<ANY` » signifie « inférieur à au moins une des valeurs » donc « inférieur au maximum ». L'opérateur « `>ANY` » signifie « supérieur à au moins une des valeurs » donc « supérieur au minimum ».
- `ALL` compare l'élément à tous ceux ramenés par la sous-interrogation. L'opérateur « `<ALL` » signifie « inférieur au minimum » et « `>ALL` » signifie « supérieur au maximum ».

Le tableau suivant détaille quelques sous-interrogations multilignes. Le dernier exemple programme une partie d'une jointure externe.

La directive `NOT IN` doit être utilisée avec prudence car elle retourne `FALSE` si un membre ramené par la sous-interrogation est `NULL`.

Tableau 4-40 Sous-interrogations multilignes

Opérateur	Besoin	Requête
IN	*R2.* Coordonnées des compagnies qui embauchent des pilotes de plus de 950 heures de vol.	`SELECT nomComp, nrue, rue, ville FROM Compagnie WHERE comp IN (SELECT compa FROM Pilote WHERE nbHVol>950);`
= et IN	*R4.* Somme des heures de vol des pilotes placés sous la responsabilité des chefs pilotes de la compagnie de nom 'Air France'.	`SELECT SUM(nbHVol) FROM Pilote WHERE chefPil IN (SELECT brevet FROM Pilote WHERE compa = (SELECT comp FROM Compagnie WHERE nomComp = 'Air France'));`
NOT IN	Compagnies n'ayant pas de pilote.	`SELECT nomComp, nrue, rue, ville FROM Compagnie WHERE comp NOT IN (SELECT compa FROM Pilote WHERE compa IS NOT NULL);`

Pour illustrer les opérateurs ANY et ALL, considérons la table suivante. Nous avons indiqué en gras les nombres d'heures minimal et maximal des A320, en grisé les nombres d'heures minimal et maximal des avions de la compagnie 'AF'.

Figure 4-16 *Table Avion*

Avions

immat	typeAv	nbHVol	compa
A1	A320	**1000**	AF
A2	A330	1500	AF
A3	A320	**550**	SING
A4	A340	1800	SING
A5	A340	200	AF
A6	A330	100	AF

Le tableau suivant détaille quelques jointures procédurales utilisant les opérateurs ALL et ANY.

Tableau 4-41 Opérateurs ALL et ANY

Opérateur	Besoin	Requête

ANY	*R11.* Avions dont le nombre d'heures de vol est inférieur à celui de n'importe quel A320.	`SELECT immat, typeAv, nbHVol`
		` FROM Avion`
		` WHERE nbHVol < ANY`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE typeAv='A320');`
		
		`IMMAT TYPE NBHVOL`
		`------ ---- ----------`
		`A3 A320 550`
		`A5 A340 200`
		`A6 A330 100`

	R12. Compagnies et leurs avions dont le nombre d'heures de vol est supérieur à celui de n'importe quel avion de la compagnie de code 'SING'.	`SELECT immat, typeAv, nbHVol, compa`
		` FROM Avion`
		` WHERE nbHVol > ANY`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE compa = 'SING');`
		
		`IMMAT TYPE NBHVOL COMP`
		`------ ---- ---------- ----`
		`A1 A320 1000 AF`
		`A2 A330 1500 AF`
		`A4 A340 1800 SING`

ALL	*R13.* Avions dont le nombre d'heures de vol est inférieur à tous les A320.	`SELECT immat, typeAv, nbHVol`
		` FROM Avion`
		` WHERE nbHVol < ALL`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE typeAv='A320');`
		
		`IMMAT TYPE NBHVOL`
		`------ ---- ----------`
		`A5 A340 200`
		`A6 A330 100`

	R14. Compagnies et leurs avions dont le nombre d'heures de vol est supérieur à tous les avions de la compagnie de code 'AF'.	`SELECT immat, typeAv, nbHVol, compa`
		` FROM Avion`
		` WHERE nbHVol > ALL`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE compa = 'AF');`
		
		`IMMAT TYPE NBHVOL COMP`
		`------ ---- ---------- ----`
		`A4 A340 1800 SING`

Jointures mixtes

Si vous avez besoin de combiner dans une requête des clauses de jointures de la forme relationnelle, des sous-interrogations dans le FROM ou WHERE, ou d'utiliser conjointement des directives SQL2 (INNER JOIN, OUTER JOIN, etc.), vous écrivez une jointure dite mixte.

La requête suivante combine une jointure relationnelle (**en gras**) avec une jointure procédurale (surlignée) pour extraire la somme des heures de vol des pilotes placés sous la responsabilité des chefs pilotes de la compagnie Air France (requête *R4*).

```
SELECT SUM(p1.nbHVol)
   FROM Pilote p1, Pilote p2
   WHERE p1.chefPil = p2.brevet
   AND   p2.compa = (SELECT comp FROM Compagnie WHERE nomComp =
   'Air France') ;
```

Ce type d'écriture peut être intéressant s'il n'est pas nécessaire d'afficher des colonnes des tables présentes dans les sous-interrogations ou si l'on désire appliquer des fonctions à des regroupements.

Sous-interrogations synchronisées

Une sous-interrogation est synchronisée si elle manipule des colonnes d'une table du niveau supérieur (on parle de requête imbriquée).

Une sous-interrogation synchronisée est exécutée une fois pour chaque enregistrement extrait par la requête de niveau supérieur. Cette technique peut être aussi utilisée dans les ordres UPDATE et DELETE.

La forme générale d'une sous-interrogation synchronisée est la suivante. Les alias des tables sont utiles pour pouvoir manipuler des colonnes de tables de différents niveaux.

```
SELECT alias1.c
  FROM nomTable1 alias1 <----------------------
  WHERE colonne(s) opérateur (SELECT alias2.x…
                              FROM nomTable2 alias2
                              WHERE alias1.x opérateur alias2.y)
  [AND (conditionsTable1)];
```

Une sous-interrogation synchronisée peut ramener une ou plusieurs lignes. Différents opérateurs peuvent être employés (=, >, <, >=, <=, EXISTS).

Opérateur mathématique

Le tableau suivant détaille un exemple d'opérateur mathématique appliqué à une sous-interrogation synchronisée.

Tableau 4-42 Sous-interrogation synchronisée

Besoin	Requête

R15. Avions dont le nombre d'heures de vol est supérieur au nombre d'heures de vol moyen des avions de leur compagnie (ici 700 h pour 'AF' et 1 115 h pour 'SING').	`SELECT avi1.*`
	`FROM Avion avi1`
	`WHERE avi1.nbHVol >`
	`(SELECT AVG(avi2.nbHVol) FROM Avion avi2`
	`WHERE avi2.compa = avi1.compa);`
	
	`IMMAT TYPE NBHVOL COMP`
	`------ ---- ---------- ----`
	`A1 A320 1000 AF`
	`A2 A330 1500 AF`
	`A4 A340 1800 SING`

Opérateur EXISTS

L'opérateur `EXISTS` permet d'interrompre la sous-interrogation dès le premier enregistrement trouvé. La valeur `FALSE` est retournée si aucun enregistrement n'est extrait par la sous-interrogation.

Utilisons la table suivante pour décrire l'utilisation de l'opérateur `EXISTS` :

Figure 4-17 *Utilisation de EXISTS*

Pilote

brevet	nom	nbHVol	compa	chefPil
PL-1	Pierre Larnothe	450	AF	
PL-2	Didier Linxe	900	AF	PL-4
PL-3	Christian Soutou	1000	SING	PL-4
PL-4	Henri Alquié	3400	AF	
PL-5	Michel Castaings			

La sous-interrogation synchronisée est surlignée dans le script suivant :

Tableau 4-43 Opérateur EXISTS

Besoin	Requête

R16. Pilotes ayant au moins un pilote sous leur responsabilité.	`SELECT pil1.brevet, pil1.nom, pil1.compa`
	`FROM Pilote pil1`
	`WHERE EXISTS`
	`(SELECT pil2.* FROM Pilote pil2`
	`WHERE pil2.chefPil = pil1.brevet);`
	
	`BREVET NOM COMP`
	`------ -------------------- ----`
	`PL-4 Henri Alquié AF`

Opérateur NOT EXISTS

L'opérateur `NOT EXISTS` retourne la valeur `TRUE` si aucun enregistrement n'est extrait par la sous-interrogation. Cet opérateur peut être utilisé pour écrire des jointures externes.

Tableau 4-44 Opérateur NOT EXISTS

Besoin	Requête

Liste des compagnies n'ayant pas de pilote.	`SELECT cpg.*`
	`FROM Compagnie cpg`
	`WHERE NOT EXISTS`
	`(SELECT compa FROM Pilote WHERE compa = cpg.comp);`
	
	`COMP NRUE RUE VILLE NOMCOMP`
	`---- ----- ---------- ---------- ----------`
	`CAST 1 G. Brassens Blagnac Castanet AL`

Autres directives SQL2

Étudions enfin les autres options des jointures SQL2 (`NATURAL JOIN`, `USING` et `CROSS JOIN`).

Considérons le schéma suivant (des colonnes portent le même nom). La colonne `typeAv` dans la table `Navigant` désigne le type d'appareil sur lequel le pilote est instructeur.

Figure 4-18 *Deux tables à mettre en jointure naturelle*

VolsControle

brevet	typeAv	validite
PL-1	A320	24-06-2005
PL-2	A320	04-04-2006
PL-2	A330	13-05-2006
PL-3	A380	20-07-2007
PL-3	A320	12-03-2005

Navigant

brevet	nom	nbHVol	typeAv
PL-1	Pierre Lamothe	450	
PL-2	Didier Linxe	900	A320
PL-3	Henri Alquié	3400	A380

Opérateur NATURAL JOIN

La jointure naturelle est programmée par la clause `NATURAL JOIN`. La clause de jointure est automatiquement construite sur la base de toutes les colonnes portant le même nom entre les deux tables.

Les concepteurs doivent donc penser à nommer d'une manière semblable clés primaires et clés étrangères. Ce principe n'est pas souvent appliqué aux schémas volumineux.

Le tableau suivant détaille deux écritures possibles d'une jointure naturelle. La clause de jointure est basée sur les colonnes (`brevet`, `typeAv`). Une clause `WHERE` aurait pu aussi être progammée.

Tableau 4-45 Jointures naturelles

Besoin	Jointures SQL2

Navigants qualifiés sur un type d'appareil et instructeurs sur ce même type.	`SELECT brevet, nom, typeAv, validite`
	`FROM Navigant NATURAL JOIN VolsControle;`
	`--équivalent à`
	`SELECT brevet, nom, typeAv, validite`
	`FROM VolsControle NATURAL JOIN Navigant;`

```
BREVET NOM                  TYPEAV VALIDITE
------ -------------------- ------ --------
PL-2   Didier Linxe         A320   04/04/06
PL-3   Henri Alquié         A380   20/07/07
```

Opérateur USING

La directive `USING(col1, col2…)` de la clause `JOIN` programme une jointure naturelle restreinte à un ensemble de colonnes. Il ne faut pas utiliser d'alias de tables dans la liste des colonnes.

Dans notre exemple, on peut restreindre la jointure naturelle aux colonnes `brevet` ou `typeAv`. Si on les positionnait (`brevet`, `typeAv`) dans la directive `USING` cela reviendrait à construire un `NATURAL JOIN`. Le tableau suivant détaille deux écritures d'une jointure naturelle restreinte :

Tableau 4-46 Jointures naturelles restreintes

Besoin	Jointures SQL2

Nom des navigants avec leurs qualifications et dates de validité.	`SELECT nom, v.typeAv, v.validite`
	`FROM Navigant`
	`JOIN VolsControle v USING(brevet);`
	
	`SELECT nom, v.typeAv, v.validite`
	`FROM VolsControle v`
	`JOIN Navigant USING(brevet) ;`

```
NOM                  TYPEAV VALIDITE
-------------------- ------ --------
Pierre Lamothe       A320   24/06/05
Didier Linxe         A320   04/04/06
Didier Linxe         A330   13/05/06
Henri Alquié         A380   20/07/07
Henri Alquié         A320   12/03/05
```

Opérateur CROSS JOIN

La directive `CROSS JOIN` programme un produit cartésien qu'on peut restreindre dans la clause `WHERE`.

Le tableau suivant présente deux écritures d'un produit cartésien (seul l'ordre d'affichage des colonnes change) :

Tableau 4-47 Produit cartésien

Besoin	Jointures SQL2

Combinaison de toutes les lignes des deux tables.	`SELECT *`
	`FROM Navigant CROSS JOIN VolsControle;`
	`--équivalent à`
	`SELECT *`
	`FROM VolsControle CROSS JOIN Navigant;`

```
BREVET NOM                  NBHVOL TYPEAV BREVET TYPEAV VALIDITE
------ -------------------- ------ ------ ------ ------ --------
PL-1   Pierre Lamothe          450        PL-1   A320   24/06/05
PL-2   Didier Linxe            900 A320   PL-1   A320   24/06/05
PL-3   Henri Alquié           3400 A380   PL-1   A320   24/06/05
PL-1   Pierre Lamothe          450        PL-2   A320   04/04/06
… 15 ligne(s) sélectionnée(s).
```

Division

La division est un opérateur algébrique et non ensembliste. Cet opérateur est semblable, sur le principe, à l'opération qu'on apprend au CE2 et qu'on a oubliée en terminale à cause des calculettes. La division est un opérateur binaire comme la jointure car il s'agit de diviser une table (ou partie de) par une autre table (ou partie de). Il est possible d'opérer une division à partir d'une seule table, en ce cas on divise deux parties de cette table (analogue aux auto-jointures).

L'opérateur de division n'est pas fourni par Oracle (ni par ses concurrents d'ailleurs). Il n'existe donc malheureusement pas d'instruction `DIVIDE`.

Est-ce la complexité ou le manque d'intérêt qui freinent les éditeurs de logiciels à programmer ce concept ? La question reste en suspens, alors si vous avez un avis à ce sujet, faites-moi signe !

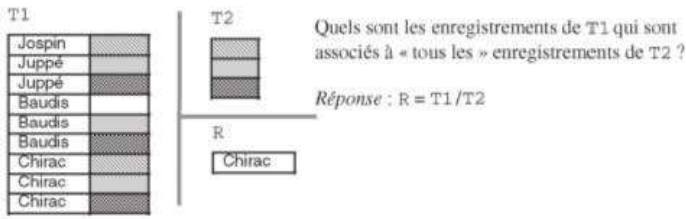
Cet opérateur permet de traduire le terme « pour tous les » des requêtes qu'on désire programmer en SQL.

On peut aussi dire que lorsque vous voulez comparer un ensemble avec un groupe de référence, il faut programmer une division.

La division se traduit sous SQL par l'opérateur ensembliste `MINUS` et la fonction `NOT EXISTS`.

La figure suivante illustre l'opérateur de division dans sa plus simple expression (nous ne parlons pas du contenu des tables bien sûr…). Le schéma fait davantage apparaître le deuxième aspect révélateur énoncé ci-dessus, à savoir comparer un ensemble (la table *T1*) avec un ensemble de référence (la table *T2*).

Figure 4-19 *Division*



Définition

La division de la table *T1[a1,…,an,b1,…,bn]* par la table *T2[b1,…,bn]* (la structure de *T2* est incluse dans la structure de *T1*) donne la table *T3[a1,…,an]* qui contient les enregistrements *ti* vérifiant *ti* ∈ T3 (de structure *[a1,…,an]*), *tj* ∈ *T2* (*tj* de structure *[b1,…,bn]*) et *ti,tj* ∈ *T1* (*ti,tj* de structure *[a1,…,an,b1,…,bn]*).

Classification

Considérons l'exemple suivant pour décrire la requête à construire. Il s'agit de répondre à la question « Quels sont les avions affrétés par **toutes** les compagnies françaises ? » L'ensemble de référence (*A*) est constitué des codes des compagnies françaises. L'ensemble à comparer (*B*) est constitué des codes des compagnies pour chaque avion.

Deux cas sont à envisager suivant la manière de comparer les deux ensembles :

- Division inexacte : un ensemble est seulement inclus dans un autre (*A inclus dans B*). La question à programmer serait « Quels sont les avions affrétés par **toutes** les compagnies

françaises ? » sans préciser si les avions ne doivent pas être aussi affrétés par des compagnies étrangères. L'avion (A3, Mercure) répondrait à cette question, que la dernière ligne de la table `Affrètements` soit présente ou pas.

- Division exacte : les deux ensembles sont égaux (B=A). La question à programmer serait « Quels sont les avions affrétés **exactement** (ou **uniquement**) par toutes les compagnies françaises ? » L'avion (A3, Mercure) répondrait à cette question si la dernière ligne de la table `Affrètements` est inexistante. Les lignes concernées dans les deux tables sont grisées.

Figure 4-20 *Divisions à programmer*

`Affrètements`

immat	typeAv	compa	dateAff
A1	A320	SING	13-05-1995
A2	A340	AF	22-06-1968
A3	Mercure	AF	05-02-1965
A4	A330	ALIB	16-01-1965
A3	Mercure	ALIB	05-03-1942
A3	Mercure	SING	01-03-1987

`Compagnie`

comp	nomComp	pays
AF	Air France	F
ALIB	Air Lib	F
SING	Singapore AL	SG

`Résultat`

immat	typeAv
A3	Mercure

L'opérateur ensembliste `MINUS` combiné à la fonction `EXISTS` permet de programmer ces deux comparaisons (un ensemble inclus dans un autre et une égalité d'ensembles). Il existe d'autres solutions à base de regroupements et de sous-interrogations (synchronisées ou pas) que nous n'étudierons pas, parce qu'elles semblent plus compliquées. Écrivons à présent ces deux divisions à l'aide de requêtes SQL.

Division inexacte en SQL

Pour programmer le fait qu'un ensemble est seulement inclus dans un autre (ici $A \subset B$), il faut qu'il n'existe pas d'élément dans l'ensemble $\{A\text{-}B\}$. La différence se programme à l'aide de l'opérateur `MINUS`, l'inexistence d'élément se programme à l'aide de la fonction `NOT EXISTS` comme le montre la requête suivante :

```
SELECT DISTINCT immat, typeAv FROM Affrètements aliasAff
  WHERE NOT EXISTS
   (SELECT comp FROM Compagnie WHERE pays = 'F'
    MINUS
    SELECT compa FROM Affrètements WHERE immat = aliasAff.immat) ;
```

Parcours de tous les avions

Ensemble A de référence

Ensemble B à comparer

Division exacte en SQL

Pour programmer le fait qu'un ensemble est strictement égal à un autre (ici $A=B$), il faut qu'il n'existe ni d'élément dans l'ensemble $\{A\text{-}B\}$ ni dans l'ensemble $\{B\text{-}A\}$. La traduction mathématique est la suivante : $A=B \Leftrightarrow (A\text{-}B=\emptyset$ et $B\text{-}A=\emptyset)$. Les opérateurs se programment de la même manière que pour la requête précédente. Le « et » se programme avec un `AND` (*of course*).

Parcours de tous les avions

```
SELECT DISTINCT immat, typeAv FROM Affrètements aliasAff
  WHERE NOT EXISTS
    (SELECT comp FROM Compagnie WHERE pays = 'F'                    A – B
    MINUS
    SELECT compa FROM Affrètements WHERE immat = aliasAff.immat)
  AND NOT EXISTS
    (SELECT compa FROM Affrètements WHERE immat = aliasAff.immat
    MINUS                                                           B – A
    SELECT comp FROM Compagnie WHERE pays = 'F');
```

Requêtes hiérarchiques

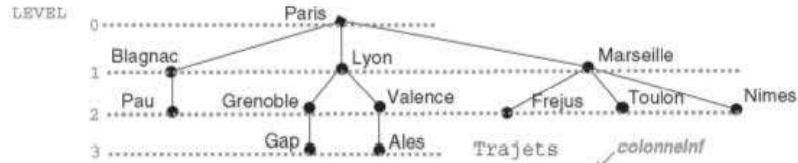
Les requêtes hiérarchiques extraient des données provenant d'une structure arborescente. Les enregistrements d'une structure arborescente appartiennent, en général, à la même table et sont reliés entre eux par une association réflexive à plusieurs niveaux.

L'exemple décrit un arbre qui comprend trois niveaux. La table `Trajets` décrit cet arbre. Des deux colonnes qui assurent l'association, il est facile de distinguer celle qui désigne l'élément supérieur (*colonneSup* ici `départ`) de celle qui désigne un élément inférieur (*colonneInf* ici `arrivée`).

La syntaxe générale d'une requête hiérarchique est la suivante. La pseudo-colonne `LEVEL` désigne le niveau de l'arbre par rapport à une racine donnée.

```
SELECT [LEVEL,] colonne, expression…
   FROM nomTable
   [WHERE condition]
   [START WITH condition]
   CONNECT BY PRIOR condition;
```

Figure 4-21 Arbre représenté par une table



Trajets

colonneInf

colonneSup

départ	arrivée	tempsVol
Paris	Blagnac	1
Paris	Lyon	0.8
Paris	Marseille	0.9
Blagnac	Pau	0.4
Lyon	Grenoble	0.3
Lyon	Valence	0.2
Grenoble	Gap	0.35
Valence	Ales	0.25
Marseille	Frejus	0.2
Marseille	Toulon	0.15
Marseille	Nimes	0.35

Point de départ du parcours (START WITH)

Le point de départ est spécifié par la directive `START WITH`. Ce n'est pas forcément la racine la plus haute de la hiérarchie.

Dans notre exemple, si on désire parcourir l'arbre en partant de la ville de Lyon, on utilisera « `START WITH départ='Lyon'` ».

Si la directive `START WITH` est omise, tous les enregistrements sont considérés comme des racines et le résultat devra être interprété comme un ensemble d'arbres.

Parcours de l'arbre (CONNECT BY PRIOR)

Il faut indiquer dans la directive `CONNECT BY` la clause de connexion qui contient les colonnes de jointure (*colonneSup* et *colonneInf*). Celles-ci peuvent être composées. Le parcours de l'arbre est le suivant :

- du bas vers le haut avec la directive `CONNECT BY PRIOR` *`colonneSup=colonneInf ;`*
- du haut vers le bas avec la directive `CONNECT BY PRIOR` *`colonneInf=colonneSup`*.

Nous verrons plus tard que la directive PRIOR permet également d'éliminer des arborescences entières du parcours.

Le tableau suivant détaille les chemins dans les deux sens de notre arbre. Les requêtes contiennent des clauses hiérarchiques (en surligné) et des clauses de connexions (en gras).

Tableau 4-48 Requêtes hiérarchiques

Besoin	Requête et résultat

Parcours de l'arbre de bas en haut en partant de la ville de Paris.	`SELECT LEVEL, arrivée, départ, tempsVol`
	`FROM Trajets`
	`START WITH départ = 'Paris'`
	`CONNECT BY PRIOR` **`départ = arrivée`**`;`
	
	`LEVEL ARRIVÉE DÉPART TEMPSVOL`
	`--------- ---------- ---------- ----------`
	`1 Blagnac Paris 1`
	`1 Lyon Paris ,8`
	`1 Marseille Paris ,9`

Parcours de l'arbre de haut en bas en partant de la ville de Lyon.	`SELECT LEVEL, départ, arrivée, tempsVol`
	`FROM Trajets`
	`START WITH départ = 'Lyon'`
	`CONNECT BY PRIOR` **`arrivée = départ`**`;`
	
	`LEVEL DÉPART ARRIVÉE TEMPSVOL`
	`-------- ---------- ---------- ----------`
	`1 Lyon Grenoble ,3`
	`2 Grenoble Gap ,35`
	`1 Lyon Valence ,2`
	`2 Valence Ales ,25`

Indentation

Pour composer un état de sortie indenté (comme pour un programme dans lequel vous indentez vos blocs dans un souci de lisibilité) en fonction du parcours de l'arbre, il faut utiliser plusieurs mécanismes :

- la pseudo-colonne LEVEL qui retourne le numéro du niveau courant de chaque enregistrement ;
- la fonction LPAD insère à gauche une expression des caractères ;
- la directive COLUMN (mise en forme du nom et de la taille des colonnes dans l'interface SQL*Plus) permet de substituer un libellé à une colonne, à l'affichage.

La requête suivante décale à gauche de quatre espaces les affichages pour chaque niveau (le premier niveau n'est pas décalé, le deuxième l'est de quatre espaces, etc.). La concaténation de ce décalage avec la colonne `arrivée` est renommée dans une variable (`DepartParis`), déclarée ici, de quinze caractères.

Tableau 4-49 Requête hiérarchique (indentation)

Besoin	Requête et résultat sous SQL*Plus

Parcours de l'arbre en entier de haut en bas en partant de la ville de Paris.	`COLUMN DepartParis FORMAT A15`
	`SELECT LPAD(' ',4*LEVEL-4) || arrivée`
	`DepartParis, tempsVol`
	`FROM Trajets`
	`START WITH départ ='Paris'`
	`CONNECT BY PRIOR arrivée = départ;`
	
	`DEPARTPARIS TEMPSVOL`
	`--------------- ----------`
	`Blagnac 1`
	`Pau ,4`
	`Lyon ,8`
	`Grenoble ,3`
	`Gap ,35`
	`Valence ,2`
	`Ales ,25`
	`Marseille ,9`
	`Frejus ,2`
	`Toulon ,15`
	`Nimes ,35`

Élagage de l'arbre (WHERE et PRIOR)

Il existe deux possibilités (qui peuvent se combiner) d'affiner le parcours d'un arbre :

- la clause `WHERE` permet d'éliminer des nœuds de l'arbre ;
- la clause `PRIOR` supprime des arborescences de l'arbre.

Le tableau suivant présente trois requêtes hiérarchiques. La première enlève un nœud, la deuxième une arborescence, la troisième combine ces deux élagages en ôtant à l'arbre un nœud et l'arborescence rattachée.

Tableau 4-50 Élagage d'arbres

Besoin	Requête et résultat sous SQL*Plus

Parcours de l'arbre en entier de haut en bas en partant de la ville de Paris sans prendre en compte Lyon ni en départ ni en arrivée.	`SELECT LPAD(' ',4*LEVEL-4)||arrivée`
	`DepartParis, tempsVol`
	`FROM Trajets`
	`WHERE NOT (départ='Lyon' OR arrivée='Lyon')`
	`START WITH départ ='Paris'`
	`CONNECT BY PRIOR arrivée = départ;`
	
	`DEPARTPARIS TEMPSVOL`
	`--------------- ----------`
	`Blagnac 1`
	`Pau ,4`
	`Gap ,35`
	`Ales ,25`
	`Marseille ,9`
	`Frejus ,2`
	`Toulon ,15`
	`Nimes ,35`

Parcours de l'arbre en entier de haut en bas en partant de la ville de Paris sans prendre en compte les trajets depuis Lyon.	`SELECT LPAD(' ',4*LEVEL-4)||arrivée`
	`DepartParis, tempsVol`
	`FROM Trajets`
	`START WITH départ ='Paris'`
	`CONNECT BY PRIOR arrivée = départ`
	`AND NOT départ = 'Lyon';`
	
	`DEPARTPARIS TEMPSVOL`
	`--------------- ----------`
	`Blagnac 1`
	`Pau ,4`
	`Lyon ,8`
	`Marseille ,9`
	`Frejus ,2`
	`Toulon ,15`
	`Nimes ,35`

Parcours de l'arbre en entier de haut en bas en partant de la ville de Paris sans prendre en compte Lyon et ses trajets.	`SELECT LPAD(' ',4*LEVEL-4)||arrivée`
	`DepartParis, tempsVol`
	`FROM Trajets`
	`WHERE NOT (arrivée = 'Lyon')`
	`START WITH départ ='Paris'`
	`CONNECT BY PRIOR arrivée = départ`
	`AND NOT départ = 'Lyon';`
	
	`DEPARTPARIS TEMPSVOL`
	`--------------- ----------`
	`Blagnac 1`
	`Pau ,4`
	`Marseille ,9`
	`Frejus ,2`
	`Toulon ,15`
	`Nimes ,35`

Jointures

Les requêtes hiérarchiques supportent les jointures mais seules des équijointures devraient être appliquées.

Si la clause `WHERE` contient une sous-interrogation (jointure procédurale), la jointure sera réalisée avant la clause `CONNECT BY`. Si la clause `WHERE` ne contient pas de sous-interrogation, le parcours de l'arbre est réalisé par le `CONNECT BY` puis les conditions du `WHERE` sont appliquées.

Dans le cas de jointures relationnelles, il faut que chaque nœud à parcourir vérifie la condition de jointure sous peine de perdre des éléments de l'arbre, non pas du fait du parcours mais de la jointure.

Supposons que nous disposions de la table `Aéroports` ci-dessous. L'équijointure relationnelle permet d'afficher les fréquences des aéroports sur les parcours.

Tableau 4-51 Requête hiérarchique (jointure relationnelle)

Table `Aéroport`

```
NOMAERO    FREQUENCETWR
---------- ------------
Ales              120,3
Blagnac           118,1
Frejus            114,7
Gap               122,7
Grenoble          115,6
Lyon              123,8
Marseille         118,7
Nimes             126,2
Paris             123,4
Pau               117,9
Toulon            119,9
Valence           126,9
```

Requête et résultat sous SQL*Plus

```
SELECT LPAD(' ',4*LEVEL-4)||arrivée
       DepartParis, tempsVol, frequenceTWR
 FROM Trajets, Aéroports
 WHERE NOT (arrivée = 'Lyon')
 AND arrivée = nomAero
 START WITH départ='Paris'
 CONNECT BY PRIOR arrivée = départ
 AND NOT départ = 'Lyon';

DEPARTPARIS       TEMPSVOL FREQUENCETWR
--------------- ---------- ------------
Blagnac                  1        118,1
    Pau                 ,4        117,9
Marseille               ,9        118,7
    Frejus              ,2        114,7
    Nimes              ,35        126,2
    Toulon             ,15        119,9
```

Ordonnancement

L'utilisation des directives `ORDER BY` ou `GROUP BY` est incompatible avec le parcours hiérarchique de l'arbre.

Pour classer des enregistrements d'une hiérarchie, il faut utiliser la directive `ORDER SIBLINGS BY`.

La requête suivante affiche tout l'arbre en triant sur les escales par ordre alphabétique inverse.

Tableau 4-52 Ordonner une requête hiérarchique

Requête	Résultat sous SQL*Plus

`COLUMN DepartParis FORMAT A15`
`SELECT LPAD(' ',4*LEVEL-4)||arrivée`
`DepartParis, tempsVol`
`FROM Trajets`
`START WITH départ='Paris'`
`CONNECT BY PRIOR arrivée = départ`
`ORDER SIBLINGS BY arrivée DESC;`
	`DEPARTPARIS TEMPSVOL`
	`--------------- ----------`
	`Marseille ,9`
	`Toulon ,15`
	`Nimes ,35`
	`Frejus ,2`
	`Lyon ,8`
	`Valence ,2`
	`Ales ,25`
	`Grenoble ,3`
	`Gap ,35`
	`Blagnac 1`
	`Pau ,4`

Extraction de chemins

Disponible depuis la version 10*g*, la fonction SYS_CONNECT_BY_PATH extrait le chemin (sous la forme d'une chaîne VARCHAR2) à partir de la racine (ou des racines si aucune clause START WITH n'est indiquée jusqu'aux feuilles terminales). La syntaxe de cette fonction est la suivante :

```
SYS_CONNECT_BY_PATH(colonne, caractère)
```

colonne et *caractère* sont de type CHAR, VARCHAR2, NCHAR, ou NVARCHAR2. Le premier paramètre désigne la colonne de la table qui compose la hiérarchie définie par la clause CONNECT BY et qu'on désire afficher. Le second paramètre indique le séparateur utilisé pour l'affichage du chemin complet.

La requête suivante extrait tous les chemins complets partant de Paris.

```
COL chemin FORMAT A30 HEADING "Hélas tout part de Paris..."
SELECT LPAD(' ',2*LEVEL-1)||SYS_CONNECT_BY_PATH(arrivée,'/')
   chemin, tempsVol
   FROM Trajets
   START WITH depart = 'Paris'
   CONNECT BY PRIOR arrivée = départ;
```

```
Hélas tout part de Paris...      TEMPSVOL
------------------------------ ----------
 /Blagnac                               1
   /Blagnac/Pau                        ,4
 /Lyon                                 ,8
   /Lyon/Grenoble                      ,3
     /Lyon/Grenoble/Gap                ,35
   /Lyon/Valence                       ,2
     /Lyon/Valence/Ales                ,25
 /Marseille                            ,9
   /Marseille/Frejus                   ,2
   /Marseille/Toulon                   ,15
   /Marseille/Nimes                    ,35
```

Extraction d'un élément

Disponible depuis la version 10*g*, l'opérateur `CONNECT_BY_ROOT` étend la fonctionnalité de la condition `CONNECT BY [PRIOR]` en permettant de qualifier une colonne et de retourner non seulement un enregistrement parent de l'enregistrement courant, mais également tous ses ancêtres. Cet opérateur ne peut pas être utilisé dans une clause `START WITH` ou `CONNECT BY`.

La requête suivante extrait les chemins complets ayant deux escales. L'opérateur `CONNECT_BY_ROOT` permet ici d'afficher la première escale.

```
COL chemin FORMAT A30 HEADING "Chemin..."
SELECT arrivée "De Paris à", CONNECT_BY_ROOT arrivée "arrivée",
       SYS_CONNECT_BY_PATH(départ,'/') chemin
     FROM Trajets WHERE LEVEL > 2
     CONNECT BY PRIOR arrivée = départ;

De Paris à arrivée    Chemin...
---------- ---------- ------------------------------
Gap        Lyon       /Paris/Lyon/Grenoble
Ales       Lyon       /Paris/Lyon/Valence
```

Nature d'un élément

Disponible depuis la version 10*g*, la pseudo-colonne `CONNECT_BY_ISLEAF` retourne la valeur 1 si l'enregistrement courant est une feuille de la hiérarchie désignée par la condition dans la clause `CONNECT BY`. Dans le cas inverse, cette pseudo-colonne vaut 0. Cette information permet de savoir si un enregistrement courant est un nœud ou une feuille de la hiérarchie.

La requête suivante extrait les chemins complets des trajets avec les destinations finales. L'opérateur `CONNECT_BY_ISLEAF` permet ici d'afficher seulement les terminaisons de la hiérarchie.

```
COL chemin FORMAT A30 HEADING "Chemin..."
SELECT arrivée, CONNECT_BY_ISLEAF "IsLeaf", LEVEL,
       SYS_CONNECT_BY_PATH(départ,'/') chemin
     FROM Trajets WHERE CONNECT_BY_ISLEAF = 1
     START WITH départ='Paris'
     CONNECT BY PRIOR arrivée = départ;
ARRIVÉE        IsLeaf      LEVEL Chemin...

---------- ---------- ---------- ------------------------------
Pau                 1          2 /Paris/Blagnac
Gap                 1          3 /Paris/Lyon/Grenoble
Ales                1          3 /Paris/Lyon/Valence
Frejus              1          2 /Paris/Marseille
Toulon              1          2 /Paris/Marseille
Nimes               1          2 /Paris/Marseille
```

La requête suivante extrait les chemins complets des trajets avec les destinations au bout de deux escales non terminales.

```
COL chemin FORMAT A35 HEADING "Chemin 2 escales non terminales..."
SELECT arrivée, SYS_CONNECT_BY_PATH(départ, '/') chemin
    FROM Trajets
    WHERE CONNECT_BY_ISLEAF = 0 AND LEVEL = 2
    START WITH depart = 'Paris'
    CONNECT BY PRIOR arrivée = départ;
ARRIVÉE    Chemin 2 escales non terminales...
---------- -----------------------------------
Grenoble   /Paris/Lyon
Valence    /Paris/Lyon
```

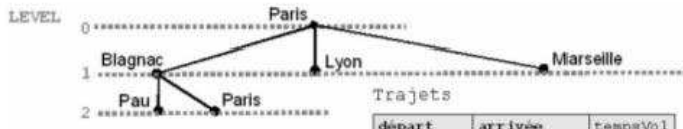
Éviter un cycle

Disponible depuis la version 10*g*, la pseudo-colonne `CONNECT_BY_ISCYCLE` retourne la valeur 1 si l'enregistrement courant est associé à un enregistrement enfant qui est également son ancêtre dans la hiérarchie désignée par la condition dans la clause `CONNECT BY`. Dans le cas inverse, cette pseudo-colonne vaut 0. Elle n'a de sens que si le paramètre `NOCYCLE` a été spécifié dans la clause `CONNECT BY`. Ce paramètre permet de retourner un résultat récursif qui échouerait sans cette option. La syntaxe de la définition du parcours de la hiérarchie est la suivante (elle est à placer après la condition `WHERE` de la requête) :

```
[ START WITH condition ]
CONNECT BY [ NOCYCLE ] condition
```

Considérons la hiérarchie suivante qui inclut un cycle. Il sera nécessaire d'utiliser le paramètre NOCYCLE et la pseudo-colonne CONNECT_BY_ISCYCLE pour que le cycle n'entraîne pas d'interférences dans les différentes requêtes qui parcourront la hiérarchie.

Figure 4-22 *Hiérarchie avec un cycle*



Trajets

départ	arrivée	tempsVol
Paris	Blagnac	1
Paris	Lyon	0.8
Paris	Marseille	0.9
Blagnac	Pau	0.4
Blagnac	Paris	0.95

La requête suivante extrait les chemins complets des trajets avec les destinations finales et intermédiaires. L'opérateur CONNECT_BY_ISCYCLE permet ici de trouver le cycle.

```
COL chemin FORMAT A30 HEADING "Chemin..."
SELECT arrivée "De Paris à", CONNECT_BY_ISCYCLE, LEVEL,
       SYS_CONNECT_BY_PATH(départ, '/') chemin
     FROM Trajets
     START WITH depart = 'Paris'
     CONNECT BY NOCYCLE PRIOR arrivée = départ;
```

```
De Paris à CONNECT_BY_ISCYCLE      LEVEL Chemin...
---------- ------------------ ---------- ------------------------
Blagnac                     0          1 /Paris
Pau                         0          2 /Paris/Blagnac
Paris                       1          2 /Paris/Blagnac
Lyon                        0          3 /Paris/Blagnac/Paris
Marseille                   0          3 /Paris/Blagnac/Paris
Lyon                        0          1 /Paris
Marseille                   0          1 /Paris
```

La requête suivante extrait les chemins complets des trajets avec les destinations finales et intermédiaires sans que le cycle n'interfère dans le résultat.

```
SELECT arrivée "De Paris à", LEVEL, SYS_CONNECT_BY_PATH(départ, '/') che-
min
       FROM Trajets
       WHERE CONNECT_BY_ISCYCLE = 0 AND LEVEL < 3
       START WITH départ = 'Paris'
       CONNECT BY NOCYCLE PRIOR arrivée = départ;
```

```
De Paris à      LEVEL Chemin...
---------- ---------- ------------------------------
Blagnac             1 /Paris
Pau                 2 /Paris/Blagnac
Lyon                1 /Paris
Marseille           1 /Paris
```

Mises à jour conditionnées (fusions)

L'instruction MERGE extrait des enregistrements d'une table source afin de mettre à jour (UPDATE) ou d'insérer (INSERT) des données dans une table cible. Cela évite d'écrire des insertions ou des mises à jour multiples en plusieurs commandes.

Vous devez avoir reçu les privilèges INSERT et UPDATE sur la table cible et le privilège SELECT sur la table source.

Syntaxe (MERGE)

La syntaxe générale de l'instruction MERGE est la suivante :

```
MERGE INTO [schéma.] nomTableCible [alias]
      USING [schéma.] { nomTableSource | nomVue | requête } [alias]
      ON (condition)
WHEN  MATCHED THEN
      UPDATE  SET col1 = { expression1 | DEFAULT }
                [,col2 = { expression2 | DEFAULT }]…
WHEN  NOT MATCHED THEN
      INSERT (col1 [, col2]…) VALUES (expression1 [,expression2]…);
```

Le choix entre la mise à jour et l'insertion dans la table cible est conditionné par la clause ON. Pour chaque enregistrement de la table cible qui vérifie la condition, l'enregistrement correspondant de la table source est modifié (UPDATE). Les données de la table cible qui ne vérifient pas la condition, déclenchent une insertion dans la table cible, basée sur des valeurs d'enregistrements de la table source.

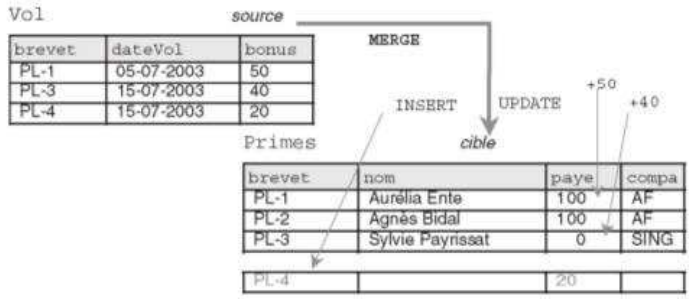
Il n'est pas possible d'utiliser la directive DEFAULT en travaillant avec des vues.

L'instruction MERGE est déterministe : il n'est pas possible de mettre à jour plusieurs fois le même enregistrement de la table cible en une seule instruction.

Exemple

Supposons qu'on désire ajouter à la paye de chaque pilote un bonus. Si on en donne un à un pilote n'ayant pas eu encore de prime, il faut ajouter ce pilote en affectant son bonus à son salaire. La figure suivante illustre cet exemple qui, sans l'utilisation de l'instruction MERGE, nécessite d'utiliser une instruction UPDATE et une instruction INSERT (multiligne si plusieurs pilotes n'étaient pas référencés dans la table Primes).

***Figure 4-23** Mises à jour conditionnées*



Le tableau suivant décrit l'instruction MERGE à utiliser et le résultat produit. L'utilisation de l'alias p permet de parcourir la table Primes et d'effectuer la jointure avec la table Vol.

Tableau 4-53 Fusion par MERGE

Requête	Résultat sous SQL*Plus

`MERGE INTO Primes p`
`  USING (SELECT brevet, bonus FROM Vol) v`
`  ON  (p.brevet = v.brevet)`
`  WHEN MATCHED THEN`
`  UPDATE SET p.paye = p.paye + v.bonus`
`  WHEN NOT MATCHED THEN`
`  INSERT (brevet, paye)`
`  VALUES (v.brevet, v.bonus);`
	`SQL> SELECT * FROM Primes ;`
	
	`BREVET NOM  PAYE COMP`
	`------ ---------- ----- ----`
	`PL-1  Aurélia Ente  150 AF`
	`PL-2  Agnès Bidal  100 AF`
	`PL-3  Sylvie Payrissat  40 SING`
	`PL-4  20`

Suppressions dans la table cible

Depuis la version 10*g*, l'instruction MERGE permet les trois types d'opération (UPDATE, DELETE, ou INSERT). Cela évite d'écrire des insertions, mises à jour ou suppressions multiples en plusieurs commandes. La nouvelle syntaxe de cette instruction est la suivante.

```
MERGE INTO [schéma.] nomTableCible [alias]
      USING [schéma.] { nomTableSource | nomVue | requête } [alias]
      ON (condition)
   [WHEN MATCHED THEN
         UPDATE SET col1 = {expression1 | DEFAULT}
                  [,col2 = {expression2 | DEFAULT}]...
         [WHERE condition]
         [DELETE WHERE condition] ]
   [WHEN NOT MATCHED THEN
         INSERT [ (col1 [, col2]...) ]
         VALUES ( {expression1 [,expression2]... | DEFAULT } )
         [WHERE condition] ] ;
```

Le choix de l'opération dans la table cible est toujours conditionné par la clause ON. Pour chaque enregistrement de la table cible qui vérifie la condition, l'enregistrement correspondant de la table source est modifié. Les données de la table cible qui ne vérifient pas la condition déclenchent une insertion dans la table cible, basée sur des valeurs d'enregistrements de la table source.

La clause DELETE permet de vider des enregistrements de la table cible, tout en la remplissant ou en la modifiant. Les seuls enregistrements affectés sont ceux qui sont concernés par la fusion. Cette clause évalue seulement les valeurs mises à jour (pas les valeurs originales qui sont évaluées par la directive UPDATE SET... WHERE *condition*). Si un enregistrement de la table cible satisfait à la condition du DELETE, mais n'est pas inclus dans la jointure définie par la directive ON, il ne sera pas détruit.

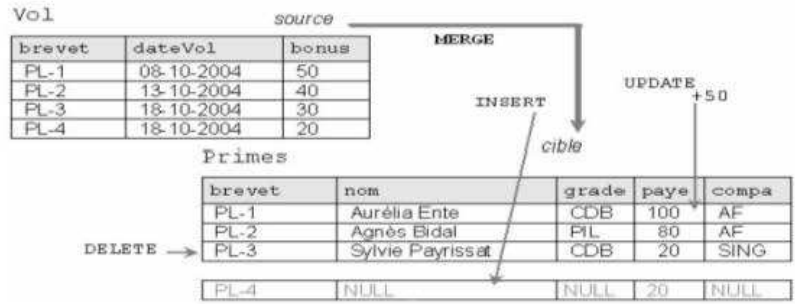
La clause WHERE de l'instruction INSERT filtre les insertions par une condition sur la table source.

Il n'est pas possible de modifier une colonne référencée dans la clause de jointure ON.

Exemple

Supposons qu'on désire ajouter à la paye de chaque pilote de grade 'CDB' un bonus. Si un bonus est donné à un pilote n'ayant pas encore eu de prime, il faudra ajouter ce pilote en affectant sa paye au bonus reçu. On désire aussi supprimer les primes des pilotes modifiés si la valeur de leur paye est inférieure à 90. La figure suivante illustre cet exemple qui, sans l'utilisation de l'instruction MERGE, requiert l'utilisation d'une instruction UPDATE, DELETE et INSERT (qui serait multiligne si plusieurs pilotes n'étaient pas référencés dans la table Primes).

Figure 4-24 Mises à jour conditionnées (à partir de 10g)



Le tableau suivant décrit l'instruction MERGE à utiliser et le résultat produit.

Tableau 4-54 Fusion par MERGE

Requête	Résultat sous SQL*Plus

`MERGE INTO Primes p`
`USING (SELECT brevet, bonus FROM Vol) v`
`ON (p.brevet = v.brevet)`
`WHEN MATCHED THEN`
`UPDATE SET p.paye = p.paye + v.bonus`
`WHERE grade = 'CDB'`
`DELETE WHERE paye < 90`
`WHEN NOT MATCHED THEN`
`INSERT (brevet, paye)`
`VALUES (v.brevet, v.bonus);`
	`SQL> SELECT * FROM Primes ;`
	
	`BREVET NOM GRADE PAYE COMP`
	`------ ------------- ----- ------ ----`
	`PL-1 Aurélia Ente CDB 150 AF`
	`PL-2 Agnès Bidal PIL 80 AF`
	`PL-4 20`

Expressions régulières

Depuis la version 10*g*, Oracle gère les expressions régulières. Ces dernières ont un fort rapport avec la notion de format de données ou de grammaire associée. Par exemple, un numéro de téléphone en France s'écrit sur 10 chiffres, le plus souvent indiqués par groupes de 2 entre tirets (exemple : 05-62-74-75-70). Les deux premiers chiffres indiquent une région (05 indique le Sud-Ouest). Un autre exemple concerne les numéros d'immatriculation des véhicules composés d'une série de chiffres, de lettres et de chiffres représentant le département d'appartenance.

Les expressions régulières sont manipulées sous SQL ou PL/SQL par les opérateurs REGEXP_LIKE, REGEXP_REPLACE, REGEXP_INSTR et REGEXP_SUBSTR. Le tableau suivant décrit les principaux éléments permettant de composer une expression régulière.

Tableau 4-55 Éléments décrivant une expression régulière

Élément	Description
\	Le caractère *backslash* (barre oblique inverse) permet d'annuler l'effet d'un caractère significatif suivant (opérateur, par exemple).
*	Désigne aucune ou plusieurs occurrences.
+	Désigne une ou plusieurs occurrences.
?	Désigne au plus une occurrence.
|	Opérateur spécifiant une alternative.
^	Désigne le début d'une ligne de caractères.
$	Désigne la fin d'une ligne de caractères.
.	Désigne tout caractère excepté la valeur `NULL`.
[ ]	Désigne une liste devant vérifier une expression continue dans la liste. Une liste ne devant pas vérifier une expression contenue dans la liste devra commencer par le caractère ^.
( )	Désigne une expression groupée et traitée comme une simple sous-expression.
{m}	Signifie exactement *m* fois.
{m,}	Signifie au moins *m* fois.
{m,n}	Signifie au moins *m* fois mais pas plus de *n* fois.
[::]	Spécifie la classe de caractères (précisée dans le tableau suivant).
[==]	Spécifie la classe d'équivalence (ex : `'[=a=]'` filtrera ă, â, à...).

Le tableau suivant recence les classes d'équivalence disponibles.

Tableau 4-56 Classes d'équivalence

Classe	Explication
[:alnum:]	Caractères alphanumériques.
[:alpha:]	Caractères alphabétiques.
[:blank:]	Caractères d'espacement.
[:cntrl:]	Caractères de contrôle.
[:digit:]	Chiffres.
[:graph:]	Caractères de la forme `[:punct:]`, `[:upper:]`, `[:lower:]` et `[:digit:]`.
[:lower:]	Caractères alphabétiques minuscules.
[:print:]	Caractères imprimables.
[:punct:]	Caractères de ponctuation.
[:space:]	Caractères espaces (non affichables).
[:upper:]	Caractères alphabétiques majuscules.
[:xdigit:]	Caractères hexadécimaux.

Quelques exemples

Considérons les données suivantes décrivant des parcs Américains (issu de [GEN 03]). La structure de la table `Parcs` est la suivante : `endroit VARCHAR2(7), telephone VARCHAR2(15), description VARCHAR2(400).`

Figure 4-25 *Jeu d'essai*

`Parcs`

endroit	telephone	description
P1	(231) 436-4100	Michigan's first state park encompasses approximately 1800 acres of Mackinac Island. The centerpiece is Fort Mackinac, built in 1780 by the British to protect the Great Lakes Fur Trade. For information by phone, dial 800-44-PARKS or 517-373-1214.
P2	(906) 289-4215	Located almost at the very tip of the Keewenaw Penninsula, Fort Wilkens is a restored army fort built during the copper rush. Camping is available. For the modern campground, phone (800) 447-2757. For group-camping, phone 906 289 4215. For information on canoe, kayak, and other boat rentals, call the concession office at (906) 289-4210.
P3	(906) 863-9747	This scenic site is centered around an impressive waterfall. A rustic, picnic area with waterpump is available.
P4	(906) 658-3338	A 217-acre park located on the site of an old lumber town, Deer Park. Shower and toilet facilities are available, as are campsites with electricity.
P5	(906) 885-5275	Michigan's largest state park consists of some 60,000 acres of mostly virgin timber. Over 90 miles of trails are available to backpackers and hikers. Downhill skiing is available in winter. Rustic cabins are available. To reserve a cabin, call (906) 885-5275.
P6	NULL	One of the largest waterfalls east of the Mississippi is found within this park's 40,000+ acres. Upper Tahquamenon Falls is some 50 feet high, 200 feet across, and supports a flow that has been known to reach 50,000 gallons/second. The park phone is 906.492.3415.

Fonction REGEXP_LIKE

La fonction booléenne `REGEXP_LIKE` permet d'identifier des enregistrements vérifiant une condition à propos d'une expression régulière. Cette fonction s'utilise majoritairement dans la clause `WHERE` d'une requête. La syntaxe de cette fonction est la suivante :

```
REGEXP_LIKE (chaineSource, grammaire [,paramètre …] )
```

`paramètre` est un texte littéral qui permet de moduler l'expression régulière. Les valeurs de ce paramètre peuvent être :

- `'i'` si on ne tient pas compte de la casse ;
- `'c'` si on tient compte de la casse ;
- `'n'` permet d'utiliser le caractère `.` en tant que fin de ligne ;
- `'m'` permet de traiter la chaîne source comme plusieurs lignes. Oracle interprète `^` et `$` comme le début et la fin de chaque sous-ligne.

Si aucun paramètre n'est utilisé, la sensibilité à la casse est définie par la valeur de NLS_SORT, le caractère . ne termine pas une ligne et la chaîne est traitée comme une seule ligne.

Exemples pour l'extraction

Le tableau suivant illustre quelques utilisations de cette fonction manipulant des expressions régulières. Le filtre porte sur la colonne description qui comporte plusieurs lignes. Nous testons ici les différents formats des numéros de téléphone.

Tableau 4-57 Utilisations de la fonction REGEXP_LIKE

Expression	Requête	Résultat SQL*Plus
xxx-xxxx	SELECT endroit FROM Parcs WHERE **REGEXP_LIKE** (description, '...-....');	ENDROIT ------- P1 P2 P4 P5
Idem, x étant un chiffre, élimine par exemple l'expression "217-acre".	SELECT endroit FROM Parcs WHERE **REGEXP_LIKE** (description, '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'); ou SELECT endroit FROM Parcs WHERE REGEXP_LIKE (description, '[0-9]{3}-[0-9]{4,4}');	ENDROIT ------- P1 P2 P5
Idem en autorisant aussi les nombres séparés par des points.	SELECT endroit FROM Parcs WHERE **REGEXP_LIKE** (description, '[0-9]{3}[-.][0-9]{4,4}');	ENDROIT ------- P1 P2 P5 P6

Le tableau suivant illustre quelques autres expressions régulières extraites du jeu d'essai décrit ci-après.

Figure 4-26 *Jeu d'essai*

Test

col
bonjour
Maître
enfant
maître
môbile
pájaro
zurück

Test2

col
resume
résumé
résume
resumé
rasumé
râsume

Tableau 4-58 Utilisation de classe de caractères

Expression	Requête	Résultat SQL*Plus
Chaînes de 6 caractères et plus en minuscules.	`SELECT col FROM Test WHERE REGEXP_LIKE(col,'([[:lower:]]){6}');`	`COL ---------- bonjour enfant maître mòbile pájaro zurück`
Chaînes de 6 caractères en minuscules.	`SELECT col FROM Test WHERE REGEXP_LIKE(col,'([[:lower:]]){6}$');`	`COL ---------- enfant maître mòbile pájaro zurück`
Chaînes de 6 caractères commençant par une majuscule, le reste en minuscules.	`SELECT col FROM Test REGEXP_LIKE(col, '([[:upper:]){1}[[:lower:]]{5}$'));`	`COL ---------- Maître`
Classe d'équivalence du « e » en deuxième et dernière position.	`SELECT col FROM Test2 WHERE REGEXP_LIKE(col,'r[[=e=]]sum[[=e=]]');`	`COL ---------- resume résumé résume resumé`
Classe d'équivalence de « a » et de « e ».	`SELECT col FROM Test2 WHERE REGEXP_LIKE(col,'r[[=a=]]sum[[=e=]]');`	`COL ---------- rasumé ràsume`

Définition d'une contrainte

La fonction `REGEXP_LIKE` permet également de définir des contraintes au niveau des colonnes de tables afin de s'assurer du format des données. L'ajout de la contrainte suivante garantit que la colonne `telephone` contient à présent des valeurs de la forme « (*xxx*) *xxx*-*xxxx* ».

```
ALTER TABLE Parcs
   ADD (CONSTRAINT ck_format_telephone
        REGEXP_LIKE(telephone,
             '^\([[:digit:]]{3}\) [[:digit:]]{3}-[[:digit:]]{4}$')));
```

Étudions à présent les fonctions par lesquelles on peut manipuler des chaînes de caractères tout en utilisant des expressions régulières.

Fonction REGEXP_REPLACE

La fonction REGEXP_REPLACE étend la fonction REPLACE en permettant de modifier une chaîne de caractères à partir d'une expression régulière. Par défaut, la fonction remplace une chaîne source par chaque occurrence d'une expression régulière donnée. Cette fonction retourne un VARCHAR2 si le premier paramètre n'est pas une donnée de type *LOB*. Dans le cas inverse, la fonction retourne une donnée de type CLOB. La syntaxe de cette fonction est la suivante :

```
REGEXP_REPLACE (source, modèle [,remplace
    [,position [, occurrence [, paramètre ] ] ] ] )
```

- *source* indique la chaîne à examiner (une colonne de type CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, ou NCLOB) ;
- *modèle* désigne l'expression régulière (jusqu'à 512 octets) ;
- *remplace* décrit, sous la forme de références arrières (jusqu'à 500 expressions « \n » avec *n* chiffres de 1 à 9), de quelle manière la chaîne source va être transformée. Si le paramètre *remplace* est un CLOB ou NCLOB, alors Oracle le tronque à 32 Ko ;
- *position* est un entier indiquant la position de début de recherche (par défaut 1) ;
- *occurrence* est un entier précisant le remplacement (0 pour remplacer toutes les occurrences qui conviennent à l'expression régulière, *n* pour remplacer la $n^{ième}$) ;
- *paramètre* a la même signification que dans l'utilisation de la fonction REGEXP_LIKE.

Le tableau suivant illustre quelques utilisations de cette fonction de remplacement. Le premier exemple remplace chaque caractère non nul par son équivalent suivi d'un tiret. Le deuxième remplace plusieurs espaces par un seul.

Dans le troisième exemple, nous rendons homogène (à l'affichage) les différents formats des numéros de téléphone de type « *xxx*xxx*xxxx* » (*x* étant un chiffre et * étant un tiret ou un point) présents dans la colonne `description` par le format « *(xxx) xxx-xxxx* ». On remarque que le numéro de téléphone codé en partie à l'aide de lettres n'a pas été modifié car il ne respecte pas l'expression régulière. Utilisée dans un UPDATE, cette fonction pourrait permettre de modifier cette colonne en conséquence.

Tableau 4-59 Utilisation de la fonction REGEXP_REPLACE

Requête	Résultat SQL*Plus

`CREATE TABLE Test (col VARCHAR2(30));`
`INSERT INTO Test VALUES ('Castanet');`
`INSERT INTO Test VALUES ('Blagnac');`
`INSERT INTO Test VALUES ('Paris');`
 	
`SELECT REGEXP_REPLACE(col, '(.)', '\1-')`
`FROM Test ;`
	`REGEXP_REPLACE(COL,'(.)','\1-')`
	`--------------------`
	`C-a-s-t-a-n-e-t-`
	`B-l-a-g-n-a-c-`
	`P-a-r-i-s-`

`SELECT REGEXP_REPLACE('IUT,1 Place G.Brassens, Blagnac',`
`'( ){2,}', ' ')) "Exemple 2" FROM DUAL;`
	`Exemple 2`
	`--------------------`
	`IUT, 1 Place G. Brassens, Blagnac`

`SELECT REGEXP_REPLACE(description,`
`'([[:digit:]]{3})[-.]([[:digit:]]{3})`
`[-.]([[:digit:]]{4})', '(\1) \2-\3')`
`FROM Parcs WHERE endroit = 'P1';`
	`DESCRIPTION`
	`--------------------`
	`Michigan's first state park`
	`encompasses approximately`
	`1800 acres of Mackinac Island.`
	`The centerpiece is Fort Mackinac,`
	`built in 1780 by the British`
	`to protect the Great Lakes Fur`
	`Trade. For information by phone,`
	`dial 800-44-PARKS or`
	`(517) 373-1214 .`

Fonction REGEXP_INSTR

La fonction REGEXP_INSTR étend la fonction INSTR en permettant de rechercher une chaîne de caractères à partir d'une expression régulière. Cette fonction retourne un entier indiquant le début (ou la fin) d'une sous-chaîne vérifiant l'expression régulière, ceci en fonction d'un paramètre de retour. Si aucune sous-chaîne ne convient, la fonction retourne 0. La syntaxe de cette fonction est la suivante :

```
REGEXP_INSTR (source, modèle
   [, position [, occurrence [, optionRetour [, paramètre ] ] ] ] )
```

- *source* indique la chaîne à examiner (une colonne de type CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB ou NCLOB) ;
- *modèle* désigne l'expression régulière (jusqu'à 512 octets) ;
- *position* est un entier positif indiquant la position de début de recherche (par défaut 1) ;
- *occurrence* est un entier positif précisant quelle est l'occurrence de l'expression recherchée (par défaut, 1 indiquant que la première occurrence est à examiner, *n* pour examiner la $n^{ième}$) ;

- *optionRetour* codifie ce qui doit être retourné :
  - 0 si la position du premier caractère de l'occurrence extraite doit être retournée (option par défaut) ;
  - 1 si la position du premier caractère suivant l'occurrence extraite doit être retournée ;
- *paramètre* a la même signification que dans l'utilisation des fonctions REGEXP_LIKE et REGEXP_REPLACE.

Le tableau suivant illustre quelques utilisations de cette fonction de recherche.

Le premier exemple examine la chaîne décrivant une adresse, recherche les occurrences des caractères non blancs en débutant au premier caractère et retourne la première position du quatrième mot (15 correspond à la position qui débute avec l'expression « 31703 »).

Le deuxième exemple examine la chaîne et analyse les mots de sept lettres commençant par s, r, ou p (casse indifférente). La recherche débute au troisième caractère et retourne la position du premier caractère suivant la seconde occurrence du type de mot recherché (ici, 28 correspond à la position du « *S* » de « *Shores* » ; « *Parkway* » et « *Redwood* » étant deux mots qui respectent l'expression régulière).

Dans le troisième exemple, nous extrayons les endroits dont la description inclut une surface (définis en acres mais hétérogènes au niveau de l'expression). Utilisées conjointement à SUBSTR (qui extrait une sous-chaîne), les fonctions REGEXP_INSTR permettent de délimiter les différentes expressions décrivant une surface (*1800 acres*, *217-acre*, *60,000 acres* et *40,000+ acres*). L'expression régulière est divisée par une barre verticale qui filtre à la fois les mots « *acres* » et « *acre* ». Les deuxième et troisième appels à REGEXP_INSTR servent à déterminer la taille de l'expression.

Tableau 4-60 Utilisations de la fonction REGEXP_INSTR

Requête	Résultat SQL*Plus

`SELECT REGEXP_INSTR('IUT Dept GTR, 31703 Blagnac', '[^ ]+', 1, 4) FROM DUAL;`	`Exemple 1`
	`----------`
	`15`

`SELECT REGEXP_INSTR('500 Oracle Parkway,Redwood Shores, CA','[s|r|p][[:alpha:]]{6}', 3, 2, 1,'i') "Exemple 2" FROM DUAL;`	`Exemple 2`
	`----------`
	`28`

`SELECT endroit, SUBSTR(description, REGEXP_INSTR(description, '[^]+ acres | [^]+-acre',1,1,0,'i'), REGEXP_INSTR(description, '[^]+ acres | [^]+-acre',1,1,1,'i') - REGEXP_INSTR(description, '[^]+ acres | [^]+-acre',1,1,0,'i')) "SURFACE" FROM Parcs WHERE REGEXP_LIKE(description, '[^]+ acres | [^]+-acre','i');`	`ENDROIT SURFACE`
	`-------- ------------`
	`P1 1800 acres`
	`P4 217-acre`
	`P5 60,000 acres`
	`P6 40,000+ acres`

Fonction REGEXP_SUBSTR

La fonction REGEXP_SUBSTR étend la fonction SUBSTR en permettant d'extraire une sous-chaîne à partir d'une expression régulière. Le fonctionnement de cette fonction est similaire à celui de REGEXP_INSTR sauf qu'au lieu de retourner la position d'une sous-chaîne, REGEXP_SUBSTR retourne la sous-chaîne elle-même. La syntaxe de cette fonction est la suivante :

```
REGEXP_SUBSTR (source, modèle
      [, position [, occurrence [, paramètre ] ] ] )
```

- *source* indique la chaîne à examiner (une colonne de type CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB ou NCLOB) ;
- *modèle* désigne l'expression régulière (jusqu'à 512 octets) ;
- *position* est un entier positif indiquant la position de début de recherche (par défaut 1) ;
- *occurrence* est un entier positif précisant quelle est l'occurrence de l'expression recherchée (par défaut, 1 indiquant que la première occurrence est à examiner, *n* pour examiner la $n^{ième}$).
- *paramètre* a la même signification que dans l'utilisation des fonctions REGEXP_LIKE et REGEXP_REPLACE et REGEXP_INSTSR.

Le tableau suivant illustre quelques utilisations de cette fonction d'extraction reposant sur les exemples précédents. Le premier exemple retourne la chaîne correspondant au quatrième mot. Le deuxième exemple retourne la chaîne correspondant à la seconde occurrence d'un mot de sept lettres commençant par s, r, ou p (casse indifférente). Dans le troisième exemple, nous simplifions l'extraction précédemment étudiée.

Tableau 4-61 Utilisations de la fonction REGEXP_SUBSTR

Requête	Résultat SQL*Plus

`SELECT REGEXP_SUBSTR('IUT Dept GTR, 31703 Blagnac', '[^ ]+', 1, 4) FROM DUAL;`	`Ex. 1`
	`-----`
	`31703`

`SELECT`
`  REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA', '[s|r|p][[:alpha:]]{6}', 1, 2, 'i')`
`  "Ex. 2" FROM DUAL;`
	`Ex. 2`
	`-------`
	`Redwood`

`COLUMN surface format a13 heading "Ex. 3"`
`SELECT endroit,`
`  REGEXP_SUBSTR(description,'[^ ]+[- ]acres?',1,1,'i')`
`  surface`
`  FROM Parcs`
`  WHERE REGEXP_LIKE(description,'[^ ]+[- ]acres?','i');`
	`ENDROIT  Ex. 3`
	`--------  -------------`
	`-`
	`P1  1800 acres`
	`P4  217-acre`
	`P5  60,000 acres`
	`P6  40,000+ acres`

Sous-expressions

Depuis la version 11*g*, les fonctions de recherche de chaîne de caractères et d'extraction de sous-chaîne de caractères à partir d'une expression régulière (REGEXP_INSTR et REGEXP_SUBSTR) sont enrichies d'une option supplémentaire qui permet de cibler une sous-expression particulière de l'expression régulière à évaluer.

La nouvelle fonction REGEXP_COUNT vient en complément de REGEXP_INSTR pour compter le nombre d'occurrences d'une expression régulière dans une chaîne de caractères.

Recherche et extraction

Concernant la recherche, l'option supplémentaire est indiquée en gras dans la syntaxe suivante :

```
REGEXP_INSTR (source, modèle
   [, position [, occurrence [, optionRetour
                  [, paramètre ] [, sousexpr ] ] ] ] )
```

- L'option sousexpr est un entier (de 0 à 9) qui permet de rechercher une position d'une sous-expression régulière (fragment d'expression entre parenthèses). Une sous-expression peut être imbriquée et est numérotée dans l'ordre d'apparition en fonction des parenthèses.
  - Si l'option sousexpr vaut zéro (valeur par défaut), la fonction se ramène à celle étudiée à la section précédente.
  - Si l'option sousexpr est différente de zéro, alors la position de la sous-chaîne (fragment) qui correspond à l'ordre de la sous-expression est retourné. Si aucune position n'est trouvée, la fonction retourne zéro.

Par exemple, considérons la chaîne (IUT)(R(ei)(ms)). Elle comporte quatre fragments qui sont respectivement (dans l'ordre des parenthèses) « IUT », « Reims », « ei » et « ms ». Ainsi, la requête suivante détermine la position de la troisième sous-expression (ici « ei ») au sein de la chaîne de caractères source (ici « IUTReims »).

```
SELECT REGEXP_INSTR('IUTReims','(IUT)(R(ei)(ms))',1,1,0,'i',3
                    "REGEXP_INSTR" FROM DUAL;
REGEXP_INSTR
---------------------
5
```

Concernant l'extraction, on retrouve la même nouvelle option dans la syntaxe suivante :

```
REGEXP_SUBSTR (source, modèle
   [, position [, occurrence
                  [, paramètre ] [, sousexpr ] ] ] ])
```

- L'option sousexpr est un entier (de 0 à 9) qui permet d'extraire une sous-expression régulière (fragment d'expression entre parenthèses). Si l'option sousexpr vaut zéro (valeur par défaut), la fonction se ramène à celle étudiée à la section précédente.

- Si aucune sous-expression n'est trouvée, la fonction retourne NULL.

Considérons l'exemple précédent, et extrayons la troisième sous-expression présente dans l'expression régulière au sein de la chaîne de caractères source.

```
SELECT REGEXP_SUBSTR('IUTReims','(IUT)(R(ei)(ms))',1,1,'i',3)
                     "REGEXP_SUBSTR" FROM DUAL;
REGEXP_SUBSTR
-------------
ei
```

Comptage

La fonction REGEXP_COUNT complète la fonction REGEXP_INSTR en permettant de compter le nombre d'occurrences d'une expression régulière dans une chaîne de caractères. Si aucune occurrence n'est trouvée, la fonction retourne zéro. La syntaxe de cette fonction est la suivante :

```
REGEXP_COUNT (source, modèle [, position [, paramètre ] ] )
```

- *source* indique la chaîne à examiner.
- *modèle* désigne l'expression régulière (jusqu'à 512 octets). Si des sous-expressions sont présentes (fragments), elles seront ignorées et considérées comme un tout.
- *position* est un entier indiquant la position de début de recherche (par défaut 1).
- *paramètre* a la même signification que dans l'utilisation de la fonction REGEXP_LIKE.

L'exemple suivant retourne le nombre de fois où l'expression IUT est présente dans la chaîne source.

```
SELECT REGEXP_COUNT('IUT-BlagnacIUT', '(IU)T', 1, 'i')
                    REGEXP_COUNT FROM DUAL;
REGEXP_COUNT
--------------------
2
```

Extractions diverses

Étudions, pour terminer ce chapitre consacré au requêtage, d'autres fonctions disponibles depuis la version 10*g* R2 avant de nous intéresser à celles proposées dans la version 11*g* R2.

Directive WITH

La directive WITH *nomRequête* permet d'assigner un nom à une sous-requête de façon à pouvoir l'utiliser à différents endroits et en particulier dans la requête finale (*main query*). Oracle optimise l'interrogation en considérant la sous-requête comme une vue ou comme une table temporaire.

Syntaxe

La syntaxe est la suivante :

```
WITH nomRequête1 AS (requêteSQL)
     [,nomRequête2 AS (requêteSQL2) ]...
SELECT...
```

Le nom d'une sous-requête est visible au niveau de la requête finale et au sein de toutes les autres sous-requêtes exceptée celle qui définie la sous-requête en question.

Chaque résultat d'une sous-requête est appelé CTE (*Common Table Expression*).

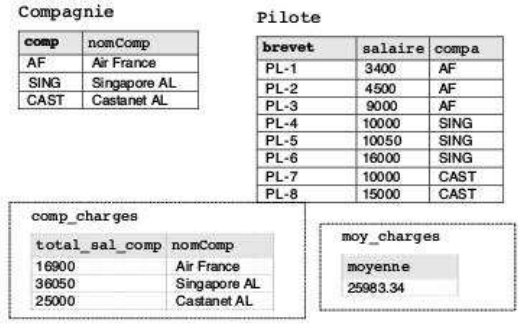
Exemple

L'exemple suivant extrait le nom des compagnies dont la masse salariale est inférieure à la masse salariale moyenne par compagnie. Nous utilisons ici deux sous-requêtes nommées. La première (`comp_charges`) construit un ensemble décrivant les compagnies avec leur masse salariale. La seconde sous-requête (`moy_charges`) se sert de la première afin d'extraire la moyenne de la masse salariale. Les deux sont utilisées par la suite par la requête finale.

```
WITH comp_charges  AS (SELECT nomComp, SUM(salaire) total_sal_comp
                       FROM Pilote p, Compagnie c
                       WHERE p.compa = c.comp GROUP BY nomComp),
     moy_charges AS (SELECT SUM(total_sal_comp)/COUNT(*) moyenne
                       FROM comp_charges )
SELECT * FROM comp_charges
      WHERE total_sal_comp < (SELECT moyenne FROM moy_charges)
      ORDER BY nomComp;
```

La figure suivante illustre cette directive à l'aide d'un exemple.

Figure 4-27 *Jeu d'exemple pour les sous-requêtes nommées*



Le résultat de cette extraction est le suivant :

```
NOMCOMP          TOTAL_SAL_COMP
---------------- --------------
Air France                16900
Castanet AL               25000
```

Il n'est pas possible d'utiliser une clause `WITH` dans une requête ou une expression (la clause `WITH` doit se trouver au plus haut niveau).

Fonction WIDTH_BUCKET

La fonction `WIDTH_BUCKET` permet de définir des plages de valeurs à partir d'intervalles calculés.

Syntaxe

La syntaxe est la suivante. Les paramètres sont explicités au tableau 4-16.

```
WIDTH_BUCKET (expression, valeurMin, valeurMax, nbrIntervalle)
```

Exemple

L'exemple suivant permet de répartir les pilotes suivant leur expérience (nombre d'heures de vol). Considérons les données suivantes.

```
SQL> SELECT brevet, nom, nbhvol  FROM Pilote ORDER BY nbHVol ;
BREVET NOM                       NBHVOL
------ ------------------- ----------
PL-1   Henri Alquié                 400
PL-2   Pierre Lamothe               500
PL-3   Didier Linxe                 900
PL-4   Christian Soutou            1000
PL-5   Gilles Laborde              1050
PL-6   Pierre Séry                 1600
PL-7   Michel Castaings            1700
PL-9   Patrick Baudry              3999
PL-8   Jules Ente                  4000
PL-10  Daniel Viel                 5000
```

La requête suivante définit 10 plages de valeurs (heures de vol) entre les chiffres 600 et 4 000 (soit 10 plages de 340 unités). La première ira de 600 à 940 (non inclus), la seconde de 940 à 1 280 (non inclus), etc. Si le chiffre est inférieur à la borne minimale, la plage est valuée à zéro, s'il est supérieur à la borne maximale, la plage est automatiquement calculée.

```
SELECT brevet, nom, nbHVol "Heures de vol",
   WIDTH_BUCKET (nbHVol,600, 4000, 10)   "Tranche Expérience"
   FROM Pilote ORDER BY nbHVol;
```

Le résultat est le suivant. Notez les deux premières lignes et les deux dernières qui sont hors intervalle prédéfini.

```
BREVET NOM                  Heures de vol Tranche Expérience
------ -------------------- ------------- ------------------
PL-1   Henri Alquié                  400                  0
PL-2   Pierre Lamothe                500                  0
PL-3   Didier Linxe                  900                  1
PL-4   Christian Soutou             1000                  2
PL-5   Gilles Laborde               1050                  2
PL-6   Pierre Séry                  1600                  3
PL-7   Michel Castaings              1700                  4
PL-9   Patrick Baudry                3999                 10
PL-8   Jules Ente                    4000                 11
PL-10  Daniel Viel                   5000                 11
```

Récursivité avec WITH (CTE)

Depuis la *release* 2 de la version 11*g*, l'opérateur `WITH` permet de programmer la récursivité d'une manière plus efficace que la clause `CONNECT BY`. En effet, une sous-requête peut désormais utiliser la requête principale. On parle d'une expression commune de table (CTE pour *Common Table Expression*). La syntaxe de cet opérateur permettant la récursivité est la suivante :

```
WITH
  nomRequete1 ([alias_col1 [,alias_col2]...])
AS
  (sousRequete1)
  [SEARCH { DEPTH FIRST BY alias_c1 [,alias_c2]...
            [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
          | BREADTH FIRST BY alias_c1 [,alias_c2]...
            [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ] }
        SET col_ordre ]
  [CYCLE alias_c1 [,alias_c2]...
    SET alias_col_cycle TO valeur_cycle
        DEFAULT valeur_non_cycle ]
  [,nomRequete2 ([alias_col1 [,alias_col2]...])
AS
  (sousRequete2) …
```

La sous-requête (*sousRequetei*) programmant la récursivité doit être composée de deux requêtes : la première est dite *anchor member* et la seconde est appelée *recursive member*. La première ne peut pas référencer la requête principale tandis que la seconde doit impérativement la référencer, mais une seule fois. La première requête peut être composée de plusieurs sous-requêtes reliées par des opérateurs ensemblistes (`UNION ALL`, `UNION`, `INTERSECT` ou `MINUS`). Par ailleurs, vous devrez utiliser l'opérateur `UNION ALL` entre la requête *anchor member* et la requête *recursive member*.

Premier exemple

Considérons l'exemple suivant décrivant la hiérarchie de quelques aéroports. Dans l'exemple, la récursivité va parcourir les liaisons entre enregistrements fils et parents (Castelnaudary dépend de Toulouse qui dépend d'Orly, lui-même sous Charles-de-Gaulle).

Tableau 4-62 Données à parcourir

Contenu de la table

```
SQL> SELECT OACI, nomAero, OACI_resp FROM Aeroport;

OACI  NOMAERO                        OACI_RESP
----- ------------------------------ ----------
LFPG  Paris Charles de Gaulle
LFPO  Paris Orly                     LFPG
LFBO  Toulouse Blagnac               LFBD
LFBD  Bordeaux Merignac              LFPO
LFCI  ALbi                           LFBO
LFCK  Castres                        LFBO
LFMW  Castelnaudary                  LFBO
LFMT  Montpellier Fregorgues         LFMM
LFMM  Marseille Marignane            LFPO
```

Le nombre d'alias de colonnes d'une requête principale doit être identique au nombre d'alias de colonnes des requêtes *anchor member* et *recursive member*.

La requête *recursive member* ne doit pas contenir `DISTINCT`, `GROUP BY`, `MODEL`, fonctions d'agrégat, sous-requêtes ou jointures externes avec la requête principale.

La requête suivante parcourt l'arbre des aéroports récursivement (la condition de liaison est basée sur l'égalité des colonnes clés. Dans l'exemple, le point de départ est l'aéroport d'Orly à partir duquel Oracle recherche tous ses subordonnés, pour chaque subordonné, Oracle recherche tous ses subordonnés, etc.

Tableau 4-63 Requête récursive

Requête

```
COL arbo FORMAT A15
WITH
  sous_Paris_Orly (OACI, OACI_resp, niveau)
  AS ( SELECT OACI, OACI_resp, 0 niveau
     FROM Aeroport
     WHERE OACI = 'LFPO'
   UNION ALL
   SELECT a.OACI, a.OACI_resp, niveau+1
     FROM sous_Paris_Orly sp, Aeroport a
     WHERE sp.OACI = a.OACI_resp )
SELECT OACI_resp,
   LPAD(' ',2*niveau)||OACI arbo, niveau
   FROM sous_Paris_Orly
   WHERE niveau>0
   ORDER BY niveau, OACI;
```

Résultat

```
OACI_RESP  ARBO                NIVEAU
---------- --------------- ----------
LFPO          LFBD                 1
LFPO          LFMM                 1
LFBD            LFBO               2
LFMM            LFMT               2
LFBO              LFCI             3
LFBO              LFCK             3
LFBO              LFMW             3
```

Constituer une liste des descendants

La requête suivante parcourt l'arbre des aéroports récursivement en partant de l'aéroport d'Orly. À chaque subordonné trouvé, la liste des descendants est complétée.

Tableau 4-64 Liste des descendants

Requête

```
COL OACI FORMAT A5
WITH
  sous_Paris_Orly
 (OACI, OACI_resp, niveau, aro_liste)
AS ( SELECT OACI, OACI_resp, 0 niveau,
        CAST(OACI_resp AS VARCHAR2(25))
     FROM Aeroport
     WHERE OACI = 'LFPO'
   UNION ALL
   SELECT a.OACI, a.OACI_resp, niveau+1,
        CAST(aro_liste || ',' ||
       a.OACI_resp AS VARCHAR2(25))
     FROM sous_Paris_Orly sp, Aeroport a
     WHERE sp.OACI = a.OACI_resp )
SELECT OACI, niveau, aro_liste
    FROM sous_Paris_Orly
    ORDER BY niveau, OACI;
```

Résultat

```
OACI       NIVEAU ARO_LISTE
----- ---------- --------------------
LFPO            0 LFPG
LFBD            1 LFPG,LFPO
LFMM            1 LFPG,LFPO
LFBO            2 LFPG,LFPO,LFBD
LFMT            2 LFPG,LFPO,LFMM
LFCI            3 LFPG,LFPO,LFBD,LFBO
LFCK            3 LFPG,LFPO,LFBD,LFBO
LFMW            3 LFPG,LFPO,LFBD,LFBO
```

Ordonner les descendants

La clause SEARCH permet d'ordonnancer les lignes extraites lors du parcours. L'option BREADTH FIRST BY retourne les lignes d'un même niveau (*sibling rows*) avant de descendre dans l'arbre (*child rows*). L'option DEPTH FIRST BY réalise l'inverse. L'ordre est indiqué par la colonne citée dans l'opérateur BY et remonte au niveau de la requête principale à l'aide d'une colonne fictive présente dans l'opérateur SET.

La requête suivante parcourt l'arbre des aéroports récursivement en profondeur d'abord, puis en largeur en partant de l'aéroport d'Orly.

Tableau 4-65 Liste ordonnée par DEPTH FIRST BY

Requête

```
COL OACI FORMAT A5
WITH
 sous_Paris_Orly (OACI, OACI_resp, niveau)
AS
   ( SELECT OACI, OACI_resp, 0 niveau
     FROM Aeroport
     WHERE OACI = 'LFPO'
   UNION ALL
   SELECT a.OACI, a.OACI_resp, niveau+1
     FROM sous_Paris_Orly sp, Aeroport a
     WHERE sp.OACI = a.OACI_resp )
 SEARCH DEPTH FIRST BY OACI_resp SET order1
SELECT OACI_resp,
    LPAD(' ',2*niveau)||OACI arbo, niveau
FROM sous_Paris_Orly
WHERE niveau>0
ORDER BY order1;
```

Résultat

```
OACI_RESP  ARBO                 NIVEAU
---------- -------------------- ----------
LFPO         LFBD                      1
LFBD           LFBO                    2
LFBO             LFCI                  3
LFBO             LFCK                  3
LFBO             LFMW                  3
LFPO         LFMM                      1
LFMM           LFMT                    2
```

La requête suivante parcourt l'arbre des aéroports récursivement en largeur d'abord, puis en profondeur en partant de l'aéroport d'Orly. Cela ne représente pas, dans notre exemple, l'arbre modélisé mais cela peut résoudre certaines problématiques.

Tableau 4-66 Liste ordonnée par BREADTH FIRST BY

Requête

```
COL OACI FORMAT A5
WITH
  sous_Paris_Orly (OACI, OACI_resp, niveau)
AS
  (SELECT OACI, OACI_resp, 0 niveau
     FROM Aeroport
     WHERE OACI = 'LFPO'
   UNION ALL
   SELECT a.OACI, a.OACI_resp, niveau+1
     FROM sous_Paris_Orly sp, Aeroport a
     WHERE sp.OACI = a.OACI_resp )
SEARCH BREADTH FIRST BY OACI_resp SET order1
SELECT OACI_resp,
    LPAD(' ',2*niveau)||OACI arbo, niveau
FROM sous_Paris_Orly
WHERE niveau>0
ORDER BY order1;
```

Résultat

```
OACI_RESP  ARBO                 NIVEAU
---------- --------------- ----------
LFPO         LFMM                   1
LFPO         LFBD                   1
LFBD           LFBO                 2
LFMM           LFMT                 2
LFBO             LFMW               3
LFBO             LFCK               3
LFBO             LFCI               3
```

Cycles de colonnes

La clause CYCLE permet de détecter les cycles de colonnes, une ligne compose un cycle lorsque l'une de ces lignes ancêtre (ascendant) a la même valeur pour une colonne donnée (celle du cycle cherché). Attention, il ne s'agit pas de cycle de l'arbre, qui, s'il existe, devra être considéré comme un graphe (voir la section qui suit).

```
CYCLE alias_c1 [,alias_c2]...
    SET alias_col_cycle TO valeur_cycle DEFAULT valeur_non_cycle
```

- Les alias suivant la clause CYCLE doivent faire référence aux colonnes de la requête principale.
- Les paramètres *valeur_cycle* et *valeur_non_cycle* doivent être des caractères de taille 1.
- Dès qu'un cycle est détecté, alors la colonne *alias_col_cycle* est initialisée à la valeur *valeur_cycle*. La récursivité s'arrête sur cette ligne (aucune ligne descendante n'est examinée) mais le traitement se poursuit sur les lignes de même niveau (et leurs descendantes).
- Si aucun cycle n'est trouvé, la colonne *alias_col_cycle* contiendra la valeur *valeur_non_cycle* pour les lignes extraites. Cette colonne est d'ailleurs automatiquement ajoutée à la requête finale.

Ajoutons à l'exemple précédent la colonne année de création et cherchons les aéroports qui ont un ancêtre construit la même année.

Tableau 4-67 Données à parcourir

Contenu de la table

```
SQL> SELECT OACI, nomAero, OACI_resp, anneeCreation FROM Aeroport;

OACI       NOMAERO                        OACI_RESP  ANNEECREATION
---------- ------------------------------ ---------- -------------
LFPG       Paris Charles de Gaulle                            1978
LFPO       Paris Orly                     LFPG                1967
LFBO       Toulouse Blagnac               LFBD                1968
LFBD       Bordeaux Merignac              LFPO                1972
LFCI       ALbi                           LFBO                1967
LFCK       Castres                        LFBO                1980
LFMW       Castelnaudary                  LFBO                1981
LFMT       Montpellier Fregorgues         LFMM                1973
LFMM       Marseille Marignane            LFPO                1975
```

La requête suivante parcourt tout l'arbre récursivement et détecte un cycle sur l'année de création entre Albi et Orly.

Tableau 4-68 Recherche d'un cycle sur une colonne

Requête et résultats

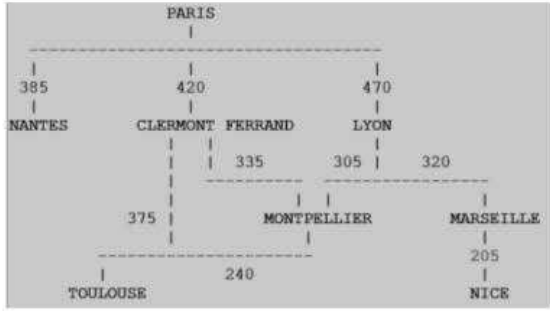
```
COL arbo FORMAT A15
COL est_cycle FORMAT A9
WITH
 sous_Paris_Orly (OACI, OACI_resp, niveau, creation)
AS
  (SELECT OACI, OACI_resp, 0 niveau, anneeCreation
     FROM Aeroport WHERE OACI = 'LFPO'
   UNION ALL
   SELECT a.OACI, a.OACI_resp, niveau+1, anneeCreation
     FROM sous_Paris_Orly sp, Aeroport a
     WHERE sp.OACI = a.OACI_resp )
SEARCH DEPTH FIRST BY OACI_resp SET order1
CYCLE creation SET est_cycle TO 'Y'  DEFAULT 'N'
SELECT OACI_resp, LPAD(' ',2*niveau)||OACI arbo,
       niveau, creation, est_cycle
FROM sous_Paris_Orly ORDER BY order1;
```

```
OACI_RESP  ARBO              NIVEAU   CREATION EST_CYCLE
---------- --------------- ---------- ---------- ---------
LFPG       LFPO                     0       1967 N
LFPO         LFBD                   1       1972 N
LFBD           LFBO                 2       1968 N
LFBO             LFCI               3       1967 Y
LFBO             LFCK               3       1980 N
LFBO             LFMW               3       1981 N
LFPO         LFMM                   1       1975 N
LFMM           LFMT                 2       1973 N
```

Parcours d'un graphe orienté

L'exemple du graphe suivant est extrait du blog de Frédéric Brouard (alias *SQLpro*), consultant expert SQL et spécialiste de SQL-Server. Je lui rends ici hommage, au regard de la foultitude d'articles de qualité qu'il a produit à propos de SQL (*http://blog.developpez.com/sqlpro*). Bon, il n'aime pas trop Oracle, mais personne n'est parfait.

Figure 4-28 *Graphe autoroutier*



La table `Autoroutes` permet d'implémenter cet état de fait. Nous allons progressivement rechercher le détail des trajets possible entre la capitale et le Capitole.

Tableau 4-69 Données à parcourir

Contenu de la table

```
SQL> SELECT ville_de, ville_vers, km FROM Autoroutes;

VILLE_DE        VILLE_VERS              KM
--------------- --------------- ----------
PARIS           NANTES                 385
PARIS           CLERMONT-FERRAND       420
PARIS           LYON                   470
CLERMONT-FERRAND MONTPELLIER           335
CLERMONT-FERRAND TOULOUSE              375
LYON            MONTPELLIER            305
LYON            MARSEILLE              320
MONTPELLIER     TOULOUSE               240
MARSEILLE       NICE                   205
```

La requête suivante parcourt le graphe récursivement pour extraire le nombre d'étapes des différents trajets, entre Paris et la ville rose. On retrouve la jointure entre la table de référence et celle construite récursivement.

Tableau 4-70 Recherche dans un graphe

Requête	Résultat

```
WITH trajets (ville_vers, etape)
AS
   (SELECT DISTINCT ville_de, 0
      FROM   Autoroutes
      WHERE  ville_de = 'PARIS'
    UNION  ALL
    SELECT  a.ville_vers, t.etape + 1
      FROM  Autoroutes a, trajets t
      WHERE t.ville_vers  = a.ville_de)
SELECT ville_vers, etape
FROM   trajets
WHERE  ville_vers = 'TOULOUSE';
```

```
VILLE_VERS             ETAPE
---------------- ----------
TOULOUSE                2
TOULOUSE                3
TOULOUSE                3
```

La requête suivante ajoute à la précédente la somme des kilométrages pour chaque ligne extraite du graphe.

Tableau 4-71 Recherche dans un graphe (incrémentation d'une variable)

Requête	Résultat

```
WITH trajets
  (ville_vers, etape, distance)
AS
   (SELECT DISTINCT ville_de, 0, 0
      FROM  Autoroutes
      WHERE ville_de = 'PARIS'
    UNION ALL
    SELECT  a.ville_vers,
       t.etape + 1, t.distance+a.km
      FROM  Autoroutes a, trajets t
      WHERE t.ville_vers  = a.ville_de)
SELECT ville_vers, etape, distance
FROM   trajets
WHERE  ville_vers = 'TOULOUSE';
```

```
VILLE_VERS             ETAPE   DISTANCE
---------------- ---------- ----------
TOULOUSE                2        795
TOULOUSE                3       1015
TOULOUSE                3        995
```

La requête suivante ajoute à la précédente la construction progressive des chemins parcourus (que j'estime à 50 caractères) et l'ordonnancement en fonction de la distance totale.

Tableau 4-72 Recherche dans un graphe (construction de chemins)

Requête et résultats

```
WITH trajets (ville_vers, etape, distance, trajet)
AS
   (SELECT DISTINCT ville_de, 0, 0, CAST('PARIS' AS VARCHAR(50))
      FROM Autoroutes WHERE  ville_de = 'PARIS'
    UNION  ALL
    SELECT  a.ville_vers, t.etape + 1, t.distance+a.km,
         CAST(t.trajet|| ',' || a.ville_vers AS VARCHAR2(50))
      FROM  Autoroutes a, trajets t
      WHERE t.ville_vers  = a.ville_de)
SELECT trajet, etape, distance
       FROM trajets WHERE  ville_vers = 'TOULOUSE' ORDER BY distance;
```

```
TRAJET                                                ETAPE   DISTANCE
----------------------------------------------------- ------- ---------
PARIS,CLERMONT-FERRAND,TOULOUSE                           2        795
PARIS,CLERMONT-FERRAND,MONTPELLIER,TOULOUSE               3        995
PARIS,LYON,MONTPELLIER,TOULOUSE                           3       1015
```

Qu'adviendrait-il si toutes les étapes inverses (exemple : `NANTES,PARIS,385`) étaient également stockées dans la même table ? Toutes ces requêtes s'y perdraient dans les cycles, renvoyant l'erreur « `ORA-32044: cycle détecté lors de l'exécution de l'interrogation WITH récursive` ». Le paragraphe suivant décrit le moyen d'éviter ces désagréments.

Parcours d'un graphe non orienté

Ajoutons d'abord les étapes inverses à la table implémentant les trajets possibles.

Tableau 4-73 Ajout des étapes inverses

Requête et résultats

```
SQL> INSERT INTO Autoroutes
           SELECT ville_vers,ville_de, km FROM Autoroutes;

SQL> SELECT ville_de, ville_vers, km
           FROM Autoroutes ORDER BY ville_de, ville_vers;

VILLE_DE        VILLE_VERS           KM
--------------- --------------- ----------
CLERMONT-FERRAND MONTPELLIER           335
CLERMONT-FERRAND PARIS                 420
CLERMONT-FERRAND TOULOUSE              375
LYON            MARSEILLE              320
LYON            MONTPELLIER            305
LYON            PARIS                  470
MARSEILLE       LYON                   320
MARSEILLE       NICE                   205
MONTPELLIER     CLERMONT-FERRAND       335
...
```

Les requêtes précédentes ne conviennent pas à cette table car Oracle détecte des cycles sans fin.

Il est possible de se débarrasser des cycles en comparant tout chemin courant avec la colonne évaluée en question. Ce n'est toutefois pas suffisant, car Oracle ne vous fait pas confiance et teste la requête avant d'exécuter cette condition. En conséquence, il faut ajouter une condition concernant un nombre maximal d'itération récursive.

La requête suivante teste la ville d'arrivée avec tout chemin construit récursivement et élimine ainsi les cycles. Par ailleurs, la recherche dans le graphe se limite à 10 niveaux.

Tableau 4-74 Recherche dans un graphe (construction de chemins)

Requête et résultats

```
WITH trajets (ville_vers, etape, distance, trajet)
AS
   (SELECT DISTINCT ville_de, 0, 0, CAST('PARIS' AS VARCHAR(50))
      FROM   Autoroutes WHERE  ville_de = 'PARIS'
    UNION  ALL
    SELECT  a.ville_vers, t.etape + 1, t.distance+a.km,
         CAST(t.trajet|| ',' || a.ville_vers AS VARCHAR2(50))
      FROM  Autoroutes a, trajets t
      WHERE t.ville_vers  = a.ville_de
      AND   t.trajet NOT LIKE '%' || a.ville_vers || '%'
      AND   t.etape <10)
SELECT trajet, etape, distance
       FROM  trajets
       WHERE ville_vers = 'TOULOUSE' ORDER BY distance;

TRAJET                                                ETAPE   DISTANCE
----------------------------------------------------- ---------- ----------
PARIS,CLERMONT-FERRAND,TOULOUSE                            2        795
PARIS,CLERMONT-FERRAND,MONTPELLIER,TOULOUSE                3        995
PARIS,LYON,MONTPELLIER,TOULOUSE                            3       1015
PARIS,LYON,MONTPELLIER,CLERMONT-FERRAND,TOULOUSE           4       1485
```

Une nouvelle route apparaît, puisque Montpellier est située sur le chemin de Clermont et de Lyon. Comme dit Frédéric Brouard, « c'est la plus longue, mais peut-être la plus belle ».

Pivots (PIVOT)

Depuis la version 11*g*, l'opérateur `PIVOT` permet, à l'aide de requêtes, de transformer des lignes en colonnes tout en opérant une fonction d'agrégat à la volée (somme, moyenne, etc.). La syntaxe de cet opérateur est la suivante.

```
table_interrogée PIVOT [ XML ]
  ( fonction_agregat ( expression ) [[AS] alias ]
     [,fonction_agregat (expression) [[AS] alias ] ]...
  FOR { colonne | (colonne [,colonne]... ) }
  IN ( { { { expression | (expression [,expression]...)} [ [ AS]
alias] }...
       | requete_SELECT | ANY [, ANY]... } ) )
```

La clause PIVOT contient une ou plusieurs fonctions d'agrégat, la clause FOR liste une ou plusieurs colonnes (à grouper puis à faire pivoter). La clause IN filtre les colonnes de la clause FOR.

Le mécanisme du pivot est le suivant : calcul de(s) agrégat(s) (sans GROUP BY devenu implicite du fait de la directive IN) puis transposition de chaque valeur calculée à la colonne correspondante.

Exemples

Considérons l'exemple suivant décrivant les vols d'une semaine.

Tableau 4-75 Données à faire pivoter

Requête	Résultat

```
CREATE TABLE vols
( id_vol  NUMBER,
  jour_id NUMBER,
  numVol VARCHAR2(6),
  nb_passagers NUMBER);
```

```
ID_VOL  JOUR_ID NUMVOL NB_PASSAGERS
------- ------- ------ ------------
1       1       AF6143 10
2       1       BA234  20
3       1       CF56   30
4       2       AF6143 40
5       2       CF56   50
6       3       AF6143 60
7       3       BA234  70
8       3       CF56   80
9       3       D009   90
10      4       AF6143 100
```

Le tableau suivant présente deux requêtes utilisant l'opérateur PIVOT. La première requête totalise le nombre de passagers transportés par vol, la seconde opère deux agrégats (somme et moyenne des passagers transportés le lundi et le mardi). Notez qu'Oracle nomme les colonnes en concaténant les noms des alias.

Tableau 4-76 Requêtes de pivot

Total des passagers transportés par vol

```
SELECT * FROM
  (SELECT numVol, nb_passagers
   FROM vols)
 PIVOT (SUM(nb_passagers) AS somme
  FOR (numVol)
  IN ('AF6143' AS AF6143,
  'BA234' AS BA234, 'CF56' AS CF56,
  'D009' AS D009 ));

AF6143_SOMME BA234_SOMME CF56_SOMME D009_SOMME
------------ ----------- ---------- ----------
210          90          160        90
```

Total et moyenne des passagers transportés lundi et mardi

```
SELECT * FROM
  (SELECT jour_id, nb_passagers
   FROM vols)
 PIVOT (SUM(nb_passagers) AS somme,
        AVG(nb_passagers) AS moy
  FOR (jour_id)
  IN (1 AS Lundi, 2 AS Mardi));

LUNDI_SOMME LUNDI_MOY MARDI_SOMME MARDI_MOY
----------- --------- ----------- ---------
60          20        90          45
```

La requête suivante construit un modèle à deux dimensions qui permet d'extraire le nombre total des passagers transportés par vol et par jour. La directive `WITH` permet de travailler temporairement avec une partie de la table `vols` (ici les vols du lundi, mardi et mercredi dont le numéro ne commence pas par la lettre 'B'.).

Tableau 4-77 Modèle à deux dimensions par pivot

Requête

```
WITH pivot_data AS
 (SELECT jour_id, numVol,
         nb_passagers FROM vols)
 SELECT * FROM  pivot_data
     PIVOT (SUM(nb_passagers)
        FOR jour_id
        IN  (1 AS Lundi,2 AS Mardi,
             3 AS Mercredi) )
    WHERE NOT (numVol LIKE 'B%');
```

Résultat

```
NUMVOL LUNDI  MARDI  MERCREDI
------ ------ ------ --------
CF56   30     50     80
AF6143 10     40     60
D009                 90
```

Il n'est pas possible de générer un nombre inconnu de valeurs pivot sans la clause XML.

Avec XML

Utilisé conjointement à l'opérateur `PIVOT`, la directive XML implique que le résultat de la requête est une unique colonne contenant un document XML de racine `PivotSet`. Les lignes sont représentés par les éléments `item` contenant autant d'éléments `column` que de colonnes extraites. Les valeurs calculées sont dans ces éléments terminaux.

Seule la directive `XML` permet d'utiliser une liste de valeurs, une sous-requête ou la clause `ANY` (qui sélectionne toutes les valeurs des colonnes présentes dans la clause `FOR` pour pivot) dans la clause `IN`.

Le tableau suivant présente deux requêtes avec l'option `XML` de l'opérateur `PIVOT`. La première requête extrait la somme des passagers transportés par vol et pour tous les vols. La seconde extrait le total et la moyenne du nombre de passagers en examinant tous les jours (présents dans la table d'origine) avec `ANY`.

Tableau 4-78 Requêtes de pivot avec l'option XML

Requête	Résultat

```
SELECT * FROM
  (SELECT numVol, nb_passagers
          FROM vols)
  PIVOT XML (SUM(nb_passagers) AS pax
    FOR (numVol)
    IN  SELECT DISTINCT numVol
                FROM vols));
```

```
NUMVOL_XML
------------------------------------
<PivotSet>
 <item>
    <column name="NUMVOL">AF6143</column>
    <column name="PAX">210</column>
 </item>
 <item>
    <column name="NUMVOL">BA234</column>
    <column name="PAX">90</column>
 </item>
 <item>
    <column name="NUMVOL">CF56</column>
    <column name="PAX">160</column>
 </item>
 <item>
    <column name="NUMVOL">D009</column>
    <column name="PAX">90</column>
 </item>
</PivotSet>
```

Tableau 4-78 Requêtes de pivot avec l'option XML *(suite)*

Requête	Résultat

`SELECT * FROM`
`  (SELECT jour_id, nb_passagers`
`  FROM vols)`
`  PIVOT XML`
`  (SUM(nb_passagers) AS somme,`
`  AVG(nb_passagers) AS moy`
`  FOR jour_id IN (ANY));`
	`<PivotSet>`
	`<item>`
	`<column name="JOUR_ID">1</column>`
	`<column name="SOMME">60</column>`
	`<column name="MOY">20</column>`
	`</item>`
	`<item>`
	`<column name="JOUR_ID">2</column>`
	`<column name="SOMME">90</column>`
	`<column name="MOY">45</column>`
	`</item>`
	`<item>`
	`<column name="JOUR_ID">3</column>`
	`<column name="SOMME">300</column>`
	`<column name="MOY">75</column>`
	`</item>`
	`<item>`
	`<column name="JOUR_ID">4</column>`
	`<column name="SOMME">100</column>`
	`<column name="MOY">100</column>`
	`</item>`
	`</PivotSet>`

Transpositions (UNPIVOT)

Comme son nom l'indique, l'opérateur `UNPIVOT` réalise l'opération inverse de l'opérateur `PIVOT` en convertissant des données disposées en colonnes sous la forme de lignes. On peut parler de désagrégation (ou transposition). La syntaxe de l'opérateur `UNPIVOT` est la suivante :

```
table_interrogée UNPIVOT [ {INCLUDE | EXCLUDE} NULLS ]
  ( { colonne | (colonne [,colonne]... ) }
  FOR { colonne | (colonne [,colonne]... ) }
  IN ( { colonne | (colonne [,colonne]... ) }
      [  AS { constante | (constante [,constante]... ) } ]
        [, { colonne | (colonne [,colonne]... ) }
          [  AS { constante | (constante [,constante]... ) } ]
        ].) )
```

La clause `UNPIVOT` nomme les colonnes qui vont accueillir des valeurs résultantes de la désagrégation. Il est possible d'inclure ou non des valeurs nulles (par défaut on les exclut). La clause `FOR` compose les colonnes résultantes de la désagrégation. La clause `IN` liste les colonnes (pas les valeurs) à transformer en lignes.

Le mécanisme inverse du pivot est le suivant : parcours des colonnes puis transposition de chaque valeur (en testant éventuellement sa nullité) calculée dans la colonne correspondante. Considérons l'exemple suivant qui décrit les vols d'une semaine (en termes de nombre de passagers transportés).

Tableau 4-79 Données à désagréger

Contenu de la table `vols2`

```
NUMVOL LUNDI  MARDI  MERCREDI  JEUDI  VENDREDI  SAMEDI
------ ------ ------ --------- ------ --------- -------
BA234  20            70
CF56   30     50     80
AF6143 60     40     60        100    180
D009                 90                       10
```

Le tableau suivant présente l'opérateur `UNPIVOT` qui transforme les colonnes des différents jours en lignes par l'apport de la colonne `JOURS`. Ici les valeurs nulles de la table d'origine ne sont pas prises en compte.

Tableau 4-80 Requête avec UNPIVOT

Détail par vol et par jour du nombre de passagers transportés

```
SELECT * FROM vols2
 UNPIVOT (passagers
       FOR jours
        IN  (Lundi, Mardi,
             Mercredi, Jeudi,
             Vendredi, Samedi)
         );
```

Résultat

```
NUMVOL JOURS    PASSAGERS
------ -------- ------------
BA234  LUNDI    20
BA234  MERCREDI 70
CF56   LUNDI    30
CF56   MARDI    50
CF56   MERCREDI 80
AF6143 LUNDI    60
AF6143 MARDI    40
AF6143 MERCREDI 60
AF6143 JEUDI    100
AF6143 VENDREDI 180
D009   MERCREDI 90
D009   SAMEDI   10
```

L'option `INCLUDE NULLS` prend en compte les valeurs nulles. Dans notre exemple, pour transposer les vols en se basant uniquement sur les valeurs nulles, il faut aussi tester la nullité dans le `WHERE` de la requête globale.

Tableau 4-81 Requête avec UNPIVOT et INCLUDE NULLS

Jours où les vols n'ont transporté aucun passager	Résultat

```
SELECT * FROM vols2
   UNPIVOT INCLUDE NULLS
   (passagers
       FOR jours
       IN  (Lundi, Mardi,
            Mercredi, Jeudi,
            Vendredi, Samedi)
   )
WHERE passagers IS NULL;
```

```
NUMVOL JOURS    PASSAGERS
------ -------- -----------
BA234  MARDI
BA234  JEUDI
BA234  VENDREDI
BA234  SAMEDI
CF56   JEUDI
CF56   VENDREDI
CF56   SAMEDI
AF6143 SAMEDI
D009   LUNDI
D009   MARDI
D009   JEUDI
D009   VENDREDI
```

Sans la restriction finale, la liste des vols par jour avec ou sans passagers aurait été retournée (résultat de l'union des deux précédentes requêtes).

Fonction LISTAGG

Analogue au mécanisme de pivot, la fonction `LISTAGG` ordonne des données au sein de chaque groupe cité par `ORDER BY` et concatène le résultat décrit par le premier paramètre sous la forme d'une chaîne de caractères. La syntaxe de cette fonction est la suivante :

```
LISTAGG (expression [, 'delimiteur'])
      WITHIN GROUP (clause_ORDER_BY) [OVER clause_partitionnement]
```

Cette fonction, qui appartient au domaine des fonctions analytiques du fait de l'existence de la clause de partitionnement, ignore les valeurs `NULL`. Le tableau ci-contre présente une utilisation de cette fonction et extrait, pour chaque compagnie, l'identité des pilotes des moins expérimentés aux plus expérimentés.

Si vous ne disposez pas de la version 11*g*, il vous est toutefois `possible` de programmer cette fonction de plusieurs façons notamment à l'aide de la directive `WITH` ou en utilisant conjointement `ROW_NUMBER` et `SYS_CONNECT_BY_PATH` (voir *http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php*).

Tableau 4-82 Extraction avec LISTAGG

Table

```
SQL> SELECT * FROM Pilote;

    BREVET PRENOM     NOM          NBHVOL COMP
---------- ---------- ---------- ------- ----
       100 Steven     King         24000 AF
       101 Neena      Kochhar      17000 AF
       102 Lex        De Haan      17000 EJ
       103 Alexander  Hunold        9000 AF
       104 Bruce      Ernst         6000 EJ
       105 David      Austin        4800 AF
       106 Valli      Pataballa     4800 EJ
       107 Diana      Lorentz       4200 TAT
       108 Nancy      Greenberg    12000 TAT
       109 Daniel     Faviet        9000 AF
```

Requête

```
COL res FORMAT A50 HEADING "Du - au + expérimenté"

SELECT comp,
  LISTAGG(prenom||' '||nom, ' - ')
          WITHIN GROUP
           (ORDER BY nbHVol,nom) AS res,
  MIN(nbHVol), MAX(nbHVol)
FROM Pilote
GROUP BY comp;
```

```
COMP Du - au + expérimenté                                MIN(NBHVOL) MAX(NBHVOL)
---- ---------------------------------------------------- ----------- -----------
AF   David Austin - Daniel Faviet - Alexander Hunold -           4800       24000
     Neena Kochhar - Steven King
EJ   Valli Pataballa - Bruce Ernst - Lex De Haan                 4800       17000
TAT  Diana Lorentz - Nancy Greenberg                             4200       12000
```

Exercices

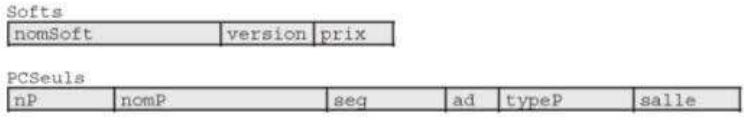
Les objectifs de ces exercices sont :

- de créer des tables et leurs données ;
- d'écrire des requêtes monotables et multitables ;
- de réaliser des modifications synchronisées ;
- de composer des jointures et des divisions.

Exercice 4.1 Création dynamique de tables

Écrivez le script `créaDynamique.sql` permettant de créer les tables `Softs` et `PCSeuls` suivantes (en utilisant la directive `AS SELECT` de la commande `CREATE TABLE`). Vous ne poserez aucune contrainte sur ces tables.

Figure 4-29 *Structures des nouvelles tables*



La table `Softs` sera construite sur la base de tous les enregistrements de la table `Logiciel` que vous avez créée et alimentée précédemment.

La table `PCSeuls` doit seulement contenir les enregistrements de la table `Poste` qui sont de type 'PCWS' ou 'PCNT'.

Vérifier :

```
SELECT * FROM Softs;
SELECT * FROM PCSeuls;
```

Exercice 4.2 Requêtes monotables

Écrivez le script `requêtes.sql`, permettant d'extraire, à l'aide d'instructions `SELECT`, les données suivantes :

1 Type du poste 'p8'.

2 Noms des logiciels Unix.

3 Nom, adresse IP, numéro de salle des postes de type 'Unix' ou 'PCWS'.

4 Même requête pour les postes du segment '130.120.80' triés par numéros de salles décroissants.

5 Numéros des logiciels installés sur le poste 'p6'.

6 Numéros des postes qui hébergent le logiciel 'log1'.

7 Nom et adresse IP complète (ex : '130.120.80.01') des postes de type TX (utiliser l'opérateur de concaténation).

Exercice 4.3 Fonctions et groupements

8 Pour chaque poste, le nombre de logiciels installés (en utilisant la table `Installer`).

9 Pour chaque salle, le nombre de postes (à partir de la table `Poste`).

10 Pour chaque logiciel, le nombre d'installations sur des postes différents.

11 Moyenne des prix des logiciels 'Unix'.

12 Plus récente date d'achat d'un logiciel.

13 Numéros des postes hébergeant 2 logiciels.

14 Nombre de postes hébergeant 2 logiciels (utiliser la requête précédente en faisant un `SELECT` dans la clause `FROM`).

Exercice 4.4 Requêtes multitables

Opérateurs ensemblistes

15 Types de postes non recensés dans le parc informatique (utiliser la table `Types`).

16 Types existant à la fois comme types de postes et de logiciels.

17 Types de postes de travail n'étant pas des types de logiciel.

Jointures procédurales

18 Adresses IP des postes qui hébergent le logiciel 'log6'.

19 Adresses IP des postes qui hébergent le logiciel de nom 'Oracle 8'.

20 Noms des segments possédant exactement trois postes de travail de type 'TX'.

21 Noms des salles ou l'on peut trouver au moins un poste hébergeant le logiciel 'Oracle 6'.

22 Nom du logiciel acheté le plus récent (utiliser la requête 12).

Jointures relationnelles

Écrire les requêtes 18, 19, 20, 21 avec des jointures de la forme relationnelle. Numéroter ces nouvelles requêtes de 23 à 26.

27 Installations (nom segment, nom salle, adresse IP complète, nom logiciel, date d'installation) triées par segment, salle et adresse IP.

Jointures SQL2

Écrire les requêtes 18, 19, 20, 21 avec des jointures SQL2 (`JOIN`, `NATURAL JOIN`, `JOIN USING`). Numéroter ces nouvelles requêtes de 28 à 31.

Exercice 4.5 Modifications synchronisées

Écrivez le script `modifSynchronisées.sql` pour ajouter les lignes suivantes dans la table `Installer` :

Figure 4-30 *Lignes à ajouter*

Installer

nPoste	nLog	numIns	dateIns	delai
...	...	...	...	...
p2	log6	*séquence...*	SYSDATE	NULL
p8	log1		SYSDATE	NULL
p10	log1		SYSDATE	NULL

Écrivez les requêtes `UPDATE` synchronisées de la forme suivante :

```
UPDATE table1 alias1
  SET colonne = (SELECT COUNT(*)
                 FROM  table2 alias2
                 WHERE alias2.colonneA = alias1.colonneB…);
```

Pour mettre à jour automatiquement les colonnes rajoutées :

- `nbSalle` dans la table `Segment` (nombre de salles traversées par le segment) ;
- `nbPoste` dans la table `Segment` (nombre de postes du segment) ;
- `nbInstall` dans la table `Logiciel` (nombre d'installations du logiciel) ;
- `nbLog` dans la table `Poste` (nombre de logiciels installés par poste).

Vérifier le contenu des tables modifiées (`Segment`, `Logiciel` et `Poste`).

Exercice 4.6 Opérateurs existentiels

Ajoutez au script `requêtes.sql`, les instructions `SELECT` pour extraire les données suivantes :

Sous-interrogation synchronisée

32 Noms des postes ayant au moins un logiciel commun au poste 'p6' (on doit trouver les postes p2, p8 et p10).

Divisions

33 Noms des postes ayant les mêmes logiciels que le poste 'p6' (les postes peuvent avoir plus de logiciels que 'p6'). On doit trouver les postes 'p2' et 'p8' (division inexacte).

34 Noms des postes ayant exactement les mêmes logiciels que le poste 'p2' (division exacte), on doit trouver 'p8'.

Exercice 4.7 Extractions dans la base *Chantiers*

Écrivez dans le script `reqchantier.sql` les requêtes SQL permettant d'extraire :

35 Numéro et nom des conducteurs qui étaient sur la route un jour donné (format *jj/mm/aaaa*).

36 Numéro et nom des passagers qui ont visités un chantier un jour donné (format *jj/mm/aaaa*).

37 En déduire le numéro et nom des employés qui n'ont pas bougés de chez eux le même jour.

38 Numéro des chantiers visités entre le 2 et le 3 du mois et d'une année donnés avec le nombre de visites pour chacun d'eux.

39 En déduire les chantiers les plus visités.

40 Nombre de visites de chaque employé (transporté ou conducteur) pour un mois donné.

41 Temps de conduite de chaque conducteur d'un mois donné.

42 Numéro du conducteur qui a fait le plus de kilométrage dans l'année avec le kilométrage total.

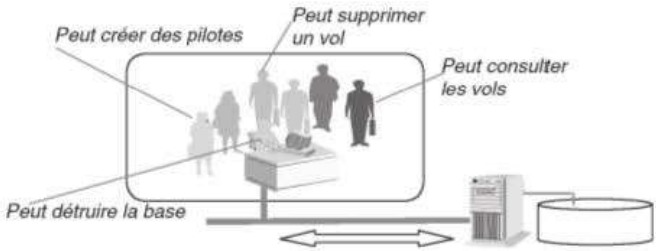
43 Nom et qualification du conducteur autorisé à piloter tous les types de véhicule.

Chapitre 5

Contrôle des données

Comme dans tout système multi-utilisateur, l'usager d'un SGBD sera toujours identifié avant d'employer des ressources restreintes (à moins d'être l'administrateur principal : le compte `sys` d'Oracle). L'accès aux données doit toujours être contrôlé à des fins de sécurité et de confidentialité. La figure suivante illustre un groupe d'utilisateurs aux profils divers.

Figure 5-1 *Conséquences de l'aspect multi-utilisateur*



Cette section décrit l'utilisation de SQL pour contrôler l'accès aux données au travers des mécanismes suivants :

- utilisateurs et espaces de stockage (*tablespaces*) ;
- privilèges système et privilèges objet ;
- rôles, vues et synonymes ;
- dictionnaire des données.

Le dernier paragraphe « Le multitenant » est spécifique à la version 12*c* dont l'architecture diffère radicalement des éditions précédentes et apporte un grand nombre de nouveautés notamment au niveau des utilisateurs et du dictionnaire des données. Sans entrer dans les détails, décrivons tout d'abord brièvement le concept de *tablespace* qui concerne directement les utilisateurs et les objets qu'ils possèdent.

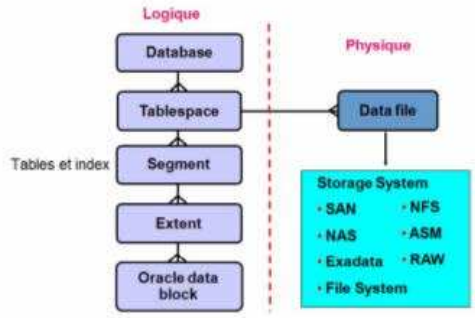
Les tablespaces

Comme dans tout SGBD relationnel, l'indépendance entre le niveau physique (les fichiers et répertoires dans le système d'exploitation) et le niveau logique (ce qu'on présente à l'utilisateur) est masqué.

Indépendance logique/physique

La figure 5-2 illustre les différents mécanismes mis en œuvre par Oracle.

Figure 5-2 *Correspondances logique-physique*



- Le bloc de données (*data block*, aussi appelé page) est la partie la plus petite qu'Oracle est capable de manipuler comme un tout, de la mémoire au disque (souvent 8 Ko par défaut). Il contiendra principalement les données des tables et des index.
- Un *extent* est un ensemble de blocs contigus sur le disque (souvent 64 Ko par défaut, soit 8 blocs). Une table ou un index a donc au minimum une taille égale à celle d'un extent.
- Un segment (table ou index) regroupe plusieurs extents, pas forcément contigus (une table peut être créée un moment, puis occuper de la place au fur et à mesure tandis qu'une autre table ou un autre index réserve aussi des extents).
- Un tablespace regroupe des segments et les stocke sur le disque dans un ou plusieurs fichiers du système d'exploitation (*datafile*).

Tablespaces déjà livrés

Le mécanisme des tablespaces présente beaucoup d'avantages pour l'administrateur : exportation et sauvegarde de parties de la base, déplacement des données d'un disque à l'autre, etc. Les tablespaces qui préexistent dans votre base sont :

- SYSTEM et SYSAUX qui contiennent notamment le dictionnaire des données, les procédures cataloguées et les déclencheurs de tout le monde ;
- USERS proposé par défaut pour stocker vos données ;
- TEMP qui peut agir en complément de la mémoire pour vos tris et jointures ;
- UNDOTBS1 qui permet la lecture consistante du mode transactionnel (*rollback segment*).

Le tableau 5-1 présente différentes interrogations du dictionnaire des données pour retrouver certaines caractéristiques des composants des tablespaces présents.

Tableau 5-1 Caractéristiques des tablespaces

Requêtes et résultats	Commentaires

`SELECT tablespace_name AS espace, block_size,`
`initial_extent, status, contents`
`FROM dba_tablespaces`
`ORDER BY tablespace_name;`
 	
`ESPACE BLOCK_SIZE INITIAL_EXTENT STATUS CONTENTS`
`SYSAUX 8192 65536 ONLINE PERMANENT`
`SYSTEM 8192 65536 ONLINE PERMANENT`
`TEMP 8192 1048576 ONLINE TEMPORARY`
`UNDOTBS1 8192 65536 ONLINE UNDO`
`USERS 8192 65536 ONLINE PERMANENT`
	La taille du bloc de chaque espace est de 8 Ko et de 64 Ko pour le premier extent de la plupart des espaces. Les données seront par ailleurs permanentes pour certains.

`SELECT tablespace_name AS espace, file_name`
`FROM dba_data_files;`
 	
`ESPACE FILE_NAME`
`USERS C:\APP\CSOUTOU\ORADATA\ORCL\USERS01.DBF`
`UNDOTBS1 C:\APP\CSOUTOU\ORADATA\ORCL\UNDOTBS01.DBF`
`SYSAUX C:\APP\CSOUTOU\ORADATA\ORCL\SYSAUX01.DBF`
`SYSTEM C:\APP\CSOUTOU\ORADATA\ORCL\SYSTEM01.DBF`
	Localisation des fichiers physiques de chaque espace.

`SELECT tablespace_name AS espace,`
`bytes/1024/1024 AS "Taille en Mo",`
`user_bytes/1024/1024 AS "Dispo en Mo"`
`FROM dba_data_files;`
 	
`ESPACE Taille en Mo Dispo en Mo`
`USERS 5 4`
`UNDOTBS1 895 894`
`SYSAUX 980 979`
`SYSTEM 790 789`
	Taille occupée et disponible de chaque espace.

Création d'un tablespace

Selon les versions, un tablespace traditionnel (*smallfile*) peut contenir au plus 1 024 datafiles. Un tablespace de type *bigfile* (depuis la version 10*g*) ne contiendra qu'un seul datafile (dont la taille n'a plus beaucoup de limites…). La syntaxe très simplifiée de création d'un tablespace (avec les options par défaut : *locally managed auto* et *allocated tablespace*) est la suivante (`K` désigne un Ko, `M` un Mo, etc.).

```
CREATE [BIGFILE | SMALLFILE] TABLESPACE nom_espace
  DATAFILE 'nom_fichier' SIZE taille_initialle K/M/G/T/P/E [ REUSE ]
  AUTOEXTEND {OFF | ON   NEXT valeur_extension K/M/G/T/P/E}
  MAXSIZE { UNLIMITED |       taille_maxi       K/M/G/T/P/E };
```

Le tableau 5-2 présente la création d'un tablespace personnalisé.

Tableau 5-2 Création d'un tablespace

Requête et résultat	Commentaires

`SQL> CREATE SMALLFILE TABLESPACE ts_eyrolles`
`DATAFILE`
`'C:\APP\CSOUTOU\ORADATA\ORCL\tblspc_eyrolles.data'`
`SIZE 4 M REUSE`
`AUTOEXTEND ON NEXT 1 M`
`MAXSIZE 1 G;`
 	
`Tablespace créé.`
	L'espace `ts_eyrolles` est créé dans le fichier `tblspc_eyrolles.data` (4 Mo), situé à l'emplacement `C:\APP\CSOUTOU\ORADATA\ORCL`. Chaque extension nécessaire demandera 1 Mo supplémentaire et le fichier ne devra pas dépasser 1 Go.

Si vous utilisez la version 12*c* avec l'option multitenant (voir la section « Le multitenant » en fin de chapitre), vous devrez vous placer dans une base enfichable (`ALTER SESSION SET CONTAINER=…`) sous peine de créer un espace dans le container *root*.

Gestion des utilisateurs

Un utilisateur (*user*) sera identifié par son nom (*username*) et un mot de passe permettant de se connecter, puis d'exécuter des instructions et d'accéder aux données sous réserve d'avoir reçu des privilèges. Comme nous l'avons présenté dans l'introduction, un schéma consiste en une collection d'objets (tables, séquences, index, procédures, etc.) dont le propriétaire (*owner*) est

l'utilisateur. L'utilisateur et le schéma ont le même nom et seule l'instruction CREATE USER permet de créer un utilisateur.

Classification

Les types d'utilisateurs, leurs fonctions et leur nombre peuvent varier d'une base à une autre. Néanmoins, pour chaque base de données en activité, on peut classifier les utilisateurs de la manière suivante.

- Le (ou les) DBA (*DataBase Administrator*), les comptes livrés par Oracle après l'installation system et sys sont prévus pour cela. De nombreuses tâches leur seront confiées, notamment :
  - l'installation et la migration des bases ;
  - la gestion du réseau, de l'espace disque et des utilisateurs ;
  - les sauvegardes et restaurations ;
  - l'optimisation des performances (*tuning*).
- Les développeurs qui interagissent avec les DBA (droits, stockage, optimisation, sécurité, etc.).
- Les utilisateurs finaux qui doivent se connecter via des applications ou des outils (génération de rapports, modifications de données, etc.).

Tous seront des utilisateurs (au sens Oracle) avec des privilèges différents.

Création d'un utilisateur (CREATE USER)

Pour pouvoir créer un utilisateur vous devez posséder le privilège CREATE USER.

La syntaxe SQL de création d'un utilisateur est la suivante :

```
CREATE USER utilisateur IDENTIFIED
   { BY motdePasse | EXTERNALLY | GLOBALLY AS 'nomExterne' }
   [ DEFAULT TABLESPACE nomTablespace
     [QUOTA { entier [ K | M ] | UNLIMITED } ON nomTablespace ] ]
   [TEMPORARY TABLESPACE nomTablespace
     [QUOTA { entier [ K | M ] | UNLIMITED } ON nomTablespace ].]
   [PROFILE nomProfil ]  [PASSWORD EXPIRE ]  [ ACCOUNT { LOCK | UNLOCK
   } ] ;
```

- IDENTIFIED BY *motdePasse* permet d'affecter un mot de passe à un utilisateur local (cas le plus courant et le plus simple).
- IDENTIFIED BY EXTERNALLY permet de se servir de l'authenticité du système d'exploitation pour s'identifier à Oracle (cas des compte OPS$ pour Unix).

- `IDENTIFIED BY GLOBALLY` permet de se servir de l'authenticité d'un système d'annuaire.
- `DEFAULT TABLESPACE nomTablespace` associe un espace disque de travail (appelé *tablespace*) à l'utilisateur.
- `TEMPORARY TABLESPACE nomTablespace` associe un espace disque temporaire (dans lequel certaines opérations se dérouleront) à l'utilisateur.
- `QUOTA` permet de limiter ou pas chaque espace alloué.
- `PROFILE nomProfil` affecte un profil (caractéristiques système relatives au CPU et aux connexions) à l'utilisateur.
- `PASSWORD EXPIRE` pour obliger l'utilisateur à changer son mot de passe à la première connexion (par défaut il est libre). Le DBA peut aussi changer ce mot de passe.
- `ACCOUNT` pour verrouiller ou libérer l'accès à la base (par défaut `UNLOCK`).

Si vous ne renseignez pas l'espace par défaut, le tablespace `SYSTEM` pourra être associé à l'utilisateur en tant qu'espace de travail (et même d'espace temporaire) ! Utilisez donc toujours explicitement soit les tablespaces prédéfinis (`USERS` et `TEMP`), soit ceux que vous aurez créés préalablement. Quantifiez également (quota) le volume d'espace prévisionnel pour chaque utilisateur sur tous les espaces qu'il utilisera, mis à part les espaces temporaires.

La clause `ALTER USER` vous permettra de modifier un utilisateur pour changer notamment l'espace par défaut et l'espace temporaire. Si aucun profil n'existe (voir la section suivante « Profils »), le profil `DEFAULT` sera affecté à l'utilisateur.

Le tableau suivant décrit la création de deux utilisateurs.

Tableau 5-3 Création d'utilisateurs

Instruction SQL	Résultat

`CREATE USER dev1`
`IDENTIFIED BY dev1`
`DEFAULT TABLESPACE users`
`QUOTA 10M ON users`
`QUOTA 1M ON ts_eyrolles`
`TEMPORARY TABLESPACE temp`
`PASSWORD EXPIRE;`
	L'utilisateur `dev1` pourra créer des objets dans les espaces `USERS` et `TS_EYROLLES`. Il devra changer son mot de passe à la première connexion.

`CREATE USER dev2`
`IDENTIFIED BY dev2`
`DEFAULT TABLESPACE users`
`QUOTA 10M ON users`
`TEMPORARY TABLESPACE temp`
`ACCOUNT LOCK;`
	L'utilisateur `dev2` ne peut utiliser que 10 Mo sur l'espace `USERS`. Son compte est pour l'instant bloqué.

Par défaut, les utilisateurs, une fois créés n'ont aucun droit sur la base de données sur laquelle ils sont connectés. La section « Privilèges » étudie ces droits.

Selon les versions et les éditions, l'installation par défaut affecte des mots de passe (pour `sys` c'est `change_on_install` et pour `system`, il s'agit de `manager`). Il est préférable que vous utilisiez `system` à la place de `sys` (plutôt réservé à de lourdes tâches comme la création d'une base, la sauvegarde, son arrêt, etc.). L'affectation du rôle `DBA` (voir la section « Rôles ») est à réaliser avec précaution. De plus, il n'inclut pas les privilèges système `SYSDBA` et `SYSOPER`.

Modification d'un utilisateur (ALTER USER)

Pour pouvoir modifier les caractéristiques d'un utilisateur (autres que celle du mot de passe) vous devez posséder le privilège `ALTER USER`.

La syntaxe simplifiée SQL pour modifier un utilisateur est la suivante. Cette instruction reprend les options étudiées lors de la création d'un utilisateur.

```
ALTER USER utilisateur
   [ IDENTIFIED { BY password [ REPLACE old_password ] |
                  EXTERNALLY | GLOBALLY AS 'external_name' } ]
   [ DEFAULT TABLESPACE nomTablespace
      [QUOTA { entier [ K | M ] | UNLIMITED } ON nomTablespace ] ]
   [ TEMPORARY TABLESPACE nomTablespace
      [QUOTA { entier [ K | M ] | UNLIMITED } ON nomTablespace ].]
   [ PROFILE nomProfil ]
   [ DEFAULT ROLE { rôle1 [,rôle2]… | ALL [EXCEPT rôle1 [,rôle2]…] |
   NONE }
   [ PASSWORD EXPIRE ]  [ ACCOUNT { LOCK | UNLOCK } ] ;
```

- `PASSWORD EXPIRE` oblige l'utilisateur à changer son mot de passe à la prochaine connexion.
- `DEFAULT ROLE` affecte à l'utilisateur des rôles qui sont en fait des ensembles de privilèges (voir la section « Rôles »).

Chaque utilisateur peut changer son propre mot de passe à l'aide de cette instruction. Les autres changements seront opérationnels aux prochaines sessions de l'utilisateur mais pas à la session courante (cas de l'utilisateur qui déclare un espace de travail alors qu'il est couramment connecté à un autre).

Le tableau suivant décrit des modifications des utilisateurs créés auparavant :

Tableau 5-4 Modification d'utilisateurs

Instruction SQL	Résultat
`ALTER USER dev1 IDENTIFIED BY mdp_dev1 QUOTA UNLIMITED ON ts_eyrolles;`	`dev1` change de mot de passe et son quota n'est plus limité sur l'un des espaces sur lesquels il pouvait accéder.
`ALTER USER dev2 QUOTA 2M ON ts_eyrolles ACCOUNT UNLOCK;`	`dev2` peut accéder à un autre espace dans la limite de 2 Mo. Le compte est débloqué.

Suppression d'un utilisateur (DROP USER)

Pour pouvoir supprimer un utilisateur vous devez posséder le privilège `DROP USER`. Un utilisateur connecté ne peut pas être supprimé en direct avec cette commande. Pour forcer cette suppression, il faut arrêter ses sessions par la commande `ALTER SYSTEM` et l'option `KILL SESSION`. Si vous désirez effacer juste l'utilisateur en tant qu'entrée dans la base sans supprimer ses objets, préférez le retrait par `REVOKE` du privilège `CREATE SESSION`.

La syntaxe SQL pour supprimer un utilisateur est la suivante :

```
DROP USER utilisateur [CASCADE];
```

Oracle ne supprime pas par défaut un utilisateur s'il possède des objets (tables, séquences, index, déclencheurs, etc.). L'option `CASCADE` force la suppression et détruit tous les objets du schéma de l'utilisateur.

Les contraintes d'intégrité d'autres schémas qui référençaient des tables du schéma à détruire sont aussi supprimées.

Les vues, synonymes, procédures ou fonctions cataloguées définis à partir du schéma détruit mais présents dans d'autres schémas ne sont pas supprimés mais invalidés.

Les rôles définis par l'utilisateur à supprimer ne sont pas détruits par l'instruction `DROP USER`.

Profils

Un profil regroupe des caractéristiques système (ressources) qu'il est possible d'affecter à un ou plusieurs utilisateurs. Un profil est identifié par son nom. Un profil est créé par `CREATE PROFILE`, modifié par `ALTER PROFILE` et supprimé par `DROP PROFILE`. Il est affecté à un utilisateur lors de sa création par `CREATE USER` ou après que l'utilisateur est créé par `ALTER USER`. Le profil `DEFAULT` est affecté par défaut à chaque utilisateur si aucun profil défini n'est précisé.

Création d'un profil (CREATE PROFILE)

Pour pouvoir créer un profil vous devez posséder le privilège CREATE PROFILE. La syntaxe SQL est la suivante :

```
CREATE PROFILE nomProfil LIMIT
{ ParamètreRessource | ParamètreMotdePasse }
[ ParamètreRessource | ParamètreMotdePasse ]…;

ParamètreRessource :
{ { SESSIONS_PER_USER | CPU_PER_SESSION | CPU_PER_CALL
  | CONNECT_TIME | IDLE_TIME | LOGICAL_READS_PER_SESSION
  | LOGICAL_READS_PER_CALL | COMPOSITE_LIMIT } { entier | UNLIMITED |
  DEFAULT }
  | PRIVATE_SGA {entier[K|M] | UNLIMITED | DEFAULT}}

ParamètreMotdePasse :
{ FAILED_LOGIN_ATTEMPTS | PASSWORD_LIFE_TIME | PASSWORD_REUSE_TIME
  | PASSWORD_REUSE_MAX | PASSWORD_LOCK_TIME | PASSWORD_GRACE_TIME }
  { expression | UNLIMITED | DEFAULT } }
```

Les options principales sont les suivantes :

- SESSIONS_PER_USER : nombre de sessions concurrentes autorisées.
- CPU_PER_SESSION : temps CPU maximal pour une session en centièmes de secondes.
- CPU_PER_CALL : temps CPU autorisé pour un appel noyau en centièmes de secondes.
- CONNECT_TIME : temps total autorisé pour une session en minutes (pratique pour les examens de TP minutés).
- IDLE_TIME : temps d'inactivité autorisé, en minutes, au sein d'une même session (pour les étudiants qui ne clôturent jamais leurs sessions).
- PRIVATE_SGA : espace mémoire privé alloué dans la SGA (*System Global Area*).
- FAILED_LOGIN_ATTEMPTS : nombre de tentatives de connexion avant de bloquer l'utilisateur (pour la carte bleue, c'est trois).
- PASSWORD_LIFE_TIME : nombre de jours de validité du mot de passe (il expire s'il n'est pas changé au cours de cette période).
- PASSWORD_REUSE_TIME : nombre de jours avant que le mot de passe puisse être utilisé à nouveau. Si ce paramètre est initialisé à un entier, le paramètre PASSWORD_REUSE_MAX doit être passé à UNLIMITED.
- PASSWORD_REUSE_MAX : nombre de modifications de mot de passe avant de pouvoir réutiliser le mot de passe courant. Si ce paramètre est initialisé à un entier, le paramètre PASSWORD_REUSE_TIME doit être passé à UNLIMITED.

- PASSWORD_LOCK_TIME : nombre de jours d'interdiction d'accès à un compte après que le nombre de tentatives de connexions a été atteint (pour la carte bleue, ça dépend de plein de choses, de toute façon vous en recevrez une autre toute neuve mais toute chère…).
- PASSWORD_GRACE_TIME : nombre de jours d'une période de grâce qui prolonge l'utilisation du mot de passe avant son changement (un message d'avertissement s'affiche lors des connexions). Après cette période le mot de passe expire.

Les limites des ressources qui ne sont pas spécifiées sont initialisées avec les valeurs du profil DEFAULT. Par défaut toutes les limites du profil DEFAULT sont à UNLIMITED. Il est possible de visualiser chaque paramètre de tout profil en interrogeant certaines vues du dictionnaire des données (voir le chapitre suivant).

Exemple

Le tableau suivant décrit la création d'un profil et l'explication de ses options.

Tableau 5-5 Modification d'utilisateurs

Instructions SQL		Explications
CREATE PROFILE profil_Etudiants **LIMIT**		• 3 sessions simultanées autorisées.
SESSIONS_PER_USER	3	• Un appel système ne peut pas consommer plus de 30 secondes de CPU.
CPU_PER_CALL	3000	• Chaque session ne peut excéder 45 minutes.
CONNECT_TIME	45	• Un appel système ne peut lire plus de 1 000 blocs de données en mémoire et sur le disque.
LOGICAL_READS_PER_CALL	1000	• Chaque session ne peut allouer plus de 15 ko de mémoire en SGA.
PRIVATE_SGA	15K	• Pour chaque session, 40 minutes d'inactivité maximum sont autorisées.
IDLE_TIME	40	• 5 tentatives de connexion avant blocage du compte.
FAILED_LOGIN_ATTEMPTS	5	• Le mot de passe est valable pendant 70 jours et il faudra attendre 60 jours avant qu'il puisse être utilisé à nouveau.
PASSWORD_LIFE_TIME	70	• 1 seul jour d'interdiction d'accès après que les 5 tentatives de connexion ont été atteintes.
PASSWORD_REUSE_TIME	60	• La période de grâce qui prolonge l'utilisation du mot de passe avant son changement est de 10 jours.
PASSWORD_REUSE_MAX	UNLIMITED	
PASSWORD_LOCK_TIME	1/24	
PASSWORD_GRACE_TIME	10;	

L'affectation de ce profil à l'utilisateur Paul est réalisée via l'instruction ALTER USER suivante :

```
ALTER USER Paul PROFILE profil_Etudiants ;
```

Modification d'un profil (ALTER PROFILE)

Pour pouvoir modifier un profil, vous devez posséder le privilège ALTER PROFILE. La syntaxe SQL est la suivante, elle utilise les options étudiées lors de la création d'un profil :

```
ALTER PROFILE nomProfil LIMIT
{ ParamètreRessource | ParamètreMotdePasse }
[ ParamètreRessource | ParamètreMotdePasse ]…;
```

Il est plus prudent de restreindre certaines valeurs du profil DEFAULT à l'aide de cette commande (ALTER PROFILE DEFAULT LIMIT…).

Suppression d'un profil (DROP PROFILE)

Pour pouvoir supprimer un profil, vous devez posséder le privilège DROP PROFILE. Le profil DEFAULT ne peut pas être supprimé. La syntaxe SQL est la suivante :

```
DROP PROFILE nomProfil [CASCADE] ;
```

- CASCADE permet de supprimer le profil même si des utilisateurs en sont pourvus (option obligatoire dans ce cas) et affecte le profil DEFAULT à ces derniers.

Privilèges

Depuis le début du livre nous avons parlé de privilèges, il est temps à présent de préciser ce que recouvre ce terme. Un privilège (sous-entendu utilisateur) est un droit d'exécuter une certaine instruction SQL (on parle de privilège système), ou un droit d'accéder à un certain objet d'un autre schéma (on parle de privilège objet). Les privilèges système diffèrent sensiblement d'un SGBD à un autre. En revanche, les privilèges objets sont les mêmes et sont tous pris en charge via les instructions GRANT et REVOKE.

Les privilèges assortis de la mention ANY donnent la possibilité au bénéficiaire de s'en servir dans tout schéma (n'incluant pas par défaut celui de l'utilisateur SYS). Par exemple le privilège CREATE ANY TABLE permet de créer des tables dans tout schéma alors que le privilège CREATE TABLE ne permet de créer des tables que dans son propre schéma.

Privilèges système

Il existe une centaine de privilèges système. Citons par exemple la création d'utilisateurs (CREATE USER), la création et la suppression de tables (CREATE/DROP TABLE), la création d'espaces (CREATE TABLESPACE), la sauvegarde des tables (BACKUP ANY TABLE), etc.

Nous indiquons ici quelques privilèges système relatifs aux notions étudiées jusqu'ici. La liste complète de tous les privilèges (système et objets, ainsi que les rôles prédéfinis) se trouve dans la documentation à la fin de la commande GRANT du livre électronique *SQL Reference*.

Tableau 5-6 Options possibles de quelques privilèges système

Privilège	ALTER	CREATE	DROP	Autre
INDEX		×		QUERY REWRITE (index basés sur des fonctions)
ANY INDEX	×	×	×	
TABLE		×		
ANY TABLE	×	×	×	BACKUP, INSERT, DELETE, SELECT, UPDATE
USER	×	×	×	BECOME (pour des importations de bases)
PROFILE	×	×	×	
SEQUENCE		×		
ANY SEQUENCE	×	×	×	SELECT (pour utiliser toute séquence)
ANY OBJECT PRIVILEGE				pour manipuler tout objet

Attribution de privilèges système (GRANT)

La commande GRANT permet d'attribuer un ou plusieurs privilèges à un ou plusieurs bénéficiaires. Nous étudierons les rôles dans la section suivante. L'utilisateur qui exécute cette commande doit avoir reçu lui-même le droit de transmettre ces privilèges. Dans le cas des utilisateurs SYS et SYSTEM, la question ne se pose pas car ils ont tous les droits. La syntaxe est la suivante :

```
GRANT { privilègeSystème | nomRôle | ALL PRIVILEGES }
   [, { privilègeSystème | nomRôle | ALL PRIVILEGES }]…
   TO { utilisateur | nomRôle | PUBLIC } [,{ utilisateur | nomRôle |
   PUBLIC } ]…
   [ IDENTIFIED BY motdePasse ]
   [ WITH ADMIN OPTION ] ;
```

- *privilègeSystème* : description du privilège système (exemple CREATE TABLE, CREATE SESSION, etc.).
- ALL PRIVILEGES : tous les privilèges système.
- PUBLIC : pour attribuer le(s) privilège(s) à tous les utilisateurs.
- IDENTIFIED BY désigne un utilisateur encore inexistant dans la base. Cette option n'est pas valide si le bénéficiaire est un rôle ou est PUBLIC.
- WITH ADMIN OPTION : permet d'attribuer aux bénéficiaires le droit de retransmettre le(s) privilège(s) reçu(s) à une tierce personne (utilisateur(s) ou rôle(s)).

Le tableau suivant décrit l'affectation de quelques privilèges système en donnant les explications associées.

Tableau 5-7 Affectation de privilèges système

Administrateur	Explications
`GRANT CREATE SESSION, CREATE SEQUENCE, CREATE TABLE TO dev1;`	`dev1` peut se connecter avec SQL*Plus, SQL Developer ou par le bais de tout programme sous réserve de disposer d'un pilote adéquat. Il peut aussi créer des séquences et des tables dans son schéma.
`GRANT CREATE SESSION, CREATE ANY TABLE, DROP ANY TABLE TO dev2;`	`dev2` peut se connecter. Il peut également créer et détruire des tables de tout schéma (de la base enfichable concernée si l'architecture multitenant est mise en œuvre).

Révocation de privilèges système (REVOKE)

La révocation d'un ou de plusieurs privilèges est réalisée par l'instruction `REVOKE`. Cette commande permet d'annuler un privilège système ou un rôle d'un utilisateur ou d'un rôle. Nous verrons aussi que cette commande est opérationnelle pour les privilèges objets. Pour pouvoir révoquer un privilège ou un rôle, vous devez détenir au préalable ce privilège avec l'option `WITH ADMIN OPTION`.

```
REVOKE
   { privilègeSystème | nomRôle | ALL PRIVILEGES }
   [,{ privilègeSystème | nomRôle }]…
   FROM { utilisateur | nomRôle | PUBLIC } [,{ utilisateur | nomRôle
   } ]…  ;
```

Les options sont les mêmes que pour la commande `GRANT`.

- `ALL PRIVILEGES` (valable si l'utilisateur ou le rôle ont tous les privilèges système).
- `PUBLIC` pour annuler le(s) privilège(s) à chaque utilisateur ayant reçu ce(s) privilège(s) par l'option `PUBLIC`.

Le tableau suivant décrit la révocation de certains privilèges acquis des utilisateurs de notre exemple.

Tableau 5-8 Révocation de privilèges système

Administrateur	Explications
`REVOKE CREATE SESSION FROM dev1, dev2;`	Les utilisateurs `dev1` et `dev2` ne peuvent plus accéder à la base tout en conservant les privilèges déjà acquis.

Privilèges objets

Les privilèges objets sont relatifs aux données de la base et aux actions sur les objets (table, vue, séquence, procédure). Chaque type d'objet a différents privilèges associés comme l'indique le tableau suivant. Nous ne montrons ici que quelques-unes des possibilités de privilèges

objets. Il existe d'autres options de cette instruction concernant le stockage de LOB, l'accès à des répertoires (DIRECTORY) et aux ressources Java.

Tableau 5-9 Options possibles de quelques privilèges objets

Privilège	Table	Vue	Séquence	Programme PL/SQL
ALTER			×	
DELETE	×	×		
EXECUTE				×
INDEX	×			
INSERT	×	×		
REFERENCES	×			
SELECT	×	×	×	
UPDATE	×	×		

Attribution de privilèges objets (GRANT)

L'instruction GRANT permet d'attribuer un (ou plusieurs) privilège à un (ou plusieurs) objet à un (ou des) bénéficiaire (ou plusieurs). L'utilisateur qui exécute cette commande doit avoir reçu lui-même le droit de transmettre ces privilèges (sauf s'il s'agit de ses propres objets pour lesquels il possède automatiquement les privilèges avec l'option GRANT OPTION).

```
GRANT { privilègeObjet | nomRôle | ALL PRIVILEGES } [(colonne1
[,colonne2]…)]
   [, { privilègeObjet | nomRôle | ALL PRIVILEGES }] [(colonne1
   [,colonne2]…)]…
ON { [schéma.]nomObjet | { DIRECTORY nomRépertoire
   | JAVA { SOURCE | RESOURCE } [schéma.]nomObjet } }
TO { utilisateur | nomRôle | PUBLIC } [,{ utilisateur | nomRôle |
PUBLIC } ]…
[WITH GRANT OPTION] ;
```

- *privilègeObjet* : description du privilège objet (ex : SELECT, DELETE, etc.).
- *colonne* précise la ou les colonnes sur lesquelles se porte le privilège INSERT, REFERENCES, ou UPDATE (exemple : UPDATE(typeAvion) pour n'autoriser que la modification de la colonne typeAvion).
- ALL PRIVILEGES donne tous les privilèges avec l'option GRANT OPTION) l'objet en question.
- PUBLIC : pour attribuer le(s) privilège(s) à tous les utilisateurs.
- WITH GRANT OPTION : permet de donner aux bénéficiaires le droit de retransmettre les privilèges reçus à une tierce personne (utilisateur(s) ou rôle(s)).

Le tableau suivant décrit un scénario d'affectation de quelques privilèges objets entre deux utilisateurs.

Tableau 5-10 Affectations de privilèges objets

laurent_navarro	*christian_soutou*

```
--Table Pilote
BREVET NOM                 AGE VILLE
------ ---------- ---------- --------
P1     Sarda                46 Balma
P2     Giaconne             64 Toulouse
P3     Calac                53 Cugnaux
P4     Gazagne              63 Toulouse
```

```
--Table Qualif
TYPEQUALIF PIL
---------- ------
CPL        P4
FI/A       P1
FI/A       P2
IR         P3
PL         P1
PPL        P4
```

Affectation à l'utilisateur *christian_soutou* des privilèges (1) de lecture sur la table `Pilote`, (2) de modification sur la colonne `ville` et (3) de référence à la clé `brevet`.

```
GRANT REFERENCES(brevet),
      UPDATE(ville),
      SELECT ON Pilote
TO christian_soutou;
```

Modification d'une ville dans la table `Pilote` du schéma *laurent_navarro*.

```
UPDATE  laurent_navarro.Pilote
SET     ville = 'Castanet'
WHERE   brevet = 'P3';
```

Lecture de la table `Pilote` du schéma *laurent_navarro*.

```
SELECT  brevet, nom, ville
FROM    laurent_navarro.Pilote
WHERE   ville = 'Castanet';

BREVET NOM        VILLE
------ ---------- ------------
P3     Calac      Castanet
```

Déclaration d'une clé étrangère vers la table `Pilote` du schéma *laurent_navarro*.

```
ALTER TABLE    Qualifications
ADD CONSTRAINT dans_Pilote_laurent_
navarro
    FOREIGN KEY(pil)
    REFERENCES laurent_
navarro.Pilote(brevet);
```

Le privilège `REFERENCES` permet de pouvoir déclarer et de bénéficier d'une contrainte d'intégrité interschémas. Dans l'exemple précédent, les qualifications sont soumises à l'existence des pilotes situés dans un autre schéma.

Révocation de privilèges objets

Pour pouvoir révoquer un privilège objet, vous devez détenir au préalable cette permission ou avoir reçu le privilège système ANY OBJECT PRIVILEGE. Il n'est pas possible d'annuler un privilège objet qui a été accordé avec l'option WITH GRANT OPTION.

```
REVOKE { privilègeObjet | ALL PRIVILEGES } [(colonne1 [,colonne2]…)]
   [, { privilègeObjet | ALL PRIVILEGES }] [(colonne1 [,colonne2]…)]…
ON { [schéma.]nomObjet | { DIRECTORY nomRépertoire
   | JAVA { SOURCE | RESOURCE } [schéma.]nomObjet } }
FROM { utilisateur | nomRôle | PUBLIC } [,{ utilisateur | nomRôle |
PUBLIC } ]…
[CASCADE CONSTRAINTS] [FORCE];
```

Certaines options sont similaires à celles de la commande GRANT. Les autres sont expliquées ci-après :

- CASCADE CONSTRAINTS concerne les privilèges REFERENCES ou ALL PRIVILEGES. Cette option permet de supprimer la contrainte référentielle entre deux tables de schémas distincts.
- FORCE : concerne les privilèges EXECUTE sur les types (extensions SQL3). En ce cas, tous les objets dépendants (types, tables ou vues) sont marqués INVALID et les index sont notés UNUSABLE.

Le tableau suivant décrit la révocation des privilèges de l'utilisateur *christian_soutou* :

Tableau 5-11 Révocation de privilèges objets

laurent_navarro	Explications
`REVOKE UPDATE, SELECT ON Pilote FROM christian_soutou;`	*christian_soutou* ne peut plus ni modifier ni lire la table `Pilote` du schéma *laurent_navarro*.
`REVOKE REFERENCES ON Pilote FROM christian_soutou CASCADE CONSTRAINTS;`	*christian_soutou* ne peut plus bénéficier de la table `Pilote` du schéma *laurent_navarro* en tant que référence (l'option `CASCADE CONSTRAINT` est requise).

Privilèges prédéfinis

Oracle propose des privilèges prédéfinis pour faciliter la gestion des droits. Le tableau suivant en décrit quelques-uns :

Tableau 5-12 Privilèges prédéfinis

Nom	Privilèges
GRANT ANY PRIVILEGE	Autorisation de donner tout privilège système.
GRANT ANY OBJECT PRIVILEGE	Autorisation de donner tout privilège objet.
COMMENT ANY TABLE	Commenter une table, vue ou colonne de tout schéma.
SELECT ANY DICTIONARY	Interroger les objets du dictionnaire des données (schéma SYS).
SYSDBA	ALTER DATABASE OPEN | MOUNT | BACKUP, CREATE DATABASE, ARCHIVELOG, RECOVERY, CREATE SPFILE, RESTRICTED SESSION
SYSOPER	Idem sauf CREATE DATABASE

Le code suivant donne puis reprend la possibilité d'autoriser tout privilège à l'utilisateur *christian_soutou*. Non, il n'y a pas d'erreur, deux GRANT se suivent, et un GRANT suit un REVOKE.

```
GRANT GRANT ANY OBJECT PRIVILEGE ,
      GRANT ANY PRIVILEGE  TO christian_soutou;

REVOKE GRANT ANY OBJECT ,
       GRANT ANY PRIVILEGE  FROM christian_soutou;
```

Les privilèges système SYSDBA et SYSOPER sont nécessaires pour qu'un utilisateur puisse démarrer (*startup*) ou arrêter (*shutdown*) la base de données. Pour une connexion avec le privilège SYSDBA, vous êtes dans le schéma de SYS. Avec SYSOPER, vous êtes dans le schéma PUBLIC. Les privilèges SYSOPER sont inclus dans ceux de SYSDBA.

Il est à noter qu'un utilisateur créé simplement (avec les rôles CONNECT et RESOURCE) ne peut pas lancer la console. Pour ce faire, il faut lui attribuer le droit SELECT ANY DICTIONARY. Sous SQL*Plus la manipulation à faire est la suivante :

Sous SYS ou SYSTEM dans SQL*Plus :

```
GRANT SELECT ANY DICTIONARY TO utilisateur;
```

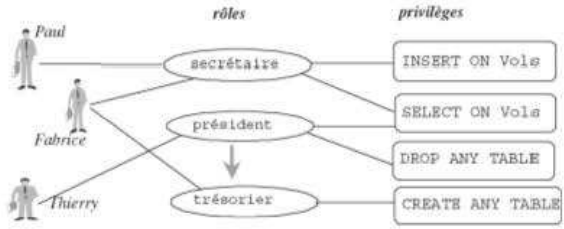
Rôles

Un rôle *(role)* est un ensemble nommé de privilèges (système ou objets). Un rôle est accordé à un ou plusieurs utilisateurs, voire à tous (utilisation de PUBLIC). Ce mécanisme facilite la gestion des privilèges.

Un rôle peut être aussi attribué à un autre rôle pour transmettre davantage de droits comme le montre la figure suivante. Le rôle president est constitué du privilège objet SELECT sur la table Vols, et du privilège système DROP de tables de tout schéma. Il hérite aussi des

privilèges du rôle `trésorier` constitué du privilège système `CREATE TABLE` dans tout schéma.

Figure 5-3 *Rôles*



La chronologie des actions à entreprendre pour travailler avec des rôles est la suivante :

- créer le rôle (`CREATE ROLE`) ;
- l'alimenter de privilèges système ou objets par `GRANT` ;
- l'attribuer par `GRANT` à des utilisateurs (voire à tous avec `PUBLIC`), ou à d'autres rôles ;
- lui ajouter éventuellement de nouveaux privilèges système ou objets par `GRANT`.

Création d'un rôle (CREATE ROLE)

Pour pouvoir créer un rôle vous devez posséder le privilège `CREATE ROLE`. La syntaxe SQL est la suivante :

```
CREATE ROLE nomRôle
[ NOT IDENTIFIED | IDENTIFIED
   { BY motdePasse | USING [schéma.]paquetage | EXTERNALLY | GLOBALLY
   } ] ;
```

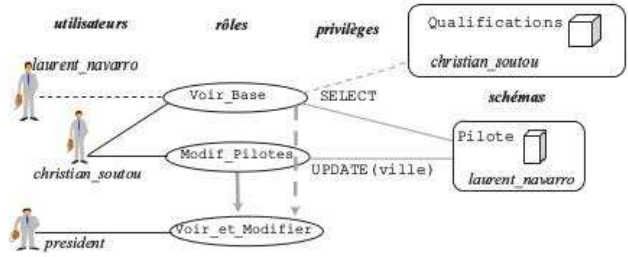
- `NOT IDENTIFIED` indique que l'utilisation de ce rôle est autorisée sans mot de passe.
- `IDENTIFIED` signale que l'utilisateur doit être autorisé par une méthode (locale par un mot de passe, applicative par un paquetage, externe à Oracle et globale par un service d'annuaire) avant que le rôle soit activé par `SET ROLE` (voir plus loin).

Il n'est pas possible de donner le privilège `REFERENCES` à un rôle.

La figure suivante décrit la mise en œuvre de trois rôles. `Voir_Base` autorise l'accès en lecture aux tables de deux schémas. `Modif_Pilotes` autorise la modification de la table `Pilote` du

schéma *laurent_navarro* pour la colonne `ville`. `Voir_et_Modifier` hérite des deux rôles précédents et est affecté à l'utilisateur `president`.

Figure 5-4 *Rôles à définir*



Le tableau suivant décrit la chronologie à respecter pour créer, alimenter et affecter ces rôles :

Tableau 5-13 Gestion de rôles

Adminstrateur	Explication

`CREATE ROLE Voir_Base NOT IDENTIFIED;`
`CREATE ROLE Modif_Pilotes NOT IDENTIFIED;`
`CREATE ROLE Voir_et_Modifier NOT IDENTIFIED;`
	Création des trois rôles.

`GRANT SELECT ON laurent_navarro.Pilote TO Voir_Base;`
`GRANT SELECT ON christian_soutou.Qualifications TO Voir_Base;`
`GRANT UPDATE(ville) ON laurent_navarro.Pilote TO Modif_Pilotes;`
	Alimentation des rôles par des privilèges.

`GRANT Voir_Base, Modif_Pilotes TO Voir_et_Modifier;`	Alimentation d'un rôle par deux autres rôles.

`GRANT Modif_Pilotes TO christian_soutou;`
`GRANT Voir_Base TO christian_soutou, laurent_navarro;`
`GRANT Voir_et_Modifier TO president;`
	Affectation des trois rôles à des utilisateurs.

Rôles prédéfinis

Selon la version, Oracle propose un certain nombre de rôles prédéfinis (qui sont tous attribués par défaut aux utilisateurs SYSTEM et SYS). Ils sont générés lors de la création de la base (scripts accessibles dans le sous-répertoire RDBMS\ADMIN). Vous pouvez utiliser ces rôles en les affectant, par exemple, à des utilisateurs ou pour définir vos propres rôles. Le tableau 5-14 résume les caractéristiques des principaux rôles prédéfinis.

Tableau 5-14 Quelques rôles prédéfinis

Nom du rôle	Commentaires
CONNECT	Se connecter (CREATE SESSION), créer des tables, vues et séquences.
RESOURCE	Créer des procédures, déclencheurs, tables et types.
DBA	Détenir tous les privilèges système avec la possibilité de les retransmettre.
EXP_FULL_DATABASE et DATAPUMP_EXP_FULL_DATABASE	Réaliser des exportations.
IMP_FULL_DATABASE et DATAPUMP_IMP_FULL_DATABASE	Réaliser des importations.
EM_EXPRESS_BASIC et EM_EXPRESS_ALL	Utiliser la console d'administration (version 12*c* Express).
OEM_ADVISOR	Régler des requêtes (voir le chapitre 12).
SELECT_CATALOG_ROLE	Accéder à tous les objets de tout schéma (en consultation).
XDBADMIN	Accéder à XML DB Repository (voir le chapitre 11).

Depuis des années, la documentation conseille de ne plus utiliser les rôles CONNECT, RESOURCE et DBA. Ils sont toujours présents car beaucoup les ont utilisés dès le début. Il est fort probable qu'ils soient toujours disponibles dans les versions à venir.

Révocation d'un rôle

La révocation de privilèges d'un rôle existant se réalise à l'aide de la commande REVOKE précédemment étudiée dans les sections « Privilèges ». Pour pouvoir annuler un rôle, vous devez détenir au préalable ce rôle avec l'option ADMIN OPTION ou avoir reçu le privilège système GRANT ANY ROLE.

```
REVOKE nomRôle [, nomRôle…]
   FROM {utilisateur | nomRôle | PUBLIC} [,{utilisateur | nomRôle |
   PUBLIC}]… ;
```

Le tableau suivant présente trois révocations. La première retire un privilège à un rôle (et affecte ainsi tous les utilisateurs qui bénéficiaient du rôle). La deuxième retire un rôle (ici Voir_Base) à un utilisateur particulier (ici, *laurent_navarro*. Enfin, la dernière retire un rôle (ici, Voir_Base) à un autre rôle Voir_et_Modifier afin de restreindre davantage ce dernier.

Tableau 5-15 Révocations de rôles et de privilèges

Administrateur	Explications
`REVOKE SELECT ON christian_soutou.Qualifications FROM Voir_Base;`	Révocation d'un privilège d'un rôle.
`REVOKE Voir_Base FROM laurent_navarro;`	Révocation d'un rôle d'un utilisateur.
`REVOKE Voir_Base FROM Voir_et_Modifier;`	Révocation du rôle d'un rôle.

Activation d'un rôle (SET ROLE)

Quand un utilisateur se connecte, il détient par défaut tous les privilèges qui lui ont été attribués soit directement soit via des rôles. Les rôles, une fois créés et alimentés, sont donc actifs par défaut. Durant la session (SQL*Plus ou programme), des rôles peuvent être désactivés puis réactivés par la commande SET ROLE. Le nombre de rôles qui peuvent être actifs en même temps est limité par le paramètre d'initialisation MAX_ENABLED_ROLES.

```
SET ROLE
{ nomRôle [IDENTIFIED BY motdePasse] [,nomRôle [IDENTIFIED BY motde-
Passe]]…
   | ALL [EXCEPT nomRôle [,nomRôle]…]
   | NONE } ;
```

- IDENTIFIED indique le mot de passe du rôle si besoin est.
- ALL active tous les rôles (non identifiés) accordés à l'utilisateur qui exécute la commande. Cette activation n'est valable que dans la session courante. La clause EXCEPT permet d'exclure des rôles accordés à l'utilisateur (mais pas via d'autres rôles) de l'activation globale.
- NONE désactive tous les rôles dans la session courante (rôle DEFAULT inclus).

Le tableau suivant décrit un scénario de désactivation et d'activation :

Tableau 5-16 Révocations de rôles et de privilèges

Administrateur	Explications
`CREATE ROLE Supprime_Pilotes IDENTIFIED BY suppil;`	Création d'un rôle identifié.
	Alimentation du rôle.
`GRANT DELETE ON laurent_navarro.Pilote TO Supprime_Pilotes;`	Attribution du rôle à un utilisateur.
`GRANT Supprime_Pilotes TO christian_soutou;`	
Connexions de *christian_soutou*	
--Possible car rôle actif par défaut `DELETE FROM laurent_navarro.Pilote;`	--Désactivation de tous les rôles `SET ROLE NONE;`
--Désactivation `SET ROLE NONE;`	--Activation `SET ROLE Supprime_Pilotes IDENTIFIED BY suppil;`

--Suppression plus permise car rôle inactif
`DELETE FROM laurent_navarro.Pilote;`
`ERREUR à la ligne 1 :`
`ORA-00942: Table ou vue inexistante`
	--Possible car rôle actif de nouveau
	`DELETE FROM laurent_navarro.Pilote;`

Modification d'un rôle (ALTER ROLE)

Nous traitons ici de la modification d'un rôle au niveau de l'identification. La modification du contenu d'un rôle (ajout ou retrait de privilèges) se programme à l'aide des commandes GRANT (pour ajouter un privilège) et REVOKE (pour enlever un privilège).

La commande ALTER ROLE permet de changer le mode d'identification d'un rôle. Vous devez être propriétaire du rôle ou l'avoir reçu avec l'option WITH ADMIN OPTION, ou détenir le privilège ALTER ANY ROLE. Les paramètres de cette commande ont les mêmes significations que dans le cas de la création d'un rôle (CREATE ROLE).

```
ALTER ROLE nomRôle
[ NOT IDENTIFIED | IDENTIFIED
   { BY motdePasse | USING
        [schéma.]paquetage | EXTERNALLY | GLOBALLY } ] ;
```

Le tableau suivant décrit le fait que l'administrateur change le mot de passe du rôle `Supprime_Pilotes` sans prévenir l'utilisateur (ça arrive) :

Tableau 5-17 Modification d'un rôle

Administrateur	Utilisateur *christian_soutou*
--Modification du rôle **ALTER ROLE** Supprime_Pilotes IDENTIFIED BY Ouille;	--Désactivation de tous les rôles SET ROLE NONE; --Activation invalide SET ROLE Supprime_Pilotes IDENTIFIED BY suppil; **ERREUR** à la ligne 1 : **ORA-01979: Mot de passe absent ou erroné pour le rôle 'SUPPRIME_PILOTES'**

Suppression d'un rôle (DROP ROLE)

Pour pouvoir supprimer un rôle vous devez en être propriétaire ou en bénéficier via l'option `WITH ADMIN OPTION`. Le privilège `DROP ANY ROLE` vous donne le droit de supprimer un rôle dans tout schéma.

La commande `DROP ROLE` supprime le rôle et le désaffecte en cascade aux bénéficiaires. Les utilisateurs des sessions en cours ne sont pas affectés par cette suppression qui sera active dès une nouvelle session. La syntaxe de cette commande est la suivante :

```
DROP ROLE nomRôle;
```

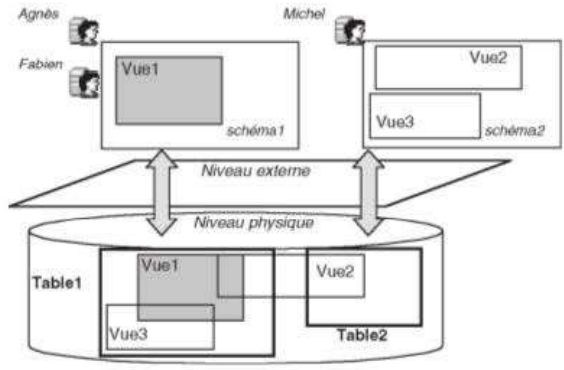
Vues

Outre le contrôle de l'accès aux données (privilèges), la confidentialité des informations est un aspect important qu'un SGBD relationnel doit prendre en compte. La confidentialité est assurée par l'utilisation de vues (*views*), qui agissent comme des fenêtres sur la base de données. Ce chapitre décrit les différents types de vues qu'on peut rencontrer.

Les vues correspondent à ce qu'on appelle *le niveau externe* qui reflète la partie visible de la base de données pour chaque utilisateur. Seules les tables contiennent des données et pourtant, pour l'utilisateur, une vue apparaît comme une table. En théorie, les utilisateurs ne devraient accéder aux informations qu'en questionnant des vues. Ces dernières masquant la structure des tables interrogées. En pratique, beaucoup d'applications se passent de ce concept en manipulant directement les tables.

La figure suivante illustre ce qui a été dit en présentant trois utilisateurs. Ils travaillent chacun sur un schéma contenant des vues qui proviennent de données de différentes tables.

Figure 5-5 *Les vues*



Une vue est considérée comme une table virtuelle car elle ne nécessite aucune allocation en mémoire pour contenir les données. Une vue n'a pas d'existence propre car seule sa structure est stockée dans le dictionnaire de données.

Une vue est créée à l'aide d'une instruction `SELECT` appelée « requête de définition ». Cette requête interroge une ou plusieurs table(s), vue(s) ou cliché(s). Une vue se recharge chaque fois qu'elle est interrogée.

Outre le fait d'assurer la confidentialité des informations, une vue est capable de réaliser des contrôles de contraintes d'intégrité et de simplifier la formulation de requêtes complexes. Même dans certains cas, la définition d'une vue temporaire est nécessaire pour écrire une requête qu'il ne serait pas possible de construire à partir des tables seules. Utilisées conjointement avec des synonymes et attribuées comme des privilèges (`GRANT`), les vues améliorent la sécurité des informations stockées.

Création d'une vue (CREATE VIEW)

Pour pouvoir créer une vue dans votre schéma vous devez posséder le privilège `CREATE VIEW`. Pour créer des vues dans d'autres schémas, le privilège `CREATE ANY VIEW` est requis. La syntaxe SQL simplifiée de création d'une vue est la suivante.

```
CREATE [OR REPLACE] [[NO]FORCE] VIEW [schéma.]nomVue
[ ( { alias [ContrainteInLine [ContrainteInLine]…] |
ContrainteOutLine }
   [, { alias ContrainteInLine [ContrainteInLine]… |
   ContrainteOutLine } ] )
]
AS requêteSELECT [WITH { READ ONLY |
                         CHECK OPTION [CONSTRAINT nomContrainte] } ];
```

- `OR REPLACE` remplace la vue par la nouvelle définition même si elle existait déjà (évite d'avoir à détruire la vue avant de la recréer).
- `FORCE` pour créer la vue sans vérifier si les tables, vues ou clichés qui l'alimentent existent, ou si les privilèges adéquats (`SELECT`, `INSERT`, `UPDATE`, ou `DELETE`) sur ces objets sont acquis par l'utilisateur qui crée la vue.
- `NOFORCE` (par défaut) pour créer la vue en vérifiant au préalable si les tables, vues ou clichés qui l'alimentent existent et que les privilèges sur ces objets sont acquis.
- *`alias`* désigne le nom de chaque colonne de la vue. Si l'alias n'est pas présent, la colonne prend le nom de l'expression renvoyée par la requête de définition.
- *`ContrainteInLine`* indique une contrainte en ligne (exemple : `nomPilote NOT NULL` avec `nomPilote` l'alias et `NOT NULL` la contrainte en ligne). La syntaxe suivante décrit les possibilités d'écriture d'une telle contrainte. Seule l'option `DISABLE NOVALIDATE` est disponible à ce jour.

```
[CONSTRAINT nomContrainte]
{ [NOT] NULL | UNIQUE | PRIMARY KEY
  | REFERENCES [schéma.]nomObjet [(col1 [,col2…])] } DISABLE
  NOVALIDATE
```

- *`ContrainteOutLine`* indique une contrainte (exemple : `CONSTRAINT id_piloteAF PRIMARY KEY (brevet) DISABLE NOVALIDATE`). La syntaxe suivante décrit les possibilités d'écriture d'une telle contrainte :

```
CONSTRAINT nomContrainte
{ UNIQUE(col1 [,col2]… ) | PRIMARY KEY(col1 [,col2]…
  | FOREIGN KEY(col1 [,col2…]) REFERENCES [schéma.]nomObjet [(col1
  [,col2…])] }
 DISABLE NOVALIDATE
```

- *`requêteSELECT`*: requête de définition interrogeant une (ou des) table(s), vue(s), cliché(s) pouvant contenir jusqu'à mille expressions dans la clause `SELECT`.

- La requête de définition ne peut inclure des fonctions sur des séquences `CURRVAL` et `NEXTVAL` ainsi qu'une clause `ORDER BY`.

- Il est nécessaire de mettre un alias, dans la requête, sur les pseudo-colonnes `ROWID`, `ROWNUM`, et `LEVEL`.
- Si la requête de définition sélectionne toutes les colonnes d'un objet source (`SELECT * FROM...`), et si des colonnes sont ajoutées par la suite à cet objet, la vue ne contiendra pas ces colonnes définies ultérieurement à elle. Il faudra recréer la vue pour prendre en compte l'évolution structurelle de l'objet source.

- `WITH READ ONLY` déclare la vue non modifiable par `INSERT`, `UPDATE`, ou `DELETE`.
- `WITH CHECK OPTION` garantit que toute mise à jour de la vue par `INSERT` ou `UPDATE` s'effectuera conformément au prédicat contenu dans la requête de définition. Il existe toutefois des situations particulières et marginales qui n'assurent pas ces mises à jour (sous-interrogation de la vue dans la requête de définition ou mises à jour à partir de déclencheurs `INSTEAD OF`).
- `CONSTRAINT` *`nomContrainte`* nomme la clause `CHECK OPTION` sous la forme d'un nom de contrainte. En l'absence de cette option, la clause porte un nom unique généré par Oracle au niveau du dictionnaire des données (`SYS_Cnnnn`, *n* entier).

Classification

On distingue les vues simples des vues complexes en fonction de la nature de la requête de définition. Le tableau suivant résume ce que nous allons détailler au cours de cette section :

Tableau 5-18 Classification des vues

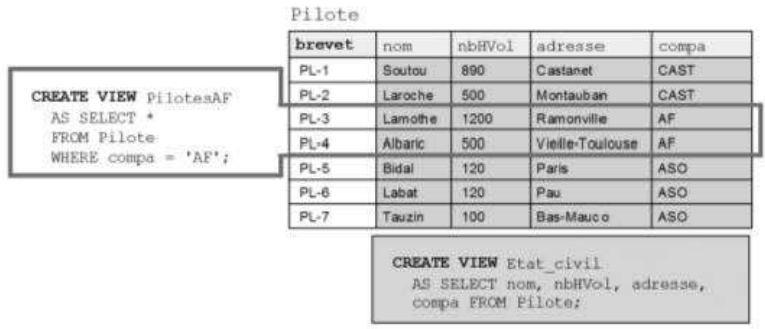
Requête de définition	Vue simple	Vue complexe
Nombre de table	1	1 ou plusieurs
Fonction	Non	Oui
Regroupement	Non	Oui
Mises à jour possibles ?	Oui	Pas toujours

Une vue monotable est définie par une requête `SELECT` ne comportant qu'une seule table dans sa clause `FROM`.

Vues monotables

Les mécanismes présentés ci-après s'appliquent aussi, pour la plupart, aux vues multitables (étudiées plus loin). Considérons les deux vues illustrées par la figure suivante et dérivées de la table `Pilote`. La vue `PilotesAF` décrit les pilotes d'Air France à l'aide d'une restriction (éléments du `WHERE`). La vue `Etat_civil` est constituée par une projection de certaines colonnes (éléments du `SELECT`).

Figure 5-6 Deux vues d'une table



Une fois créée, une vue s'interroge comme une table par tout utilisateur, sous réserve qu'il ait obtenu le privilège en lecture directement (`GRANT SELECT ON nomVue TO...`) ou via un rôle. Le tableau suivant présente une interrogation des deux vues.

Tableau 5-19 Interrogation d'une vue

Besoin et requête	Résultat

Somme des heures de vol des pilotes d'Air France.
`SELECT SUM(nbHVol) FROM PilotesAF;`
	`SUM(NBHVOL)`
	`----------`
	`1700`

Nombre de pilotes.
`SELECT COUNT(*) FROM Etat_civil;`
	`COUNT(*)`
	`----------`
	`7`

À partir de cette table et de ces vues, nous allons étudier certaines autres options de l'instruction `CREATE VIEW`.

Alias

Les alias, s'ils sont utilisés, désignent le nom de chaque colonne de la vue. Ce mécanisme permet de mieux contrôler les noms de colonnes. Quand un alias n'est pas présent la colonne prend le nom de l'expression renvoyée par la requête de définition. Ce mécanisme sert à masquer les noms des colonnes de l'objet source.

Les vues suivantes sont créées avec des alias qui masquent le nom des colonnes de la table source. Les deux écritures sont équivalentes.

Tableau 5-20 Vue avec alias

Écriture 1

```
CREATE OR REPLACE VIEW
    PilotesPasAF
   (codepil,nomPil,heuresPil,
    adressePil, société)
 AS SELECT *
 FROM Pilote
 WHERE NOT (compa = 'AF');
```

Écriture 2

```
CREATE OR REPLACE VIEW
 PilotesPasAF
 AS SELECT brevet codepil, nom nomPil,
 nbHVol heuresPil, adresse adressePil,
 compa société
 FROM  Pilote
 WHERE NOT (compa = 'AF');
```

Contenu de la vue :

CODEPIL	NOMPIL	HEURESPIL	ADRESSEPIL	SOCIÉTÉ
PL-1	Soutou	890	Castanet	CAST
PL-2	Laroche	500	Montauban	CAST
PL-5	Bidal	120	Paris	ASO
PL-6	Labat	120	Pau	ASO
PL-7	Tauzin	100	Bas-Mauco	ASO

Vue d'une vue

L'objet source d'une vue est en général une table mais peut aussi être une vue ou un cliché. La vue suivante est définie à partir de la vue `PilotesPasAF` précédemment créée. Notez qu'il aurait été possible d'utiliser des alias pour renommer à nouveau les colonnes de la nouvelle vue.

Tableau 5-21 Vue d'une vue

Création

```
CREATE OR REPLACE VIEW
  EtatCivilPilotesPasAF
  AS SELECT nomPil,heuresPil,adressePil
  FROM PilotesPasAF;
```

Contenu de la vue

NOMPIL	HEURESPIL	ADRESSEPIL
Soutou	890	Castanet
Laroche	500	Montauban
Bidal	120	Paris
Labat	120	Pau
Tauzin	100	Bas-Mauco

Vues en lecture seule

L'option `WITH READ ONLY` déclare la vue non modifiable par `INSERT`, `UPDATE`, ou `DELETE`.

Redéfinissons la vue `PilotesPasAF` à l'aide de cette option. Les messages d'erreur induits par la clause de lecture seule, générés par Oracle ne sont pas très parlants.

Tableau 5-22 Vue en lecture seule

Création	Opérations impossibles
`CREATE OR REPLACE VIEW PilotesPasAFRO AS SELECT * FROM Pilote WHERE NOT (compa = 'AF') WITH READ ONLY;`	`INSERT INTO PilotesPasAFRO VALUES ('PL-8', 'Ferry', 5, 'Paris', 'ASO');` `ORA-01733: les colonnes virtuelles ne sont pas autorisées ici` `UPDATE PilotesPasAFRO SET nbHvol=nbHvol+2;` `ORA-01733: les colonnes virtuelles ne sont pas autorisées ici` `DELETE FROM PilotesPasAFRO ;` `ORA-01752: Impossible de supprimer de la vue sans exactement une table protégée par clé`

Vues modifiables

Lorsqu'il est possible d'exécuter des instructions `INSERT`, `UPDATE` ou `DELETE` sur une vue, cette dernière est dite modifiable (*updatable view*). Vous pouvez créer une vue qui est modifiable intrinsèquement. Si elle ne l'est pas, il est possible de programmer un déclencheur `INSTEAD OF` (voir la partie 2) qui permet de rendre toute vue modifiable. Les mises à jour sont automatiquement répercutées au niveau d'une ou de plusieurs tables.

Pour mettre à jour une vue, il doit exister une correspondance biunivoque entre les lignes de la vue et celles de l'objet source. De plus certaines conditions doivent être remplies.

Pour qu'une vue simple soit modifiable, sa requête de définition doit respecter les critères suivants :

- pas de directive `DISTINCT`, de fonction (`AVG`, `COUNT`, `MAX`, `MIN`, `STDDEV`, `SUM`, ou `VARIANCE`), d'expression ou de pseudo-colonne (`ROWNUM`, `ROWID`, `LEVEL`) dans le `SELECT` ;
- pas de `GROUP BY`, `ORDER BY`, `HAVING` ou `CONNECT BY`.

Dans notre exemple, nous constatons qu'il ne sera pas possible d'ajouter un pilote à la vue `Etat_civil`, car la clé primaire de la table source ne serait pas renseignée. Ceci est contradictoire avec la condition de correspondance biunivoque.

En revanche, il sera possible de modifier les colonnes de cette vue. On pourra aussi ajouter, modifier (sous réserve de respecter les éventuelles contraintes issues des colonnes de la table source), ou supprimer des pilotes en passant par la vue `PilotesAF`.

La dernière instruction est paradoxale car elle permet d'ajouter un pilote de la compagnie 'ASO' en passant par la vue des pilotes de la compagnie 'AF'. La directive `WITH CHECK OPTION` permet d'éviter ces effets de bord indésirables pour l'intégrité de la base.

Tableau 5-23 Mises à jour de vues

Opérations possibles	Opérations impossibles

Suppression des pilotes de ASO
`DELETE FROM Etat_civil`
`WHERE compa = 'ASO';`
Le pilote *Lamothe* double ses heures
`UPDATE Etat_civil`
`SET nbHVol = nbHVol*2`
`WHERE nom = 'Lamothe';`
	Ajout d'un pilote
	`INSERT INTO Etat_civil`
	`VALUES('Raffarin',10,'Poitiers','ASO');`
	ORA-01400: impossible d'insérer NULL dans ("SOUTOU"."PILOTE"."BREVET")

Ajout d'un pilote
`INSERT INTO PilotesAF VALUES`
`('PL-8', 'Ferry', 5, 'Paris', 'AF');`
Modification
`UPDATE PilotesAF`
`SET nbHVol = nbHVol*2;`
Suppression
`DELETE FROM PilotesAF`
`WHERE nom = 'Ferry';`
Ajout d'un pilote qui n'est pas de 'AF' !
`INSERT INTO PilotesAF VALUES`
`('PL-9', 'Raffarin', 10, 'Poitiers', 'ASO');`
	Toute mise à jour qui ne respecterait pas les contraintes de la table `Pilote`

Directive CHECK OPTION

La directive `WITH CHECK OPTION` empêche un ajout ou une modification non conformes à la définition de la vue.

Interdisons l'ajout (ou la modification de la colonne `compa`) d'un pilote au travers de la vue `PilotesAF`, si le pilote n'appartient pas à la compagnie de code 'AF'.

Il est nécessaire de redéfinir la vue `PilotesAF`. Le script suivant décrit la redéfinition de la vue, l'ajout d'un pilote et les tentatives d'addition et de modification ne respectant pas les caractéristiques de la vue :

Tableau 5-24 Vue avec CHECK OPTION

Opérations possibles	Opérations impossibles

Recréation de la vue
`CREATE OR REPLACE VIEW PilotesAF`
`AS SELECT * FROM pilote`
`WHERE compa = 'AF' WITH CHECK OPTION;`
Nouveau pilote
`INSERT INTO PilotesAF VALUES`
`('PL-11','Teste',900,'Revel','AF');`
`1 ligne créée.`
	Ajout d'un pilote
	`INSERT INTO PilotesAF VALUES`
	`('PL-10','Juppé',10,'Bordeaux','ASO');`
	ORA-01402: vue WITH CHECK OPTION - violation de clause WHERE
	Modification de pilotes
	`UPDATE PilotesAF SET compa='ASO'`
	ORA-01402: vue WITH CHECK OPTION - violation de clause WHERE

Vues avec contraintes

Comme il est indiqué dans la clause de création d'une vue, il est possible de définir au niveau de chaque colonne une ou plusieurs contraintes (en ligne ou complète).

Oracle n'assure pas encore l'activation de ces contraintes. Elles sont créées avec l'option `DISABLE NOVALIDATE` et ne peuvent être modifiées par la suite. Les contraintes sur les vues sont donc déclaratives (comme l'étaient les clés étrangères de la version 6).

Les deux vues suivantes sont déclarées avec une contrainte de chaque type. Il sera possible néanmoins d'y insérer des pilotes de même nom.

Tableau 5-25 Contraintes déclaratives d'une vue

In line	*Out of line*

`CREATE OR REPLACE VIEW`
`PilotesPasAF_inLine (codepil,`
`nomPil UNIQUE DISABLE NOVALIDATE,`
`heuresPil, adressePil, société)`
`AS SELECT * FROM Pilote`
`WHERE NOT (compa = 'AF')`
`WITH CHECK OPTION;`
	`CREATE OR REPLACE VIEW`
	`PilotesPasAF_outLine`
	`(codepil, nomPil, heuresPil,`
	`adressePil, société,`
	`CONSTRAINT un_nomPil UNIQUE(nomPil)`
	`DISABLE NOVALIDATE)`
	`AS SELECT * FROM Pilote`
	`WHERE NOT (compa = 'AF')`
	`WITH CHECK OPTION;`

Vues complexes

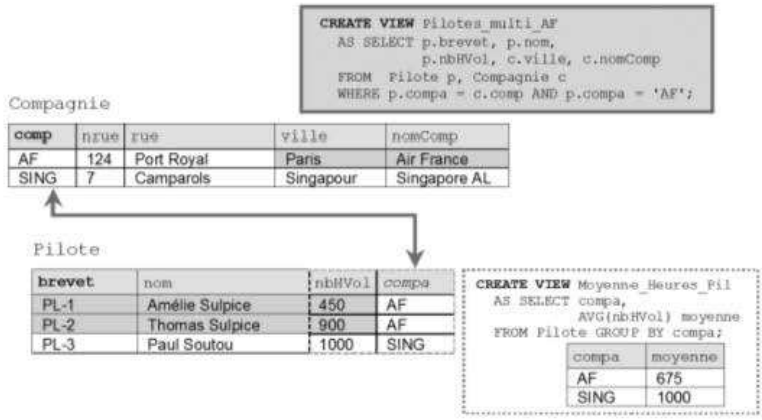
Une vue complexe est caractérisée par le fait de contenir, dans sa définition, plusieurs tables (jointures), et une fonction appliquée à des regroupements, ou des expressions. La mise à jour de telles vues n'est pas toujours possible.

Les restrictions de création sont les suivantes :

- Si la requête de définition contient une sous-interrogation (jointure procédurale), elle ne doit pas être synchronisée ou faire intervenir la table source.
- Il n'est pas possible d'utiliser les opérateurs ensemblistes (`UNION [ALL]`, `INTERSECT` ou `MINUS`).

La figure suivante présente deux vues complexes qui ne sont pas modifiables. La vue multi-table `Pilotes_multi_AF` est créée à partir d'une jointure entre les tables `Compagnie` et `Pilote`. La vue `Moyenne_Heures_Pil` est créée à partir d'un regroupement de la table `Pilote`.

Figure 5-7 *Vues complexes*



Mises à jour

Il apparaît clairement qu'on ne peut pas insérer dans les deux vues car il manquerait les clés primaires. Les messages d'erreurs générés par Oracle sont différents suivant la nature de la vue (monotable ou multitable).

Tableau 5-26 Tentatives d'insertions dans des vues complexes

Vue monotable	Vue multitable
`INSERT INTO Moyenne_Heures_Pil VALUES('TAT',50);` **`ORA-01732`**`: les manipulations de données sont interdites sur cette vue`	`INSERT INTO Pilotes_multi_AF VALUES('PL-4','Test',400,'Castanet', 'Castanet Air Lines');` **`ORA-01776`**`: Impossible de modifier plus d'une table de base via une vue jointe`

On pourrait croire qu'il en est de même pour les modifications et les suppressions. Il n'en est rien. Alors que la vue monotable `Moyenne_Heures_Pil` n'est pas modifiable, ni par `UPDATE` ni par `DELETE` (message d'erreur `ORA-01732`), la vue multitable `Pilotes_multi_AF` est transformable dans une certaine mesure, car la table `Pilote` (qui entre dans sa composition) est dite « protégée par clé » (*key preserved*). Nous verrons dans le prochain paragraphe la signification de cette notion.

Modifions et supprimons des enregistrements à travers la vue multitable `Pilotes_multi_AF`. Il est à noter que seules les colonnes de la vue correspondant à la table protégée par clé peuvent être modifiées (ici `nbHVol` peut être mise à jour, en revanche, `ville` ne peut pas

l'être). Les suppressions se répercutent aussi sur les enregistrements de la table protégée par clé (Pilote).

Tableau 5-27 Mises à jour d'une vue multitable

Mise à jour	Résultats

`UPDATE Pilotes_multi_AF SET nbHVol = nbHVol * 2;`
 	
`2 ligne(s) mise(s) à jour.`
	`SQL> SELECT * FROM Pilotes_multi_AF;`
	`BREVET NOM NBHVOL VILLE NOMCOMP`
	`PL-1 Amélie Sulpice 900 Paris Air France`
	`PL-2 Thomas Sulpice 1800 Paris Air France`

`DELETE FROM Pilotes_multi_AF;`
 	
`2 ligne(s) supprimée(s).`
	`SQL> SELECT * FROM Pilote;`
	`BREVET NOM NBHVOL COMP`
	`PL-3 Paul Soutou 1000 SING`
	
	`SQL> SELECT * FROM Compagnie;`
	`COMP NRUE RUE VILLE NOMCOMP`
	`SING 7 Camparols Singapour Singapore AL`
	`AF 124 Port Royal Paris Air France`

Tables protégées (key preserved tables)

Une table est dite protégée par sa clé (*key preserved*) si sa clé primaire est préservée dans la clause de jointure et se retrouve en tant que colonne de la vue multitable (peut jouer le rôle de clé primaire de la vue).

En considérant les données initiales, pour la vue multitable Vue_Multi_Comp_Pil, la table préservée est la table Pilote, car la colonne brevet identifie chaque enregistrement extrait de la vue alors que la colonne comp ne le fait pas.

Tableau 5-28 Vue multitable

Création de la vue	Résultats

`CREATE VIEW Vue_Multi_Comp_Pil`
`AS SELECT c.comp, c.nomComp,`
`p.brevet, p.nom, p.nbHVol`
`FROM Pilote p, Compagnie c`
`WHERE p.compa = c.comp;`
	`SQL> SELECT * FROM Vue_Multi_Comp_Pil;`
	
	`COMP NOMCOMP BREVET NOM NBHVOL`
	`AF Air France PL-1 Amélie Sulpice 450`
	`AF Air France PL-2 Thomas Sulpice 900`
	`SING Singapore AL PL-3 Paul Soutou 1000`

Cela ne veut pas dire que cette vue est modifiable de toute manière. Aucune insertion n'est permise, seules les modifications des colonnes de la table Pilote sont autorisées. Les suppressions se répercuteront sur la table Pilote.

Afin de savoir dans quelle mesure les colonnes d'une vue sont modifiables (en insertion ou suppression), il faut interroger la vue USER_UPDATABLE_COLUMNS du dictionnaire des données (aspect étudié dans le prochain chapitre).

L'interrogation suivante illustre ce principe. La fonction UPPER est utilisée pour convertir en majuscules le nom de la table (tout est codé en majuscules dans le dictionnaire des données). Les caractéristiques des colonnes apparaissent clairement.

Tableau 5-29 Caractéristiques des colonnes d'une vue

Requête	Résultat

```
SELECT COLUMN_NAME, INSERTABLE,
  UPDATABLE, DELETABLE
  FROM  USER_UPDATABLE_COLUMNS
  WHERE TABLE_NAME = UPPER('Vue_Multi_Comp_Pil');
```

```
COLUMN_NAME      INS UPD DEL
---------------- --- --- ---
COMP             NO  NO  NO
NOMCOMP          NO  NO  NO
BREVET           YES YES YES
NOM              YES YES YES
NBHVOL           YES YES YES
```

Étudions à présent les conditions qui régissent ces limitations.

Critères

Une vue multitable modifiable (*updatable join view* ou *modifiable join view*) est une vue qui n'est pas définie avec l'option WITH READ ONLY et est telle que la requête de définition contient plusieurs tables dans la clause FROM.

Pour qu'une vue multitable soit modifiable, sa requête de définition doit respecter les critères suivants :

- La mise à jour (INSERT, UPDATE, DELETE) n'affecte qu'une seule table.
- Seuls des enregistrements de la table protégée peuvent être insérés. Si la clause WITH CHECK OPTION est utilisée, aucune insertion n'est possible (message d'erreur : ORA-01733: les colonnes virtuelles ne sont pas autorisées ici).
- Seules les colonnes de la table protégée peuvent être modifiées.
- Seuls les enregistrements de la table protégée peuvent être supprimés.

Autres utilisations de vues

Les vues peuvent également servir pour renforcer la confidentialité, simplifier des requêtes complexes et programmer une partie de l'intégrité référentielle.

Variables d'environnement

Une requête de définition d'une vue peut utiliser des fonctions SQL relatives aux variables d'environnement d'Oracle. Le tableau suivant décrit ces variables :

Tableau 5-30 Fonctions et variables d'environnement

Variable / Fonction	Signification	
`USER`	Nom de l'utilisateur connecté.	
`UID`	Numéro d'identification de l'utilisateur connecté.	
`USERENV('paramètre')`	Fonction utilisant un des paramètres ci-contre.	`SESSIONID` : numéro de la session.
		`TERMINAL` : nom du terminal dans le système d'exploitation hôte.
		`ENTRYID` : numéro chronologique de la commande SQL dans la session.
		`LANGUAGE` : langage utilisé.

La vue `Soutou_Camparols_PilotesAF` restituera les pilotes de la compagnie 'AF' pour l'utilisateur *Soutou*, ou pour un utilisateur connecté au terminal *Camparols* sous une version Oracle française.

```
CREATE VIEW Soutou_Camparols_PilotesAF
   AS SELECT * FROM  Pilote WHERE compa = 'AF'
   AND USER = 'SOUTOU'
   OR (USERENV('TERMINAL')      =    'CAMPAROLS'
       AND USERENV('LANGUAGE')  LIKE 'FRENCH_FRANCE%');
```

Contrôles d'intégrité référentielle

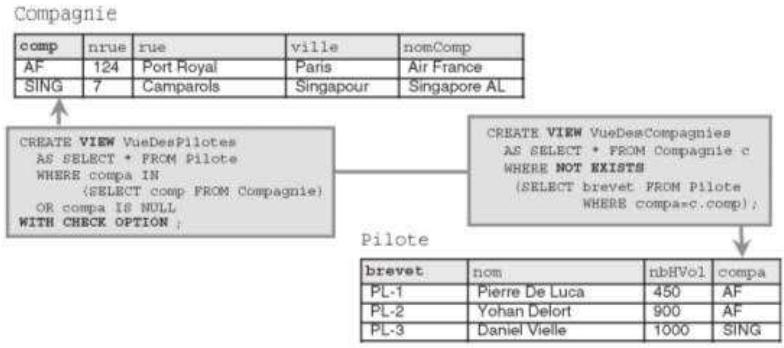
En plus de contraintes de vérification (`CHECK`), il est possible de contrôler l'intégrité référentielle par des vues. Avant la version 7 d'Oracle, et en l'absence des clés étrangères, c'était un moyen de programmer l'intégrité référentielle (une autre façon étant l'utilisation des déclencheurs).

La cohérence référentielle entre deux tables *t1* (table « père ») et *t2* (table « fils ») se programme :

- du « père » vers le « fils » par l'utilisation d'une vue *v1* de la table *t1* définie avec la clause `NOT EXISTS` ;
- du « fils » vers le « père » par l'utilisation d'une vue *v2* de la table *t2* définie avec la clause `WITH CHECK OPTION`.

Considérons les tables `Compagnie` (« père ») et `Pilote` (« fils ») définies sans clés étrangères et programmons la contrainte référentielle à l'aide des vues `VueDesCompagnies` et `VueDesPilotes`. Le raisonnement fait ici sur deux tables peut se généraliser à une hiérarchie d'associations.

Figure 5-8 *Vues qui simulent l'intégrité référentielle*



La vue `VueDesCompagnies` restitue les compagnies qui n'embauchent aucun pilote. La vue `VueDesPilotes` restitue les pilotes dont la colonne `compa` est référencée dans la table `Compagnie`, ou ceux n'ayant pas de compagnie attitrée (la condition `IS NULL` peut être omise dans la définition de la vue si chaque pilote doit être obligatoirement rattaché à une compagnie).

Les règles à respecter pour manipuler les objets côté « père » (table *t1*, vue *v1*) et côté « fils » (table *t2*, vue *v2*) sont les suivantes :

- côté « père » : modification, insertion et suppression via la vue *v1*, lecture de la table *t1* ;
- côté « fils » modification, insertion, suppression et lecture via la vue *v2*.

Manipulons à présent les vues de notre exemple.

Tableau 5-31 Manipulations des vues pour l'intégrité référentielle

Cohérence fils→père	Cohérence père→fils

Insertion incorrecte (père absent) :
`INSERT INTO VueDesPilotes VALUES ('PL-4','Jean',1000,'Rien')`
`ORA-01402: vue WITH CHECK OPTION - violation de clause WHERE`
Insertions correctes :
`INSERT INTO VueDesPilotes VALUES ('PL-4','Paul Soutou',1000,NULL);`
`INSERT INTO VueDesPilotes VALUES ('PL-5','Oliver Blanc',500,'SING');`
	Toute insertion à travers la vue `VueDesCompagnies` est possible (sous réserve de la validité des valeurs du type des colonnes).
	Insertion correcte :
	`INSERT INTO VueDesCompagnies VALUES ('EASY',1,'G. Brassens','Blagnac', 'Easy Jet');`

Modification incorrecte (père absent) :
`UPDATE VueDesPilotes SET compa = 'Toto' WHERE brevet = 'PL-4'`
`ORA-01402: vue WITH CHECK OPTION - violation de clause WHERE`
Modification correcte :
`UPDATE VueDesPilotes SET compa = 'AF' WHERE brevet = 'PL-4';`
	Modification incorrecte (fils présent) :
	`UPDATE VueDesCompagnies SET comp = 'AF2' WHERE comp = 'AF';`
	`0 ligne(s) mise(s) à jour.`
	Modifications correctes :
	`UPDATE VueDesCompagnies SET ville = 'Perpignan' WHERE comp = 'EASY';`
	`UPDATE VueDesCompagnies SET comp = 'EJET' WHERE comp = 'EASY';`

Toute suppression est possible à travers la vue `VueDesPilotes`.	Suppression incorrecte (fils présent) :
	`DELETE FROM VueDesCompagnies WHERE comp = 'AF';`
	`0 ligne(s) supprimée(s).`
	Suppression correcte :
	`DELETE FROM VueDesCompagnies WHERE comp = 'EJET';`
	`1 ligne supprimée.`

Confidentialité

La confidentialité est une des vocations premières des vues. Outre l'utilisation de variables d'environnement, il est possible de restreindre l'accès à des tables en fonction de moments.

Les vues suivantes limitent temporellement les accès en lecture et en écriture à des tables.

Tableau 5-32 Vues pour restreindre l'accès à des moments précis

Définition de la vue	Accès
`CREATE VIEW VueDesCompagniesJoursFériés AS SELECT * FROM Compagnie WHERE TO_CHAR(SYSDATE, 'DAY') IN ('SAMEDI', 'DIMANCHE');`	Restriction, en lecture de la table `Compagnie`, les samedi et dimanche. Mises à jour possibles à tout moment.
`CREATE VIEW VueDesPilotesJoursOuvrables AS SELECT * FROM Pilote WHERE TO_CHAR(SYSDATE,'HH24:MI') BETWEEN '8:30' AND '17:30' AND TO_CHAR(SYSDATE,'DAY') NOT IN ('SAMEDI', 'DIMANCHE') WITH CHECK OPTION;`	Restriction, en lecture et en écriture (à cause de `WITH CHECK OPTION`), de la table `Pilote` les jours ouvrables de 8 h 30 à 17 h 30.

Notez qu'il est possible, en plus, de limiter l'accès à un utilisateur particulier en utilisant des variables d'environnement précédemment étudiées (exemple : ajout de la condition `AND USER='SOUTOU'` à la vue).

Transmission de droits

Les mécanismes de transmission et de révocation de privilèges que nous avons étudiés s'appliquent également aux vues. Ainsi, si un utilisateur désire transmettre des droits sur une partie d'une de ses tables, il utilisera une vue. Seules les données appartenant à la vue seront accessibles aux bénéficiaires.

Les privilèges objets qu'il est possible d'attribuer sur une vue sont les mêmes que ceux applicables sur les tables (`SELECT`, `INSERT`, `UPDATE` sur une ou plusieurs colonnes, `DELETE`).

Tableau 5-33 Privilèges sur les vues

Attribution du privilège	Signification
`GRANT SELECT ON VueDesCompagniesJoursFériés TO PUBLIC;`	Accès pour tous en lecture sur la vue `VueDesCompagniesJoursFériés`.
`GRANT INSERT ON VueDesCompagniesJoursFériés TO Paul;`	Accès pour *Paul* en écriture sur la vue `VueDesCompagniesJoursFériés`.

Modification d'une vue (ALTER VIEW)

Pour pouvoir modifier une vue, vous devez en être propriétaire ou posséder le privilège `ALTER ANY VIEW`. La syntaxe SQL est la suivante :

```
ALTER VIEW [schéma.]nomVue
{ ADD ContrainteOutLine | DROP
{ CONSTRAINT nomContrainte | PRIMARY KEY | UNIQUE(col1 [, col2]… ) }
COMPILE ;
```

Les modifications concernent l'ajout ou la suppression de contraintes qui ne sont pas encore opérationnelles (voir la section « Vues avec contraintes »).

Suppression d'une vue (DROP VIEW)

Pour pouvoir supprimer une vue, vous devez en être propriétaire ou posséder le privilège `DROP ANY VIEW`. La suppression d'une vue n'entraîne pas la perte des données qui résident toujours dans les tables. La syntaxe SQL est la suivante :

```
DROP VIEW [schéma.]nomVue [CASCADE CONSTRAINTS];
```

Les vues ou synonymes qui dépendent de la vue supprimée ne sont pas détruits, ils sont seulement marqués comme invalides.

L'option CASCADE CONSTRAINTS est semblable à celle de la commande DROP TABLE et concerne la suppression des clés primaires ou uniques pour lesquelles il faut répercuter la suppression des clés étrangères associées.

Synonymes

Un synonyme est un alias d'un objet (table, vue, séquence, procédure, fonction ou paquetage). Les avantages d'utiliser des synonymes sont les suivants :

- simplifier l'accès aux objets en abrégeant les noms de tables, par exemple, ou en regroupant dans un même alias les noms du schéma et de l'objet, pour les objets qui ne vous appartiennent pas, mais dont vous avez accès ;
- masquer le vrai nom des objets ou la localisation des objets distants (réunis par liens de base de données : *database links*) ;
- améliorer la maintenance des applications dans la mesure où la nature du synonyme peut être modifiée sans mettre à jour tous les programmes qui l'utilisent (le synonyme garde le même nom tout en référençant un nouvel objet).

Il est ainsi possible d'attribuer plusieurs noms à un même objet. Il est aussi permis de créer des synonymes publics (en utilisant la directive PUBLIC) qui seront visibles et utilisables par tous. Les autres synonymes (privés) ne seront pas accessibles par d'autres utilisateurs à moins de donner les autorisations nécessaires (par GRANT).

Création d'un synonyme (CREATE SYNONYM)

Pour pouvoir créer un synonyme dans votre schéma, il faut que vous ayez reçu le privilège CREATE SYNONYM. Si vous avez le privilège CREATE ANY SYNONYM, vous pouvez créer des synonymes dans tout schéma. Enfin, pour pouvoir créer un synonyme public, il faut que vous ayez reçu le privilège CREATE PUBLIC SYNONYM.

La syntaxe SQL est la suivante :

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schéma.]nomSynonyme
   FOR [schéma.]nomObjet [@lienBaseDonnées];
```

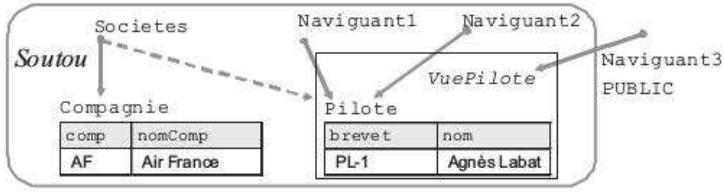
- OR REPLACE recrée le synonyme même s'il en existe déjà un de ce nom (cela vous évite de le détruire puis de le créer). Il existe une restriction pour les synonymes de types dont dépend une table (extension objet non étudiée dans ce livre).
- PUBLIC crée un synonyme public, accessible par tous, sous réserve que les utilisateurs aient les privilèges adéquats sur les objets concernés par le synonyme (par exemple *Paul*

déclare un synonyme public nommé `NavigantPublic` référençant sa table `Pilote`. Ce synonyme est théoriquement accessible par l'utilisateur *Jean*. Pratiquement il faut que *Jean* ait le privilège de lecture sur la table `soutou.Pilote`. Si la clause `PUBLIC` n'est pas appliquée le synonyme est privé et son nom doit être unique dans le schéma.

- *schéma* : le premier désigne le schéma dans lequel va se trouver le synonyme (s'il n'est pas renseigné, vous le créez dans votre schéma). Le deuxième désigne le schéma dans lequel se trouve l'objet à référencer (s'il n'est pas renseigné, vous référencez un objet de votre schéma). Pour les synonymes publics les deux options ne doivent pas être utilisées.
- *nomSynonyme* : nom du synonyme, alias qui va désigner l'objet référencé.
- *nomObjet* : nom de l'objet référencé. Peuvent être concernés : tables, vues, séquences, paquetages, procédures ou fonctions cataloguées, classes Java, types ou autres synonymes.
- *LienBaseDonnées* : désigne un objet distant via un *database link*.

Considérons les tables et la vue suivantes appartenant au schéma *Soutou*, et définissons trois synonymes privés (`Societes`, `Navigant1` et `Navigant2`) et un synonyme public (`Navigant3`).

Figure 5-9 *Synonymes de l'utilisateur Soutou*



Les instructions SQL sont les suivantes :

Tableau 5-34 Création de synonymes

Utilisateur *Soutou*	Signification
`CREATE SYNONYM Navigant1 FOR Pilote;` `CREATE SYNONYM Navigant2 FOR soutou.Pilote;`	Deux synonymes privés équivalents de la table `Pilote`.
`CREATE PUBLIC SYNONYM Navigant3 FOR VuePilote;`	Un synonyme public de la vue `VuePilote` vision de la table `Pilote`.
`CREATE SYNONYM Societes FOR Pilote;` `CREATE OR REPLACE SYNONYM Sociétés FOR Compagnie;`	Remplacement du synonyme privé `Societes` de la table `Compagnie` à la place de la table `Pilote`.

Pour tout synonyme public créé qui référence une table, il n'est pas possible d'ajouter un autre objet du même nom dans le même schéma.

Il n'est pas non plus possible de créer un synonyme public du nom d'un schéma existant (*Soutou* par exemple).

Transmission de droits

La transmission et la révocation des privilèges objets (`SELECT`, `INSERT`, `UPDATE` sur une ou plusieurs colonnes, `DELETE`) s'appliquent également aux synonymes.

Tableau 5-35 Privilèges sur les synonymes

Utilisateur *Soutou*	Utilisateur *Paul*

`GRANT INSERT,SELECT ON Navigant2 TO dev1;`	Écriture incorrecte car il manque le nom du schéma :
	`SELECT * FROM Navigant2;`
	Écritures correctes :
	`SELECT * FROM Soutou.Navigant2;`
	`INSERT INTO Soutou.Navigant2 VALUES('PL-2', 'Jean Turcat');`

`GRANT SELECT ON Navigant3 TO dev1;`	Écriture correcte car synonyme public : `SELECT * FROM Navigant3;`

Suppression d'un synonyme (DROP SYNONYM)

Pour pouvoir supprimer un synonyme, il faut qu'il se trouve dans votre schéma ou que vous ayez reçu le privilège `DROP ANY SYNONYM`. Pour pouvoir supprimer un synonyme public il faut que vous ayez reçu le privilège `DROP PUBLIC SYNONYM`.

La syntaxe SQL est la suivante :

```
DROP [PUBLIC] SYNONYM [schéma.]nomSynonyme [FORCE];
```

- `PUBLIC` : pour détruire un synonyme public (en ce cas ne pas utiliser le préfixe *schéma* pour désigner le synonyme).
- `FORCE` concerne les synonymes de types pour lesquels il existe des tables ou des types qui en dépendent.

Dictionnaire des données

Le dictionnaire des données (*data dictionary*) est une partie majeure d'une base de données Oracle qu'on peut assimiler à une structure centralisée. Le dictionnaire est constitué d'un ensemble de tables système à partir desquelles sont définies environ six cents vues distinctes. Celles-ci stockent toutes les informations décrivant tous les objets de la base de données.

Constitution

Le dictionnaire des données contient :

- la définition des tables, vues, index, clusters, synonymes, séquences, procédures, fonctions, paquetages, déclencheurs, etc. ;
- la description de l'espace disque alloué et occupé pour chaque objet ;
- les valeurs par défaut des colonnes (`DEFAULT`) ;
- la description des contraintes de vérification et d'intégrité référentielle ;
- le nom des utilisateurs de la base ;
- les privilèges et rôles pour chaque utilisateur ;
- des informations d'audit (accès aux objets) et d'autre nature (commentaires par exemple).

Toutes les tables du dictionnaire des données sont accessibles en lecture seulement, elles appartiennent à l'utilisateur `SYS` et sont situées dans l'espace de stockage (*tablespace*) `SYSTEM`. Ce sont plutôt les vues de ces tables qui sont intéressantes car bien structurées. L'interrogation du dictionnaire des données ne peut se faire qu'au travers de requêtes `SELECT`.

Toutes les informations contenues dans les tables système du dictionnaire des données et accessibles au travers de vues sont codées en `MAJUSCULES`.

Le dictionnaire des données est mis automatiquement à jour après chaque instruction SQL du LMD (`INSERT`, `UPDATE`, `DELETE`, `LOCK TABLE`, `MERGE`).

Classification des vues

Soit la vue *v*. Trois classes de vues sont proposées par Oracle (le nom de la classe de vue préfixe le nom de la vue du dictionnaire de données) :

- `USER_`*v* décrit les objets du schéma de l'utilisateur connecté (qui interroge le dictionnaire) ;
- `ALL_`*v* (extension de la précédente) décrit les objets du schéma de l'utilisateur connecté et les objets sur lesquels il a reçu des privilèges ;
- `DBA_`*v* décrit les objets de tous les schémas (de plus il faut préfixer le nom de la vue par celui du propriétaire, ici `SYS.DBA_`*v*).

La structure de ces vues ne diffère que par les points suivants :

- les vues préfixées par `USER_` ne comportent pas la colonne `OWNER` identifiant le propriétaire de l'objet. Cette colonne est implicitement paramétrée par le nom de l'utilisateur connecté ;
- certaines vues préfixées par `DBA_` ont des colonnes supplémentaires décrivant des aspects système.

Démarche à suivre

La démarche à suivre afin d'interroger correctement le dictionnaire des données à propos d'un objet est la suivante :

- trouver le nom de la vue ou des vues qui sont pertinentes à partir de la vue DICTIONARY situé au niveau le plus haut de la hiérarchie ;
- choisir les colonnes de la vue à sélectionner en affichant la structure de la vue (par la commande DESC) ;
- interroger la vue en exécutant une requête SELECT contenant les colonnes intéressantes.

La première étape peut être omise si on connaît déjà le nom de la vue (ce sera le cas pour les vues usuelles que vous aurez déjà utilisées à plusieurs reprises).

Recherche du nom d'une vue

L'extraction du nom des vues qui concernent un objet est rendue possible par l'interrogation de la vue DICTIONARY (de synonyme DICT). Le tableau suivant décrit dans un premier temps la structure de la vue DICTIONARY. La requête interroge cette vue pour extraire automatiquement le nom des trois vues qui concernent les séquences (notez l'utilisation des majuscules dans la condition).

Tableau 5-36 Recherche du nom des vues du dictionnaire des données (à partir de TABLE_NAME)

Commande SQL	Résultat

`DESC DICTIONARY`	`Nom NULL ? Type`
	`---------- ------- ------------`
	`TABLE_NAME VARCHAR2(30)`
	`COMMENTS VARCHAR2(4000)`

`SELECT * FROM DICTIONARY`
`WHERE table_name`
`LIKE '%SEQUENCE%';`
	`TABLE_NAME COMMENTS`
	`---------- -----------`
	`ALL_SEQUENCES Description of SEQUENCEs accessible to the user`
	`DBA_SEQUENCES Description of all SEQUENCEs in the database`
	`USER_SEQUENCES Description of the user's own SEQUENCEs`

On aurait pu interroger la vue DICTIONARY à propos des tables (TABLE), index (INDEX), synonymes (SYNONYM), contraintes (CONSTRAINT), déclencheurs (TRIGGER), etc. Il est aussi possible de tester la colonne COMMENTS qui décrit, sous la forme d'une phrase, la vue. Ce principe de recherche ramène plus de résultats que l'interrogation en testant le nom de colonne TABLE_NAME (notamment à cause des synonymes de vues, ici SEQ). Interrogeons de cette manière le dictionnaire, en s'intéressant aux séquences comme le montre l'exemple suivant.

Tableau 5-37 Recherche du nom des vues du dictionnaire des données (à partir de COMMENTS)

Commande SQL	Résultat

`SELECT * FROM DICTIONARY`
`  WHERE UPPER(comments)`
`  LIKE '%SEQUENCE%';`
	`TABLE_NAME  COMMENTS`
	`---------- ----------`
	`ALL_CATALOG  All tables, views, synonyms, sequences accessible to the user`
	`ALL_SEQUENCES`
	`DBA_CATALOG  All database Tables, Views, Synonyms, Sequences`
	`DBA_SEQUENCES`
	`USER_AUDIT_OBJECT  Audit trail records for statements concerning objects, specifically: table, cluster, …, sequences …`
	`USER_CATALOG  Tables, Views, Synonyms and Sequences owned by the user`
	`USER_SEQUENCES`
	`SEQ  Synonym for USER_SEQUENCES`

Choisir les colonnes

Le choix des colonnes d'une vue du dictionnaire des données s'effectue après avoir listé la structure de cette vue (par DESC). Le nom de la colonne est en général assez parlant. Dans notre exemple, la vue USER_SEQUENCES contient huit colonnes. La colonne SEQUENCE_NAME désignera le nom des séquences du schéma courant, MIN_VALUE les valeurs minimales des séquences, etc.

Si vous avez du mal à interpréter la signification d'une colonne d'une vue du dictionnaire des données, consultez la documentation *Database Reference*, chapitre 2 *Static Data Dictionary Views*.

Tableau 5-38 Choix des colonnes d'une vue du dictionnaire des données

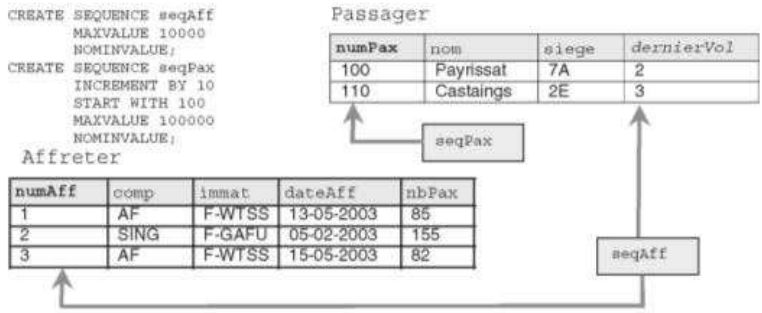
Commande SQL	Résultat

`DESC USER_SEQUENCES`	`Nom  NULL ?  Type`
	`---------- -------- ----------`
	`SEQUENCE_NAME  NOT NULL VARCHAR2(30)`
	`MIN_VALUE  NUMBER`
	`MAX_VALUE  NUMBER`
	`INCREMENT_BY  NOT NULL NUMBER`
	`CYCLE_FLAG  VARCHAR2(1)`
	`ORDER_FLAG  VARCHAR2(1)`
	`CACHE_SIZE  NOT NULL NUMBER`
	`LAST_NUMBER  NOT NULL NUMBER`

Interroger la vue

L'interrogation de la vue sur la base des colonnes choisies est l'étape finale de la recherche de données dans le dictionnaire. Il convient d'écrire une requête monotable ou multitable (jointures) qui extrait des données contenues dans la vue. Ces données sont en fait renfermées dans des tables système qui sont plus difficilement interrogeables du fait de la complexité de leur structure. Supposons que notre schéma contienne les deux séquences suivantes (étudiées au chapitre 2) :

Figure 5-10 *Séquences*



Interrogeons le dictionnaire des données à travers les quatre premières colonnes de la vue USER_SEQUENCES pour retrouver les caractéristiques de ces deux séquences. La valeur courante de la séquence n'est pas stockée dans cette vue, elle est, en revanche, accessible via la fonction CURRVAL.

Tableau 5-39 Interrogation de la vue USER_SEQUENCES

Commande SQL	Résultat

```
SELECT SEQUENCE_NAME,
       MIN_VALUE, MAX_VALUE,
       INCREMENT_BY
  FROM USER_SEQUENCES;
```

```
SEQUENCE_NAME    MIN_VALUE  MAX_VALUE  INCREMENT_BY
---------------- ---------- ---------- ------------
SEQAFF                    1      10000            1
SEQPAX                    1     100000           10
```

Principales vues

Nous listons ici les principales vues qui concernent un utilisateur donné (préfixées par USER_), pour s'intéresser aux objets sur lesquels on a reçu des privilèges. Il faut préfixer ces vues par ALL_, le préfixe DBA_ permettra d'extraire les objets dans tout schéma. Nous approfondirons par la suite l'étude de certaines de ces vues.

Tableau 5-40 Principales vues du dictionnaire des données

Nature de l'objet	Vues
Objets (au sens général)	USER_OBJECTS : objets appartenant à l'utilisateur (synonyme OBJ). USER_ERRORS : erreurs après compilation des objets PL/SQL stockés (procédures, fonctions, paquetages, déclencheurs). USER_STORED_SETTINGS : paramètres des objets PL/SQL stockés. USER_SOURCE : source des objets PL/SQL stockés.
Tables	USER_TABLES : description des tables relationnelles de l'utilisateur (synonyme TABS). USER_ALL_TABLES : description des tables relationnelles et objets de l'utilisateur.
Colonnes	USER_TAB_COLUMNS : colonnes des tables et vues (synonyme COLS). USER_UNUSED_COL_TABS : colonnes éliminées des tables.
Index	USER_INDEXES : description des index (synonyme IND). USER_IND_EXPRESSIONS : expressions fonctionnelles des index. USER_IND_COLUMNS : colonnes qui composent les index.
Contraintes	USER_CONSTRAINTS : définition des contraintes de tables. USER_CONS_COLUMNS : composition des contraintes (colonnes).
Vues	USER_VIEWS : description des vues de l'utilisateur.
Synonymes	USER_SYNONYMS : description des synonymes privés d'un utilisateur (synonyme SYN). DBA_SYNONYMS et ALL_SYNONYMS : description de tous les synonymes (privés et publics).
Séquences	Déjà étudié en début de section.
Commentaires	USER_TAB_COMMENTS : commentaires à propos des tables ou des vues. USER_COL_COMMENTS : commentaires à propos des colonnes des tables et vues.
Utilisateurs	USER_USERS : caractéristiques de l'utilisateur courant. DBA_USERS et ALL_USERS : caractéristiques de tous les utilisateurs.
Privilèges	USER_TAB_GRANTS : liste des autorisations sur les tables et les vues pour lesquelles l'utilisateur est le propriétaire, ou ayant donné ou reçu l'autorisation. USER_TAB_GRANTS_MADE : liste des autorisations sur les objets appartenant à l'utilisateur. USER_COL_GRANTS : colonnes autorisées à l'accès. USER_COL_GRANTS_MADE : liste des autorisations sur les colonnes des tables ou des vues appartenant à l'utilisateur. USER_COL_PRIVS_MADE : informations sur les colonnes pour lesquelles l'utilisateur est propriétaire ou bénéficiaire. USER_TAB_GRANTS_RECD : liste des objets pour lesquels l'utilisateur a reçu une autorisation. USER_COL_PRIVS_RECD : informations sur les colonnes pour lesquelles l'utilisateur a reçu une autorisation.
Rôles	DBA_ROLES : tous les rôles existants. DBA_ROLE_PRIVS : rôles donnés aux utilisateurs et aux autres rôles. USER_ROLE_PRIVS : rôles donnés à l'utilisateur. ROLE_ROLE_PRIVS : rôles donnés aux autres rôles. ROLE_SYS_PRIVS : privilèges système donnés aux rôles. ROLE_TAB_PRIVS : privilèges sur les tables donnés aux rôles. SESSION_ROLES : rôles actifs à un instant *t*.

Interrogeons à présent quelques-unes de ces vues dans le cadre d'exemples concrets.

Objets d'un schéma

La requête suivante interroge la vue USER_OBJECTS et permet de retrouver tous les objets du schéma courant (avec la date de création). L'instruction SQL*Plus COL précise le nombre de caractères à éditer pour une colonne à l'affichage.

```
COL OBJECT_NAME FORMAT A30
SELECT OBJECT_NAME, OBJECT_TYPE, CREATED FROM USER_OBJECTS;

OBJECT_NAME                    OBJECT_TYPE        CREATED
------------------------------ ------------------ --------
ACCES_SECURISE                 PACKAGE            03/09/03
ACCES_SECURISE                 PACKAGE BODY       03/09/03
AFFICHEAVIONS                  PROCEDURE          03/09/03
Compagnies                     JAVA CLASS         17/08/03
EFFECTIFSHEURE                 FUNCTION           16/09/03
ESPIONCONNEXION                TRIGGER            16/09/03
PILOTE                         TABLE              18/09/03
PK_PILOTE                      INDEX              18/09/03
VUEMULTICOMPPIL                VIEW               14/09/03
...
```

Structure d'une table

Il est aisé d'extraire le nom des tables en ajoutant la condition « WHERE TABLE_NAME= 'TABLE' » à l'interrogation précédente. Une fois qu'on connaît le nom d'une table, il est possible de retrouver sa structure (équivalent de ce que produit la commande SQL*Plus DESC) à l'aide de la vue USER_TAB_COLUMNS.

La requête suivante décrit en partie la table INSTALLER qui fait partie du schéma des exercices de ce livre.

```
COL COLUMN_NAME FORMAT A15
COL DATA_TYPE FORMAT A30
SELECT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, DATA_PRECISION
       FROM USER_TAB_COLUMNS WHERE TABLE_NAME = 'INSTALLER';
COLUMN_NAME  DATA_TYPE                      DATA_LENGTH DATA_PRECISION
------------ ------------------------------ ----------- --------------
NPOSTE       VARCHAR2                                 7
NLOG         VARCHAR2                                 5
NUMINS       NUMBER                                  22              5
DATEINS      DATE                                     7
DELAI        INTERVAL DAY(5) TO SECOND(2)            11              5
```

Recherche des contraintes d'une table

La vue USER_CONSTRAINTS décrit la nature des contraintes. Pour retrouver la liste des contraintes d'une table, il faut utiliser les colonnes CONSTRAINT_NAME et CONSTRAINT_TYPE de la vue. Trois valeurs sont possibles au niveau de la colonne CONSTRAINT_TYPE (P désigne la clé primaire, R désigne une clé étrangère et C une contrainte CHECK, UNIQUE ou NOT NULL). La requête suivante liste les contraintes de la table INSTALLER :

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE
       FROM USER_CONSTRAINTS WHERE TABLE_NAME = 'INSTALLER';

CONSTRAINT_NAME                CONSTRAINT_TYPE
------------------------------ ---------------
PK_INSTALLER                   P
FK_INSTALLER_NPOSTE_POSTE      R
FK_INSTALLER_NLOG_LOGICIEL     R
```

Composition des contraintes d'une table

La vue USER_CONS_COLUMNS décrit la composition des contraintes. Pour retrouver la composition d'une clé primaire d'une table, il faut utiliser la colonne POSITION de la vue. La requête suivante permet d'extraire la composition des contraintes et en particulier celle de la clé primaire :

```
SELECT CONSTRAINT_NAME, POSITION, COLUMN_NAME
       FROM USER_CONS_COLUMNS WHERE TABLE_NAME = 'INSTALLER';

CONSTRAINT_NAME                POSITION COLUMN_NAME
------------------------------ ---------- ---------------
FK_INSTALLER_NLOG_LOGICIEL              1 NLOG
FK_INSTALLER_NPOSTE_POSTE               1 NPOSTE
PK_INSTALLER                            1 NPOSTE
PK_INSTALLER                            2 NLOG
```

Détails des contraintes référentielles

La vue USER_CONSTRAINTS permet également de retrouver la nature de la référence pour chaque clé étrangère. La colonne R_CONSTRAINT_NAME (comme *Remote* CONSTRAINT_NAME) désigne le nom de la contrainte de la clé primaire cible. La requête suivante retrouve le nom de la clé primaire des clés étrangères de la table INSTALLER :

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, R_CONSTRAINT_NAME
       FROM USER_CONSTRAINTS WHERE TABLE_NAME = 'INSTALLER';
```

```
CONSTRAINT_NAME              CONSTRAINT_TYPE   R_CONSTRAINT_NAME
-------------------------- --------------- -----------------
PK_INSTALLER                 P
FK_INSTALLER_NPOSTE_POSTE    R                 PK_POSTE
FK_INSTALLER_NLOG_LOGICIEL   R                 PK_LOGICIEL
```

Vous allez me dire qu'on ne voit pas clairement de quelle table et de quelle colonne cible il s'agit. Vous avez raison, le nom de la contrainte peut ne pas être parlant. Afin d'extraire ces éléments manquants, il faut faire une jointure avec la vue USER_CONS_COLUMNS. La requête suivante extrait le détail de chaque clé étrangère de la table INSTALLER :

```
COL OBJECT_NAME FORMAT A30
SELECT OBJECT_NAME, OBJECT_TYPE, CREATED FROM USER_OBJECTS;
COL CONSTRAINT_NAME FORMAT A26 HEADING "Clé étrangère"
COL R_CONSTRAINT_NAME FORMAT A17 HEADING "Nom cible"
COL COLUMN_NAME FORMAT A15 HEADING "Clé cible"
COL TABLE_NAME FORMAT A15 HEADING "Table cible"
SELECT u1.CONSTRAINT_NAME , u1.R_CONSTRAINT_NAME,
       u2.TABLE_NAME, u2.COLUMN_NAME
       FROM  USER_CONSTRAINTS u1, USER_CONS_COLUMNS u2
       WHERE u1.TABLE_NAME         = 'INSTALLER'
       AND   u1.R_CONSTRAINT_NAME  = u2.CONSTRAINT_NAME
       AND   u1.CONSTRAINT_TYPE    = 'R';

Clé étrangère              Nom cible        Table cible     Clé cible
-------------------------- ---------------- --------------- -------
FK_INSTALLER_NPOSTE_POSTE  PK_POSTE          POSTE           NPOSTE
FK_INSTALLER_NLOG_LOGICIEL PK_LOGICIEL       LOGICIEL        NLOG
```

Recherche du code source d'un sous-programme

La vue USER_SOURCE décrit la composition des sous-programmes PL/SQL (procédures, fonctions, paquetages et déclencheurs). La colonne NAME précise le nom du sous-programme. La requête suivante permet d'extraire le code source de la procédure de nom CHERCHEPILOTE :

```
SET LINESIZE 90
COL TEXT FORMAT A70
SELECT LINE,TEXT FROM USER_SOURCE WHERE NAME = 'CHERCHEPILOTE';

     Ligne TEXT
---------- ----------------------------------------------------------
           1  PROCEDURE cherchePilote(p_brevet IN VARCHAR2) IS
           2  var1    Pilote.nbHvol%TYPE;
           3  BEGIN
```

```
4  SELECT nbHvol INTO var1 FROM Pilote WHERE brevet =
p_brevet;
5  IF var1 <= 1000 THEN
6  RAISE_APPLICATION_ERROR(-20777,'Désolé, le pilote manque
d''expérience');
7  END IF;
8  DBMS_OUTPUT.PUT_LINE('Ce pilote a plus de 1000 heures');
9  EXCEPTION
10 WHEN NO_DATA_FOUND THEN
11 DBMS_OUTPUT.PUT_LINE('Pas de pilote avec ce numéro de
brevet');
12 END cherchePilote;
```

Recherche des utilisateurs d'une base de données

La vue ALL_USERS liste les utilisateurs de la base avec la date de leur création :

```
SELECT * FROM ALL_USERS;

USERNAME                       USER_ID CREATED
------------------------------ ---------- --------
SYS                                 0 12/05/02
SYSTEM                              5 12/05/02
…
SCOTT                              59 12/05/02
SOUTOU                             61 16/08/03
TESTE                              63 17/09/03
```

Rôles reçus

La vue USER_ROLE_PRIVS recense les rôles reçus pour un utilisateur. La colonne GRANTED_ROLE contient le nom du rôle attribué. La colonne ADMIN_OPTION précise la nature du rôle (transmissible à d'autres ou pas). La requête suivante liste les rôles détenus par l'utilisateur connecté (ici *Soutou*). Cet utilisateur possède trois rôles (dont le superpuissant DBA).

```
SELECT USERNAME, GRANTED_ROLE, ADMIN_OPTION FROM USER_ROLE_PRIVS;

USERNAME                       GRANTED_ROLE                   ADM
------------------------------ ------------------------------ ---
SOUTOU                         CONNECT                        NO
SOUTOU                         DBA                            NO
SOUTOU                         RESOURCE                       NO
```

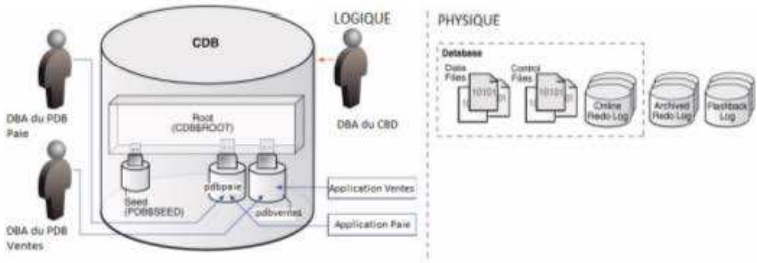
Le multitenant

Depuis la version 12*c*, il est possible d'opter pour l'architecture multitenant (option payante et disponible uniquement pour l'édition *Enterprise*). Le choix vous est donné à l'étape suivante.

Figure 5-11 *Choix de l'installation en tant que multitenant*

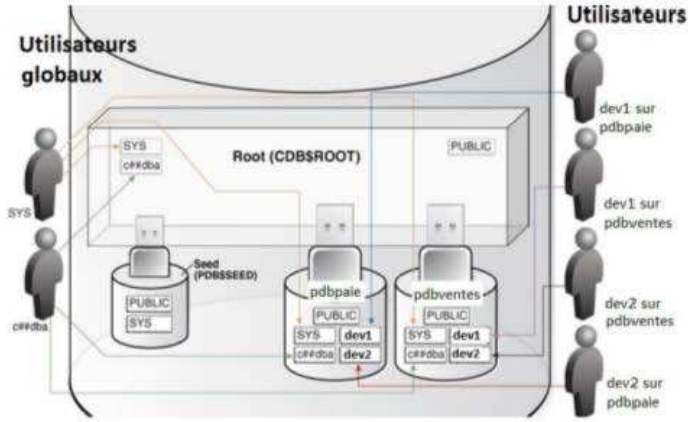
Cette option permet qu'une instance héberge plusieurs bases de données distinctes. Jusqu'à la version 11*g* R2, instance et base de données étaient bien souvent confondues sous la forme d'un ensemble contenant à la fois les données utilisateur et les données système. Avec le multitenant, une instance est composée d'un unique CDB (*Container DataBase*) appelé `CDB$ROOT` qui peut contenir plusieurs bases enfichables (PDB : *Pluggable DataBase*). Chaque PDB pourra être dédiée à une application et contiendra à la fois les données utilisateur et les programmes les manipulant. Le CDB sera consacré à la gestion des données système. La base `PDB$SEED` est également livrée ; elle vous permettra de créer rapidement votre base par clonage.

Figure 5-12 *Architecture multitenant*



Ce type d'architecture vise à réduire les coûts, à consommer moins de ressources (RAM, CPU et stockage) et à offrir davantage de souplesse dans l'administration (migration et mises à jour au niveau PDB). Ce type d'architecture impacte également les utilisateurs et leurs droits. On distingue les utilisateurs au niveau CDB (*common users* dont le nom est préfixé par `C##` ou `c##`) des utilisateurs locaux cantonnés à leur PDB.

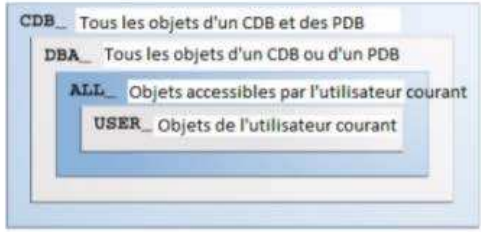
Figure 5-13 *Utilisateurs dans le multitenant*



Enfin, depuis la version 12*c*, il est possible d'installer une instance non-CDB pour rester à l'ancienne architecture (une seule base de données éventuellement répartie au niveau des tablespaces). Avec le multitenant, chaque PDB peut être associée à plusieurs tablespaces applicatifs mais aussi système (le tablespace `system` est présent dans chaque PDB mais n'est pas identique entre deux PDB).

L'architecture multitenant induit un niveau supplémentaire au dictionnaire des données : il s'agit des vues statiques préfixées par `CDB_`, qui renseignent au niveau du container et permet d'accéder à toutes les PDB. Des nouvelles vues dynamiques (`V$`) sont également fournies (`V$pdbs`, par exemple).

Figure 5-14 *Niveaux du dictionnaire des données avec le multitenant*



Dans la plupart des vues du plus haut niveau, la colonne con_id identifie le container (qui peut être le CDB ou une PDB). La valeur 0 est réservée au CDB (c'est le cas pour les vues d'une instance sans multitenant), la valeur 1 à la base CDB$ROOT, 2 à la base PDB$SEED. Ensuite, chaque PDB se voit affecter séquentiellement un chiffre de 3 à 254. Le tableau suivant présente quelques utilisations de cette colonne.

Tableau 5-41 Quelques vues du multitenant

Interrogation du dictionnaire	Commentaires

`SQL> SELECT name, cdb, open_mode, con_id`
`FROM V$DATABASE;`
 	
`NAME CDB OPEN_MODE CON_ID`
`--------- --- ---------- ----------`
`ORCL YES READ WRITE 0`
	L'instance ORCL est multitenant (de type container).

`SQL> SELECT name, open_mode, con_id`
`FROM V$PDBS;`
 	
`NAME OPEN_MODE CON_ID`
`---------- ---------- --------`
`PDB$SEED READ ONLY 2`
`PDBORCL READ WRITE 3`
	Deux PDB sont disponibles : la base PDBORCL qui est ouverte en écriture et celle pour le clonage.

`SQL> SELECT table_name, owner, tablespace_name`
`FROM CDB_TABLES`
`WHERE con_id = (SELECT con_id FROM V$PDBS`
`WHERE name='PDBORCL')`
`ORDER BY owner, table_name;`
 	
`TABLE_NAME OWNER TABLESPACE_NAME`
`---------- ---------- ----------`
`PILOTE EYROLLES USERS`
`BILLETS SOUTOU USERS`
`CARACTERESUNICODE SOUTOU USERS`
`...`
	Liste des tables (avec leur schéma et leur tablespace) contenues dans la base PDBORCL.

`SQL> SELECT username, default_tablespace, con_id`
`FROM CDB_USERS`
`WHERE con_id = (SELECT con_id FROM V$PDBS`
`WHERE name='PDBORCL')`
`AND default_tablespace NOT IN`
`('SYSTEM', 'SYSAUX')`
`ORDER BY username;`
 	
`USERNAME DEFAULT_TABLESPACE CON_ID`
`---------- ---------- --------`
`APEX_PUBLIC_USER USERS 3`
`AUDSYS USERS 3`
`BI EXAMPLE 3`
`DEV1 USERS 3`
`DEV2 USERS 3`
`...`
	Liste des utilisateurs locaux (avec leur tablespace par défaut) de la base PDBORCL.

Les consoles d'administration

Depuis Oracle 9*i*, chaque nouvelle version a apporté un nouvel outil graphique d'administration, notamment la console Enterprise Manager. Au début, il s'agissait d'une interface Java qui est devenue au fur et à mesure des versions une interface web de plus en plus sophistiquée. Depuis la version 12*c*, deux consoles sont proposées : Enterprise Manager Database Express, qui convient pour des architectures restreintes, et Enterprise Manager Cloud Control pour centraliser la gestion de plusieurs serveurs.

Enterprise Manager Database Express

Cette console permet de gérer des bases non-CDB, CDB ou de type PDB. Pour chacune de ces bases, un port HTTPS unique doit être configuré (l'assistant DBCA réalise ces configurations quand vous l'utilisez). La connexion à la console s'effectue à l'aide de l'URL suivante : `https://nom_serveur:port/em`. En vous connectant à `sys AS sysdba`, vous pouvez retrouver avec la fonction `gethttpsport` du paquetage `DBMS_XDB_CONFIG`, le port assigné à chacune de vos bases. La procédure `sethttpsport` vous permettra d'affecter un port s'il n'était pas déjà configuré.

Tableau 5-42 Port de la console EM Express

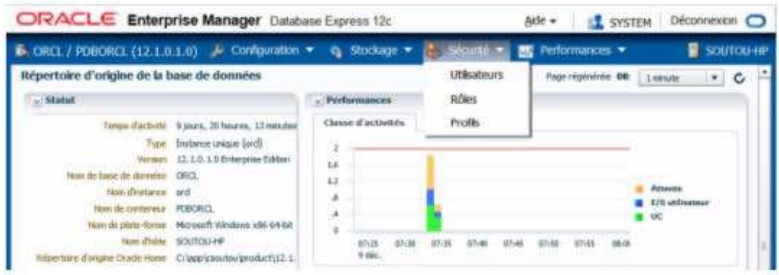
Configuration de la session	Commentaires

`SQL> alter session set container=CDB$ROOT;`
`Session modifiée.`
 	
`SQL> SELECT DBMS_XDB_CONFIG.gethttpsport()`
`  FROM DUAL;`
`DBMS_XDB_CONFIG.GETHTTPSPORT()`
`------------------------------`
`5500`
	La connexion à la console au niveau du CDB s'effectue via *https://localhost:5500/em*.

`SQL> alter session set container=PDBORCL;`
`Session modifiée.`
 	
`SQL> SELECT DBMS_XDB_CONFIG.gethttpsport()`
`  FROM DUAL;`
 	
`DBMS_XDB_CONFIG.GETHTTPSPORT()`
`------------------------------`
`5501`
	La connexion à la console au niveau de la PDB s'effectue via *https://localhost:5501/em*.

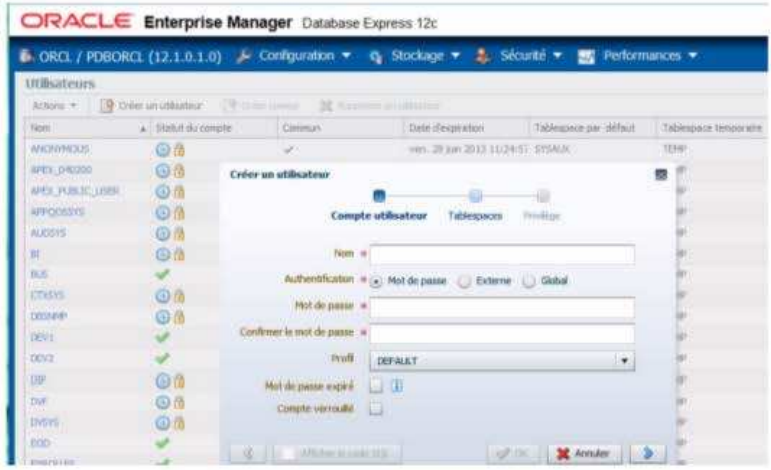
Une fois connecté avec l'utilisateur `system`, vous retrouverez intuitivement le moyen d'agir sur la base (paramètres d'initialisation, espaces de stockage, utilisateurs, rôles, profils, etc.).

Figure 5-15 *Console EM Database Express*



Concernant un utilisateur local, l'instruction `CREATE USER` précédemment étudiée sera générée en fonction de vos choix :

Figure 5-16 *Création d'un utilisateur*



L'onglet Stockage vous permettra de visualiser et de créer vos espaces de stockage.

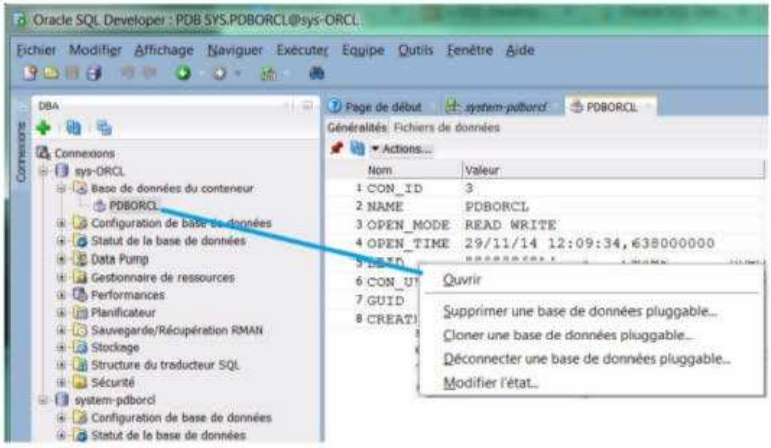
Figure 5-17 *Visualisation des tablespaces*

SQL Developer

Depuis la version 4 (décembre 2013), l'outil SQL Developer est fourni avec une « vue » DBA. En créant une connexion au container principal et une autre à la base enfichable (ici, PDBORCL), vous pourrez gérer graphiquement vos espaces de stockage, utilisateurs, etc.

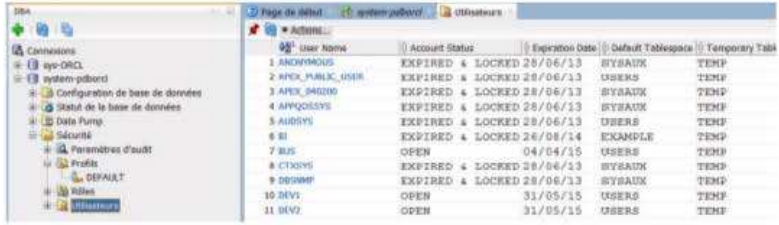
L'écran suivant présente deux connexions : sur le CDB et sur la base enfichable. Sur la première, il est possible d'opérer sur l'état d'une PDB (ouverture, fermeture, déconnexion, suppression ou clonage).

Figure 5-18 *Vue DBA de SQL Developer*



L'écran suivant présente l'affichage des utilisateurs locaux de la PDB. Une fois sur un utilisateur, vous pourrez attribuer des privilèges, changer un mot de passe ou un profil, supprimer le schéma, créer un nouvel utilisateur, etc.

Figure 5-19 *Gestion des utilisateurs avec SQL Developer*



Exercices

Les objectifs de ces exercices sont :

- de créer des vues monotables et multitables ;
- d'insérer des enregistrements dans des vues ;
- d'effectuer une mise à jour conditionnée via une vue.

Exercice 5.1 Vues monotables

Vues sans contraintes

Écrivez le script `vues.sql`, permettant de créer :

- La vue `LogicielsUnix` qui contient tous les logiciels de type 'Unix' (toutes les colonnes sont conservées). Vérifier la structure et le contenu de la vue (`DESC` et `SELECT`).
- La vue `Poste_0` de structure (`nPos0`, `nomPoste0`, `nSalle0`, `TypePoste0`, `indIP`, `ad0`) qui contient tous les postes du rez-de-chaussée (`etage=0` au niveau de la table `Segment`). Faire une jointure procédurale sinon la vue sera considérée comme une vue multitable. Vérifier la structure et le contenu de la vue.

Insérez deux nouveaux postes dans la vue, tels qu'un poste soit connecté au segment du rez-de-chaussée et l'autre à un segment d'un autre étage. Vérifier le contenu de la vue et celui de la table. Conclusion ?

Supprimez ces deux enregistrements de la table `Poste`.

Résoudre une requête complexe

Créez la vue `SallePrix` de structure (`nSalle`, `nomSalle`, `nbPoste`, `prixLocation`) qui contient les salles et leur prix de location pour une journée (fonction du nombre de postes). Le montant de la location d'une salle à la journée sera d'abord calculé sur la base de 100 € par poste. Servez-vous de l'expression `100*nbPoste` dans la requête de définition.

Vérifiez le contenu de la vue, puis afficher les salles dont le prix de location dépasse 150 €.

Ajoutez la colonne `tarif` de type `NUMBER(3)` à la table `Types`. Mettez à jour cette table de manière à insérer les valeurs suivantes :

Tableau 5-43 Tarifs des postes

Type du poste	Tarif en €
TX	50
PCWS	100
PCNT	120
UNIX	200
NC	80
BeOS	400

Créez la vue `SalleIntermédiaire` de structure (`nSalle`, `typePoste`, `nombre`, `tarif`), de telle sorte que le contenu de la vue reflète le tarif ajusté des salles en fonction du nombre et du type des postes de travail. Il s'agit de grouper par salle, type et tarif (tout en faisant une jointure avec la table `Types` pour les tarifs), et de compter le nombre de postes pour avoir le résultat suivant :

```
NSALLE  TYPEPOSTE     NOMBRE       TARIF
------- --------- ---------- ----------
s01     TX                 2         50
s01     UNIX               1        200
s02     PCWS               2        100
...
```

À partir de la vue `SalleIntermédiaire`, créez la vue `SallePrixTotal(nSalle, PrixRéel)` qui reflète le prix réel de chaque salle (par exemple la s01 sera facturée 2*50 + 1*200 = 300). Vérifiez le contenu de cette vue.

Affichez les salles les plus économiques à la location.

Vues avec contraintes

Remplacez la vue `Poste0` en rajoutant l'option de contrôle (`CHECK OPTION`). Tenter d'insérer un poste appartenant à un étage différent du rez-de-chaussée.

Créez la vue `Installer0` de structure (`nPoste`, `nLog`, `num`, `dateIns`) ne permettant de travailler qu'avec les postes du rez-de-chaussée, tout en interdisant l'installation d'un logiciel de type 'PCNT'. Tentez d'insérer deux postes dans cette vue ne correspondant pas à ces deux contraintes : un poste d'un étage, puis un logiciel de type 'PCNT'. Insérer l'enregistrement 'p6', 'log2' qui doit passer à travers la vue.

Exercice 5.2 Vue multitable

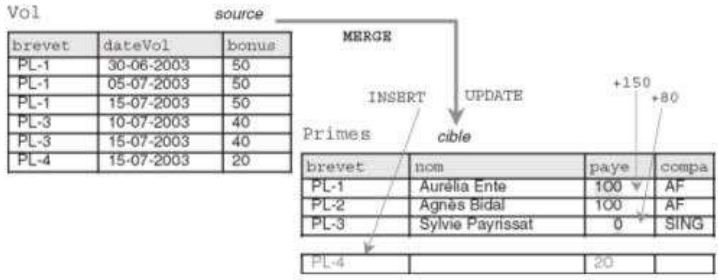
Créez la vue `SallePoste` de structure (`nomSalle`, `nomPoste`, `adrIP`, `nomTypePoste`) permettant d'extraire toutes les installations sous la forme suivante :

```
NOMSALLE        NOMPOSTE            ADRIP         NOMTYPEPOSTE
--------------- ------------------- ------------  ----------------
Salle 1         Poste 1             130.120.80.01 Terminal X-Window
Salle 1         Poste 2             130.120.80.02 Système Unix
...
```

Exercice 5.3 Mises à jour conditionnées

À partir de la table `Vol` ci-dessous, définissez la vue `v_Vols` qui permettra, à l'aide d'une instruction `MERGE`, de mettre correctement à jour la table `Primes`.

Figure 5-20 *Mises à jour conditionnées*



Exercice 5.4 Vues de la base *Chantiers*

Créez la vue `chantier_passagers` permettant d'extraire le détail des visites des employés en tant que passagers d'un mois donné sous la forme suivante (ici pour Avril 2008) :

```
CHANTIER   JOUR       VEHICULE   PASSAGER   CONDUCTEUR       TEMPS
---------- ---------- ---------- ---------- ---------- ----------
CH1        01/04/08   V1         E7         E1                2,5
CH1        01/04/08   V1         E8         E1                2,5
CH1        02/04/08   V2         E1         E10                 2
...
```

Créez la vue `chantier_conducteur` permettant d'extraire le temps passé sur la route par les conducteurs des visites d'un mois donné sous la forme suivante :

```
CHANTIER   CONDUCTEUR JOUR            TEMPS
---------- ---------- ---------- ----------
CH1        E1         01/04/08          2,5
...
```

Créez la vue `chantier_conducteur_passagers` permettant d'extraire le temps passé sur la route par les employés (conducteur ou passager) d'un mois donné sous la forme suivante :

```
EMPLOYE    Temps passé
---------- -----------
E1                 8,5
E2               4,875
```

...

En utilisant ces vues, écrivez la requête qui permet de facturer le temps passé par les employés sur tous les chantiers. La formule à programmer est la suivante : pour tout chantier, le prix est égal au nombre d'employés multiplié par le temps passé (sur la base de 30 euros de l'heure).

Un exemple est donné ci-après :

```
CHANTIER   SUM(TEMPS) COUNT(EMPLOYE)       PRIX
---------- ---------- -------------- ----------
CH1              15,5              7       3255
CH2                 9             11       2970
...
```

Partie II

PL/SQL

Chapitre 6

Bases du PL/SQL

Ce chapitre décrit les caractéristiques générales du langage PL/SQL :

- structure d'un programme ;
- déclaration et affectation de variables ;
- structures de contrôle (*si*, *tant que*, *répéter*, *pour*) ;
- mécanismes d'interaction avec la base ;
- programmation de transactions.

Généralités

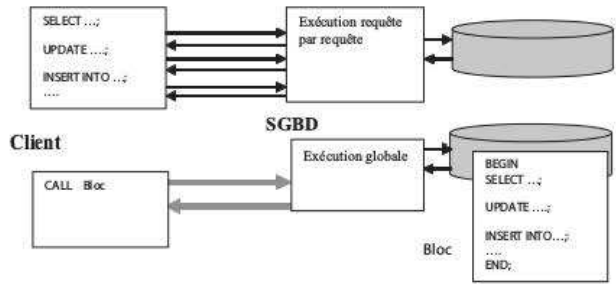
Les structures de contrôle habituelles d'un langage (`IF`, `WHILE`...) ne font pas partie intégrante de la norme SQL. Elles apparaissent dans une sous-partie optionnelle de la norme (ISO/IEC 9075-5:1996. *Flow-control statements*). Oracle les prend en compte dans PL/SQL. Nombre de concepts de PL/SQL proviennent du langage Ada.

Le langage PL/SQL (*Procedural Langage/Structured Query Langage*) est le langage de prédilection d'Oracle depuis la version 6. Ce langage est une extension de SQL car il permet de faire cohabiter des structures de contrôle (*si*, *pour* et *tant que*) avec des instructions SQL (principalement `SELECT`, `INSERT`, `UPDATE` et `DELETE`). PL/SQL est aussi utilisé par des outils d'Oracle (*Forms*, *Report* et *Graphics*).

Environnement client-serveur

Dans un environnement client-serveur, chaque instruction SQL donne lieu à l'envoi d'un message du client vers le serveur suivi de la réponse du serveur vers le client. Il est préférable de travailler avec un bloc PL/SQL plutôt qu'avec une suite d'instructions SQL susceptibles d'encombrer le trafic réseau. En effet, un bloc PL/SQL donne lieu à un seul échange sur le réseau entre le client et le serveur. Les résultats intermédiaires sont traités côté serveur et seul le résultat final est retourné au client.

Figure 6-1 Différentes approches du client-serveur



Avantages

Les principaux avantages de PL/SQL sont :

- La modularité (un bloc d'instruction peut être composé d'un autre, etc.) : un bloc peut être nommé pour devenir une procédure ou une fonction cataloguée, donc réutilisable. Une procédure, ou fonction, cataloguée peut être incluse dans un paquetage (*package*) pour mieux contrôler et réutiliser ces composants logiciels.
- La portabilité : un programme PL/SQL est indépendant du système d'exploitation qui héberge le serveur Oracle. En changeant de système, les applicatifs n'ont pas à être modifiés.
- L'intégration avec les données des tables : on retrouvera avec PL/SQL tous les types de données et instructions disponibles sous SQL, et des mécanismes pour parcourir des résultats de requêtes (curseurs), pour traiter des erreurs (exceptions), pour manipuler des données complexes (paquetages `DBMS_xxx`) et pour programmer des transactions (`COMMIT`, `ROLLBACK`, `SAVEPOINT`).

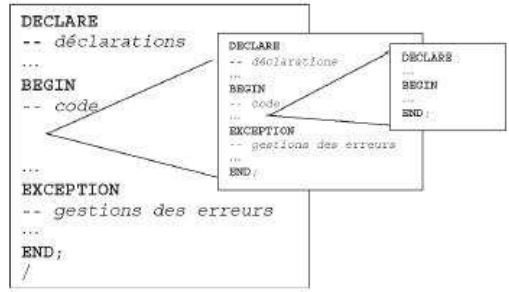
Structure d'un programme

Un programme PL/SQL qui n'est pas nommé (aussi appelé bloc) est composé de trois sections comme le montre la figure ci-contre :

- `DECLARE` (section optionnelle) déclare les variables, types, curseurs, exceptions, etc. ;
- `BEGIN` (section obligatoire) contient le code PL/SQL incluant ou non des directives SQL (jusqu'à l'instruction `END;`). Le caractère « / » termine un bloc pour son exécution dans l'interface SQL*Plus. Nous n'indiquons pas ce signe dans nos exemples pour ne pas surcharger le code, mais vous devrez l'inclure à la fin de vos blocs ;

- EXCEPTION (section optionnelle) permet de traiter les erreurs retournées par le SGBD à la suite d'exécutions d'instructions SQL.

Figure 6-2 *Strcture d'un bloc PL/SQL*

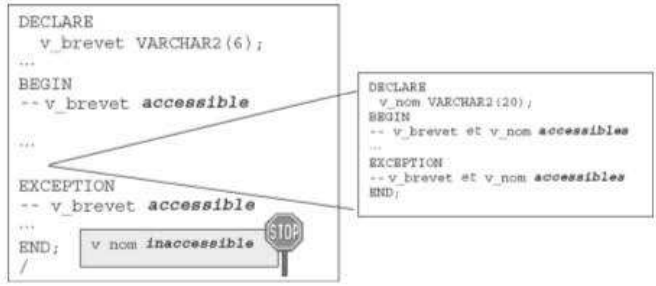


Portée des objets

Un bloc peut être imbriqué dans le code d'un autre bloc (on parle de sous-bloc). Un sous-bloc peut aussi se trouver dans la partie des exceptions. Un sous-bloc commence par BEGIN et se termine par END.

La portée d'un objet (variable, type, curseur, exception, etc.) est la zone du programme qui peut y accéder. Un bloc qui déclare qu'un objet peut y accéder, ainsi que les sous-blocs. En revanche, un objet déclaré dans un sous-bloc n'est pas visible du bloc supérieur (principe des accolades de C et Java).

Figure 6-3 *Visibilité des objets*



Jeu de caractères

Comme SQL, les programmes PL/SQL sont capables d'interpréter les caractères suivants :

- lettres A à Z et a à z ;
- chiffres de 0 à 9 ;
- symboles ( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ] ;
- tabulations, espaces et retours-chariot.

Comme SQL, PL/SQL n'est pas sensible à la casse (*not case sensitive*). Ainsi `numéroBrevet` et `NuméroBREVET` désignent le même identificateur (tout est traduit en majuscules au niveau du dictionnaire des données). Les règles d'écriture concernant l'indentation et les espaces entre variables, mots-clés et instructions doivent être respectées dans un souci de lisibilité.

Tableau 6-1 Lisibilité du code

Peu lisible	C'est mieux

`IF x>y THEN max:=x;ELSE max:=y;END IF;`	`IF x > y THEN`
	`  max := x;`
	`ELSE`
	`  max := y;`
	`END IF;`

Identificateurs

Avant de parler des différents types de variables PL/SQL, décrivons comment il est possible de nommer des objets PL/SQL (variables, curseurs, exceptions, etc.).

Un identificateur commence par une lettre suivie (optionnel) de symboles (lettres, chiffres, $, _, #). Un identificateur peut contenir jusqu'à trente caractères. Les autres signes pourtant connus du langage sont interdits comme le montre le tableau suivant :

Tableau 6-2 Identificateurs

Autorisés	Interdits
`X`	`moi&toi` (symbole &)
`t2`	`debit-credit` (symbole -)
`téléphone#`	`on/off` (symbole /)
`code_brevet`	`code brevet` (symbole espace)
`codeBrevet`	
`oracle$nombre`	

Commentaires

PL/SQL supporte deux types de commentaires :

- monolignes, commençant au symbole -- et finissant à la fin de la ligne ;
- multilignes, commençant par /* et finissant par */.

Le tableau suivant décrit quelques exemples :

Tableau 6-3 Commentaires

Sur une ligne	Sur plusieurs lignes

`-- Lecture de la table`
`SELECT salaire INTO v_salaire`
`FROM Pilote -- Extraction du salaire`
`WHERE nom = 'Thierry Albaric';`
`v_bonus := v_salaire * 0.15; -- Calcul`
	`/* Lecture de la table Pilote */`
	`SELECT salaire INTO v_salaire`
	`FROM Pilote`
	`/* Extraction du salaire`
	`pour calculer le bonus */`
	`WHERE nom = 'Thierry Albaric';`
	`v_bonus := v_salaire * 0.15; /*Calcul*/`

Il n'est pas possible d'imbriquer des commentaires. Pour les programmes PL/SQL qui sont utilisés par des précompilateurs, il faut employer des commentaires multilignes.

Variables

Un programme PL/SQL est capable de manipuler des variables et des constantes (dont la valeur est invariable). Les variables et les constantes sont déclarées (et éventuellement initialisées) dans la section DECLARE. Ces objets permettent de transmettre des valeurs à des sous-programmes via des paramètres, ou d'afficher des états de sortie sous l'interface SQL*Plus.

Plusieurs types de variables sont manipulés par un programme PL/SQL :

- Variables PL/SQL :
  - scalaires recevant une seule valeur d'un type SQL (exemple : colonne d'une table) ;
  - composites (%ROWTYPE, RECORD et TYPE) ;
  - références (REF) ;
  - LOB (*locators*).
- Variables non PL/SQL : définies sous SQL*Plus (de substitution et globales), variables hôtes (déclarées dans des programmes précompilés).

Variables scalaires

La déclaration d'une variable scalaire est de la forme suivante :

```
identificateur [CONSTANT] typeDeDonnée [NOT NULL] [:= | DEFAULT
expression];
```

- CONSTANT précise qu'il s'agit d'une constante ;
- NOT NULL pose une contrainte en ligne sur la variable ;
- DEFAULT permet d'initialiser la variable (équivaut à l'affectation :=).

Le tableau suivant décrit quelques exemples :

Tableau 6-4 Déclarations

Variables	Constantes et expressions

DECLARE
v_dateNaissance DATE;
/* équivaut à
v_dateNaissance DATE:= NULL;
*/
v_capacité NUMBER(3) := 999;
v_téléphone CHAR(14)
NOT NULL := '06-76-85-14-89';
v_trouvé BOOLEAN NOT NULL := TRUE;
BEGIN
…
	DECLARE
	c_pi **CONSTANT** NUMBER := 3.14159;
	v_rayon NUMBER := 1.5;
	v_aire NUMBER := c_pi * v_rayon**2;
	--
	v_groupeSanguin CHAR(3) := 'O+';
	/* équivaut à
	v_groupeSanguin CHAR(3) **DEFAULT** 'O+';
	*/
	v_dateValeur DATE := SYSDATE + 2;
	BEGIN
	…

Il n'est pas possible d'affecter une valeur nulle à une variable définie NOT NULL (l'erreur renvoyée est l'exception prédéfinie VALUE_ERROR).

La contrainte NOT NULL doit être suivie d'une clause d'initialisation.

Affectations

Il existe plusieurs possibilités pour affecter une valeur à une variable :

- l'affectation comme on la connaît dans les langages de programmation (*variable* := *expression*) ;
- par la directive DEFAULT ;
- par la directive INTO d'une requête (SELECT … **INTO** *variable* FROM …).

Le tableau suivant décrit quelques exemples.

Tableau 6-5 Affectations

Code PL/SQL	Commentaires

DECLARE
  v_brevet VARCHAR2(6);
  v_brevet2 VARCHAR2(6);
  v_prime NUMBER(5,2);
  v_naissance DATE;
  v_trouvé BOOLEAN NOT NULL
  DEFAULT FALSE;
	Déclarations de variables.

BEGIN v_brevet := 'PL-1';	Affectation d'une chaîne de caractères.
v_brevet2 := v_brevet;	Affectation d'une variable.
v_prime := 500.50;	Affectation d'un nombre.
v_naissance := '04-07-2003'; v_naissance := TO_DATE('04-07-2003 17:30', 'DD:MM:YYYY HH24:MI');	Affectation de dates.
v_trouvé := TRUE;	Affectation d'un booléen.
SELECT brevet **INTO** v_brevet FROM Pilote WHERE nom = 'Gratien Viel'; …	Affectation d'une chaîne de caractères par une requête.

Restrictions

Il est impossible d'utiliser un identificateur dans une expression s'il n'est pas déclaré au préalable. Ici, la déclaration de la variable `maxi` est incorrecte :

```
DECLARE
  maxi NUMBER := 2 * mini;
  mini NUMBER := 15;
```

À l'inverse de la plupart des langages récents, les déclarations multiples ne sont pas permises. Celle qui suit est incorrecte :

```
DECLARE
  i, j, k NUMBER;
```

Variables %TYPE

La directive `%TYPE` déclare une variable selon la définition d'une colonne d'une table ou d'une vue existante. Elle permet aussi de déclarer une variable conformément à une autre variable précédemment déclarée.

Il faut faire préfixer la directive %TYPE avec le nom de la table et celui de la colonne (*identificateur nomTable.nomColonne*%TYPE) ou avec le nom d'une variable existante (*identificateur2 identificateur1*%TYPE). Le tableau suivant décrit cette syntaxe :

Tableau 6-6 Utilisation de %TYPE

Code PL/SQL	Commentaires
DECLARE v_brevet Pilote.brevet%TYPE;	v_brevet prend le type de la colonne brevet de la table Pilote.

v_prime NUMBER(5,2) := 500.50;
v_prime_min v_prime%TYPE := v_prime*0.9;
BEGIN
…
	v_prime_min prend le type de la variable v_prime et est initialisée à 450,45.

Variables %ROWTYPE

La directive %ROWTYPE permet de travailler au niveau d'un enregistrement (*record*). Ce dernier est composé d'un ensemble de colonnes. L'enregistrement peut contenir toutes les colonnes d'une table ou seulement certaines.

Cette directive est très utile du point de vue de la maintenance des applicatifs. Utilisés à bon escient, elle diminue les changements à apporter au code en cas de modification des types des colonnes de la table. Il est aussi possible d'insérer dans une table ou de modifier une table en utilisant une variable du type %ROWTYPE. Nous détaillerons, au chapitre 7, le mécanisme des curseurs qui emploient beaucoup cette directive. Le tableau suivant décrit ces cas d'utilisation :

Tableau 6-7 Utilisations de %ROWTYPE

Code PL/SQL	Commentaires
DECLARE rty_pilote Pilote%ROWTYPE;	La structure rty_pilote est composée de toutes les colonnes de la table Pilote.
v_brevet Pilote.brevet%TYPE; BEGIN	
SELECT * INTO rty_pilote FROM Pilote WHERE brevet='PL-1'; v_brevet := rty_pilote.brevet; …	Chargement de l'enregistrement rty_pilote à partir d'une ligne de la table Pilote. Accès à des valeurs de l'enregistrement par la notation pointée.
rty_pilote.brevet := 'PL-9'; rty_pilote.nom := 'Pierre Bazex'; … INSERT INTO Pilote VALUES rty_pilote;	Insertion dans la table Pilote à partir d'un enregistrement.

Les colonnes récupérées par la directive %ROWTYPE n'héritent pas des contraintes NOT NULL qui seraient éventuellement déclarées au niveau de la table.

Variables RECORD

Alors que la directive %ROWTYPE permet de déclarer une structure composée de colonnes de tables, elle ne convient pas à des structures de données personnalisées. Le type de données RECORD (disponible depuis la version 7) définit vos propres structures de données (l'équivalent du `struct` en C). Depuis la version 8, les types RECORD peuvent inclure des LOB (BLOB, CLOB et BFILE) ou des extensions objets (REF, TABLE ou VARRAY).

La syntaxe générale pour déclarer un RECORD est la suivante :

```
TYPE nomRecord IS RECORD
( nomChamp typeDonnées [[NOT NULL] {:= | DEFAULT} expression]
 [,nomChamp typeDonnées… ]… );
```

L'exemple suivant décrit l'utilisation d'un `record` :

Tableau 6-8 Manipulation de RECORD

Code PL/SQL	Commentaires

`DECLARE`
`  TYPE avionAirbus_rec IS RECORD`
`   (nserie CHAR(10), nomAvion CHAR(20),`
`    usine  CHAR(10) := 'Blagnac',`
`    nbHVol NUMBER(7,2));`
	Déclaration du RECORD contenant quatre champs ; initialisation du champ `usine` par défaut.

`  r_unA320  avionAirbus_rec;`
`  r_FGLFS  avionAirbus_rec;`
`BEGIN`
	Déclaration de deux variables de type RECORD.

`  r_unA320.nserie  := 'A1';` `  r_unA320.nomAvion := 'A320-200';`	Initialisation des champs d'un RECORD.
`  r_unA320.nbHVol  := 2500.60;` `  r_FGLFS  := r_unA320;`	Affectation d'un RECORD.
…	

Les types RECORD ne peuvent pas être stockés dans une table. En revanche, il est possible qu'un champ d'un RECORD soit lui-même un RECORD, ou soit déclaré avec les directives %TYPE ou %ROWTYPE. L'exemple suivant illustre le RECORD `r_vols` déclaré avec ces trois possibilités :

```
DECLARE
   TYPE avionAirbus_rec  IS RECORD
     (nserie CHAR(10), nomAvion CHAR(20),
      usine  CHAR(10) := 'Blagnac', nbHVol NUMBER(7,2));
```

```
TYPE vols_rec IS RECORD
  (r_aéronef avionAirbus_rec , dateVol DATE,
   rty_coPilote Pilote%ROWTYPE, affrétéPar Compagnie.comp%TYPE);
```

Les RECORD ne peuvent pas être comparés (nullité, égalité et inégalité), ainsi les tests suivants sont incorrects :

```
v1 avionAirbus_rec;
v2 vols_rec;
v3 vols_rec;
BEGIN
…
IF v1 IS NULL THEN …
IF v2 > v3 THEN …
```

Variables tableaux (type TABLE)

Les variables de type TABLE (tableaux associatifs : *associative arrays*) permettent de définir et de manipuler des tableaux dynamiques (définis sans dimension initiale). Un tableau est composé d'une clé primaire (de type BINARY_INTEGER, PLS_INTEGER, ou chaîne de caractères) pour accéder à chaque élément de type scalaire ou composé (RECORD ou ROWTYPE).

Syntaxe

La syntaxe générale pour déclarer un type de tableau et une variable tableau est la suivante :

```
TYPE nom_type_tableau IS TABLE OF
   {type_scalaire          | variable%TYPE  | nom_RECORD
   nom_table.colonne%TYPE  | nom_table.%ROWTYPE } [NOT NULL]
  [INDEX BY {BINARY_INTEGER | PLS_INTEGER} | VARCHAR2(taille) ];
nom_tableau nom_type_tableau;
```

L'option INDEX BY BINARY_INTEGER est facultative depuis la version 8 de PL/SQL. Si elle est omise, le type déclaré est considéré comme une *nested table* (extension objet). Si elle est présente, l'indexation ne commence pas nécessairement à 1 et peut être même négative (l'intervalle de valeurs du type BINARY_INTEGER va de -2 147 483 647 à 2 147 483 647).

L'exemple suivant décrit la déclaration de trois tableaux et l'affectation de valeurs à différents indices (-1, -2 et 7 800). L'accès à des champs d'éléments complexes se fait à l'aide de la notation pointée (voir la dernière instruction).

Tableau 6-9 Tableaux PL/SQL

Code PL/SQL	Commentaires

`DECLARE`
`  TYPE brevets_tytab IS TABLE OF VARCHAR2(6)`
`  INDEX BY BINARY_INTEGER;`
	Type de tableaux de chaînes de six caractères.

`TYPE nomPilotes_tytab IS TABLE OF Pilote.nom%TYPE` `  INDEX BY BINARY_INTEGER;`	Type de tableaux de colonnes de type `nom` de la table `Pilote`.
`TYPE pilotes_tytab IS TABLE OF Pilote%ROWTYPE` `  INDEX BY BINARY_INTEGER;`	Type de tableaux d'enregistrements de type de la table `Pilote`.

`  tab_brevets  brevets_tytab;`
`  tab_nomPilotes nomPilotes_tytab;`
`  tab_pilotes  pilotes_tytab;`
	Déclaration des tableaux.

`BEGIN`
`  tab_brevets(-1)  := 'PL-1';`
`  tab_brevets(-2)  := 'PL-2';`
`  tab_nomPilotes(7800)  := 'Bidal';`
`  tab_pilotes(0).brevet := 'PL-0';`
`END;`
	Initialisations.

Fonctions pour les tableaux

PL/SQL propose un ensemble de fonctions qui permettent de manipuler des tableaux (également disponibles pour les *nested tables* et *varrays*). Ces fonctions sont les suivantes (les trois dernières sont des procédures) :

Tableau 6-10 Fonctions pour les tableaux

Fonction	Description
`EXISTS(x)`	Retourne `TRUE` si le x^e élément du tableau existe.
`COUNT`	Retourne le nombre d'éléments du tableau.
`FIRST / LAST`	Retourne le premier/dernier indice du tableau (`NULL` si tableau vide).
`PRIOR(x) / NEXT(x)`	Retourne l'élément avant/après le x^e élément du tableau.

`DELETE`
`DELETE(x)`
`DELETE(x,y)`
	Supprime un ou plusieurs éléments au tableau.

Il n'est pas possible actuellement d'appeler une de ces fonctions dans une instruction SQL (`SELECT`, `INSERT`, `UPDATE` ou `DELETE`).

Les exemples suivants décrivent l'utilisation de ces fonctions.

Tableau 6-11 Fonctions PL/SQL pour les tableaux

Code PL/SQL	Commentaires
`IF tab_pilotes.EXISTS(0) THEN…`	Renvoie « vrai » car il existe un élément à l'indice 0.
`v_nombre := tab_brevets.COUNT;`	La variable `v_nombre` contient 2.
`v_premier := tab_brevets.FIRST;` `v_dernier := tab_brevets.LAST;`	La variable `v_premier` contient -2, `v_dernier` contient -1.
`v_avant := tab_brevets.PRIOR(-1);`	La variable `v_avant` contient -2.
`tab_brevets.DELETE;`	Suppression de tous les éléments de `tab_brevets`.

Résolution de noms

Lors des conflits potentiels de noms (variables ou colonnes) dans des instructions SQL (principalement `INSERT`, `UPDATE`, `DELETE` et `SELECT`), le nom de la colonne de la table est prioritairement interprété au détriment de la variable (de même nom).

Dans l'exemple suivant, l'instruction `DELETE` supprime tous les pilotes (et non pas seulement le pilote 'Pierre Lamothe'), car Oracle considère les deux identificateurs comme la colonne de la table et non pas comme deux variables différentes !

```
DECLARE
   nom CHAR(20) := 'Pierre Lamothe';
BEGIN
   DELETE FROM Pilote WHERE nom = nom ;
   …
```

Pour se prémunir de tels effets de bord, deux solutions existent. La première consiste à nommer toutes les variables explicitement et différemment des colonnes. La deuxième consiste à utiliser une étiquette de bloc (*block label*) pour lever les ambiguïtés. Le tableau suivant illustre ces solutions concernant notre exemple :

Tableau 6-12 Éviter les ambiguïtés

Préfixer les variables	Étiquette de bloc

`DECLARE`
`  v_nom CHAR(20) := 'Pierre Lamothe';`
`BEGIN`
`  DELETE FROM Pilote WHERE nom = v_nom;`
`END;`
	`<<principal>>`
	`DECLARE`
	`  nom CHAR(20) := 'Pierre Lamothe';`
	`BEGIN`
	`  DELETE FROM Pilote`
	`  WHERE nom = principal.nom;`
	`END;`

Opérateurs

Les opérateurs SQL étudiés au chapitre 4 (logiques, arithmétiques, concaténation…) sont disponibles aussi avec PL/SQL. Les règles de priorité sont les mêmes que dans le cas de SQL.

L'opérateur `IS NULL` permet de tester une expression avec la valeur `NULL`. Toute expression arithmétique contenant une valeur nulle est évaluée à `NULL`.

Le tableau suivant illustre quelques utilisations possibles d'opérateurs logiques :

Tableau 6-13 Utilisation d'opérateurs

Code PL/SQL	Commentaires

`DECLARE`
`  v_compteur NUMBER(3) DEFAULT 0;`
`  v_booléen  BOOLEAN;`
`  v_nombre   NUMBER(3);`
`BEGIN`	

`  v_compteur := v_compteur+1;`	Incrémentation, opérateur +
`  v_booléen := (v_compteur = v_nombre);`	`v_booléen` reçoit `NULL` du fait de la condition.
`  v_booléen  := (v_nombre IS NULL);`	`v_booléen` reçoit `TRUE` car la condition est vraie.

Variables de substitution

Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*Plus. Ces variables sont dites de substitution. On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable du symbole « & » (avec ou sans guillemets simples suivant qu'il s'agit d'un nombre ou pas).

Le tableau suivant illustre un exemple de deux variables de substitution. La directive `ACCEPT` permet la saisie au clavier de variables dans l'interface SQL*Plus. Elle doit être utilisée conjointement à la directive `PROMPT` toutes deux placées en amont d'un bloc PL/SQL qui devra être exécuté via la commande `START`. Dans cet exemple on extrait le nom et le nombre d'heures de vol d'un pilote. Son numéro de brevet et la durée du vol sont lus au clavier et la durée est ajoutée au nombre d'heures de vol du pilote. Il est à noter qu'il ne faut pas déclarer des variables de substitution.

Tableau 6-14 Variables de substitution

Code PL/SQL	Sous SQL*Plus

`ACCEPT s_brevet PROMPT 'Entrer code Brevet : '`
`ACCEPT s_duréeVol PROMPT 'Entrer durée du vol : '`
`DECLARE`
`  v_nom    Pilote.nom%TYPE;`
`  v_nbHVol Pilote.nbHVol%TYPE;`
`BEGIN`
`   SELECT nom, nbHVol INTO v_nom, v_nbHVol`
`        FROM Pilote WHERE brevet = '&s_brevet';`
`   v_nbHVol := v_nbHVol + &s_duréeVol;`
`   DBMS_OUTPUT.PUT_LINE`
`      ('Total heures vol : ' || v_nbHVol ||`
`       ' de ' || v_nom);`
`END;`
	`Entrer code Brevet : PL-2`
	`Entrer durée du vol : 27`
	`Total heures vol : 927 de`
	`Didier Linxe`
	
	`Procédure PL/SQL terminée`
	`avec succès.`

Il faut exécuter le bloc à l'aide de la commande `START` et non pas par copier-coller d'un éditeur de texte vers la fenêtre SQL*Plus (à cause des instructions d'entrée `ACCEPT`).

Variables de session

Il est possible de définir des variables de session (*bind variables*) déclarées sous SQL*Plus à l'extérieur d'un programme PL/SQL tout en pouvant être utilisées dans le corps du programme. La directive SQL*Plus à utiliser en début de bloc est `VARIABLE`. Dans le code PL/SQL, il faut faire préfixer le nom de la variable de session du symbole `:`. L'affichage de la variable sous SQL*Plus est réalisé par la directive `PRINT`.

Le tableau suivant illustre un exemple de variable de session :

Tableau 6-15 Variable de session

Code PL/SQL	Sous SQL*Plus

`VARIABLE g_compteur NUMBER;`
`DECLARE`
`  v_compteur NUMBER(3) := 99;`
`BEGIN`
`  :g_compteur := v_compteur+1;`
`END;`
	`SQL> PRINT g_compteur;`
	
	`G_COMPTEUR`
	`----------`
	`       100`

Conventions recommandées

Adoptez les conventions d'écriture suivantes pour que vos programmes PL/SQL soient plus facilement lisibles et maintenables :

Tableau 6-16 Conventions PL/SQL

Objet	Convention	Exemple
Variable	**v**_*nomVariable*	v_compteur
Constante	**c**_*nomConstante*	c_pi
Exception	**e**_*nomException*	e_pasTrouvé
Type `RECORD`	*nomRecord*_**tyrec**	pilote_tyrec
Variable `RECORD`	**v**_*nomVariable*_*nomRecord*_*rec*	V_pil_pilote_rec
Variable `ROWTYPE`	**rty**_*nomVariable*	rty_pilote
Type-tableau	*nomTypeTableau*_**tytab**	pilotes_tytab
Variable tableau	**tab**_*nomTableau*	tab_pilotes
Curseur	*nomCurseur*_**cur**	pilotes_cur
Variable de substitution (SQL*Plus)	**s**_*nomVariable*	s_brevet
Variable de session (globale)	**g**_*nomVariable*	g_brevet

Types de données PL/SQL

PL/SQL inclut tous les types de données SQL que nous avons étudiés aux chapitres 1 et 2 (NUMBER, CHAR, BOOLEAN, VARCHAR2, DATE, TIMESTAMP, INTERVAL, BLOB, ROWID...). Nous verrons ici les nouveaux types de données propres à PL/SQL.

Types prédéfinis

Les types BINARY_INTEGER et PLS_INTEGER conviennent aux entiers signés (domaine de valeurs de -2^{31} à 2^{31}, soit -2147483647 à +2147483647). Ces types requièrent moins d'espace de stockage que le type NUMBER.

Les types PLS_INTEGER et BINARY_INTEGER ne se comportent pas de la même manière lors d'erreurs de dépassement (*overflow*). PLS_INTEGER déclenchera l'exception ORA-01426 : dépassement numérique. BINARY_INTEGER ne provoque aucune exception si le résultat est affecté à une variable NUMBER.

PLS_INTEGER est plus performant au niveau des opérations arithmétiques que les types NUMBER et BINARY_INTEGER qui utilisent des librairies mathématiques.

Sous-types

Chaque type de données PL/SQL prédéfini a ses caractéristiques (domaine de valeurs, fonctions applicables...). Les sous-types de données permettent de restreindre certaines de ces caractéristiques à des données. Un sous-type n'introduit pas un nouveau type mais en restreint un existant. Les sous-types servent principalement à rendre compatibles des applications à la norme SQL ANSI/ISO ou plus pertinentes certaines déclarations de variables.

PL/SQL propose plusieurs sous-types prédéfinis et il est possible de définir des sous-types personnalisés.

Prédéfinis

Le tableau suivant décrit les sous-types prédéfinis par PL/SQL.

Tableau 6-17 Sous-types prédéfinis

Sous-type	Type restreint (sur-type)	Caractéristiques
CHARACTER	CHAR	Mêmes caractéristiques.
INTEGER	NUMBER(38,0)	Entiers sans décimales.
PLS_INTEGER	NUMBER	Déjà étudié.
BINARY_INTEGER	NUMBER	Déjà étudié.

Tableau 6-17 Sous-types prédéfinis *(suite)*

Sous-type	Type restreint (sur-type)	Caractéristiques
NATURAL, POSITIVE	BINARY_INTEGER	Non négatif.
NATURALN, POSITIVEN		Non négatif et non nul.
SIGNTYPE		Domaine de valeurs {-1, 0, 1}.
DEC, DECIMAL, NUMERIC	NUMBER	Décimaux, précision de 38 chiffres.
DOUBLE PRECISION, FLOAT, REAL		Flottants.
INTEGER, INT, SMALLINT		Entiers sur 38 chiffres.

Personnalisés

Il est possible de définir un sous-type (dit « personnalisé » car n'existant que durant le programme) par la syntaxe suivante :

```
SUBTYPE nomSousType IS typeBase[(contrainte)] [NOT NULL];
```

- *typeBase* est un type prédéfini ou personnalisé.
- *contrainte* s'applique au type de base et concerne seulement la précision ou la taille maximale.

Des exemples de déclarations de sous-types sont présentés dans le tableau suivant.

Tableau 6-18 Sous-types PL/SQL

Code PL/SQL	Commentaires
DECLARE **SUBTYPE** dateNaiss_sty IS DATE **NOT NULL**; **SUBTYPE** compteur_sty IS NATURAL; **SUBTYPE** insee_sty IS NUMBER(**13**) **SUBTYPE** nombre_sty IS NUMBER(**1,0**); TYPE r_tempsCourse IS RECORD (minutes NUMBER(**2**), secondes INTEGER); **SUBTYPE** finishTime IS r_tempsCourse; **SUBTYPE** brevet_sty IS Pilote.brevet%**TYPE**;	Directive NOT NULL. Contraintes sur les tailles de NUMBER, nombre_sty ira de -9 à 9. Sous-type d'un RECORD. Sous-type d'un %TYPE.

Des exceptions sont levées lorsque les valeurs des variables ne respectent pas les contraintes des sous-types. Par exemple, l'initialisation « v1 nombre_sty:=10; » déclencherait une exception ORA-06502 (voir plus haut).

Le sous-type SIMPLE_INTEGER

Le sous-type SIMPLE_INTEGER dérive du type PLS_INTEGER. Bien que son domaine de valeurs soit identique à celui de PLS_INTEGER (-2 147 483 648 à 2 147 483 647), il est affecté d'une

contrainte NOT NULL et diffère de son prédécesseur du fait de sa robustesse de capacité de dépassement (*overflow*). En effet, l'erreur ORA-01426 : numeric overflow n'est plus levée en cas de dépassement en positif ou en négatif d'une variable de type SIMPLE_INTEGER.

Les sous-types flottants

Les sous-types SIMPLE_FLOAT et SIMPLE_DOUBLE dérivent respectivement des types BINARY_FLOAT et BINARY_DOUBLE (mêmes domaines de valeurs). Chacun diffère de son prédécesseur du fait de l'existence d'une contrainte NOT NULL.

Sans utiliser de ressources gérant la nullité, ces nouveaux sous-types sont plus performants, lors d'opérations, que leurs prédécesseurs dans un mode opératoire par défaut (PLSQL_CODE_TYPE='NATIVE').

Variable de type séquence

Il est désormais possible d'utiliser les directives CURRVAL et NEXTVAL au sein d'un bloc PL/SQL (qui ne sont donc plus limitées aux instructions SELECT, INSERT et UPDATE comme indiqué au chapitre 2). Les expressions *séquence*.CURRVAL et *séquence*.NEXTVAL peuvent être présentes à tout endroit où une expression de type NUMBER peut apparaître.

En considérant l'exemple de séquence du chapitre 2, le tableau suivant présente un bloc PL/SQL exploitant la séquence à l'aide de deux affectations.

Tableau 6-19 Variable de type séquence

Code SQL et PL/SQL

```
CREATE TABLE Affreter
  (numAff NUMBER(5),comp CHAR(4),
   immat CHAR(6), dateAff DATE,
   nbPax NUMBER(3),CONSTRAINT
   pk_Affreter PRIMARY KEY (numAff));

CREATE SEQUENCE seqAff
   MAXVALUE 10000
   NOMINVALUE;

DECLARE
   seq_valeur NUMBER;
   BEGIN
   seq_valeur := seqAff.CURRVAL;
   DBMS_OUTPUT.PUT_LINE('Pour l''ins-
tant, il y a '||TO_CHAR(seq_valeur)||'
affrètements.');
   seq_valeur := seqAff.NEXTVAL;
   INSERT INTO Affreter
       VALUES(seq_valeur,'AF',
              'F-WOWW',SYSDATE-5,490);
END;
/
```

Exécution sous SQL*Plus

```
SQL> INSERT INTO Affreter VALUES
(seqAff.NEXTVAL,'AF','F-WTSS',SYS-
DATE,85);
SQL> INSERT INTO Affreter VALUES
(seqAff.NEXTVAL,'SING','F-GAFU','05-
02-2007',155);

SQL> SELECT * FROM Affreter ;
NUMAFF COMP IMMAT  DATEAFF       NBPAX
------ ---- ------ -------- ----------
     1 AF   F-WTSS 25/11/07         85
     2 SING F-GAFU 05/02/07        155

-- exécution du bloc ici
Pour l'instant, il y a 2 affrètements.
Procédure PL/SQL terminée avec succès.

SQL> SELECT * FROM Affreter ;
NUMAFF COMP IMMAT  DATEAFF       NBPAX
------ ---- ------ -------- ----------
     1 AF   F-WTSS 25/11/07         85
     2 SING F-GAFU 05/02/07        155
     3 AF   F-WOWW 20/11/07        490
```

Conversions de types

Comme pour SQL, les conversions de types PL/SQL sont implicites ou explicites. Les principales fonctions de conversion ont déjà été étudiées au chapitre 4, section « Conversions ».

L'exception levée en cas d'affectation incorrecte pour les sous-types de `BINARY_INTEGER` est : `ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur.`

L'exception qui est levée en cas d'affectation de la valeur nulle pour les sous-types `NATURALN` et `POSITIVEN` est : `PLS-00218: une variable déclarée NOT NULL doit avoir une affectation d'initialisation.`

Structures de contrôles

En tant que langage procédural, PL/SQL offre la possibilité de programmer :

- les structures conditionnelles *si* et *cas* (`IF`... et `CASE`) ;
- les structures répétitives *tant que*, *répéter* et *pour* (`WHILE`, `LOOP`, `FOR`).

Structures conditionnelles

PL/SQL propose deux structures pour programmer une action conditionnnelle : la structure `IF` et la structure `CASE`.

Trois formes de IF

Suivant les tests à programmer, on peut distinguer trois formes de structure `IF` : `IF-THEN` (*si-alors*) `IF-THEN-ELSE` (avec le *sinon* à programmer), et `IF-THEN-ELSIF` (imbrications de conditions).

Le tableau suivant décrit l'écriture des différentes structures conditionnelles `IF`. Notez « `END IF` » en fin de structure et non pas « `ENDIF` ». L'exemple affiche un message différent selon la nature du numéro de téléphone contenu dans la variable `v_téléphone`. La fonction `PUT_LINE` du paquetage `DBMS_OUTPUT` permet d'afficher une chaîne de caractères dans l'interface SQL*Plus. Nous étudierons plus loin les fonctions de ce paquetage.

Tableau 6-20 Structures IF

IF-THEN	IF-THEN-ELSE	IF-THEN-ELSIF
IF *condition* **THEN** *instructions;* **END IF;**	**IF** *condition* **THEN** *instructions;* **ELSE** *instructions;* **END IF;**	**IF** *condition1* **THEN** *instructions;* **ELSIF** *condition2* **THEN** *instructions;* **ELSE** *instructions;* **END IF;**

```
DECLARE
 v_téléphone CHAR(14) NOT NULL := '06-76-85-14-89';
BEGIN
 IF SUBSTR(v_téléphone,1,2)='06' THEN
   DBMS_OUTPUT.PUT_LINE ('C''est un portable!');
 ELSE
   DBMS_OUTPUT.PUT_LINE ('C''est un fixe…');
 END IF;
END;
```

Conditions booléennes

Les tableaux suivants précisent le résultat d'opérateurs logiques qui mettent en jeu des variables booléennes pouvant prendre trois valeurs (`TRUE`, `FALSE`, `NULL`). Il est à noter que la négation de `NULL` (`NOT NULL`) renvoie une valeur nulle.

Tableau 6-21 Opérateur AND

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Tableau 6-22 Opérateur OR

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Structure CASE

Comme l'instruction IF, la structure CASE permet d'exécuter une séquence d'instructions en fonction de différentes conditions. La structure CASE est utile lorsqu'il faut évaluer une même expression et proposer plusieurs traitements pour diverses conditions.

En fonction de la nature de l'expression et des conditions, une des deux écritures suivantes peut être utilisée :

Tableau 6-23 Structures CASE

CASE	*searched* CASE
[<<*étiquette*>>] **CASE** variable WHEN *expr1* THEN *instructions1*; WHEN *expr2* THEN *instructions2*; … WHEN *exprN* THEN *instructionsN*; [ELSE *instructionsN+1*;] **END CASE** [*étiquette*];	[<<*étiquette*>>] **CASE** WHEN *condition1* THEN *instructions1*; WHEN *condition2* THEN *instructions2*; … WHEN *conditionN* THEN *instructionsN*; [ELSE *instructionsN+1*;] **END CASE** [*étiquette*];

Le tableau suivant décrit l'écriture avec IF d'une programmation qu'il est plus rationnel d'effectuer avec une structure CASE (de type *searched*) :

Tableau 6-24 Différentes programmations

IF	CASE

	DECLARE
	v_mention CHAR(2);
	v_note NUMBER(4,2) := 9.8;
	BEGIN
	…

IF v_note >= 16 THEN v_mention := 'TB'; ELSIF v_note >= 14 THEN v_mention := 'B'; ELSIF v_note >= 12 THEN v_mention := 'AB'; ELSIF v_note >= 10 THEN v_mention := 'P'; ELSE v_mention := 'R'; END IF; …	CASE WHEN v_note >= 16 THEN v_mention := 'TB'; WHEN v_note >= 14 THEN v_mention := 'B'; WHEN v_note >= 12 THEN v_mention := 'AB'; WHEN v_note >= 10 THEN v_mention := 'P'; ELSE v_mention := 'R'; END CASE; …

La clause ELSE est optionnelle. Si elle n'est pas présente, PL/SQL ajoute par défaut l'instruction « ELSE RAISE CASE_NOT_FOUND; ». Celle-ci lève l'exception du même nom quand le code exécuté passe par cette instruction.

Structures répétitives

Les trois structures répétitives *tant que*, *répéter* et *pour* utilisent l'instruction `LOOP... END LOOP`.

Structure tant que

La structure *tant que* se programme à l'aide de la syntaxe suivante. Avant chaque itération (et notamment avant la première), la condition est évaluée. Si elle est vraie, la séquence d'instructions est exécutée, puis la condition est réévaluée pour un éventuel nouveau passage dans la boucle. Ce processus continue jusqu'à ce que la condition soit fausse pour passer en séquence après le `END LOOP`. Quand la condition n'est jamais fausse, on dit que le programme boucle…

```
WHILE condition LOOP
   instructions;
END LOOP;
```

Le tableau suivant décrit la programmation de deux *tant que*. Le premier calcule la somme des 100 premiers entiers. Le second recherche le premier numéro 4 dans une chaîne de caractères.

Tableau 6-25 Structures tant que

Condition simple	Condition composée

`DECLARE`
`  v_somme  NUMBER(4) := 0;`
`  v_entier  NUMBER(3) := 1;`
`BEGIN`
`  WHILE (v_entier <= 100) LOOP`
`  v_somme  := v_somme+v_entier;`
`  v_entier := v_entier + 1;`
`  END LOOP;`
`  DBMS_OUTPUT.PUT_LINE`
`  ('Somme = ' || v_somme);`
`END;`
	`DECLARE`
	`  v_téléphone CHAR(14)  := '06-76-85-14-89';`
	`  v_trouvé  BOOLEAN  := FALSE;`
	`  v_indice  NUMBER(2) := 1;`
	`BEGIN`
	`  WHILE (v_indice <= 14 AND NOT v_trouvé) LOOP`
	`  IF SUBSTR(v_téléphone,v_indice,1) = '4' THEN`
	`  v_trouvé := TRUE;`
	`  ELSE`
	`  v_indice := v_indice + 1;`
	`  END IF;`
	`  END LOOP;`
	`  IF v_trouvé THEN`
	`  DBMS_OUTPUT.PUT_LINE`
	`  ('Trouvé 4 à l''indice : ' || v_indice);`
	`  END IF;`
	`END;`

Somme = 5 050	Trouvé 4 à l'indice : 11

Cette structure est la plus puissante car elle permet de programmer aussi un *répéter* et un *pour*. Elle doit être utilisée quand il est nécessaire de tester une condition avant d'exécuter les instructions contenues dans la boucle.

Structure répéter

La structure *répéter* se programme à l'aide de la syntaxe LOOP EXIT suivante :

```
LOOP
   instructions;
   EXIT [WHEN condition;]
END LOOP;
```

La particularité de cette structure est que la première itération est effectuée quelles que soient les conditions initiales. La condition n'est évaluée qu'en fin de boucle.

- Si aucune condition n'est spécifiée (WHEN *condition* absent), la sortie de la boucle est immédiate dès la fin des instructions.
- Si la condition est fausse, la séquence d'instructions est de nouveau exécutée. Ce processus continue jusqu'à ce que la condition soit vraie pour passer en séquence après le END LOOP.
- Quand la condition n'est jamais fausse, on dit aussi que le programme boucle…

Le tableau suivant décrit la programmation de la somme des 100 premiers entiers et de la recherche du premier numéro 4 dans une chaîne de caractères à l'aide de la structure *répéter*.

Tableau 6-26 Structures répéter

Condition simple	Condition composée

DECLARE
  v_somme NUMBER(4) := 0;
  v_entier NUMBER(3) := 1;
BEGIN
  LOOP
  v_somme := v_somme+v_entier;
  v_entier := v_entier + 1;
  EXIT WHEN v_entier > 100;
  END LOOP
  DBMS_OUTPUT.PUT_LINE
  ('Somme = ' || v_somme);
END;
	DECLARE
	  v_téléphone CHAR(14) := '06-76-85-14-89';
	  v_trouvé BOOLEAN := FALSE;
	  v_indice NUMBER(2) := 1;
	BEGIN
	  LOOP
	  IF SUBSTR(v_téléphone,v_indice,1) = '4' THEN
	  v_trouvé := TRUE;
	  ELSE
	  v_indice := v_indice + 1;
	  END IF;
	  EXIT WHEN (v_indice > 14 OR v_trouvé);
	  END LOOP;
	  IF v_trouvé THEN
	  DBMS_OUTPUT.PUT_LINE
	  ('Trouvé 4 à l''indice : ' || v_indice);
	  END IF;
	END;

Cette structure doit être utilisée quand il n'est pas nécessaire de tester la condition avec les données initiales avant d'exécuter les instructions contenues dans la boucle.

Structure pour

Célèbre pour les parcours de vecteurs, tableaux et matrices en tout genre, la structure *pour* se caractérise par la connaissance a priori du nombre d'itérations que le programmeur souhaite faire effectuer à son algorithme.

La syntaxe générale de cette structure est la suivante :

```
FOR compteur IN [REVERSE] valeurInf..valeurSup LOOP
   instructions;
END LOOP;
```

Le nombre d'itérations est calculé dès le premier passage dans la condition et n'est jamais réévalué par la suite quelles que soient les instructions contenues dans la boucle. À la première itération le compteur reçoit automatiquement la valeur initiale (*valeurInf*). Après chaque passage le compteur est de fait incrémenté (ou décrémenté si l'option REVERSE a été choisie). La sortie de la boucle est automatique après l'itération correspondant à la valeur finale du compteur (*valeurSup*). La déclaration de la variable *compteur* n'est pas obligatoire.

Il n'est pas possible de modifier le pas de la variable d'itération dans le corps d'une boucle FOR...LOOP.

Le tableau suivant décrit la programmation de la somme des 100 premiers entiers et de la recherche du premier numéro 4 dans une chaîne de caractères à l'aide de la structure *pour*.

Tableau 6-27 Structures pour

Condition simple

```
DECLARE
  v_somme      NUMBER(4) := 0;
BEGIN
  FOR v_entier IN 1..100 LOOP
    v_somme := v_somme+v_entier;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE
    ('Somme = ' || v_somme);
END;
```

Condition composée

```
DECLARE
  v_téléphone CHAR(14)   := '06-76-85-14-89';
  v_trouvé    BOOLEAN    := FALSE;
  v_indice    NUMBER(2);
  v_compteur  NUMBER(2);
BEGIN
  FOR v_compteur IN 1..14 LOOP
    IF SUBSTR(v_téléphone,v_compteur,1)= '4' AND
       NOT v_trouvé THEN
      v_trouvé := TRUE;
      v_indice := v_compteur;
    END IF;
  END LOOP;
  IF v_trouvé THEN
    DBMS_OUTPUT.PUT_LINE
      ('Trouvé 4 à l''indice : ' || v_indice);
    END IF;
END;
```

Cette structure convient bien pour le premier exemple car on sait a priori qu'il faut faire 100 itérations. Pour le second, cette structure peut être utilisée mais est moins efficace car elle impose de parcourir tous les éléments de la chaîne alors qu'on pourrait interrompre le traitement

dès le numéro trouvé. De plus il est nécessaire de modifier le test dans la boucle de manière à ne garder que le premier numéro trouvé (et pas le dernier si le test n'était pas changé).

Boucles avec étiquettes

Comme les blocs de traitements, les boucles peuvent être étiquetées. L'étiquette est notée par un identifiant qui apparaît après l'instruction de fin de boucle par la syntaxe suivante :

```
<<etiquette>>
LOOP
   instructions;
END LOOP etiquette;
```

Ce mécanisme présente les deux avantages suivants :

- meilleure lisibilité du code ;
- sortie possible de plusieurs boucles imbriquées : de la boucle courante et de celle(s) qui l'inclut(ent).

L'exemple suivant décrit la programmation de la recherche d'un code d'une carte bleue (ici 8595) en considérant tous les codes possibles (en partant de 0000). Quatre boucles sont imbriquées et on doit sortir du programme dès que le code est trouvé pour ne pas examiner les autres combinaisons.

Tableau 6-28 Boucle étiquetée

Déclarations

```
DECLARE
  v_carteBleue NUMBER(4) := 8595;
  v_test       NUMBER(4) := 0000;
  v_unité      NUMBER(2);
  v_dizaine    NUMBER(3);
  v_centaine   NUMBER(4);
  v_millier    NUMBER(5);
…
```

Code PL/SQL

```
BEGIN
v_millier := 0;
<<principal>>
LOOP
 v_centaine := 0;
 LOOP
 v_dizaine := 0;
 LOOP
 v_unité := 0;
 LOOP
  EXIT principal WHEN v_test=v_carteBleue;
  EXIT WHEN v_unité = 11;
  v_test  := v_test + 1;
  v_unité := v_unité + 1;
 END LOOP;
 v_dizaine  := v_dizaine + 10;
 EXIT WHEN v_dizaine = 100;
 END LOOP;
 v_centaine := v_centaine + 100;
 EXIT WHEN v_centaine = 1000;
 END LOOP;
 v_millier := v_millier + 1000;
 EXIT WHEN v_millier = 10000;
END LOOP principal;
DBMS_OUTPUT.PUT_LINE
    ('le code CB est : ' || v_test);
END;
```

L'étiquette <<principal>> marque la première boucle. La boucle la plus imbriquée possède deux conditions de sortie : la nominale EXIT WHEN... et la sortie forcée EXIT principal WHEN...

La directive CONTINUE

La version 11*g* de PL/SQL propose la directive CONTINUE. Comme pour Java, cette directive, au sein d'une structure répétitive, interrompt l'itération en cours et revient au début de la structure (à la condition pour un WHILE, à l'itération suivante pour un FOR ou à l'instruction qui suit le LOOP) pour éventuellement refaire une nouvelle itération (à l'inverse, la directive EXIT interrompt à la fois l'itération mais aussi la structure répétitive).

La syntaxe revêt une forme inconditionnelle et une forme conditionnelle (avec WHEN).

```
CONTINUE [ etiquette ] [ WHEN condition ];
```

Dans le bloc PL/SQL suivant, la directive CONTINUE déroute le programme après l'instruction LOOP.

Tableau 6-29 Directive CONTINUE

Bloc PL/SQL	Résultat

DECLARE
  x NUMBER := 0;
BEGIN
 LOOP
  -- On arrive ici après CONTINUE
  DBMS_OUTPUT.PUT_LINE
  ('Dans la boucle,x = '||TO_CHAR(x));
  x := x + 1;
  IF (x < 3) THEN
  CONTINUE;
  END IF;
  DBMS_OUTPUT.PUT_LINE
  ('Après CONTINUE,x = '||TO_CHAR(x));
  EXIT WHEN (x = 5);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE
  ('Après la structure, x = '|| TO_CHAR(x));
END;
/
	Dans la boucle, x = 0
	Dans la boucle, x = 1
	Dans la boucle, x = 2
	Après CONTINUE, x = 3
	Dans la boucle, x = 3
	Après CONTINUE, x = 4
	Dans la boucle, x = 4
	Après CONTINUE, x = 5
	Après la structure, x = 5
	
	Procédure PL/SQL terminée
	avec succès.

La forme conditionnelle de l'instruction CONTINUE permet de remplacer une structure IF *condition* THEN CONTINUE. Ainsi, en remplaçant la structure conditionnelle dans le bloc précédent par l'instruction « CONTINUE WHEN x < 3; », on obtient le même résultat.

Si vous utilisez la directive CONTINUE dans une boucle FOR manipulant un curseur (étudié au chapitre 7), vous fermez automatiquement le curseur.

Interactions avec la base

Cette section décrit les mécanismes offerts par Oracle pour interfacer un programme PL/SQL avec une base de données.

Extraire des données

La seule instruction capable d'extraire des données à partir d'un programme PL/SQL est SELECT. Étudiée au chapitre 4 dans un contexte SQL, la particularité de cette instruction au niveau de PL/SQL est la directive INTO comme le montre la syntaxe suivante :

```
SELECT liste INTO { nomVariablePLSQL [,nomVariablePLSQL]… |
nomRECORD } FROM nomTable …;
```

La clause INTO est obligatoire et permet de préciser les noms des variables (PL/SQL, globales ou hôtes) contenant les valeurs renvoyées par la requête (une variable par colonne ou une expression sélectionnée en respectant l'ordre). A contrario, la clause INTO est interdite sous SQL.

Le tableau suivant décrit l'extraction de différentes données dans diverses variables :

Tableau 6-30 Extraction de données

Code PL/SQL	Commentaires

`VARIABLE g_nom CHAR(20);`
`DECLARE`
`  rty_pilote  Pilote%ROWTYPE;`
`  v_compa  Pilote.compa%TYPE;`
`BEGIN`
`  SELECT *  INTO rty_pilote`
`  FROM Pilote WHERE brevet='PL-1';`
	Extraction d'un enregistrement entier dans la variable rty_pilote.

`  SELECT compa  INTO v_compa` `  FROM Pilote WHERE brevet='PL-2';`	Extraction de la valeur d'une colonne dans la variable v_compa.

`SELECT nom  INTO :g_nom`
`  FROM Pilote WHERE brevet='PL-2';`
`END;`
	Extraction de la valeur d'une colonne dans la variable globale g_nom.

Une requête `SELECT ... INTO` doit renvoyer un seul enregistrement (conformément à la norme ANSI du code SQL intégré).

Pour traiter des requêtes renvoyant plusieurs enregistrements, il faut utiliser des curseurs (étudiés au chapitre suivant).

Une requête qui renvoie plusieurs enregistrements, ou qui n'en renvoie aucun, génère une erreur PL/SQL en déclenchant des exceptions (respectivement `ORA-01422 TOO_MANY_ROWS` et `ORA-01403 NO_DATA_FOUND`). Le traitement des exceptions est détaillé dans le chapitre suivant.

Le tableau ci-après décrit l'extraction de différentes données dans diverses variables. La première requête ramène la liste des codes des compagnies qui ne peuvent pas être affectées à la simple variable `v_compa`. La deuxième requête n'extrait aucun résultat car aucun pilote n'a un tel code brevet.

Tableau 6-31 Extractions par SELECT

Erreur `TOO_MANY_ROWS`	**Erreur** `NO_DATA_FOUND`

`DECLARE`
`  v_compa Pilote.compa%TYPE;`
`BEGIN`
`  SELECT compa  INTO v_compa`
`  FROM Pilote;`
`END;`
	`DECLARE`
	`  rty_pilote Pilote%ROWTYPE;`
	`BEGIN`
	`  SELECT * INTO rty_pilote`
	`  FROM Pilote WHERE brevet = '$£*';`
	`END;`

`ORA-01422: l'extraction exacte ramène plus que le nombre de lignes demandé`	`ORA-01403: Aucune donnée trouvée`

Il va de soi que les fonctions SQL (mono et multilignes) étudiées au chapitre 4 sont également disponibles sous PL/SQL à condition de les utiliser au sein d'une instruction `SELECT`. Deux exemples sont décrits dans le tableau suivant :

Tableau 6-32 Utilisation de fonctions

Monolignes	**Multilignes**

`DECLARE`
`  v_nomEnMAJUSCULES Pilote.nom%TYPE;`
`BEGIN`
`  SELECT UPPER(nom)`
`  INTO v_nomEnMAJUSCULES`
`  FROM Pilote WHERE brevet = 'PL-1';`
`END;`
	`VARIABLE g_plusGrandHVol NUMBER;`
	`DECLARE`
	`BEGIN`
	`  SELECT MAX(nbHVol) INTO :g_plusGrandHVol`
	`  FROM Pilote;`
	`END;`

Manipuler des données

Les seules instructions disponibles pour manipuler, sous PL/SQL, les éléments d'une base de données sont les mêmes que celles proposées par SQL : INSERT, UPDATE, DELETE et MERGE. Pour libérer les verrous au niveau d'un enregistrement (et des tables), il faudra ajouter les instructions COMMIT ou ROLLBACK (aspects étudiés en fin de chapitre).

Insertions

Le tableau suivant décrit l'insertion de différents enregistrements sous plusieurs écritures (il est aussi possible d'utiliser des variables de substitution) :

Tableau 6-33 Insertions d'enregistrements

Code PL/SQL	Commentaires

`DECLARE`
`  rty_pilote  Pilote%ROWTYPE;`
`  v_brevet  Pilote.brevet%TYPE;`
`BEGIN`
`  INSERT INTO Pilote VALUES`
`  ('PL-5', 'José Bové', 500, 'AF');`
	Insertion d'un enregistrement dans la table Pilote (toutes les colonnes sont renseignées et les valeurs sont figées).

`v_brevet := 'PL-6';`
`INSERT INTO Pilote VALUES (v_brevet,`
`  'Richard Virenque', 100, 'AF');`
	Insertion d'un enregistrement en utilisant une variable (toutes les colonnes sont renseignées).

`rty_pilote.brevet := 'PL-7';`
`rty_pilote.nom  := 'Serge Miranda';`
`rty_pilote.nbHVol := 1340.90;`
`rty_pilote.compa := 'AF';`
`INSERT INTO Pilote`
`(brevet, nom, nbHVol, compa)`
`VALUES (rty_pilote.brevet,`
`  rty_pilote.nom, rty_pilote.nbHVol,`
`  rty_pilote.compa);`
	Insertion d'un enregistrement en utilisant un ROWTYPE et en spécifiant les colonnes.

Comme sous SQL, il faut respecter les noms, types et domaines de valeurs des colonnes. De même, les contraintes de vérification (CHECK et NOT NULL) et d'intégrité (PRIMARY KEY et FOREIGN KEY) doivent être immédiatement valides (si elles ne sont pas différées).

Dans le cas inverse, une exception qui précise la nature du problème est levée et peut être interceptée dans la section EXCEPTION (voir chapitre suivant). Si une telle partie n'existe pas dans le bloc de code qui contient l'instruction INSERT, la première exception fera s'interrompre le programme.

Modifications

Concernant la mise à jour de colonnes par UPDATE, la clause SET peut être ambiguë dans le sens où l'identificateur à gauche de l'opérateur d'affectation est toujours une colonne de base de données, alors que celui à droite de l'opérateur peut correspondre à une colonne ou une variable.

```
UPDATE nomTable
   SET nomColonne = { nomVariablePLSQL | expression | nomColonne |
   (requête) }
   [,nomColonne2 = … ]
   [WHERE …];
```

Si aucun enregistrement n'est modifié, aucune erreur ne se produit et aucune exception n'est levée (contrairement à l'instruction SELECT).

Un curseur implicite permet de savoir combien d'enregistrements ont été modifiés (voir plus loin SQL%ROWCOUNT).

Les affectations dans le code PL/SQL utilisent obligatoirement l'opérateur := tandis que les comparaisons ou affectations SQL nécessitent l'opérateur =.

Le tableau suivant décrit la modification de différents enregistrements (il est aussi possible d'employer des variables de substitution).

Tableau 6-34 Modifications d'enregistrements

Code PL/SQL	Commentaires

`DECLARE`
`  v_duréeVol NUMBER(3,1) := 4.8;`
`BEGIN`
`  UPDATE  Pilote`
`   SET   nbHVol = nbHVol + v_duréeVol`
`   WHERE brevet= 'PL-6';`
	Modification d'un enregistrement de la table Pilote en utilisant une variable.

`  UPDATE  Pilote`
`   SET   nbHVol= nbHVol + 10`
`   WHERE compa = 'AF';`
`END;`
	Modification de plusieurs enregistrements de la table Pilote en utilisant une constante.

Suppressions

La suppression par DELETE peut être ambiguë (même raison que pour l'instruction UPDATE) au niveau de la clause WHERE.

```
DELETE FROM nomTable
   WHERE nomColonne =
   { nomVariablePLSQL | expression | nomColonne | (requête) }
        [,nomColonne2 = … ] …;
```

Si aucun enregistrement n'est modifié, aucune erreur ne se produit et aucune exception n'est levée.

Un curseur implicite permet de savoir combien d'enregistrements ont été modifiés.

Le tableau suivant décrit la modification de différents enregistrements (il est aussi possible d'utiliser des variables de substitution).

Tableau 6-35 Suppressions d'enregistrements

Code PL/SQL	Commentaires

`DECLARE`
`  v_hVolMini NUMBER(4) := 1000;`
`BEGIN`
`  DELETE FROM Pilote`
`  WHERE nbHVol < v_hVolMini;`
	Supprime les enregistrements de la table `Pilote` dont le nombre d'heures de vol est inférieur à 1 000.

`  DELETE FROM Pilote`
`  WHERE brevet = '$£*';`
`END;`
	Ne supprime aucun enregistrement de la table `Pilote`.

Curseurs implicites

PL/SQL utilise un curseur implicite pour chaque opération du LMD de SQL (`INSERT`, `UPDATE` et `DELETE`). Ce curseur porte le nom `SQL` et il est exploitable après avoir exécuté l'instruction. La commande qui suit le LMD remplace l'ancien curseur par un nouveau.

Il existe aussi le mécanisme des curseurs explicites (auxquels le programmeur affecte un nom) qui servent principalement à parcourir un ensemble d'enregistrements. Nous étudierons ce type de curseurs au chapitre suivant.

Les attributs de curseurs implicites permettent de connaître un certain nombre d'informations qui ont été renvoyées après l'instruction du LMD et qui peuvent être utiles au programmeur. Ces attributs peuvent être employés dans une section de traitement ou d'exception. Les principaux attributs sont les suivants :

Tableau 6-36 Attributs d'un curseur implicite

Attribut	Explication
`SQL%ROWCOUNT`	Nombre de lignes affectées par la dernière instruction LMD.
`SQL%FOUND`	Booléen valant `TRUE` si la dernière instruction LMD affecte au moins un enregistrement.
`SQL%NOTFOUND`	Booléen valant `TRUE` si la dernière instruction LMD n'affecte aucun enregistrement.

Le tableau suivant décrit la suppression de plusieurs données et l'extraction du nombre d'enregistrements supprimés par la commande LMD (ici `DELETE`).

Tableau 6-37 Modifications d'enregistrements

Code PL/SQL	Commentaires

`VARIABLE g_pilotesAFDétruits NUMBER`
`BEGIN`
`  DELETE FROM Pilote WHERE compa = 'AF';`
	Supprime les enregistrements de la table `Pilote` de la compagnie 'AF'.

`  :g_pilotesAFDétruits := SQL%ROWCOUNT;`
`END;`
`/`
`PRINT g_pilotesAFDétruits`
	Initialise puis affiche la variable globale `g_pilotesAFDétruits` qui contient le nombre d'enregistrements supprimés.

Paquetage DBMS_OUTPUT

Nous avons vu qu'il était possible d'afficher sous SQL*Plus des résultats calculés par un bloc PL/SQL avec des variables de session (globales). Une autre possibilité, plus riche, consiste à utiliser des procédures du paquetage `DBMS_OUTPUT`. Ce paquetage assure la gestion des entrées/sorties de blocs ou sous-programmes PL/SQL (fonctions et procédures cataloguées, paquetages ou déclencheurs).

Il existe par ailleurs plus de cent paquetages prédéfinis à certaines tâches. Citons `DBMS_LOCK` pour gérer des verrous, `DBMS_RANDOM` pour générer des nombres aléatoires, `DBMS_ROWID` pour manipuler des *rowids*, `DBMS_SQL` pour construire statiquement ou dynamiquement des ordres SQL.

Le tableau suivant décrit les procédures du paquetage `DBMS_OUTPUT`. Au niveau des paramètres, la directive `IN` désigne un paramètre d'entrée alors que `OUT` en désigne un en sortie. La procédure que vous utiliserez le plus est probablement `PUT_LINE` (équivalent du `println` Java) ; elle vous aidera à déboguer vos programmes.

Tableau 6-38 Procédures disponibles de DBMS_OUTPUT

Procédure	Explication
`ENABLE (taille_tampon IN INTEGER DEFAULT 2000)`	Activation du paquetage dans la session.
`DISABLE`	Désactivation du paquetage dans la session.
`PUT(ligne IN VARCHAR2 | DATE | NUMBER);`	Mise dans le tampon d'un paramètre.
`NEW_LINE(ligne OUT VARCHAR2, statut OUT INTEGER)`	Écriture du caractère fin de ligne dans le tampon.
`PUT_LINE(ligne IN VARCHAR2 | DATE NUMBER);`	`PUT` puis `NEW_LINE`.
`GET_LINE(ligne OUT VARCHAR2, statut OUT INTEGER)`	Affectation d'une chaîne du tampon dans une variable.
`GET_LINES(tab OUT DBMS_OUTPUT.CHARARR, nombreLignes IN OUT INTEGER);` `CHARARR table de VARCHAR2(255)`	Affectation de chaînes du tampon dans un tableau.

Sans parler de sorties sur l'écran, les procédures PUT et PUT_LINE disposent dans un tampon des informations qui peuvent être lues par d'autres bloc, déclencheur, procédure, fonction ou paquetage par les procédures GET_LINE ou GET_LINES.

Au niveau de l'interface SQL*Plus, le paquetage doit être activé au préalable dans la session avec la commande SQL*Plus SET SERVEROUTPUT ON. Une fois exécutée, cette option reste valable durant toute la session SQL*Plus.

L'appel de toute procédure d'un paquetage se réalise avec l'instruction nomPaquetage.nomProcédure(paramètres). Dans notre exemple, l'appel de la procédure PUT_LINE s'écrira donc DBMS_OUTPUT.PUT_LINE(*texte*).

Gestion des sorties (PUT_LINE)

Le tableau suivant décrit l'affichage de différentes variables :

Tableau 6-39 Affichage de résultats

Code PL/SQL	Commentaires

`SET SERVEROUTPUT ON`
`DECLARE`
`  v_nbrPil NUMBER;`
`BEGIN`
`  DBMS_OUTPUT.ENABLE;`
`  DBMS_OUTPUT.PUT_LINE('Nous sommes le  : ' || SYSDATE);`
`  DBMS_OUTPUT.PUT_LINE('La racine de 2 = ' || SQRT(2));`
`  SELECT COUNT(*) INTO v_nbrPil FROM Pilote;`
`  DBMS_OUTPUT.PUT_LINE`
`     ('Il y a ' || v_nbrPil || ' pilotes dans la table');`
`END;`
	Activation du paquetage sous SQL*Plus.
	
	Activation du paquetage sous PL/SQL.
	Affichage.

`Nous sommes le  : 14/07/03`
`La racine de 2 = 1,414213562373095048801688724209698078 57`
`Il y a 4 pilotes dans la table`
 	
`Procédure PL/SQL terminée avec succès.`
	Résultats.

Gestion des entrées (GET_LINE et GET_LINES)

Il est possible d'extraire une ou plusieurs lignes à partir du tampon (*buffer*). La procédure GET_LINE permet d'en retirer une seule (de type VARCHAR2(255)). Cette ligne est la première qui a été mise dans le tampon.

L'exemple suivant illustre un appel de GET_LINE(*ligne* OUT VARCHAR2, *statut* OUT INTEGER). Si l'exécution est correcte, le paramètre *statut* reçoit 0. S'il n'y a plus de lignes dans le tampon, le paramètre *statut* reçoit 1.

Tableau 6-40 Utilisation de GET_LINE

Code PL/SQL	Commentaires
DECLARE v_nbrPil NUMBER; v_ligne VARCHAR2(255); v_résultat INTEGER; BEGIN SELECT COUNT(*) INTO v_nbrPil FROM Pilote; DBMS_OUTPUT.PUT_LINE('Première ligne'); DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbrPil || ' pilotes'); DBMS_OUTPUT.**GET_LINE**(v_ligne, v_résultat); END;	Deux lignes sont mises dans le tampon. GET_LINE dépile la ligne du tampon.
Résultat dans v_ligne 'Première ligne', v_résultat 0	

La procédure GET_LINES(*tab* OUT DBMS_OUTPUT.CHARARR, *nombreLignes* IN OUT INTEGER) permet d'extraire plusieurs lignes vers le tableau tab. Le deuxième paramètre indique le nombre de lignes à retirer. Ces lignes sont les premières à y être mises.

L'exemple suivant illustre un appel de GET_LINES. Ici nous extrayons les trois premières lignes du tampon dans le tableau tab.

Tableau 6-41 Utilisation de GET_LINES

Code PL/SQL	Commentaires
DECLARE tab DBMS_OUTPUT.**CHARARR**; v_resultat INTEGER := 3; v_nbrPil NUMBER; BEGIN SELECT COUNT(*) INTO v_nbrPil FROM Pilote; DBMS_OUTPUT.PUT_LINE('Première ligne'); DBMS_OUTPUT.PUT_LINE('Deuxième ligne'); DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbrPil || ' pilotes'); DBMS_OUTPUT.PUT_LINE('Quatrième ligne'); DBMS_OUTPUT.**GET_LINES**(tab, v_resultat);	Quatre lignes sont mises dans le tampon. GET_LINES dépile trois lignes du tampon.
Résultat tab Première ligne Deuxième ligne Il y a 4 pilotes	

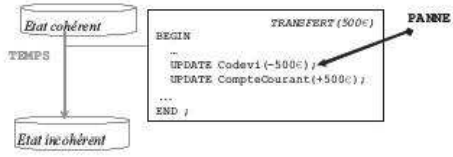
Transactions

Au sens SGBD du terme, une transaction est un bloc d'instructions LMD faisant passer la base de données d'un état initial (cohérent) à un état intermédiaire ou final cohérent. Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions de la transaction n'est réellement effectuée, quel que soit l'endroit de la transaction où intervient l'erreur. En invalidant toutes les opérations depuis le début de la transaction, la base retourne à un état initial cohérent (principe du tout ou rien).

Pour réaliser cela, Oracle dipose de plusieurs mécanismes : le segment d'annulation (*undo segment*) qui contient les blocs modifiés mais pas validés, les journaux de transactions (*redo log*) qui inscrivent les transactions validées, et le mécanisme de verrouillage qui assure que deux transactions ne modifient pas la même donnée sans donner la priorité à l'une ou l'autre. Le segment d'annulation rend possible l'isolation des données au cours de la transaction. Les langages plus évolués permettent également de programmer des transactions à travers leur API (`COMMIT` est implémenté dans le paquetage `java.sql`, par exemple).

Un exemple typique de transaction consiste au transfert d'une somme d'un compte épargne vers un compte courant. Imaginez qu'après une panne (logicielle ou matérielle) votre compte épargne ait été débité sans que votre compte courant soit crédité du même montant ! Vous ne seriez pas très content des services de votre banque (à moins que l'erreur ne soit intervenue dans l'autre sens !). La réservation d'une place au théatre ne permet pas non plus que plusieurs personnes partagent le même siège (il est vrai que l'exemple est mal choisi car le surbooking permet précisément de vendre le même siège à plusieurs personnes sachant qu'une seule en bénéficiera). Le mécanisme transactionnel empêche tout scénario fâcheux.

Figure 6-4 *Risque de procédure fâcheuse*



Caractéristiques

Une transaction assure les mécanismes ACID.

- Atomicité des instructions qui sont considérées comme une seule opération (principe du tout ou rien).
- Cohérence (passage d'un état de la base cohérent à un autre état cohérent).

- Isolation des transactions entre elles (lecture consistante).
- Durabilité (les transactions attendent tant que nécessaire pour assurer la cohérence de l'ensemble).

Une extraction (SELECT) génère un verrou partagé (S) sur tout ou partie de la table. Si une écriture (UPDATE ou MERGE) concerne cette partie de table, un verrou exlusif (X) est posé, et il sera impossible d'obtenir le verrou (S) tant que la modification n'est pas validée. S'il s'agissait d'une autre lecture, la requête initiale pourra s'exécuter. L'idée de base est qu'une lecture ne doit pas en bloquer une autre, mais qu'une écriture peut bloquer une autre écriture (ou lecture), et qu'une lecture peut bloquer une écriture.

Début et fin d'une transaction

Il existe deux modes de gestion des transactions depuis la version SQL:1999 de la norme.

- Le mode implicite dans lequel la connexion à la base démarre une transaction que l'on doit finaliser par une validation ou annulation (COMMIT ou ROLLBACK) et qui redémarre une nouvelle transaction après finalisation.
- Le mode explicite qui implique de spécifier quand démarre la transaction (éventuellement avec BEGIN) et quand l'arrêter (avec COMMIT ou ROLLBACK).

Toute transaction se termine en échec à la fin anormale d'une session (forcée ou anomalie logicielle ou matérielle).

Oracle ne fonctionne pas nativement en mode *autocommit*, c'est-à-dire que chaque instruction SQL du LMD doit être explicitement validée sinon elle risque d'être perdue.

Tout ordre SQL du *Data Definition Language* (CREATE, ALTER, DROP, COMMENT, RENAME, TRUNCATE, GRANT et REVOKE) génère une fin normale de la transaction en cours. En d'autres termes, si vous avez modifié des données et que vous décidez de créer une table, vos mises à jour seront validées.

Si vous désirez maîtriser votre code et vous prémunir des incohérences dues à des accès concurrents, vous devez programmer vos transactions explicitement à l'aide des primitives décrites au tableau 6-42. Si vous voulez monitorer de longues transactions, il est préférable de les nommer.

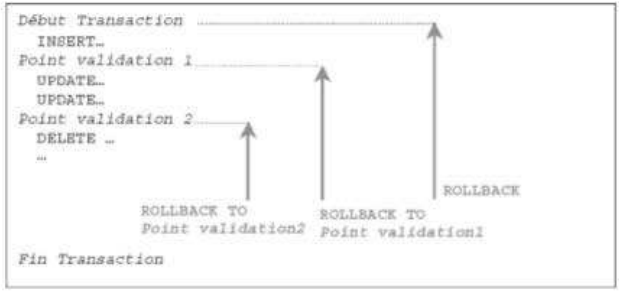
Tableau 6-42 Instructions de gestion des transactions

Instructions	Commentaires
`SET TRANSACTION NAME 'transaction_nom'; [BEGIN]`	Début de la transaction (nommée ou pas).
`COMMIT [WORK];`	Termine avec succès la transaction (validation). Libération des verrous.
`ROLLBACK [WORK];`	Termine avec échec la transaction (invalidation). Libération des verrous.
`SAVEPOINT nom_savepoint;`	Déclare un point de validation au cours de la transaction.
`ROLLBACK TO nom_savepoint;`	Invalide les instructions réalisées depuis le point de validation.

Contrôle des transactions

Il est intéressant de pouvoir découper une transaction en insérant des points de validation (*savepoints*) qui permettent d'annuler tout ou partie de la transaction. La figure 6-5 illustre une transaction découpée en trois parties. L'annulation centrale permettra d'invalider les modifications (UPDATE et DELETE) tout en laissant la possibilité de valider l'instruction INSERT (si un COMMIT suit).

Figure 6-5 *Points de validation*



Niveaux d'isolation

La concurrence des accès aux données induit des problèmes inévitables. Prises isolément et exécutées les unes après les autres, un ensemble de transactions modifiant des données en commun ne générera aucune incohérence. Le problème est que cet ensemble de transactions est susceptible de s'exécuter simultanément. La mise en place d'un niveau d'isolation pour

chaque transaction permet de gérer au mieux cet état de fait. Chaque niveau permet de résoudre un type d'anomalie. Trois types d'anomalies classiques sont recensés :

- la lecture sale de données (*dirty reads*) qui se produit lorsqu'une transaction accède à des données qui sont modifiées par une autre transaction et qui n'ont pas été encore validées ;
- la lecture non répétable de données (*non repeatable reads*) qui se produit quand deux lectures successives d'une même donnée au sein d'une transaction ne génère pas le même résultat parce qu'une autre transaction a modifié les données déjà lues entre temps ;
- la lecture de données fantômes (*phantom reads*) qui intervient lorsque de nouvelles données apparaissent au cours de lectures successives (insertion de données effectuées par une autre transaction).

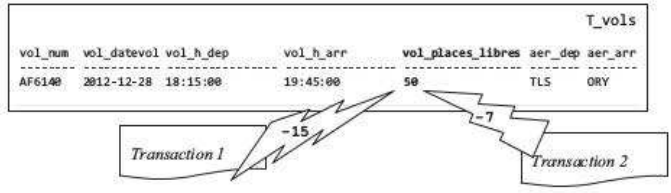
Il est possible de changer de niveau d'isolation en cours de traitement, y compris pendant la transaction, mais cela ne constitue pas généralement une bonne pratique. Selon le mode choisi, vous disposerez des fonctionnalités suivantes.

Tableau 6-43 Types de transactions (norme SQL)

Niveau d'isolation	Lectures sales	Lectures non répétables	Lectures fantômes
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Impossible	Possible	Possible
`REPEATABLE READ`	Impossible	Impossible	Possible
`SERIALIZABLE`	Impossible	Impossible	Impossible

L'exemple suivant correspond à la réservation de places pour un vol entre Toulouse et Paris. Supposons qu'il reste 50 places et que deux transactions tentent de réserver simultanément 7 places pour l'une et 15 places pour l'autre. L'état idéal serait que ces deux transactions laissent au final 28 places disponibles pour le vol…

Figure 6-6 *Modification concurrente*



La mise en place du niveau de transaction s'opère au niveau par l'instruction `SET TRANSACTION` suivante :

```
SET TRANSACTION [ READ ONLY | READ WRITE ]
                [ ISOLATION LEVEL [ SERIALIZE | READ COMMITED ]
                [ USE ROLLBACK SEGMENT 'segment_nom' ]
                [ NAME 'transaction_nom' ];
```

Le niveau d'isolation s'applique à la transaction concernée et vous ne devez pas raisonner par rapport aux autres transactions (dont vous ne pouvez maîtriser le mode). Oracle ne fournit que les niveaux `READ COMMITTED` et `SERIALIZABLE` (le mode le plus fort).

Par défaut, le mode `READ COMMITTED` est adopté ; il est le plus polyvalent et répond à la plupart des cas. Dans les deux modes, chaque requête ne verra que des données validées (par les autres transactions) avant la requête (et pas avant la transaction).

Le niveau `READ COMMITTED` repose sur un contrôle plutôt « optimiste » de la concurrence alors que le mode `SERIALIZABLE` est davantage « pessimiste » du fait qu'aucune modification, par une transaction extérieure, n'est possible sur des données mises à jour par la transaction et depuis son début (l'erreur détectée est `ORA-08177: impossible de sérialiser l'accès pour cette transaction`). Ce dernier mode implémente l'illusion pour une transaction qu'aucune autre session ne modifie les données. Le mode `SERIALIZABLE` convient aux transactions courtes modifiant peu de lignes et où les modifications concurrentes sont rares.

Il va de soi que, quel que soit le mode, chaque requête voit ses propres modifications même si la validation n'est pas encore réalisée. Par ailleurs, la bête noire des deux modes sont les modifications concurrentes (surtout en mode `SERIALIZABLE`).

Enfin, un mode en lecture seule est également fournit (`READ ONLY`) ; il équivaut à `SERIALIZABLE` qui n'opère aucune modification.

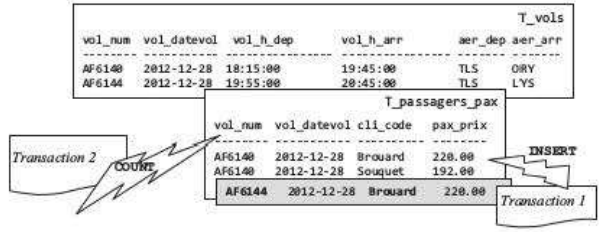
Au niveau d'une session, il est également prévu de définir le niveau d'isolation qui concernera toutes les transactions qui s'y déroulent (`ALTER SESSION SET ISOLATION_LEVEL ...`).

Lecture fantôme

La figure 6-7 présente une lecture fantôme. La transaction 1 compte le nombre de vols d'un passager et, dans l'intervalle, la transaction 2 ajoute un vol au passager en question. Quand la transaction 1 débute avant la transaction 2 et se prolonge dans le temps, le nouveau vol peut apparaître au cours du traitement ce qui ne reflètait pas l'état de la base au début de la transaction de lecture.

En se plaçant au niveau d'isolation `READ COMMITTED`, cette lecture fantôme apparaît. Le mode `SERIALIZABLE` évitera ce comportement.

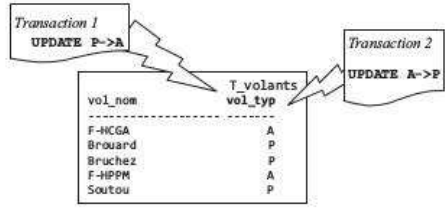
Figure 6-7 Lecture non consistante entre deux transactions en action



D'autres niveaux d'isolation

Pourquoi faut-il sérialiser les transactions au lieu d'autoriser un mode entrelacé ? En s'inspirant de l'exemple de Jim Gray (chercheur au sein de Microsoft et récompensé du prix Turing en 1998), imaginons deux transactions : l'une changeant les pilotes en avions et l'autre les avions en pilotes.

Figure 6-8 Lectures de clichés



En mode `SERIALIZABLE`, l'exécution revient à exécuter l'une, puis l'autre indifférement. À la fin, le monde ne sera peuplé que de pilotes seuls ou d'avions seuls (tout le monde reste au sol donc). En mode *snapshot* (proposé avec SQL Server), il est possible que les mises à jour soient opérées simultanément. En conséquence, les pilotes sont devenus des avions et inversement (les voyages peuvent reprendre en théorie ; c'est ça le pur optimisme). Le mode `SERIALIZABLE` permet de garder les pieds sur terre mais peut induire (comme le mode `READ COMMITED` d'ailleurs) une étreinte fatale (*deadlock*) dès lors qu'une transaction doit attendre indéfiniment que l'autre ne relâche ses verrous (entrelacement des modifications).

Le problème du verrou mortel (deadlock)

Le phénomène de *deadlock*, aussi appelé « étreinte fatale » se produit lorsque deux transactions qui ont posé des verrous sur des objets distincts tentent d'acquérir un nouveau verrou sur

un objet déjà verrouillé par l'autre transaction. En plus du fait que les verrous mortels nécessitent d'être gérés comme des exceptions, ils sont gourmands en ressources CPU.

Le tableau 6-44 illustre deux transactions en interblocage (et ce, quel que soit le niveau d'isolation). En supposant que la transaction 1 (*t1*) démarre avant la transaction 2 (*t2*), le vol AF6140 est d'abord verrouillé par *t1* jusqu'à la validation, ce qui n'empêche pas le verrouillage du vol AF6144 par *t2*. Par la suite, *t1* pose un verrou sur le vol AF6144 qui sera relâché à la fin de *t2*, qui a posé un verrou sur le vol AF6140 qui sera quant à lui relâché à la fin de *t1*. Tout le monde attend ainsi l'autre.

Tableau 6-44 Transactions en interblocage

Transaction 1	Transaction 2

`BEGIN`
`UPDATE T_vols`
`SET vol_places_libres=vol_places_libres-7`
`WHERE vol_num = 'AF6140'`
`AND vol_datevol = TO_DATE('20121228', 'YYYYMMDD');`
	`BEGIN`
	`UPDATE T_vols`
	`SET vol_places_libres=vol_places_libres-15`
	`WHERE vol_num = 'AF6144'`
	`AND vol_datevol = TO_DATE('20121228', 'YYYYMMDD');`

`-- pendant ce temps, d'autres réservations arrivent DBMS_LOCK.SLEEP(10);`	`DBMS_LOCK.SLEEP (10);`

`UPDATE T_vols`
`SET vol_places_libres=vol_places_libres-4`
`WHERE vol_num = 'AF6144'`
`AND vol_datevol = TO_DATE('20121228', 'YYYYMMDD');`
`…`
	`UPDATE T_vols`
	`SET vol_places_libres=vol_places_libres-5`
	`WHERE vol_num = 'AF6140'`
	`AND vol_datevol = TO_DATE('20121228', 'YYYYMMDD');`
	`…`

Quand un tel phénomène est identifié, il est mis fin à la transaction la moins coûteuse en ressources (équivaut à `ROLLBACK`). Le message `ORA-00060: détection d'interblocage pendant l'attente d'une ressource` est retourné au client malheureux tandis que l'autre transaction continue.

Adoptez les règles suivantes pour limiter les risques de verrous mortels :

- normalisez correctement le modèle de données et soignez l'indexation des tables ;
- réduisez la durée de votre code en limitant les entrées-sorties, extrayant les données (peut-être en utilisant des tableaux associatifs) en un minimum de lecture, en verrouillant au plus tard et libérant les verrous au plus tôt ;
- limitez l'escalade potentielle de verrous en gérant manuellement des verrous si cela est approprié.

Verrouillage manuel

La gestion manuelle des verrous est possible mais plus complexe à mettre en œuvre et à maintenir. Elle peut poser plus de problèmes qu'elle n'en résoud.

- Le moniteur de verrouillage peut faire de l'escalade afin de minimiser les ressources (passer d'un verrou de ligne à un verrou de page ou de table, par exemple, lorsque différentes lignes sont mises à jour dans une même transaction).
- Le verrouillage manuel implique que le verrou devient statique, ce qui peut se révéler inadéquat en fonction de la charge (modification de la volumétrie ou de la concurrence) ou d'éventuelles modifications de la structure logique ou physique des données (création d'index, par exemple).
- Oracle ne garantit pas l'application du verrou posé dans certaines circonstances.
- Le fait de verrouiller manuellement ne permet pas de profiter des éventuelles évolutions et peut s'avérer non portable d'une version à l'autre.

Pour effectuer un verrouillage manuel, une première possibilité utilise `LOCK TABLE` qui prévient des accès concurrents au niveau de la table (ou des tables associées si la commande concerne une vue). Une autre possibilité, plus fine, consiste à verrouiller seulement une partie des lignes avec l'option `FOR UPDATE` de la commande `SELECT` (voir au chapitre 7 un exemple avec les curseurs). Ces verrous seront libérés à la fin de la transaction (suite à un `COMMIT`, `ROLLBACK` ou une interruption involontaire).

Si vous désirez verrouiller une table, vous disposez des options suivantes.

```
LOCK TABLE [schema.] {table | vue}
  IN {ROW SHARE | SHARE UPDATE | ROW EXCLUSIVE |SHARE SHARE |
      SHARE ROW EXCLUSIVE | EXCLUSIVE } MODE
  [ NOWAIT | WAIT secondes];
```

- `SHARE UPDATE` (synonyme de `ROW SHARE`, ancienne dénomination) autorise les accès concurrents mais interdit aux autres transactions de verrouiller exclusivement.
- `ROW EXCLUSIVE` se comporte comme `SHARE UPDATE` en interdisant également le verrouillage partagé (ce sont ces types de verrous qui se posent automatiquement lors de mises à jour SQL).
- `SHARE SHARE` autorise les lectures concurrentes mais interdit les mises à jour de la table.
- `SHARE ROW EXCLUSIVE` autorise les lectures concurrentes mais interdit le verrouillage partagé ou la mise à jour.
- `EXCLUSIVE` ne permet que les lectures concurrentes.

L'interblocage présenté précédemment serait résolu, par exemple, en disposant au début de chaque transaction :

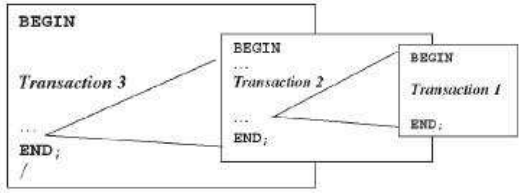
- soit un verrouillage au niveau de la table `LOCK TABLE T_vols IN SHARE ROW EXCLUSIVE MODE WAIT 5 ;`

- soit un verrouillage au niveau ligne par `SELECT... INTO ... FROM T_vols WHERE vol_num=... AND vol_datevol=... FOR UPDATE OF vol_places_libres WAIT 5` en sélectionnant en préventif le deuxième vol à mettre à jour.

Transactions imbriquées

Il est possible de programmer plusieurs transactions se déroulant dans des blocs imbriqués comme l'illustre la figure 6-9. Les mécanismes d'atomicité, de cohérence, d'isolation et de durabilité seront également respectés.

Figure 6-9 *Transactions imbriquées*



Où placer les transactions ?

L'idée de manipuler des transactions depuis un code client (VB, Delphi, Java, C++...) est séduisante mais peut entraîner un blocage du serveur du fait d'une non-libération des verrous (si le client perd la connexion sans validation ou invalidation). Un autre problème concerne les entrées-sorties qui peuvent devenir également bloquantes. La logique transactionnelle doit donc se trouver au plus près du serveur et qui mieux que les procédures cataloguées (voir chapitre 7) peuvent implémenter les transactions ?

Il est aussi possible d'utiliser des objets métier dédiés (EJB, par exemple, dans une architecture J2EE), mais ces derniers posent à nouveau des problématiques de *round-trips* (latence du fait du réseau séparant les objets du serveur) pendant lequels les tables peuvent être bloquées.

D'une manière analogue, l'utilisation massive des ORM peut être très nocive d'un point de vue transactionnel du fait de l'empilement des couches logicielles qui masquent ou empêchent d'utiliser les fonctionnalités natives du SGBD qui ont fait leurs preuves depuis de nombreuses années. Si les ORM font gagner du temps lors du développement, le bénéfice en termes de performances n'est pas souvent équivalent.

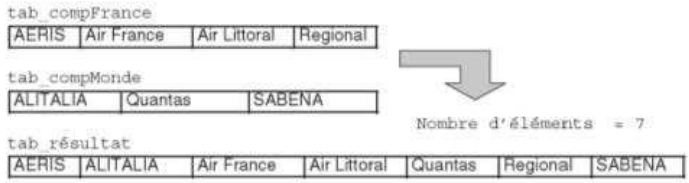
Exercices

L'objectif de ces exercices est d'écrire des blocs PL/SQL puis des transactions PL/SQL manipulant des tables du schéma *Parc Informatique*.

Exercice 6.1 Tableaux et structures de contrôle

Écrivez le bloc PL/SQL qui programme la fusion de deux tableaux (déjà triés par ordre croissant) en un seul (utiliser des structures `WHILE...`). Il faudra afficher ce nouveau tableau (utiliser une structure `FOR...`) ainsi que le nombre d'éléments de ce dernier.

Figure 6-10 *Fusion de deux tableaux*



Exercice 6.2 Bloc PL/SQL et variables %TYPE

Écrivez le bloc PL/SQL qui affiche, à l'aide du paquetage `DBMS_OUTPUT`, les détails de la dernière installation sous la forme suivante :

```
Dernière installation en salle : numérodeSalle
-----------------------------------------------
Poste : numérodePoste Logiciel : nomduLogiciel en date du dateInstallation
```

Vous utiliserez les directives `%TYPE` pour extraire directement les types des colonnes et pour améliorer ainsi la maintenance du bloc.

Ne tenez pas compte, pour le moment, des erreurs qui pourraient éventuellement se produire (aucune installation de logiciel, poste ou logiciel non référencés dans la base, etc.).

Exercice 6.3 Variables de substitution et globales

Écrivez le bloc PL/SQL qui saisit un numéro de salle et un type de poste, et qui retourne un message indiquant les nombres de postes et d'installations de logiciels correspondantes sous la forme suivante :

```
Numéro de Salle : numérodeSalle
Type de poste : typedePoste
```

```
G_NBPOSTE
----------
nombredePostes

G_NBINSTALL
-----------
nombred'installations
```

Vous utiliserez des variables de substitution pour la saisie et des variables globales pour les résultats. Vous exécuterez le bloc à l'aide de la commande `start` et non pas par copier-coller (à cause des ordres `ACCEPT`). Ne tenez pas compte pour le moment d'éventuelles erreurs (aucun poste trouvé ou aucune installation réalisée, etc.).

Exercice 6.4 Transaction

Écrivez une transaction permettant d'insérer un nouveau logiciel dans la base après avoir saisi toutes ses caractéristiques (numéro, nom, version et type du logiciel). La date d'achat doit être celle du jour. Tracer avec `PUT_LINE` l'insertion du logiciel (message `Logiciel` inséré dans la base).

Il faut ensuite procéder à l'installation de ce logiciel sur le poste de numéro 'p7' (utiliser une variable pour pouvoir plus facilement modifier ce paramètre). L'installation doit se faire à la date du jour.

Pensez à actualiser correctement la colonne `delai` qui mesure le délai (`INTERVAL`) entre l'achat et l'installation. Pour ne pas que ce délai soit nul (les deux insertions se font dans la même seconde dans cette transaction), placer une attente de 5 secondes entre les insertions avec l'instruction `DBMS_LOCK.SLEEP(5);`. Utiliser la fonction `NUMTODSINTERVAL` pour calculer ce délai. Tracer avec `PUT_LINE` l'insertion de l'installation.

La trace suivante donne un exemple de ce que vous devez produire (les champs en gras sont ceux saisis) :

```
SQL> START exo3plsql
Numéro de logiciel : log15
Nom du logiciel    : Oracle Web Agent
Version du logiciel : 15.5
Type du logiciel    : Unix
Prix du logiciel (en euros) : 1500
Logiciel inséré dans la base
Date achat : 17-07-2003 13:48:08
Date installation : 17-07-2003 13:48:13
Logiciel installé sur le poste

Procédure PL/SQL terminée avec succès.
```

Attente de 5 secondes à ce niveau

Vérifiez l'état des tables mises à jour après la transaction. Ne tenez pas compte pour le moment d'éventuelles erreurs (numéro du logiciel déjà référencé, type du logiciel incorrect, installation déjà réalisée, etc.).

Chapitre 7

Programmation avancée

Sous-programmes

Les sous-programmes sont des blocs PL/SQL nommés et capables d'inclure des paramètres en entrée et en sortie. Il existe deux types de sous-programmes PL/SQL qui sont les procédures et les fonctions. Comme dans tous les langages de programmation, les procédures réalisent des actions alors que les fonctions retournent un unique résultat. Seule la procédure peut avoir plusieurs paramètres en sortie.

Les sous-programmes sont en général écrits en PL/SQL (ce chapitre leur est consacré), mais ils peuvent être codés en Java (voir chapitre 11) ou en C.

Généralités

Dans le vocabulaire des bases de données, on appelle les sous-programmes « fonctions » ou « procédures cataloguées » (ou stockées), car ils sont compilés et résident dans la base de données. Il est possible de retrouver leur code au niveau du dictionnaire des données. Le sous-programme peut être ainsi partagé dans un contexte multi-utilisateur.

Lors d'un appel d'une fonction ou d'une procédure, le noyau recompile le programme si un objet cité dans le code a été modifié (ajout d'une colonne dans une table, modification de la taille d'une colonne…) et le charge en mémoire.

Les avantages des sous-programmes catalogués sont nombreux :

- sécurité : les droits d'accès ne portent plus sur des objets (table, vue, variable…) mais sur des programmes stockés. Ces droits sont délégués (`GRANT EXECUTE ON NOMPROCÉDURE TO UTILISATEUR`) ;
- intégrité : les traitements dépendants sont exécutés dans le même bloc (transactions) ;
- performance : réduction du nombre d'appels à la base (utilisation d'un programme partagé) ;
- productivité : simplicité de la maintenance des programmes (modularité, extensibilité, réutilisabilité) notamment par l'utilisation de paquetages.

Comme les blocs PL/SQL, nous verrons que les sous-programmes ont une partie de déclaration des variables, une autre contenant les instructions et éventuellement une dernière pour gérer les exceptions (erreurs produites durant l'exécution).

Une procédure, comme une fonction, peut être appelée à l'aide de l'interface de commande SQL*Plus (commande EXECUTE) ou par l'intermédiaire d'un outil d'Oracle (*Forms* par exemple), dans un programme externe (Java, C...), par d'autres procédures ou fonctions ou dans le corps d'un déclencheur. Les fonctions peuvent être appelées dans une instruction SQL (SELECT, INSERT, et UPDATE).

Le cycle de vie d'un sous-programme est le suivant : création de la procédure ou fonction (compilation et stockage dans la base), appels et éventuellement suppression du sous-programme de la base. Il est à noter qu'un sous-programme se recompile automatiquement dès que la structure d'un objet qu'il manipule est modifiée (tables, vues, séquences, index...). Dans certains cas de dépendances indirectes, il est prévu de pouvoir recompiler manuellement un sous-programme (ALTER PROCEDURE | FUNCTION ... COMPILE).

Procédures cataloguées

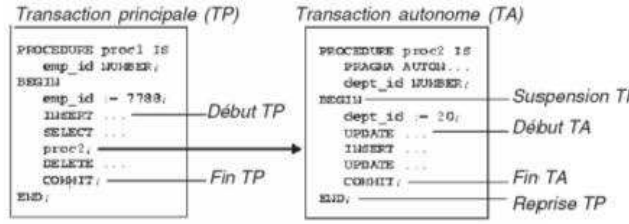
La syntaxe de création d'une procédure cataloguée est la suivante. Pour créer une procédure dans son propre schéma, le privilège CREATE PROCEDURE est requis (inclus dans le rôle RESOURCE). Pour créer une procédure dans un autre schéma, il faut posséder le privilège CREATE ANY PROCEDURE.

```
CREATE [OR REPLACE] PROCEDURE [schéma.]nomProcédure
          [(paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
                [{:= | DEFAULT} expression]
          [,paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
                [{:= | DEFAULT} expression]... ) ] ]
 [AUTHID { CURRENT_USER | DEFINER }]
 { IS | AS }
[PRAGMA AUTONOMOUS_TRANSACTION;]
    { corpsduSousProgrammePL/SQL | LANGUAGE {
      JAVA NAME 'nomMéthodeJava' |
      C [NAME nomSourceC] LIBRARY nomLibrairie [AGENT IN (paramètre)]
        [WITH CONTEXT] [PARAMETERS ( paramètres )] } };
```

- IN désigne un paramètre d'entrée, out un paramètre de sortie et in out un paramètre d'entrée et de sortie. Il est possible d'initialiser chaque paramètre par une valeur.
- NOCOPY permet de transmettre directement le paramètre. On l'utilise pour améliorer les performances lors du passage de volumineux paramètres de sortie comme les record, les tables index-by (les paramètres IN sont toujours passés en NOCOPY).

- La clause AUTHID détermine si la procédure s'exécute avec les privilèges de son propriétaire (option par défaut, on parle de *definer-rights procedure*) ou de l'utilisateur courant (on parle de *invoker-rights procedure*).
- PRAGMA AUTONOMOUS_TRANSACTION déclare le sous-programme en tant que transaction autonome (lancée par une autre transaction dite « principale »). Les transactions autonomes permettent de mettre en suspens la transaction principale puis de reprendre la transaction principale (voir la figure suivante).

Figure 7-1 *Transaction autonome*



- *corpsduSousProgrammePL/SQL* contient la déclaration et les instructions de la procédure, toutes deux écrites en PL/SQL.
- JAVA NAME '*nomMéthodeJava*', désignation de la méthode Java correspondante (voir chapitre 11).
- C [NAME *nomSourceC*]..., désignation du programme C correspondant (voir chapitre 8).

Fonctions cataloguées

La syntaxe de création d'une fonction cataloguée est CREATE FUNCTION. Les prérogatives et les options sont les mêmes que pour les procédures.

```
CREATE [OR REPLACE ] FUNCTION [schéma.]nomFonction
          [(paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
                [{:= | DEFAULT} expression]
          [,paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
                [{:= | DEFAULT} expression]... ) ] ]
 RETURN typeSQL
   [ AUTHID { DEFINER | CURRENT_USER } ]
   {IS | AS}
```

```
{ corpsduSousProgrammePL/SQL                    |
  LANGUAGE {
  JAVA NAME 'nomMéthodeJava' |
  C [NAME nomSourceC] LIBRARY nomLibrairie [AGENT IN (paramètre)]
    [WITH CONTEXT] [PARAMETERS ( paramètres )] } };
```

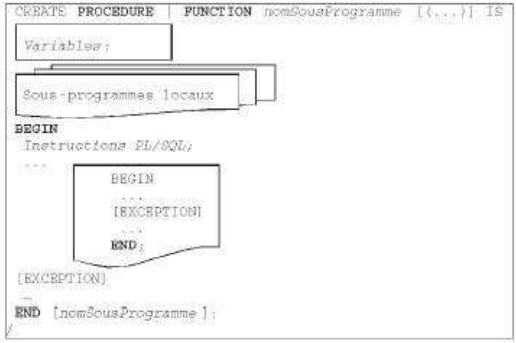
- *corpsduSousProgrammePL/SQL* contient la déclaration et les instructions de la fonction (il doit se trouver une instruction RETURN dans le code), toutes deux écrites en PL/SQL.

Codage d'un sous-programme PL/SQL

Dans une procédure, comme dans une fonction, il n'existe pas de section DECLARE ; les déclarations des variables, curseurs et exceptions suivent directement l'en-tête du programme (après la directive IS ou AS). Nous verrons aussi qu'il est possible de définir un sous-programme dans la section de déclaration d'un autre sous-programme. La figure suivante illustre la structure d'une spécification et d'un corps d'un sous-programme PL/SQL.

Le bloc d'instructions doit contenir au moins une instruction PL/SQL (si vous désirez ne pas en définir une utilisez l'instruction NULL;).

Figure 7-2 *Structure d'un sous-programme*



Exemples

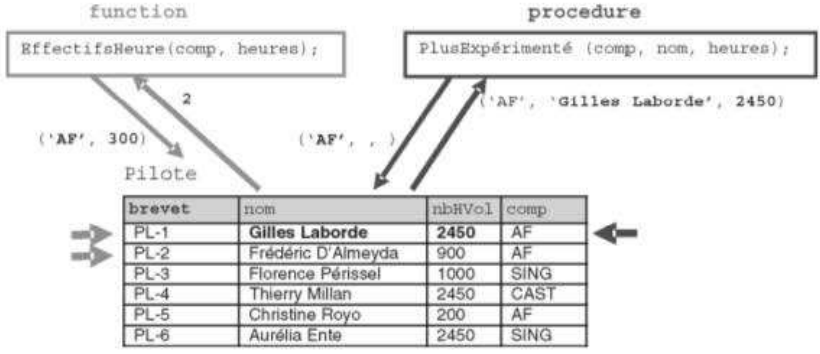
Considérons la table Pilote. Nous allons écrire les sous-programmes suivants :

- la fonction EffectifsHeure(comp,heures) retourne le nombre de pilotes d'une compagnie donnée (premier paramètre) qui ont plus d'heures de vol que la valeur du deuxième paramètre (si aucun pilote, retourne 0). Si aucune compagnie n'est passée en

paramètre (mettre NULL), le calcul inclut toutes les compagnies. Les éventuelles erreurs ne sont pas encore traitées (compagnie de code inexistant par exemple).

- La procédure PlusExpérimenté(comp,nom,heures) retourne le nom et le nombre d'heures de vol du pilote (par l'intermédiaire des deuxième et troisième paramètres) le plus expérimenté d'une compagnie donnée (premier paramètre). Si plusieurs pilotes ont la même expérience, un message d'erreur est affiché. Si aucune compagnie n'est passée en paramètre (mettre NULL), la procédure retourne le nom du plus expérimenté et le code de sa compagnie (par l'intermédiaire du premier paramètre).

Figure 7-3 *Fonction et procédure*



La création de la fonction est réalisée à l'aide du script suivant (EffectifsHeure.sql). Notez les deux paramètres d'entrée définis par la directive IN et la clause RETURN en fin de codage.

```
CREATE OR REPLACE FUNCTION
   EffectifsHeure(pcomp IN VARCHAR2, pheuresVol IN NUMBER) RETURN
   NUMBER
IS
résultat NUMBER := 0;
BEGIN
   IF (pcomp IS NULL) THEN
      SELECT COUNT(*) INTO résultat FROM Pilote
      WHERE    nbHVol >  pheuresVol ;
   ELSE
      SELECT COUNT(*) INTO résultat FROM Pilote
      WHERE    nbHVol >  pheuresVol
```

```
      AND      comp = pcomp;
   END IF;
   RETURN résultat;
END EffectifsHeure  ;
```

La création de la procédure est réalisée à l'aide du script suivant (`PlusExpérimenté.sql`). Notez les deux derniers paramètres de sortie définis par la directive `OUT` et le premier servant d'entrée ou de sortie avec la directive `IN OUT`.

```
CREATE OR REPLACE PROCEDURE PlusExpérimenté
   (pcomp IN OUT VARCHAR2, pnomPil OUT VARCHAR2, pheuresVol OUT NUM-
   BER)
IS
   p1 NUMBER;
BEGIN
   IF (pcomp IS NULL) THEN
      SELECT COUNT(*) INTO p1 FROM Pilote
      WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote);
   ELSE
      SELECT COUNT(*) INTO p1 FROM Pilote
      WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote WHERE comp = pcomp)
      AND   comp = pcomp;
   END IF;
   IF p1 = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Aucun pilote n''est le plus expérimenté');
   ELSIF p1 > 1 THEN
      DBMS_OUTPUT.PUT_LINE('Plusieurs pilotes sont les plus
      expérimentés');
    ELSE
      IF (pcomp IS NULL) THEN
       SELECT nom, nbHVol, comp INTO pnomPil, pheuresVol, pcomp
       FROM  Pilote
       WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote);
      ELSE
       SELECT nom, nbHVol INTO pnomPil, pheuresVol FROM Pilote
       WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote WHERE comp =
       pcomp)
       AND   comp = pcomp;
      END IF;
 END IF;
END PlusExpérimenté  ;
```

Compilation

Pour compiler ces sous-programmes à partir de l'interface SQL*Plus, il faut rajouter le symbole / en première colonne après chaque dernier END. Si le message suivant apparaît, Avertissement : *Fonction*/*Procédure* créée avec erreurs de compilation, deux techniques peuvent être utilisées pour visualiser les erreurs de compilation :

- faire SHOW ERRORS sous SQL*Plus ;
- interroger la vue USER_ERRORS (SELECT LINE, POSITION, TEXT FROM USER_ERRORS WHERE NAME='*nomFonction*/*nomProcédure*';).

Une fois que le message *Fonction*/*Procédure* créée. apparaît, le sous-programme est correctement compilé et stocké en base.

Appels

Le propriétaire d'un sous-programme peut exécuter ce dernier à la demande et sans aucune condition préalable. Pour exécuter un sous-programme d'un autre schéma les conditions suivantes doivent être respectées :

- détenir le privilège EXECUTE sur le sous-programme en question ou EXECUTE ANY PROCEDURE ;
- mentionner le nom du schéma contenant le sous-programme à l'appel de ce dernier (exemple de l'appel de la procédure AugmenteCapacité du schéma jean pour l'avion d'immatriculation 'F-GLFS' jean.AugmenteCapacité('F-GLFS');).

Décrivons l'appel d'un sous-programme sous l'interface de commande SQL*Plus, dans un programme PL/SQL et dans une instruction SQL. Les chapitres suivants décriront comment coder un tel appel dans un programme externe (Java et C).

*Sous SQL*Plus*

En phase de tests, il est intéressant de pouvoir appeler un sous-programme directement dans l'interface de commande. La commande EXECUTE permet d'appeler une procédure ou une fonction (qui peut aussi être appelée dans une instruction SQL, ici un SELECT).

Le tableau suivant décrit l'appel et le résultat des deux sous-programmes.

Tableau 7-1 Appels sous SQL*Plus

Procédure	Fonction

`VARIABLE g_comp VARCHAR2(4);`
`VARIABLE g_nom VARCHAR2(16);`
`VARIABLE g_heuresVol NUMBER;`
`BEGIN`
`:g_comp := 'AF';`
`END;`
`/`
`EXECUTE PlusExpérimenté(:g_comp, :g_nom, :g_heuresVol);`
	`VARIABLE g_comp VARCHAR2(4);`
	`VARIABLE g_heuresVol NUMBER;`
	`VARIABLE g_résultat NUMBER;`
	`BEGIN`
	`:g_comp := 'AF';`
	`:g_heuresVol:= 300;`
	`END;`
	`/`
	`EXECUTE :g_résultat :=`
	`EffectifsHeure(:g_comp, :g_heuresVol);`

`SQL> PRINT g_nom;`
`G_NOM`
`------------------------------`
`Gilles Laborde`
 	
`SQL> PRINT g_heuresVol ;`
`G_HEURESVOL`
`-----------`
`2450`
	`SQL> PRINT :g_résultat;`
	`G_RÉSULTAT`
	`----------`
	`2`
	
	`SQL> SELECT comp, EffectifsHeure(comp,300)`
	`FROM Pilote GROUP BY comp;`
	`COMP EFFECTIFSHEURE(COMP,300)`
	`---- -------------------------`
	`AF 2`
	`CAST 1`
	`SING 2`

Dans un programme PL/SQL

Nous appelons les deux sous-programmes à présent dans un bloc PL/SQL. Le même principe peut être adopté pour l'appel dans un sous-programme PL/SQL ou dans un déclencheur.

Tableau 7-2 Appels dans un bloc PL/SQL

Procédure	Fonction

`SET SERVEROUT ON`
`DECLARE`
`v_comp VARCHAR2(4) := 'AF';`
`v_nom VARCHAR2(16);`
`v_heuresVol NUMBER(7,2);`
`BEGIN`
`PlusExpérimenté(v_comp,`
`v_nom, v_heuresVol);`
`DBMS_OUTPUT.PUT_LINE`
`('Nom, heures de vol ' || v_nom ||`
`' : ' || v_heuresVol);`
`END;`
`/`
	`SET SERVEROUT ON`
	`DECLARE`
	`v_comp VARCHAR2(4) := 'AF';`
	`v_heuresVol NUMBER(7,2) := 300;`
	`v_résultat NUMBER;`
	`BEGIN`
	`v_résultat :=`
	`EffectifsHeure(v_comp,v_heuresVol);`
	`DBMS_OUTPUT.PUT_LINE('Pour AF et`
	`300h résultat : ' || v_résultat );`
	`END;`
	`/`

`Nom, heures de vol Gilles Laborde : 2450` `Procédure PL/SQL terminée avec succès.`	`Pour AF et 300h résultat : 2` `Procédure PL/SQL terminée avec succès.`

Types d'appel

L'appel d'un sous-programme peut être positionnel, nommé ou mixte (qui combine les deux précédentes approches). Le tableau suivant décrit ces trois notations pour l'appel de la procédure :

Tableau 7-3 Différents appels d'une procédure

Type d'appel	Code PL/SQL
Positionnel	`PlusExpérimenté(v_comp, v_nom, v_heuresVol);`
Nommé	`PlusExpérimenté(pnomPil => v_nom, pheuresVol => v_heuresVol, pcomp => v_comp );`
Mixte	`PlusExpérimenté(v_comp, pheuresVol => v_heuresVol, pnomPil => v_nom );`

Pour tous les appels mixtes, il faut que les notations positionnelles précèdent les notations nommées.

À propos des paramètres

Le passage par valeur d'un paramètre se réalise par la directive `IN`. On peut assimiler le passage d'un paramètre par référence à l'utilisation de la directive `IN OUT`. La directive `NOCOPY` restreint le champ d'un paramètre comme le montre l'exemple suivant.

Dans cet exemple, les deuxième et troisième paramètres (`n2` et `n3`) passent en référence. Seul `n3` est déclaré. `NOCOPY` et son affectation à la valeur `30` dans la procédure répercutent la modification en local de `n1` et `n2`, (à 30). Cependant `n2` retrouve sa valeur affectée auparavant (20) au retour de l'appel du fait du caractère `NOCOPY` de `n3`.

Tableau 7-4 Passage par valeur et par référence

Code PL/SQL	Commentaires

`DECLARE`
`  n NUMBER := 10;`
`BEGIN`
`  changeEtAffiche(n, n, n);`
`  DBMS_OUTPUT.PUT_LINE(n);`
`END;`
`/`
	Affiche 20.

`CREATE PROCEDURE changeEtAffiche`
`(n1 IN NUMBER,n2 IN OUT NUMBER, n3 IN OUT NOCOPY NUMBER) IS`
`  BEGIN`
`  n2 := 20;`
`  DBMS_OUTPUT.PUT_LINE(n1);`
`  n3 := 30;`
`  DBMS_OUTPUT.PUT_LINE(n1);`
`END;`
	Affiche 10.
	Affiche 30.

`10`
`30`
`20`
`Procédure PL/SQL terminée avec succès.`
	Résultat obtenu.

Récursivité

La récursivité est permise dans PL/SQL. Comme dans tout programme récursif, il ne faut pas oublier la condition de terminaison ! L'exemple suivant décrit la programmation à l'aide d'une fonction récursive du calcul de la factorielle d'un entier positif. Nous appelons cette fonction, ici, dans un SELECT.

Tableau 7-5 Récursivité

Code PL/SQL	Commentaires

`CREATE FUNCTION factorielle(n POSITIVE) RETURN INTEGER IS`
`BEGIN`
` IF n = 1 THEN`
`  RETURN 1;`
`  ELSE`
`  RETURN n * factorielle(n - 1);`
` END IF;`
`END factorielle;`
`/`
	Condition de terminaison.
	
	Appel récursif.

`SQL> SELECT factorielle(30) "Factorielle 30" FROM DUAL;`
 	
`Factorielle 30`
`--------------`
`    2,6525E+32`
	Appel de la fonction.

Sous-programmes imbriqués

Il est possible de créer un sous-programme (*nested subprogram*) dans la partie déclarative d'un autre sous-programme. C'est aussi valable pour les blocs PL/SQL dont la section DECLARE peut inclure un sous-programme. Ces sous-programmes imbriqués n'ont d'existence que le temps de l'exécution du sous-programme qui l'inclut. Les sous-programmes imbriqués doivent être les derniers éléments de la section déclarative. Il n'est pas possible de déclarer, derrière un *nested subprogram*, une variable, un curseur ou une exception.

Le tableau suivant décrit la déclaration et l'appel du sous-programme imbriqué `Mouchard` dans la procédure `PlusExpérimenté`. Ce sous-programme insère une ligne dans une table pour tracer l'appel de la procédure en fonction de l'utilisateur et du moment de l'exécution.

Dans le cas où plusieurs sous-programmes imbriqués s'appellent entre eux, il est possible de définir des références avant (*forward declaration*) pour éviter de respecter un ordre à la déclaration et pour se prémunir de tout problème de cohérence.

Tableau 7-6 Sous-programme imbriqué

Code PL/SQL	Commentaires

`CREATE OR REPLACE PROCEDURE PlusExpérimenté`
`(pcomp IN OUT VARCHAR2,`
`pnomPil OUT VARCHAR2,`
`pheuresVol OUT NUMBER)`
`IS`
`p1 NUMBER;`
	Déclaration du sous-programme.

`PROCEDURE Mouchard IS`
`BEGIN`
`INSERT INTO Trace VALUES (USER ||`
`' a lancé PlusExpérimenté le ' ||SYSDATE);`
`END Mouchard;`
	Déclaration du sous-programme imbriqué.

`BEGIN` …	Début du sous-programme.

`Mouchard;`
…
`END PlusExpérimenté;`
	Appel du sous-programme imbriqué.

Il suffit de noter la signature des sous-programmes (nom et paramètres) avant de les redéfinir au niveau du codage. Le code suivant décrit un exemple de la procédure `KGB` qui appelle `Mouchard` et qui est toutefois définie avant :

Tableau 7-7 Référence avant d'un sous-programme

Code PL/SQL	Commentaires
`DECLARE` `PROCEDURE Mouchard;`	Signature (référence avant).

`PROCEDURE KGB IS`
`BEGIN`
`Mouchard;`
`END KGB;`
	Déclaration de la procédure `KGB`.

`PROCEDURE Mouchard IS`
`BEGIN`
`INSERT INTO Trace VALUES`
`(USER || ' a lancé le Bloc ' || SYSDATE);`
`END Mouchard;`
	Déclaration de la procédure `Mouchard`.

`BEGIN`
`KGB;`
`END;`
`/`
	Codage du bloc.

Recompilation d'un sous-programme

Oracle recompile automatiquement un sous-programme quand un objet qui en dépend directement (table, vue, synonyme, séquence, etc.) a été modifié dans sa structure. Les dépendances peuvent aussi être indirectes (exemple de modification de la structure d'une table qui définit une vue utilisée dans un sous-programme). En ce cas, il peut être nécessaire de recompiler manuellement chaque sous-programme potentiellement affecté.

La recompilation manuelle d'un sous-programme s'exécute par la commande `ALTER`. Pour pouvoir recompiler un sous-programme d'un autre schéma, vous devez détenir le privilège `ALTER ANY PROCEDURE`. Les syntaxes suivantes permettent de recompiler manuellement une procédure et une fonction :

```
ALTER PROCEDURE nomProcédure COMPILE;
ALTER FUNCTION nomFonction COMPILE;
```

Destruction d'un sous-programme

La syntaxe de suppression d'un sous-programme est la suivante. Pour supprimer une procédure ou une fonction dans un autre schéma, le privilège `DROP ANY PROCEDURE` est requis.

```
DROP PROCEDURE [schéma.]nomProcédure;
DROP FUNCTION [schéma.]nomFonction;
```

Il ne faut pas utiliser cette commande pour enlever une procédure ou une fonction d'un paquetage (notion abordée à la section suivante). Pour cela, nous verrons qu'il faudra redéfinir la spécification et le corps du nouveau paquetage en utilisant la directive `OR REPLACE`.

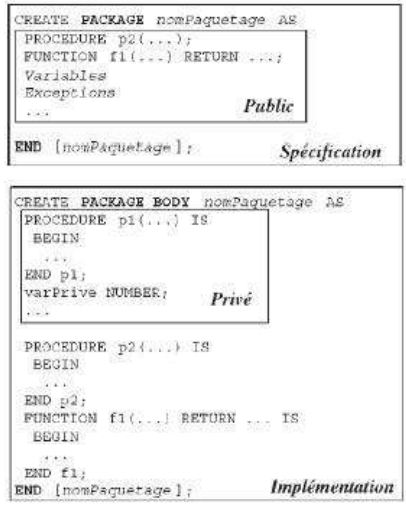
Paquetages (packages)

Un paquetage (*package*) est un composant qui regroupe plusieurs objets (variables, exceptions, curseurs, fonctions, procédures, etc.) formant un ensemble de services homogènes. C'est parce qu'un paquetage permet d'utiliser des objets publics ou privés qu'il s'apparente au concept de classe en programmation objet. L'avantage principal d'un paquetage est qu'il facilite la maintenance de l'application (modularité, extensibilité, réutilisabilité).

Généralités

La figure suivante illustre les deux parties d'un paquetage. La spécification contient les signatures des sous-programmes, la déclaration de variables, curseurs, d'exceptions, etc. L'implémentation (le corps) contient le code des sous-programmes. Ici, la procédure `p1` n'est pas définie dans la spécification et seuls les sous programmes du paquetage pourront y faire référence (ici `p2` et `f1`).

Figure 7-4 Structure d'un paquetage



Spécification

Pour créer un paquetage dans son propre schéma, il faut détenir le privilège CREATE PROCEDURE. Pour pouvoir créer un paquetage dans un autre schéma, le privilège CREATE ANY PROCEDURE doit être requis. La syntaxe simplifiée de la déclaration de la spécification d'un paquetage (CREATE PACKAGE) est la suivante :

```
CREATE [OR REPLACE] PACKAGE nomPaquetage
   [AUTHID {CURRENT_USER | DEFINER}]   {IS | AS}
   [déclarationTypeRECORD…]  [déclarationSUBTYPE …]
   [déclarationCONSTANT …]   [déclarationEXCEPTION …]
   [déclarationRECORD …]  [déclarationVariable …]
   [déclarationCURSOR …] [déclarationFonction …]
   [déclarationProcédure …]
END [nomPaquetage];
```

Créons la spécification du paquetage GestionPilotes qui inclut trois objets publics : la fonction EffectifsHeure, la procédure PlusExpérimenté et la variable résultat.

```
CREATE PACKAGE GestionPilotes AS
   résultat NUMBER := 0;
   FUNCTION EffectifsHeure(pcomp IN VARCHAR2, pheuresVol IN NUMBER)
```

```
            RETURN NUMBER;
    PROCEDURE PlusExpérimenté(pcomp IN OUT VARCHAR2,
                              pnomPil OUT VARCHAR2, pheuresVol
                              OUT NUMBER);
END GestionPilotes ;
/
```

Compilation

Pour compiler la spécification, comme l'implémentation du paquetage, à partir de l'interface SQL*Plus, il faut procéder comme pour un sous-programme. En cas d'erreurs, il faut exécuter `SHOW ERRORS` sous SQL*Plus ou interroger la vue `USER_ERRORS`. Une fois que les messages `Package créé` puis `Corps de package créé` apparaissent, le paquetage est opérationnel.

Implémentation

Pour implémenter un paquetage, il faut détenir le privilège `CREATE PROCEDURE`. Pour créer un paquetage dans un autre schéma, le privilège `CREATE ANY PROCEDURE` doit être requis. La syntaxe simplifiée de l'implémentation d'un paquetage (`CREATE PACKAGE BODY`) est la suivante :

```
CREATE [OR REPLACE] PACKAGE BODY nomPaquetage (IS | AS)
   [définition objets privés]
   [définition sous-programmes privés]
   [définition procédures publiques]
   [définition fonctions publiques]
END [nomPaquetage];
```

Créons le corps du paquetage `GestionPilotes` en codant la fonction `EffectifsHeure` et la procédure `PlusExpérimenté` :

```
CREATE PACKAGE BODY GestionPilotes AS
  FUNCTION EffectifsHeure(pcomp IN VARCHAR2, pheuresVol IN NUMBER)
           RETURN NUMBER IS
  BEGIN
   IF (pcomp IS NULL) THEN
       SELECT COUNT(*) INTO résultat FROM Pilote WHERE  nbHVol >
       pheuresVol ;
  ELSE
       SELECT COUNT(*) INTO résultat FROM Pilote
              WHERE  nbHVol >  pheuresVol AND comp = pcomp;
   END IF;
   RETURN résultat;
  END EffectifsHeure;
```

```
  PROCEDURE PlusExpérimenté(pcomp IN OUT VARCHAR2, pnomPil OUT
  VARCHAR2, pheuresVol OUT NUMBER) IS
  BEGIN
   …voir section précédente
  END PlusExpérimenté;

END GestionPilotes ;
/
```

Appel

L'accès à un sous-programme `sp` d'un paquetage `paq` s'écrit `paq.sp`. L'appel de ce sous-programme suit les mêmes règles que celles étudiées dans les sections précédentes (procédures et fonctions cataloguées). Les prérogatives d'exécution d'un sous-programme d'un paquetage sont identiques à celles des sous-programmes classiques.

L'appel de la procédure `PlusExpérimenté` du paquetage `GestionPilotes` sera codé `GestionPilotes.PlusExpérimenté(...)` dans un programme PL/SQL, et la fonction `EffectifsHeure` sera codée `GestionPilotes.EffectifsHeure(...)`.

Surcharge

Il est possible de surcharger une fonction ou une méthode d'un paquetage. Les deux sous-programmes doivent avoir le même nom mais différents paramètres. La spécification du paquetage liste tous les sous-programmes et contient un codage différent pour chacun.

Recompilation

Pour recompiler la spécification ou le corps d'un paquetage, il faut utiliser l'option `OR REPLACE` de la commande `CREATE PACKAGE` après avoir modifié une des deux parties (ou les deux) et réexécuté l'une ou l'autre partie du paquetage.

Destruction d'un paquetage

La syntaxe de suppression de la spécification et du corps d'un paquetage est la suivante : pour supprimer une partie d'un paquetage d'un autre schéma, le privilège `DROP ANY PROCEDURE` est requis.

```
DROP PACKAGE BODY [schéma.]nomPaquetage;
DROP PACKAGE [schéma.]nomPaquetage;
```

Comment retourner une table ?

Il est intéressant d'utiliser une fonction au sein d'un paquetage pour retourner tout ou partie d'une table. La fonction retournera un tableau, le paquetage contiendra la description du tableau et la déclaration de la table. La méthode la plus appropriée, depuis la version 10*g*, est celle du *bulk collect* qui permet d'extraire sous la forme d'une collection un grand volume de données.

Le tableau suivant décrit d'une part la déclaration et le codage du paquetage contenant la fonction qui retourne les pilotes d'une compagnie dont le code passe en paramètre, et d'autre part, l'appel de la fonction et le résultat obtenu sous la forme d'un tableau (dont on extrait seulement la colonne nom). Le jeu d'essai est dans le script en téléchargement.

Tableau 7-8 Comment retourner une table ?

Description et codage du paquetage

```
CREATE PACKAGE PKG_Pilotes IS
 TYPE Pilote_tytab
  IS TABLE OF Pilote%ROWTYPE
  INDEX BY BINARY_INTEGER;
  FUNCTION f_pilotes_compagnie
    (v_comp IN VARCHAR2)
    RETURN Pilote_tytab;
END PKG_Pilotes;
/

CREATE PACKAGE BODY PKG_Pilotes IS
  FUNCTION f_pilotes_compagnie
   (v_comp IN VARCHAR2)
     RETURN Pilote_tytab
  IS
     tab Pilote_tytab;
  BEGIN
   SELECT * BULK COLLECT
      INTO tab FROM Pilote
       WHERE compa=v_comp;
   RETURN tab;
  END;
END PKG_Pilotes;
/
```

Appel de la fonction

```
DECLARE
    tab_sortie PKG_Pilotes.Pilote_tytab;
    nb_pil NUMBER;
    i NUMBER;
BEGIN
 tab_sortie :=
   PKG_Pilotes.f_pilotes_compagnie('AF');
 nb_pil := tab_sortie.COUNT;
 FOR i IN 1..nb_pil LOOP
    DBMS_OUTPUT.PUT_LINE(tab_sortie(i).nom);
 END LOOP;
END;
/

Henri Alquié
Pierre Lamothe
Didier Linxe

Procédure PL/SQL terminée avec succès.
```

Curseurs

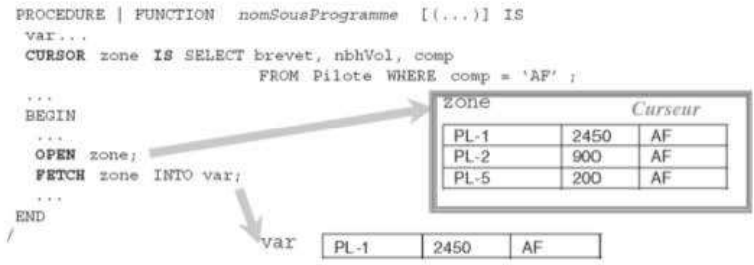
Au chapitre précédent nous avons parlé des curseurs implicites, ici nous allons étudier les curseurs explicites (que les programmeurs appellent *curseurs* tout simplement). Ils sont très utilisés, pour ne pas dire qu'ils sont présents, dans toute procédure d'une application importante. Le concept analogue au niveau de JDBC est programmé à l'aide de la classe `Resultset`, et sous ASP de Microsoft, à l'aide de la classe `RecordSet` (appelée `DataSet` avec *.Net*).

Généralités

Un curseur est une zone mémoire qui permet de traiter individuellement chaque ligne renvoyée par un SELECT. Un programme PL/SQL peut travailler avec plusieurs curseurs en même temps. Un curseur, durant son existence (de l'ouverture à la fermeture), contient en permanence l'adresse de la ligne courante.

La figure suivante illustre la manipulation de base d'un curseur. Le curseur est décrit dans la partie déclarative. Il est ouvert dans le code du programme, il s'évalue alors et va se charger en extrayant les données de la base. Le programme peut parcourir tout le curseur en récupérant les lignes une par une dans une variable locale. Le curseur est ensuite fermé.

Figure 7-5 *Principes d'un curseur*



Il existe plusieurs manières de parcourir un curseur, comme il existe plusieurs types de curseurs à parcourir. Nous allons aborder toutes ces notions par difficulté croissante.

Instructions

Les instructions propres aux curseurs sont définies dans le tableau page suivante.

Tableau 7-9 Instructions pour les curseurs

Instruction	Commentaires et exemples
CURSOR *nomCurseur* **IS** *requête*;	Déclaration du curseur. `CURSOR zone1 IS SELECT brevet, nbHVol, comp FROM Pilote WHERE comp = 'AF';`
OPEN *nomCurseur*;	Ouverture du curseur (chargement des lignes). Aucune exception n'est levée si la requête ne ramène aucune ligne. `OPEN zone1;`
FETCH *nomCurseur* **INTO** *listeVariables* | *nomRECORD*;	Positionnement sur la ligne suivante et chargement de l'enregistrement courant dans une ou plusieurs variables. `FETCH zone1 INTO var1,var2,var3;`
CLOSE *nomCurseur*;	Ferme le curseur. L'exception INVALID_CURSOR se déclenche si des accès au curseur sont opérés après sa fermeture. `CLOSE zone1;`
nomCurseur%**ISOPEN**	Retourne TRUE si le curseur est ouvert, FALSE sinon. `IF zone1%ISOPEN THEN …`
nomCurseur%**NOTFOUND**	Retourne TRUE si le dernier FETCH n'a pas renvoyé de ligne (fin de curseur). `EXIT WHEN zone1%NOTFOUND;`
nomCurseur%**FOUND**	Retourne TRUE si le dernier FETCH a renvoyé une ligne. `WHILE (zone1%FOUND) LOOP`
nomCurseur%**ROWCOUNT**	Retourne le nombre total de lignes traitées jusqu'à présent (pointeur absolu).

Parcours d'un curseur

Suivant le traitement à effectuer sur le curseur à parcourir, vous pouvez choisir d'utiliser une structure répétitive *tant que*, *répéter* ou *pour*. Étudions dans un premier temps les deux premières solutions. Le paragraphe suivant traitera de la dernière (structure FOR).

Le tableau ci-après présente le parcours d'un curseur à l'aide des deux techniques (*tant que* et *répéter*). Ici, il s'agit de faire la somme des heures de vol des pilotes de la compagnie de code 'AF'.

Avant la première extraction, *nomCurseur*%NOTFOUND revoie toujours NULL. Si l'instruction FETCH ne parvient jamais à s'exécuter correctement, la boucle *répéter* devient infinie. Il est conseillé de programmer la sortie d'une structure *répéter* à l'aide de la condition composée : EXIT WHEN *nomCurseur*%NOTFOUND OR **nomCurseur%NOTFOUND IS NULL**.

Tableau 7-10 Parcours d'un curseur

Tant que

```
DECLARE
  CURSOR zone1 IS SELECT brevet, nbHVol, comp
                  FROM Pilote WHERE comp = 'AF';
  var1   Pilote.brevet%TYPE;
  var2   Pilote.nbHVol%TYPE;
  var3   Pilote.comp%TYPE;
  totalHeures NUMBER := 0;
BEGIN
  OPEN zone1;
  FETCH zone1 INTO var1,var2,var3;
  WHILE (zone1%FOUND) LOOP
    totalHeures := totalHeures + var2;
    FETCH zone1 INTO var1,var2,var3;
  END LOOP;
  CLOSE zone1;
...
```

Répéter

```
DECLARE
  CURSOR zone1 IS SELECT brevet, nbHVol, comp
                  FROM Pilote WHERE comp = 'AF';
  var1   Pilote.brevet%TYPE;
  var2   Pilote.nbHVol%TYPE;
  var3   Pilote.comp%TYPE;
  totalHeures NUMBER := 0;
BEGIN
  OPEN zone1;
  LOOP
    FETCH zone1 INTO var1,var2,var3;
    EXIT WHEN zone1%NOTFOUND;
    totalHeures := totalHeures + var2;
  END LOOP;
  CLOSE zone1;
...
```

Utilisation de structures (%ROWTYPE)

Accès par la notation pointée

Il est possible de définir un enregistrement en fonction de la liste des colonnes d'un curseur. Cela évite de déclarer autant de variables que de colonnes contenues dans le curseur. L'accès aux valeurs des colonnes se fait par la notation pointée comme l'illustre l'exemple suivant qui affiche le nom des pilotes n'appartenant pas à la compagnie de code 'AF'.

Tableau 7-11 Utilisation d'une variable structurée

Code PL/SQL

```
DECLARE
  CURSOR zone2 IS SELECT brevet, nom
                  FROM Pilote WHERE NOT (comp='AF');
  enreg  zone2%ROWTYPE;
BEGIN
  OPEN zone2;
  FETCH zone2 INTO enreg;
  WHILE (zone2%FOUND) LOOP
    DBMS_OUTPUT.PUT_LINE('nom : ' || enreg.nom ||
                         ' (' || enreg.brevet || ').');
    FETCH zone2 INTO enreg ;
  END LOOP;
  CLOSE zone2;
END;
/
```

Commentaires

Déclaration de la structure.

Chargement de la structure.

Accès aux éléments de la structure.

Résultat.

```
nom : Florence Périssel (PL-3)
nom : Thierry Millan (PL-4)
nom : Aurélia Ente (PL-6)

Procédure PL/SQL terminée avec succès.
```

Utilisation de la clause RETURN

La clause RETURN permet de préciser le type de retour d'un curseur. Il est intéressant de combiner l'utilisation de cette clause avec une structure de données %ROWTYPE si le curseur est défini dans la spécification d'un paquetage. L'avantage de cette technique est de pouvoir recompiler le corps sans avoir à modifier la spécification.

Le tableau suivant décrit une spécification de curseur qui peut être implémentée de différentes manières dans le temps :

Tableau 7-12 Curseur défini avec RETURN

Code PL/SQL	Commentaires

`CREATE PACKAGE paquet_curseur AS`
`CURSOR zone3 RETURN Pilote%ROWTYPE;`
`END paquet_curseur;`
	Spécification du curseur.

`CREATE PACKAGE BODY paquet_curseur AS`
`CURSOR zone3 RETURN Pilote%ROWTYPE IS`
`SELECT * FROM Pilote WHERE comp = 'AF';`
…
`END paquet_curseur;`
	Implémentation du curseur.

`CREATE OR REPLACE PACKAGE BODY paquet_curseur AS`
`CURSOR zone3 RETURN Pilote%ROWTYPE IS`
`SELECT * FROM Pilote WHERE nbhVol > 500;`
…
`END paquet_curseur;`
	Autre implémentation du curseur.

Boucle FOR (gestion semi-automatique)

L'utilisation d'une boucle FOR de curseur facilite la programmation (évite les directives OPEN, FETCH et CLOSE). La boucle s'arrête d'elle-même à la fin de l'extraction de la dernière ligne du curseur. De plus, la variable de réception du curseur est aussi automatiquement déclarée (%ROWTYPE du curseur). L'accès aux valeurs des colonnes se fait également par la notation pointée.

Les lignes suivantes affichent le nom des pilotes qui n'appartiennent pas à la compagnie de code 'AF' en utilisant une boucle FOR :

Tableau 7-13 Utilisation d'une boucle FOR

Code PL/SQL	Commentaires

`DECLARE`
`CURSOR zone3 IS SELECT brevet, nom`
`FROM Pilote WHERE NOT (comp='AF');`
	Déclaration du curseur.

`BEGIN`
`FOR enreg IN zone3 LOOP`
`DBMS_OUTPUT.PUT_LINE('nom : ' || enreg.nom ||`
`' (' || enreg.brevet || ').');`
`END LOOP;`
`END;`
`/`
	Itération dans le curseur.

Pour ceux qui ne veulent pas perdre de temps à déclarer le curseur, Oracle offre la possibilité de le manipuler tout en le déclarant à l'intérieur de l'instruction FOR. Ici, il ne sera pas possible de réutiliser le curseur puisqu'il n'a d'existence que dans la boucle. Il ne sera pas possible non plus d'utiliser des paramètres de curseur. Le code suivant réalise la même action que le bloc précédent, en utilisant un curseur temporaire :

Tableau 7-14 Curseur temporaire

Code PL/SQL	Commentaires

`BEGIN`
`FOR enreg IN`
`(SELECT brevet, nom FROM Pilote WHERE`
`NOT (comp = 'AF')) LOOP`
`DBMS_OUTPUT.PUT_LINE('nom : ' || enreg.nom || '`
`(' || enreg.brevet || ').');`
`END LOOP;`
`END;`
`/`
	Itération dans un curseur temporaire.

Utilisation de tableaux (type TABLE)

Il est possible d'utiliser des tableaux PL/SQL (étudiés au chapitre précédent) pour récupérer tout ou partie du contenu d'un curseur. Ceci est bien sûr valable pour les curseurs qui renvoient un nombre raisonnable de lignes.

Le bloc suivant décrit le chargement du tableau `tab_nomPilote` à partir des noms de tous les pilotes de la compagnie de code 'AF ', et l'accès direct au deuxième élément du tableau.

Tableau 7-15 Utilisation de tableau

Code PL/SQL	Commentaires

`DECLARE`
`TYPE nomPilotes_tytab IS TABLE OF`
`Pilote.nom%TYPE INDEX BY BINARY_INTEGER;`
`tab_nomPilote nomPilotes_tytab;`
`CURSOR zone4 IS SELECT brevet, nom`
`FROM Pilote WHERE comp = 'AF';`
`i NUMBER := 1;`
	Déclaration du tableau.

`BEGIN`
`FOR enreg IN zone4 LOOP`
`tab_nomPilote(i) := enreg.nom;`
`i := i + 1;`
`END LOOP;`
`…`
`DBMS_OUTPUT.PUT_LINE('2ème pilote : ' ||`
`tab_nomPilote(2));`
`END;`
`/`
	Chargement du tableau.
	
	Accès au deuxième élément.

`2ème pilote : Frédéric D'Almeyda`
 	
`Procédure PL/SQL terminée avec succès.`
	Résultat du bloc.

Utilisation de LIMIT et BULK COLLECT

Les options LIMIT et BULK COLLECT de l'instruction FETCH permettent de traiter de grands volumes de données sans pour autant pénaliser la mémoire centrale ou le cache par le fait de ne pas monter toutes les lignes d'une table en une fois. Ainsi, afin de retourner tout ou partie d'une table d'une potentielle grande volumétrie, la méthode la plus appropriée consiste à utiliser un tableau PL/SQL qu'on chargera dans une boucle en limitant le nombre d'enregistrement tout en parcourant toute la table.

Le tableau suivant décrit une bonne et une mauvaise manière de faire cette extraction. Le jeu d'essai est dans le script en téléchargement.

- Dans la procédure correcte, par défaut, 100 enregistrements pilotes au plus sont dans le tableau, et pour chaque itération du curseur on réalise une boucle du nombre réel de pilotes chargés dans le tableau. Supposons qu'il existe 2 343 pilotes, la boucle du curseur s'exécutera 24 fois, et à sa dernière itération, la boucle interne sera effectuée 43 fois.
- Dans la procédure incorrecte, par défaut, 100 enregistrements pilotes au plus sont dans le tableau, mais si moins de 100 pilotes (ou le nombre indiqué dans la variable `limite`) sont chargés dans le tableau, la boucle du curseur s'interrompt (`EXIT WHEN pilotes_cur%NOTFOUND`). Pour les 2 343 pilotes, la boucle du curseur ne s'exécutera que 23 fois, donc les 43 derniers pilotes ne seront pas traités.

Tableau 7-16 Exemple de parcours d'une table d'un grand volume

La bonne manière

```
CREATE PROCEDURE p_traite_all_rows
   (limite IN PLS_INTEGER DEFAULT 100)
IS
  CURSOR pilotes_cur IS
         SELECT * FROM Pilote;
  TYPE Pilote_tytab IS
       TABLE OF Pilote%ROWTYPE
       INDEX BY BINARY_INTEGER;
  tab_pilote Pilote_tytab;
BEGIN
    OPEN pilotes_cur;
    LOOP
      FETCH pilotes_cur
         BULK COLLECT
         INTO tab_pilote LIMIT limite;
      FOR i IN 1 .. tab_pilote.COUNT
       LOOP
        -- traitement de chaque ligne
       END LOOP;
    EXIT WHEN tab_pilote.COUNT < limite;
   END LOOP;
   CLOSE pilotes_cur;
END p_traite_all_rows;
/
```

La mauvaise manière

```
CREATE PROCEDURE p_traite_bug_rows
   (limite IN PLS_INTEGER DEFAULT 100)
IS
    CURSOR pilotes_cur IS
          SELECT * FROM Pilote;
    TYPE Pilote_tytab IS
         TABLE OF Pilote%ROWTYPE
         INDEX BY BINARY_INTEGER;
    tab_pilote Pilote_tytab;
BEGIN
    OPEN pilotes_cur;
    LOOP
      FETCH pilotes_cur
         BULK COLLECT INTO tab_pilote
         LIMIT limite;
    EXIT WHEN pilotes_cur%NOTFOUND;
      FOR i IN 1 .. tab_pilote.COUNT
      LOOP
        -- traitement de chaque ligne
      END LOOP;
   END LOOP;
   CLOSE pilotes_cur;
END p_traite_bug_rows;
/
```

Si vous chargez un tableau à l'aide de `BULK COLLECT` dans vos instructions `FETCH`, méfiez-vous des `directives %NOTFOUND` et `%FOUND` dans les structures `EXIT` et `WHILE`. Préférez la méthode `COUNT` pour tester la taille réelle du tableau chargé par le curseur.

Paramètres d'un curseur

Un curseur peut posséder des paramètres d'entrée. Cette technique est très utile lorsqu'un même curseur doit être utilisé plusieurs fois sous des critères différents. Il faudra en ce cas fermer le curseur s'il était déjà utilisé, avant de l'ouvrir à nouveau en lui passant des paramètres différents.

Le passage des paramètres peut se faire à l'ouverture du curseur (`OPEN`) ou dans la boucle `FOR` (si le curseur est utilisé en mode semi-automatique). Comme les paramètres d'un sous-programme, ceux d'un curseur ne doivent pas être restreints au niveau de la taille, seul le type est important.

Le tableau suivant décrit un bloc qui utilise deux fois le même curseur en affichant d'abord les pilotes de la compagnie de code 'AF' puis ceux de la compagnie de code 'SING'. Nous utilisons les deux écritures possibles pour passer les paramètres.

Tableau 7-17 Curseur paramétré

Code PL/SQL	Commentaires

`DECLARE`
`CURSOR zone5(p_codecomp IN VARCHAR2) IS`
`SELECT brevet, nom`
`FROM Pilote WHERE comp = p_codecomp;`
`enregbis zone5%ROWTYPE;`
`BEGIN`
	Déclaration du curseur avec un paramètre.

`FOR enreg IN zone5('AF') LOOP`
`DBMS_OUTPUT.PUT_LINE('AF, nom : ' || enreg.nom ||`
`' (' || enreg.brevet || ').');`
`END LOOP;`
	Chargement et parcours du curseur en passant le paramètre 'AF'.

`OPEN zone5('SING');`
`FETCH zone5 INTO enregbis ;`
`WHILE (zone5%FOUND) LOOP`
`DBMS_OUTPUT.PUT_LINE('SING, nom : ' ||`
`enregbis.nom||'('||enregbis.brevet||').');`
`FETCH zone5 INTO enregbis ;`
`END LOOP;`
`CLOSE zone5;`
`END;`
`/`
	Chargement et parcours du curseur en passant le paramètre 'SING'.

Accès concurrents (FOR UPDATE et CURRENT OF)

Si vous voulez verrouiller les lignes d'une table interrogée par un curseur dans le but de mettre à jour la table, sans qu'un autre utilisateur ne la modifie en même temps, il faut utiliser la clause FOR UPDATE. Elle s'utilise lors de la déclaration du curseur et verrouille les lignes concernées lorsque le curseur est ouvert. Les verrous sont libérés à la fin de la transaction.

La déclaration d'un curseur FOR UPDATE, qu'on peut qualifier de « modifiable », est la suivante :

```
CURSOR nomCurseur[(paramètres)] IS
            SELECT … FROM {nomTable | nomVue } WHERE …
FOR UPDATE [OF [[schéma.] {nomTable | nomVue }.]colonne [, …]
           [ NOWAIT | WAIT entier ]
```

- La directive OF permet de connaître les colonnes à verrouiller. Sans elle, toutes les colonnes issues de la requête seront verrouillées.
- NOWAIT précise de ne pas faire attendre le programme et de retourner un message d'erreur si les lignes demandées sont verrouillées par une autre session.
- WAIT spécifie le nombre de secondes à attendre au maximum avant que les lignes soient déverrouillées par une autre session. Sans NOWAIT et WAIT, le programme attend que les lignes soient disponibles.

- Une validation (COMMIT) avant la fermeture d'un curseur FOR UPDATE déclenchera une erreur.
- Il n'est pas possible de déclarer un curseur FOR UPDATE en utilisant dans la requête les directives DISTINCT ou GROUP BY, un opérateur ensembliste, ou une fonction d'agrégat.

Il est souvent intéressant de pouvoir modifier facilement la ligne courante d'un curseur (UPDATE ou DELETE à répercuter au niveau de la table). La clause WHERE CURRENT OF, située au niveau de l'instruction de mise à jour (UPDATE ou DELETE), permet de référencer la ligne courante d'un curseur. Il est conseillé d'utiliser un curseur FOR UPDATE pour verrouiller les lignes à actualiser.

Le tableau suivant décrit un bloc qui utilise le curseur FOR UPDATE pour :

- augmenter le nombre d'heures de 100 pour les pilotes de la compagnie de code 'AF' ;
- diminuer ce nombre de 100 pour les pilotes de la compagnie de code 'SING' ;
- supprimer les pilotes des autres compagnies.

Notez qu'il n'y a pas d'autre condition que WHERE CURRENT OF dans les instructions de mise à jour de la table.

Tableau 7-18 Curseur modifiable

Code PL/SQL	Commentaires

`DECLARE`
`CURSOR zoneModifiable IS SELECT * FROM Pilote`
`FOR UPDATE OF nbHVol NOWAIT;`
`BEGIN`
	Déclaration du curseur modifiable.

`FOR enreg IN zoneModifiable LOOP`
`IF enreg.comp = 'AF' THEN`
`UPDATE Pilote SET nbHVol = nbHVol + 100`
`WHERE CURRENT OF zoneModifiable;`
`ELSIF enreg.comp = 'SING' THEN`
`UPDATE Pilote SET nbHVol = nbHVol - 100`
`WHERE CURRENT OF zoneModifiable;`
`ELSE`
`DELETE FROM Pilote`
`WHERE CURRENT OF zoneModifiable;`
`END IF;`
`END LOOP;`
`COMMIT;`
`END;`
`/`
	Chargement et parcours du curseur.
	
	Mises à jour de la table Pilote par l'intermédiaire du curseur.
	
	Validation de la transaction.

Variables curseurs (REF CURSOR)

Une variable curseur (REF CURSOR) définit un curseur dynamique qui n'est pas associé à une requête donnée comme un curseur classique (statique). Une variable curseur permet au curseur d'évoluer au cours du programme.

Une variable curseur est déclarée en deux étapes : déclaration du type et de la variable du type. Une variable REF CURSOR peut être définie dans un bloc ou un sous-programme PL/SQL par les instructions suivantes :

```
TYPE nomTypeCurseurDynamique IS REF CURSOR [RETURN typeRetourSQL];
nomCurseurDynamique nomTypeCurseurDynamique;
```

Le type de retour représente en général la structure d'un enregistrement d'une table. Le curseur dynamique est dit « typé » (*strong*) s'il inclut un type de retour. Dans le cas inverse, il est non typé (*weak*) et permet une grande flexibilité car toute requête peut y être associée. L'ouverture d'un curseur dynamique est commandée par l'instruction `OPEN FOR` *`requête`*. La lecture du curseur s'opère toujours avec l'instruction `FETCH`.

Curseurs non typés

Le tableau suivant décrit un bloc qui utilise le curseur dynamique non typé `zone6`. Ce curseur sert à afficher dans un premier temps les numéros de brevet et noms des pilotes qui ne sont pas de la compagnie de code 'AF'. Dans un second temps, le curseur est rechargé afin d'extraire les numéros de brevet et le nombre d'heures de vol de tous les pilotes de la compagnie de code 'AF'.

Tableau 7-19 Curseur non typé

Code PL/SQL	Commentaires

`DECLARE`
`  TYPE ref_zone6 IS REF CURSOR;`
`  zone6 ref_zone6;`
`  var1  Pilote.brevet%TYPE;`
`  var2  Pilote.nom%TYPE;`
`  var3  Pilote.nbHVol%TYPE;`
	Déclaration du curseur dynamique et des variables de réception.

`BEGIN`
`  OPEN zone6 FOR SELECT brevet, nom`
`  FROM Pilote WHERE NOT (comp = 'AF');`
`  FETCH zone6 INTO var1, var2;`
`  WHILE (zone6%FOUND) LOOP`
`  DBMS_OUTPUT.PUT_LINE('nom : ' || var2 ||`
`  ' (' || var1 || ').');`
`  FETCH zone6 INTO var1, var2;`
`  END LOOP;`
`  CLOSE zone6;`
	Chargement et parcours du curseur dynamique.

`  OPEN zone6 FOR SELECT brevet, nbHVol`
`  FROM Pilote WHERE comp = 'AF';`
`  FETCH zone6 INTO var1, var3;`
`  …`
`  CLOSE zone6;`
`END;`
`/`
	Autre chargement du curseur dynamique.

Curseurs typés

Le tableau suivant décrit un bloc qui utilise le curseur dynamique typé `zone7`. Celui-ci sert à extraire toutes les colonnes de la table `Pilote`. Dans un premier temps le curseur dynamique est chargé avec les pilotes qui ne sont pas de la compagnie de code 'AF'. Ensuite, le curseur est rechargé avec les pilotes qui sont de la compagnie de code 'AF'.

Tableau 7-20 Curseur typé

Code PL/SQL	Commentaires

`DECLARE`
`TYPE ref_zone7 IS REF CURSOR`
`RETURN Pilote%ROWTYPE;`
`zone7 ref_zone7;`
`enreg zone7%ROWTYPE;`
	Déclaration du curseur dynamique et de la structure de réception.

`BEGIN`
`OPEN zone7 FOR SELECT * FROM Pilote`
`WHERE NOT (comp = 'AF');`
`FETCH zone7 INTO enreg;`
`WHILE (zone7%FOUND) LOOP`
`DBMS_OUTPUT.PUT_LINE('nom : ' || enreg.nom`
`||(' ||enreg.comp || ').');`
`FETCH zone7 INTO enreg;`
`END LOOP;`
`CLOSE zone7;`
	Chargement et parcours du curseur dynamique.

`OPEN zone7 FOR SELECT * FROM Pilote`
`WHERE comp = 'AF';`
`FETCH zone7 INTO enreg;`
`…`
`CLOSE zone7;`
`END;`
`/`
	Autre chargement du curseur dynamique.

Fonctions table pipelined

Les fonctions *table pipelined* font jouer à PL/SQL le rôle de source de données. La fonction appelée le plus souvent dans la clause `FROM` d'une requête (du fait que le retour de cette fonction est une table, il faudra utiliser l'opérateur `TABLE` qui convertit une collection en table). Il est, dans certains cas, possible de l'invoquer dans une clause `SELECT`.

Ces fonctions peuvent accepter en paramètre une collection d'enregistrements : table PL/SQL, `VARRAY` (extension objet). L'exécution d'une telle fonction se voit ainsi « parallélisée » du fait que chaque ligne est retournée à l'appelant (directive `PIPE ROW`) sans attendre la fin de la fonction. Ainsi ne vous préoccupez pas de placer un `RETURN`, il ne peut pas être présent. Une telle fonction est déclarée à l'aide de l'option `PIPELINED`.

Le type de collection retourné par une fonction *table pipelined* est une table PL/SQL (sans l'option `INDEX BY BINARY_INTEGER`), une *nested table* ou un *varray*. Dans le code de la fonction, vous devez retourner des éléments de la collection en question (les types de données supportés sont les types SQL `NUMBER` et `VARCHAR2`. N'utilisez pas les types de données PL/SQL, tels que `PLS_INTEGER` ou `BOOLEAN`.

L'exemple suivant réalise la même fonctionnalité que celle étudiée dans le paragraphe « Comment retourner une table ? » La fonction *table pipelined* retourne à chaque itération du curseur un des pilotes d'une compagnie dont le code passe en paramètre. L'appel de la fonction se réalise dans la requête (dont on extrait les colonnes `brevet, nom et salaire`). Le jeu d'essai est dans le script en téléchargement.

Tableau 7-21 Fonction table pipelined

Description et codage du paquetage

```
REATE PACKAGE PKG_Pilotes IS
  TYPE Pilote_tytab
     IS TABLE OF Pilote%ROWTYPE;
  FUNCTION f_pilotes_comp_pipelined
  (v_comp IN VARCHAR2)
  RETURN Pilote_tytab PIPELINED;
END PKG_Pilotes;
/
CREATE PACKAGE BODY PKG_Pilotes IS
  FUNCTION f_pilotes_comp_pipelined
       (v_comp IN VARCHAR2)
  RETURN Pilote_tytab PIPELINED IS
    CURSOR Pilote_Comp_Cur
           IS SELECT * FROM Pilote
                 WHERE compa=v_comp;
    rty_pilote Pilote%ROWTYPE;
  BEGIN
     OPEN Pilote_Comp_Cur;
     FETCH Pilote_Comp_Cur INTO rty_pilote;
     WHILE (Pilote_Comp_Cur%FOUND) LOOP
        PIPE ROW(rty_pilote);
       FETCH Pilote_Comp_Cur INTO rty_pilote;
  END LOOP;
    CLOSE Pilote_Comp_Cur;
    RETURN;
END;
END PKG_Pilotes;
/
```

Appel de la fonction *table pipelined*

```
SELECT brevet,nom,salaire FROM TABLE
(PKG_Pilotes.f_pilotes_comp_pipeli-
ned('SING'));

BREVET NOM                       SALAIRE
------ -------------------- ----------
PL-4   Christian Soutou          10000
PL-5   Gilles Laborde            10050
PL-6   Pierre Séry               16000
.
```

Exceptions

Afin d'éviter qu'un programme s'arrête à la première erreur (requête ne retournant aucune ligne, valeur incorrecte à écrire dans la base, conflit de clés primaires, division par zéro, etc.), il est indispensable de prévoir tous les cas potentiels d'erreurs et d'associer à chacun de ces cas la programmation d'une exception PL/SQL. Dans le vocabulaire des programmeurs on dit qu'on *garde la main* pendant l'exécution du programme. Le mécanisme des exceptions (*handling errors*) est largement utilisé par tous les programmeurs car il est prépondérant dans la mise en œuvre des transactions.

Les exceptions peuvent se programmer dans un bloc PL/SQL, un sous-programme (fonction ou procédure cataloguée), dans un paquetage ou un déclencheur.

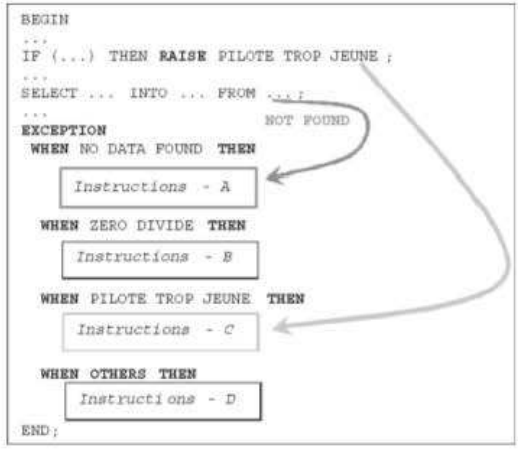
Généralités

Une exception PL/SQL correspond à une condition d'erreur et est associée à un identificateur. Une exception est détectée (aussi dite « levée ») au cours de l'exécution d'une partie de programme (entre un `BEGIN` et un `END`). Une fois levée, l'exception termine le corps principal des instructions et renvoie au bloc `EXCEPTION` du programme en question.

La figure suivante illustre les deux mécanismes qui peuvent déclencher une exception :

- Une erreur Oracle se produit, l'exception associée est déclenchée automatiquement (exemple du `SELECT` ne ramenant aucune ligne, ce qui déclenche l'exception `ORA-01403` d'identificateur `NO_DATA_FOUND`).

Figure 7-6 *Principe général des exceptions*



- Le programmeur désire dérouter volontairement (par l'intermédiaire de l'instruction RAISE) son programme dans le bloc des exceptions sous certaines conditions. L'exception est ici manuellement déclenchée et peut appartenir à l'utilisateur (ici la condition PILOTE_TROP_JEUNE) ou être prédéfinie au niveau d'Oracle (division par zéro d'identificateur ZERO_DIVIDE qui sera automatiquement déclenchée).

Si aucune erreur ne se produit, le bloc est ignoré et le traitement se termine (ou retourne à son appelant s'il s'agit d'un sous-programme).

La syntaxe générale d'un bloc d'exceptions est la suivante. Il est possible de grouper plusieurs exceptions pour programmer le même traitement. La dernière entrée (OTHERS) doit être éventuellement toujours placée en fin du bloc d'erreurs.

```
EXCEPTION
   WHEN exception1 [OR exception2 …] THEN
        instructions;
  [WHEN exception3 [OR exception4 …] THEN
        instructions;  ]
  [WHEN OTHERS THEN
        instructions; ]
```

Si une anomalie se produit, le bloc EXCEPTION s'exécute.

- Si le programme prend en compte l'erreur dans une entrée WHEN…, les instructions de cette entrée sont exécutées et le programme se termine.
- Si l'exception n'est pas prise en compte dans le bloc EXCEPTION :
  - il existe une section OTHERS où des instructions s'exécutent ;
  - il n'existe pas une section OTHERS et l'exception sera propagée au programme appelant (une section traite de la propagation des exceptions).

Étudions à présent les trois types d'exceptions qui existent sous PL/SQL, en programmant des procédures simples interrogeant la table Pilote illustrée à la figure 7-3.

Exception interne prédéfinie

Les exceptions prédéfinies sont celles qui se produisent le plus souvent. Oracle affecte un nom de manière à les traiter plus facilement dans le bloc EXCEPTION. Le tableau suivant les décrit :

Tableau 7-22 Exceptions prédéfinies

Nom de l'exception	Numéro	Commentaires
ACCESS_INTO_NULL	ORA-06530	Affectation d'une valeur à un objet non initialisé.
CASE_NOT_FOUND	ORA-06592	Aucun des choix de la structure CASE sans ELSE n'est effectué.
COLLECTION_IS_NULL	ORA-06531	Utilisation d'une méthode autre que EXISTS sur une collection (*nested table* ou *varray*) non initialisée.
CURSOR_ALREADY_OPEN	ORA-06511	Ouverture d'un curseur déjà ouvert.
DUP_VAL_ON_INDEX	ORA-00001	Insertion d'une ligne en doublon (clé primaire).
INVALID_CURSOR	ORA-01001	Ouverture interdite sur un curseur.
INVALID_NUMBER	ORA-01722	Échec d'une conversion d'une chaîne de caractères en NUMBER.
LOGIN_DENIED	ORA-01017	Connexion incorrecte.
NO_DATA_FOUND	ORA-01403	Requête ne retournant aucun résultat.
NOT_LOGGED_ON	ORA-01012	Connexion inexistante.
PROGRAM_ERROR	ORA-06501	Problème PL/SQL interne (invitation au contact du support…).
ROWTYPE_MISMATCH	ORA-06504	Incompatibilité de types entre une variable externe et une variable PL/SQL.
SELF_IS_NULL	ORA-30625	Appel d'une méthode d'un type sur un objet NULL (extension objet).
STORAGE_ERROR	ORA-06500	Dépassement de capacité mémoire.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Référence à un indice incorrect d'une collection (*nested table* ou *varray*) ou variables de type TABLE.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	
SYS_INVALID_ROWID	ORA-01410	Échec d'une conversion d'une chaîne de caractères en ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Dépassement du délai alloué à une ressource.
TOO_MANY_ROWS	ORA-01422	Requête retournant plusieurs lignes.
VALUE_ERROR	ORA-06502	Erreur arithmétique (conversion, troncature, taille) d'un NUMBER.
ZERO_DIVIDE	ORA-01476	Division par zéro.

Le code d'erreur (SQLCODE) qui peut être récupéré par un programme d'application (Java par exemple sous JDBC), est inclus dans le numéro interne de l'erreur (pour la deuxième exception, il s'agit de -6 592).

Concernant l'erreur NO_DATA_FOUND, rappelez-vous qu'elle n'est opérationnelle qu'avec l'instruction SELECT. Une mise à jour ou une suppression (UPDATE et DELETE) d'un enregistrement inexistant ne déclenche pas l'exception. Pour gérer ces cas d'erreurs, il faut utiliser un curseur implicite et une exception utilisateur (voir la section « Utilisation du curseur implicite »).

Si vous désirez programmer une erreur qui n'apparaît pas dans cette liste (exemple : erreur référentielle pour une suppression d'un enregistrement d'une table identifiée par une clé étrangère), il faudra programmer une exception non prédéfinie (voir la section suivante).

Plusieurs erreurs

Le tableau suivant décrit une procédure qui gère deux erreurs : aucun pilote n'est associé à la compagnie de code passé en paramètre (NO_DATA_FOUND) et plusieurs pilotes le sont (TOO_MANY_ROWS). Le programme se termine correctement si la requête retourne une seule ligne (cas de la compagnie de code 'CAST').

Tableau 7-23 Deux exceptions traitées

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException1 (p_comp IN VARCHAR2) IS`
`  var1 Pilote.nom%TYPE;`
`BEGIN`
`  SELECT nom INTO var1 FROM Pilote`
`  WHERE comp = p_comp;`
`  DBMS_OUTPUT.PUT_LINE('Le pilote de la compagnie '`
`  || p_comp || ' est ' || var1);`
	Requête déclenchant potentiellement deux exceptions prévues.

`EXCEPTION`
`  WHEN NO_DATA_FOUND THEN`
`  DBMS_OUTPUT.PUT_LINE('La compagnie ' ||`
`  p_comp || ' n''a aucun pilote!');`
	Aucun résultat renvoyé.

`  WHEN TOO_MANY_ROWS THEN`
`  DBMS_OUTPUT.PUT_LINE('La compagnie ' ||`
`  p_comp || ' a plusieurs pilotes!');`
`END;`
	Plusieurs résultats renvoyés.

La trace de l'exécution de cette procédure est la suivante :

```
SQL> EXECUTE procException1('AF');
```

```
La compagnie AF a plusieurs pilotes!
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE procException1('RIEN');
La compagnie RIEN n'a aucun pilote!
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE procException1('CAST');
Le pilote de la compagnie CAST est Thierry Millan
Procédure PL/SQL terminée avec succès.
```

Si une autre erreur se produit, en l'absence de la directive OTHERS dans le bloc d'exceptions, le programme se termine anormalement en renvoyant l'erreur en question. Dans notre exemple, seule une erreur interne pourrait éventuellement se produire (PROGRAM_ERROR, STORAGE_ERROR, TIMEOUT_ON_RESOURCE).

Même erreur sur différentes instructions

Le tableau 7-24 décrit une procédure qui gère deux fois l'erreur non trouvée (NO_DATA_FOUND) sur deux requêtes distinctes. La première requête extrait le nom du pilote de code passé en paramètre. La deuxième extrait le nom du pilote ayant un nombre d'heures de vol égal à celui passé en paramètre. Le programme se termine correctement si les deux requêtes ne retournent qu'un seul enregistrement.

La directive OTHERS permet d'afficher en clair une autre erreur déclenchée par une des deux requêtes (ici notamment TOO_MANY_ROWS qui n'est pas prise en compte). Notez ici l'utilisation des deux variables d'Oracle : SQLERRM qui contient le message en clair de l'erreur et SQLCODE le code associé.

La trace de l'exécution de cette procédure est la suivante :

```
SQL> EXECUTE procException2('PL-1', 1000);
Le pilote de  PL-1 est Gilles Laborde
Le pilote ayant 1000 heures est Florence Périssel
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE procException2('PL-0', 2450);
Pas de pilote de brevet : PL-0
Procédure PL/SQL terminée avec succès.
```

Dans cette procédure, une erreur sur la première requête fait sortir le programme (après avoir traité l'exception) et de ce fait la deuxième requête n'est pas évaluée. Pour cela, il est intéressant d'utiliser des blocs imbriqués pour poursuivre le traitement après avoir traité une ou plusieurs exceptions.

Tableau 7-24 Une exception traitée pour deux instructions

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException2`
`(p_brevet IN VARCHAR2, p_heures IN NUMBER) IS`
`var1 Pilote.nom%TYPE;`
`requete NUMBER := 1;`
`BEGIN`
`SELECT nom INTO var1 FROM Pilote`
`WHERE brevet = p_brevet;`
`DBMS_OUTPUT.PUT_LINE('Le pilote de ' ||`
`p_brevet || ' est ' || var1);`
	Requêtes déclenchant potentiellement une exception prévue.

`requete := 2;`
`SELECT nom INTO var1 FROM Pilote`
`WHERE nbHVol = p_heures;`
`DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' ||`
`p_heures || ' heures est ' || var1);`	

`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`IF requete = 1 THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote de brevet : '`
`|| p_brevet);`
`ELSE`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote ayant ce`
`nombre d''heures de vol : ' || p_heures);`
`END IF;`
	Aucun résultat. Traitement pour savoir quelle requête a déclenché l'exception.

`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM || ' (' || SQLCODE || ')');`
`END;`
	Autre erreur.

Imbrication de blocs d'erreurs

Le tableau suivant décrit une procédure qui inclut un bloc d'exceptions imbriqué au code principal. Ce mécanisme permet de poursuivre l'exécution après qu'Oracle a levé une exception. Dans cette procédure, les deux requêtes sont évaluées indépendamment du résultat retourné par chacune d'elles.

L'exécution suivante de cette procédure déclenche les deux exceptions. Le message d'erreur est contrôlé par le dernier cas d'exception, il ne s'agit pas d'une interruption anormale du programme.

```
SQL> EXECUTE procException3('PL-0', 2450);
Pas de pilote de brevet : PL-0
Erreur d'Oracle ORA-01422: l'extraction exacte ramène plus que le nombre
de lignes demandé (-1422)
```

Tableau 7-25 Bloc d'exceptions imbriqué

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException3`
`(p_brevet IN VARCHAR2, p_heures IN NUMBER) IS`
`var1 Pilote.nom%TYPE;`
`BEGIN`	

`BEGIN`
`SELECT nom INTO var1 FROM Pilote`
`WHERE brevet = p_brevet;`
`DBMS_OUTPUT.PUT_LINE('Le pilote de ' || p_brevet`
`|| ' est ' || var1);`
	Bloc imbriqué.

`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote de brevet : '`
`|| p_brevet);`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM || ' (' || SQLCODE || ')');`
`END;`
	Gestion des exceptions de la première requête.

`SELECT nom INTO var1 FROM Pilote`
`WHERE nbHVol = p_heures ;`
`DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' || p_heures ||`
`' heures est ' || var1);`
	Suite du traitement.

`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote ayant ce nombre`
`d''heures de vol : ' || p_heures);`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM || ' (' || SQLCODE || ')');`
`END;`
	Gestion des exceptions de la deuxième requête.

Exception utilisateur

Il est possible de définir ses propres exceptions. Cela pour bénéficier des blocs de traitements d'erreurs et aborder une erreur applicative comme une erreur renvoyée par la base. Cela améliore et facilite la maintenance et l'évolution des programmes car les erreurs applicatives peuvent très facilement être propagées aux programmes appelants.

Déclaration

La déclaration du nom de l'exception doit se trouver dans la section déclarative du sous-programme.

```
nomException EXCEPTION;
```

Déclenchement

Une exception utilisateur ne sera pas levée de la même manière qu'une exception interne. Le programme doit explicitement dérouter le traitement vers le bloc des exceptions par la directive RAISE. L'instruction RAISE permet également de déclencher des exceptions prédéfinies.

Dans notre exemple, programmons les deux exceptions suivantes :

- erreur_piloteTropJeune qui va interdire l'insertion des pilotes ayant moins de 200 heures de vol ;
- erreur_piloteTropExpérimenté qui va interdire l'insertion des pilotes ayant plus de 20 000 heures de vol.

Le tableau suivant décrit cette procédure qui intercepte ces deux erreurs applicatives :

Tableau 7-26 Exceptions utilisateur

Code PL/SQL	Commentaires

`CREATE PROCEDURE saisiePilote`
`(p_brevet IN VARCHAR2,p_nom IN VARCHAR2,`
`p_nbHVol IN NUMBER, p_comp IN VARCHAR2) IS`
`erreur_piloteTropJeune EXCEPTION;`
`erreur_piloteTropExpérimenté EXCEPTION;`
	Déclaration de l'exception.

`BEGIN`
`INSERT INTO Pilote (brevet,nom,nbHVol,comp)`
`VALUES (p_brevet,p_nom,p_nbHVol,p_comp);`
`IF p_nbHVol < 200 THEN RAISE erreur_piloteTropJeune;`
`END IF;`
`IF p_nbHVol > 20000 THEN`
`RAISE erreur_piloteTropExpérimenté;`
`END IF;`
`COMMIT;`
	Corps du traitement (validation).

`EXCEPTION`
`WHEN erreur_piloteTropJeune THEN`
`ROLLBACK;`
`DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote manque`
`d''expérience');`
`WHEN erreur_piloteTropExpérimenté THEN`
`ROLLBACK;`
`DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote a`
`trop d''expérience');`
`WHEN OTHERS THEN`
`ROLLBACK;`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM`
`|| '(' || SQLCODE || ')');`
`END;`
	Gestion de l'exception.
	
	Gestion des autres exceptions.

La trace de l'exécution de cette procédure où l'on passe des valeurs en paramètres qui déclenchent les deux exceptions est la suivante.

```
SQL> EXECUTE saisiePilote('PL-9','Tuffery Michel', 199, 'AF');
Désolé, le pilote manque d'expérience
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE saisiePilote('PL-9','Tuffery Michel', 20001, 'AF');
Désolé, le pilote a trop d'expérience
Procédure PL/SQL terminée avec succès.
```

Utilisation du curseur implicite

Étudiés dans le chapitre 6, les curseurs implicites permettent ici de pallier le fait qu'Oracle ne lève pas l'exception NO_DATA_FOUND pour les instructions UPDATE et DELETE. Ce qui est en théorie valable (aucune action sur la base peut ne pas être considérée comme une erreur), en pratique il est utile de connaître le code retour de l'instruction de mise à jour.

Considérons à nouveau la procédure détruitCompagnie en prenant en compte l'erreur applicative erreur_compagnieInexistante qui intercepte une suppression non réalisée. Le test du curseur implicite de cette instruction déclenche l'exception utilisateur associée.

Tableau 7-27 Utilisation du curseur implicite

Code PL/SQL	Commentaires

`CREATE OR REPLACE PROCEDURE détruitCompagnie (p_comp IN VARCHAR2) IS`
`erreur_ilResteUnPilote EXCEPTION;`
`PRAGMA EXCEPTION_INIT(erreur_ilResteUnPilote , -2292);`
`erreur_compagnieInexistante EXCEPTION;`
	Déclaration des exceptions.

`BEGIN`
`DELETE FROM Compagnie WHERE comp = p_comp;`
`IF SQL%NOTFOUND THEN`
`RAISE erreur_compagnieInexistante;`
`END IF;`
`COMMIT;`
`DBMS_OUTPUT.PUT_LINE('Compagnie '||p_comp|| ' détruite.');`
	Corps du traitement (validation).

`EXCEPTION`
`WHEN erreur_ilResteUnPilote THEN`
`DBMS_OUTPUT.PUT_LINE ('Désolé, il reste encore un pilote à la compagnie ' || p_comp);`
`WHEN erreur_compagnieInexistante THEN`
`DBMS_OUTPUT.PUT_LINE ('La compagnie ' || p_comp || ' n''existe pas dans la base!');`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM || '(' || SQLCODE || ')');`
`END;`
	Gestion des exceptions.
	
	Gestion des autres exceptions.

L'exécution de cette procédure où l'on passe un code compagnie inexistant fait maintenant dérouler la section des exceptions.

```
SQL> EXECUTE détruitCompagnie('rien');
La compagnie rien n'existe pas dans la base!
```

Exception interne non prédéfinie

Pour intercepter une erreur Oracle qui n'a pas été prédéfinie (pour laquelle Oracle n'a pas associé de nom), et être ainsi plus précis qu'avec la clause `OTHERS`, il faut utiliser la directive `PRAGMA EXCEPTION_INIT`. Celle-ci indique au compilateur d'associer un nom d'exception, que vous aurez choisi, à un code d'erreur Oracle existant. La directive `PRAGMA` (appelée aussi pseudo-instruction) est un mot-clé signifiant que l'instruction est destinée au compilateur (elle n'est pas traitée au moment de l'exécution).

Déclaration

Deux commandes sont nécessaires dans la section déclarative à la mise en œuvre de ce mécanisme : déclarer le nom de l'exception et associer cet identificateur à l'erreur Oracle.

```
nomException EXCEPTION;
PRAGMA EXCEPTION_INIT(nomException, numéroErreurOracle);
```

Pour connaître le numéro de l'erreur qui vous intéresse, consultez la liste des erreurs dans la documentation d'Oracle (*Error Messages* qui est classée par numéros croissants et non pas par fonctionnalités). Cherchez par exemple les entrées correspondant à *foreign key* dans le chapitre des erreurs `ORA-02100 to ORA-04099`.

Vous pouvez aussi écrire un bloc PL/SQL qui programme volontairement l'erreur pour voir sous SQL*Plus le numéro qu'Oracle renvoie.

Déclenchement

Une exception non prédéfinie sera levée de la même manière qu'une exception prédéfinie, à savoir suite à une instruction SQL pour laquelle le serveur aura renvoyé une erreur.

Considérons les deux tables suivantes. La colonne `comp` de la table `Pilote` est clé étrangère vers la table `Compagnie`. Programmons une procédure qui supprime une compagnie de code passé en paramètre.

Figure 7-7 *Deux tables*

`Compagnie`

comp	ville	nomComp
AF	Paris	Air France
SING	Singapour	Singapore AL
CAST	Blagnac	Castanet AL
EJET	**Dublin**	**Easy Jet**

à détruire

`Pilote`

brevet	nom	nbHVol	comp
PL-1	Gilles Laborde	2450	AF
PL-2	Frédéric D'Almeyda	900	AF
PL-3	Florence Périssel	1000	SING
PL-4	Thierry Millan	2450	CAST
PL-5	Christine Royo	200	AF
PL-6	Aurélia Ente	2450	SING

Le tableau suivant décrit la procédure détruitCompagnie qui intercepte l'erreur ORA-02292: enregistrement fils existant. Il s'agit de contrôler le programme si la compagnie à détruire possède encore des pilotes référencés dans la table Pilote.

Tableau 7-28 Exception interne non prédéfinie

Code PL/SQL	Commentaires

`CREATE PROCEDURE détruitCompagnie(p_comp IN VARCHAR2) IS`
`  erreur_ilResteUnPilote EXCEPTION;`
`  PRAGMA EXCEPTION_INIT(erreur_ilResteUnPilote , -2292);`
	Déclaration de l'exception.

`BEGIN`
`  DELETE FROM Compagnie WHERE comp = p_comp;`
`  COMMIT;`
`  DBMS_OUTPUT.PUT_LINE ('Compagnie ' || p_comp ||`
`  ' détruite.');`
	Corps du traitement (validation).

`EXCEPTION`
`  WHEN erreur_ilResteUnPilote THEN`
`  DBMS_OUTPUT.PUT_LINE ('Désolé, il reste encore un`
`  pilote à la compagnie ' || p_comp);`
	Gestion de l'exception.

`  WHEN OTHERS THEN`
`  DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM`
`  || '(' || SQLCODE || ')');`
	Gestion des autres exceptions.

`END;`	

La trace de l'exécution de cette procédure est la suivante. Notez que si on applique cette procédure à une compagnie inexistante, le programme se termine normalement sans passer dans la section des exceptions.

```
SQL> EXECUTE détruitCompagnie('AF');
Désolé, il reste encore un pilote à la compagnie AF
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE détruitCompagnie('EJET');
Compagnie EJET détruite.
Procédure PL/SQL terminée avec succès.
```

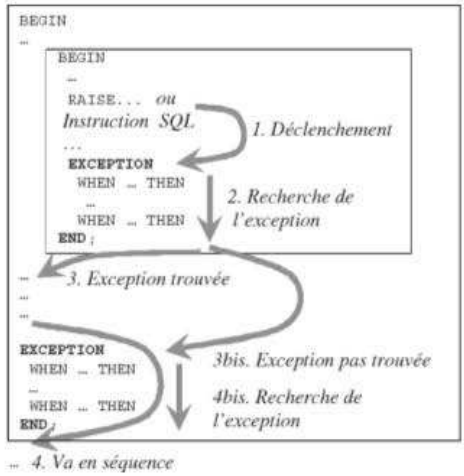
Propagation d'une exception

Nous avons vu jusqu'à présent que lorsqu'un bloc EXCEPTION traite correctement une exception (car il existe soit une entrée dans le bloc correspondant à l'exception, soit l'entrée OTHERS), l'exécution du traitement se poursuit en séquences après l'instruction END du bloc EXCEPTION.

Mécanisme général

Si une exception se déclenche mais qu'aucune entrée n'est prévue dans le bloc `EXCEPTION` (et qu'il n'existe pas l'entrée `OTHERS`), l'exception se propage successivement au niveau des blocs `EXCEPTION` contenus dans le code appelant (ou englobant), jusqu'à ce qu'une entrée corresponde (ou l'entrée `OTHERS`). Si aucun des blocs d'erreurs ne peut traiter l'exception, le programme principal se termine anormalement en renvoyant une erreur. La figure suivante illustre ce processus :

Figure 7-8 *Propagation des exceptions*

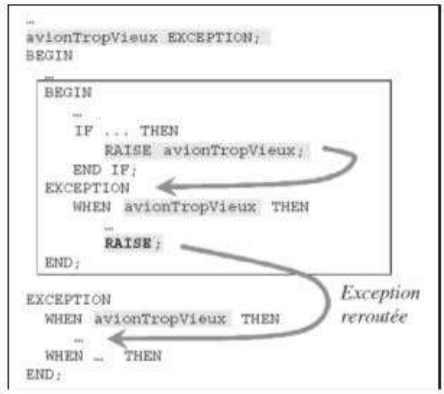


Notez que lorsque l'exception se propage à un bloc englobant, les actions exécutables restantes de ce bloc sont ignorées. Un des avantages de ce mécanisme est de pouvoir gérer des exceptions spécifiques dans leur propre bloc, tout en laissant le bloc englobant gérer les exceptions plus générales.

Exceptions reroutées (reraise)

Il est, dans certains cas, intéressant d'exécuter plusieurs blocs d'erreurs pour la même exception. On déclenche plusieurs fois l'exception (*exception reraised*). Le principe consiste à utiliser la directive `RAISE` sans spécifier le nom de l'exception à traiter de nouveau (voir la figure suivante dans laquelle l'exception `avionTropVieux` est reroutée). Si l'exception ne peut être traitée dans le bloc englobant, alors elle est propagée à l'environnement appelant ou englobant (voir section précédente).

Figure 7-9 Exception reroutée



Procédure RAISE_APPLICATION ERROR

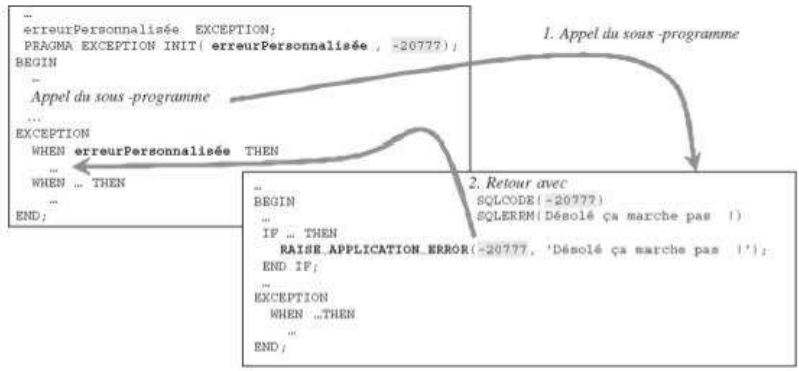
La procédure RAISE_APPLICATION_ERROR permet de définir ses propres messages et codes d'erreurs. Cette procédure évite le renvoi d'exceptions non traitées car le numéro d'erreur (inclus dans RAISE_APPLICATION_ERROR) sera communiqué à l'environnement appelant.

```
RAISE_APPLICATION_ERROR(numéroErreur, message [, {TRUE | FALSE}]);
```

- *numéroErreur* : valeur définie pour l'utilisateur pour l'exception, comprise entre -20 000 et -20 999.
- *message* : chaîne de caractères (max 2 048 octets) décrivant l'erreur.
- TRUE | FALSE : booléen facultatif. TRUE pour positionne l'erreur dans une pile si plusieurs exceptions doivent être propagées en cascade., FALSE par défaut remplace toutes les erreurs précédentes dans la pile.

La procédure RAISE_APPLICATION_ERROR peut être utilisée dans le code ou dans la section de traitement des exceptions d'un programme PL/SQL. L'appel à la procédure RAISE_APPLICATION_ERROR interrompt le programme et retourne le numéro et le message d'erreur qui peuvent être récupérés par l'environnement englobant (variables SQLCODE et SQLERRM). La figure suivante illustre ce mécanisme qui est aussi programmable dans le cas des déclencheurs.

Figure 7-10 *Utilisation de RAISE_APPLICATION_ERROR*



Déclencheurs

Les déclencheurs (*triggers*) existent depuis la version 6 d'Oracle. Ils sont compilables depuis la version 7.3 (auparavant, ils étaient évalués lors de l'exécution). Depuis la version 8, il existe un nouveau type de déclencheur (`INSTEAD OF`) qui permet la mise à jour de vues multitables.

La plupart des déclencheurs peuvent être vus comme des programmes résidents associés à un événement particulier (insertion, modification d'une ou de plusieurs colonnes, suppression) sur une table (ou une vue). Une table (ou une vue) peut « héberger » plusieurs déclencheurs ou aucun. Nous verrons qu'il existe d'autres types de déclencheurs que ceux associés à une table (ou à une vue) afin de répondre à des événements qui ne concernent pas les données.

À la différence des sous-programmes, l'exécution d'un déclencheur n'est pas explicitement opérée par une commande ou dans un programme, c'est l'événement de mise à jour de la table (ou de la vue) qui exécute automatiquement le code programmé dans le déclencheur. On dit que le déclencheur « se déclenche » (l'anglais le traduit mieux : *fired trigger*).

La majorité des déclencheurs sont programmés en PL/SQL (langage très bien adapté à la manipulation des objets Oracle), mais il est possible d'utiliser un autre langage (C ou Java par exemple).

À quoi sert un déclencheur ?

Un déclencheur permet de :

- Programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables. Par exemple, la condition : *une compagnie ne fait voler un pilote que s'il a totalisé plus de 60 heures de vol dans les 2 derniers mois sur le type*

d'appareil du vol en question, ne pourra pas être programmée par une contrainte et nécessitera l'utilisation d'un déclencheur.

- Déporter des contraintes au niveau du serveur et alléger ainsi la programmation client.
- Renforcer des aspects de sécurité et d'audit.
- Programmer l'intégrité référentielle et la réplication dans des architectures distribuées avec l'utilisation de liens de bases de données (*database links*).

Généralités

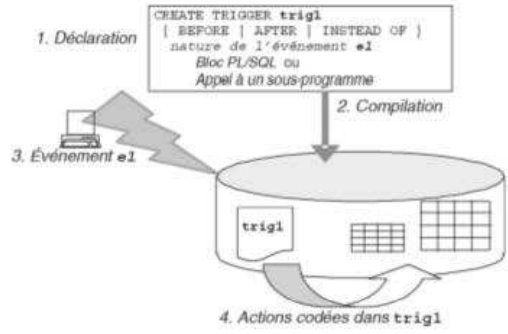
Les événements déclencheurs peuvent être :

- une instruction `INSERT`, `UPDATE`, ou `DELETE` sur une table (ou une vue). On parle de déclencheurs LMD ;
- une instruction concernant des structures (`CREATE`, `ALTER`, `DROP`) et les prérogatives (`GRANT` et `REVOKE`) sur un objet (table, index, séquence, etc.). On parle de déclencheurs LDD ;
- le démarrage ou l'arrêt de la base (*startup* ou *shutdown*), une erreur spécifique (`NO_DATA_FOUND`, `DUP_VAL_ON_INDEX`, etc.), une connexion ou une déconnexion d'un utilisateur. On parle de déclencheurs d'instances.

Mécanisme général

La figure suivante illustre les étapes à suivre pour mettre en œuvre un déclencheur. Il faut d'abord le coder (comme un sous-programme), puis le compiler (il sera stocké ainsi en base). Par la suite, au cours du temps, et si le déclencheur est actif (nous verrons qu'il est possible de désactiver un déclencheur même s'il est compilé), chaque événement (qui caractérise le déclencheur) aura pour conséquence son exécution.

Figure 7-11 *Mécanisme des déclencheurs*



Syntaxe

Pour pouvoir créer un déclencheur dans votre schéma, vous devez disposer du privilège CREATE TRIGGER (qui est inclus dans le rôle RESOURCE mais pas dans CONNECT). Pour créer un déclencheur dans un autre schéma, le privilège CREATE ANY TRIGGER est requis. En plus de ces conditions, pour fabriquer un déclencheur d'instances, il faut détenir le privilège ADMINISTER DATABASE TRIGGER.

Un déclencheur est composé de trois parties : la description de l'événement traqué, une éventuelle restriction (condition) et la description de l'action à réaliser lorsque l'événement se produit. La syntaxe de création d'un déclencheur est la suivante :

```
CREATE [OR REPLACE] TRIGGER [schéma.] nomDéclencheur
 { BEFORE | AFTER | INSTEAD OF }
 {    { DELETE | INSERT | UPDATE [OF col1 [,col2]…] }
  [OR { DELETE | INSERT | UPDATE [OF col1 [,col2]…] }]…
  ON { [schéma.] nomTable | nomVue }
  [REFERENCING
       { OLD [AS] nomVieux | NEW [AS] nomNew | PARENT [AS] nomParent }
       [ OLD [AS] nomVieux | NEW [AS] nomNew | PARENT [AS] nomParent]… ]
  [FOR EACH ROW] }
 |
 { événementBase [OR événementBase]… |
   actionStructureBase [OR actionStructureBase]… }
   ON { [schéma.] SCHEMA | DATABASE } }
```

```
[WHEN ( condition ) ]
```

```
{   Bloc PL/SQL (DECLARE variables BEGIN instructions END ; )
  | CALL nomSousProgramme(paramètres) }
```

Les options de cette commande sont les suivantes :

- BEFORE | AFTER | INSTEAD OF précise la chronologie entre l'action à réaliser par le déclencheur LMD et la réalisation de l'événement (exemple BEFORE INSERT programmera l'exécution du déclencheur avant de réaliser l'insertion).
- DELETE | INSERT | UPDATE précise la nature de l'événement pour les déclencheurs LMD.
- ON {[schéma.] nomTable | nomVue} spécifie la table, ou la vue, associée au déclencheur LMD.
- REFERENCING permet de renommer des variables.
- FOR EACH ROW différencie les déclencheurs LMD au niveau ligne ou au niveau état.
- *événementBase* identifie la nature d'un déclencheur d'instance (STARTUP ou SHUTDOWN pour exécuter le déclencheur au démarrage ou à l'arrêt de la base), d'un déclencheur

d'erreurs (SERVERERROR ou SUSPEND pour exécuter le déclencheur dans le cas d'une erreur particulière ou quand une transaction est suspendue) ou d'un déclencheur de connexion (LOGON ou LOGOFF pour exécuter le déclencheur lors de la connexion ou de la déconnexion à la base).

- *actionStructureBase* spécifie la nature d'un déclencheur LDD (CREATE, ALTER, DROP, etc. pour exécuter par exemple le déclencheur lors de la création, la modification ou la suppression d'un objet de la base).
- ON {[schéma.]SCHEMA | DATABASE} précise le champ d'application du déclencheur (de type LDD, erreur ou connexion). Utilisez DATABASE pour les déclencheurs qui s'exécutent pour quiconque commence l'événement, ou SCHEMA pour les déclencheurs qui ne doivent s'exécuter que dans le schéma courant.
- WHEN conditionne l'exécution du déclencheur.

Il est conseillé de limiter la taille (partie instructions) d'un déclencheur à soixante lignes de code PL/SQL (la taille d'un déclencheur ne peut excéder 32 ko). Pour contourner cette limitation, appeler des sous-programmes dans le code du déclencheur.

Un déclencheur ne peut valider aucune transaction, ainsi les instructions suivantes sont interdites : COMMIT, ROLLBACK, SAVEPOINT, et SET CONSTRAINT.

Attention à ne pas créer de déclencheurs récursifs (exemple d'un déclencheur qui exécute une instruction lançant elle-même le déclencheur ou deux déclencheurs s'appelant en cascade jusqu'à l'occupation de toute la mémoire réservée).

Étudions à présent plus précisément les caractéristiques de chaque type de déclencheur qu'il est possible de programmer.

Déclencheurs LMD

Pour ce type de déclencheurs, l'événement à déterminer est une mise à jour particulière de la base (ajout, modification ou suppression dans une table ou une vue). L'exécution est dépendante ou non du nombre de lignes concernées par l'événement. On programme un déclencheur de lignes (*row trigger*) quand on désire exécuter autant de fois le déclencheur qu'il y a de lignes concernées par une mise à jour. Si on désire exécuter une seule fois le déclencheur quel que soit le nombre de lignes concernées, on utilisera un déclencheur d'état (*statement trigger*). La directive FOR EACH ROW distingue ces deux familles de déclencheurs.

Dans l'exemple d'une table *t1* ayant cinq enregistrements, si on programme un déclencheur de niveau ligne avec l'événement AFTER DELETE, et qu'on lance DELETE FROM t1, le déclencheur exécutera cinq fois ses instructions (une fois après chaque suppression). Le tableau suivant explique ce mécanisme.

Tableau 7-29 Exécutions des déclencheurs LMD

Nature de l'événement	État (*statement trigger*) sans `FOR EACH ROW`	Ligne (*row trigger*) avec `FOR EACH ROW`
`BEFORE`	Exécution une fois avant la mise à jour.	Exécution avant chaque ligne mise à jour.
`AFTER`	Exécution une fois après la mise à jour.	Exécution après chaque ligne mise à jour.

Déclencheurs de lignes (row triggers)

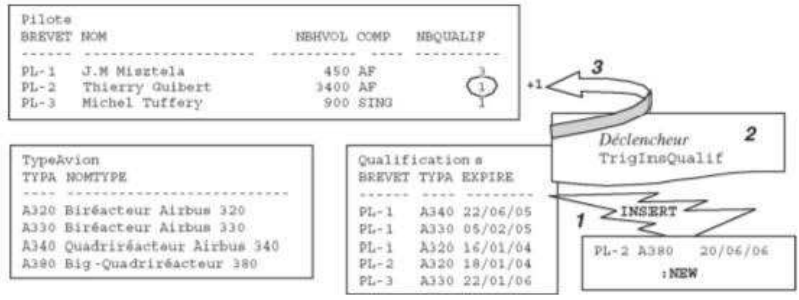
Un déclencheur de lignes est déclaré avec la directive `FOR EACH ROW`. Ce n'est que dans ce type de déclencheur qu'on a accès aux anciennes valeurs et aux nouvelles valeurs des colonnes de la ligne affectée par la mise à jour prévue par l'événement.

Quand utiliser la directive :NEW ?

Considérons l'exemple suivant, et programmons la règle de gestion *tout pilote ne peut être qualifié sur plus de trois types d'appareils*. Ici, il s'agit d'assurer la cohérence entre la valeur de la colonne `nbQualif` de la table `Pilote` et les lignes de la table `Qualifications`.

Programmons le déclencheur `TrigInsQualif` qui surveille les insertions arrivant sur la table `Qualifications` et incrémente de 1 la colonne `nbQualif` pour le pilote concerné, ou refuse l'insertion pour le pilote ayant déjà trois qualifications (cas du pilote de code 'PL-1' dans la figure suivante).

Figure 7-12 *Principe du déclencheur TrigInsQualif*



L'événement déclencheur est ici `BEFORE INSERT` car il faudra s'assurer, avant de faire l'insertion, que le pilote n'est pas déjà qualifié sur trois types d'appareils. On utilise un déclencheur `FOR EACH ROW` car on désire qu'il s'exécute autant de fois qu'il y a de lignes concernées par l'événement déclencheur. S'il se produit une insertion multiple de type `INSERT INTO Qualifications SELECT...`, on préfère lancer plusieurs fois le déclencheur.

Chaque enregistrement qui tente d'être ajouté dans la table Qualifications est désigné par :NEW au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

Le code minimal de ce déclencheur (on ne prend pas en compte l'éventuelle erreur du SELECT ne renvoyant aucun pilote) est décrit dans le tableau suivant :

Tableau 7-30 Déclencheur avant insertion

Code PL/SQL	Commentaires

`CREATE TRIGGER TrigInsQualif`
`BEFORE INSERT ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`DECLARE`
`v_compteur Pilote.nbHVol%TYPE;`
`v_nom Pilote.nom%TYPE;`
	Déclaration des variables locales.

`BEGIN`
`SELECT nbQualif, nom INTO v_compteur, v_nom`
`FROM Pilote WHERE brevet = :NEW.brevet;`
`IF v_compteur < 3 THEN`
`UPDATE Pilote SET nbQualif = nbQualif + 1`
`WHERE brevet = :NEW.brevet;`
`ELSE`
	Corps du déclencheur. Extraction et mise à jour du pilote concerné par la qualification.

`RAISE_APPLICATION_ERROR(-20100, 'Le pilote '` `|| v_nom || ' a déjà 3 qualifications!');`	Renvoi d'une erreur utilisateur.

`END IF;`
`END;`
`/`	

Le test de ce déclencheur peut être réalisé sous SQL*Plus comme le montre la trace suivante. On retrouve l'erreur utilisateur qui est levée en premier.

Tableau 7-31 Test du déclencheur

Événement déclencheur	Sortie SQL*Plus

`SQL> INSERT INTO Qualifications`
`VALUES ('PL-2', 'A380',`
`'20-06-2006');`
	`1 ligne créée.`
	`SQL> SELECT * FROM Pilote;`
	`BREVET NOM NBHVOL COMP NBQUALIF`
	`------ ------------ ------ ---- ---------`
	`PL-1 J.M Misztela 450 AF 3`
	`PL-2 Thierry Guibert 3400 AF 2`
	`PL-3 Michel Tuffery 900 SING 1`

`SQL> INSERT INTO Qualifications`
`VALUES ('PL-1', 'A380',`
`'20-06-2006');`
	`ERREUR à la ligne 1 :`
	`ORA-20100: Le pilote J.M Misztela a déjà 3 qualifications!`
	`ORA-06512: à "SOUTOU.TRIGINSQUALIF", ligne 9`
	`ORA-04088: erreur lors d'exécution du déclencheur 'SOUTOU.TRIGINSQUALIF'`

Comme l'instruction RAISE, la procédure RAISE_APPLICATION_ERROR passe par la section EXCEPTION (s'il en existe une) avant de terminer le déclencheur. En conséquence, si vous utilisez aussi une section exception dans le même bloc, il faut forcer la sortie du déclencheur par la directive RAISE pour ne pas perdre le message d'erreur et surtout ne pas réaliser la mise à jour de la base.

Afin d'illustrer cette importante remarque, ajoutons une section EXCEPTION au précédent exemple. Cette section vérifiera l'existence du pilote.

Tableau 7-32 Déclencheur avec exceptions

Code PL/SQL	Commentaires

`CREATE TRIGGER TrigInsQualif`
`BEFORE INSERT ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`DECLARE`
`v_compteur Pilote.nbHVol%TYPE;`
`v_nom Pilote.nom%TYPE;`
	Déclaration des variables locales.

`BEGIN`
`SELECT nbQualif, nom INTO v_compteur, v_nom`
`FROM Pilote WHERE brevet = :NEW.brevet;`
`IF v_compteur < 3 THEN`
`UPDATE Pilote SET nbQualif = nbQualif + 1`
`WHERE brevet = :NEW.brevet;`
`ELSE`
`RAISE_APPLICATION_ERROR(-20100, 'Le pilote ' ||`
`:NEW.brevet || ' a déjà 3 qualifications!');`
`END IF;`
`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`RAISE_APPLICATION_ERROR(-20101, 'Pas de pilote`
`de code brevet ' || :NEW.brevet);`
`WHEN OTHERS THEN`
`RAISE;`
`END;`
`/`
	Corps du déclencheur. Extraction et mise à jour du pilote concerné par la qualification.
	
	Renvoie une erreur utilisateur et annule les mises à jour.
	
	Si erreur au SELECT.
	
	Retour de l'erreur courante.

Le test d'erreur de ce déclencheur sous SQL*Plus est illustré dans le tableau suivant :

Tableau 7-33 Test du déclencheur avec exceptions

Événement déclencheur	Sortie SQL*Plus

`SQL> INSERT INTO Qualifications`
`VALUES ('Qui?', 'A380',`
`'20-06-2006');`
	`ERREUR à la ligne 1 :`
	`ORA-20101: Pas de pilote de code brevet Qui?`
	`ORA-06512: à "SOUTOU.TRIGINSQUALIF", ligne 13`
	`ORA-04088: erreur lors d'exécution du déclencheur 'SOUTOU.TRIGINSQUALIF'`

Pour que la cohérence soit plus complète, il faudrait aussi programmer le déclencheur qui décrémente la valeur de la colonne `nbQualif` pour chaque pilote concerné par une suppression de lignes dans la table `Qualifications`. Il faut raisonner ici sur la directive `:OLD`.

Quand utiliser la directive :OLD ?

Chaque enregistrement qui tente d'être supprimé d'une table qui inclut un déclencheur de type `DELETE FOR EACH ROW`, est désigné par `:OLD` au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

Programmons le déclencheur `TrigDelQualif` qui surveille les suppressions de la table `Qualifications` et décrémente de 1 la colonne `nbQualif` pour le pilote concerné par la suppression de sa qualification.

L'événement déclencheur est ici `AFTER DELETE` car il faudra s'assurer que la suppression n'est pas entravée par d'éventuelles contraintes référentielles. On utilise un déclencheur `FOR EACH ROW`, car s'il se produit une suppression de toute la table (`DELETE FROM Qualifications;`) on exécutera autant de fois le déclencheur qu'il y a de lignes supprimées.

Le code minimal de ce déclencheur (on ne prend pas en compte le fait qu'il n'existe plus de pilote de ce code brevet) est décrit dans le tableau suivant :

Tableau 7-34 Déclencheur après suppression

Code PL/SQL	Commentaires

`CREATE TRIGGER TrigDelQualif`
`AFTER DELETE ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`BEGIN`
`  UPDATE Pilote SET nbQualif = nbQualif - 1`
`  WHERE brevet = :OLD.brevet;`
`END;`
`/`
	Corps du déclencheur. Mise à jour du pilote concerné par la suppression.

En considérant les données initiales des tables, le test de ce déclencheur sous SQL*Plus est le suivant :

Tableau 7-35 Test du déclencheur

Événement déclencheur	Sortie SQL*Plus

`SQL> DELETE FROM Qualifications`
`WHERE typa = 'A320';`
	`2 ligne(s) supprimée(s).`
	`SQL> SELECT * FROM Pilote;`
	`BREVET NOM NBHVOL COMP NBQUALIF`
	`------ --------------- ------ ---- --------`
	`PL-1 J.M Misztela 450 AF 2`
	`PL-2 Thierry Guibert 3400 AF 0`
	`PL-3 Michel Tuffery 900 SING 1`

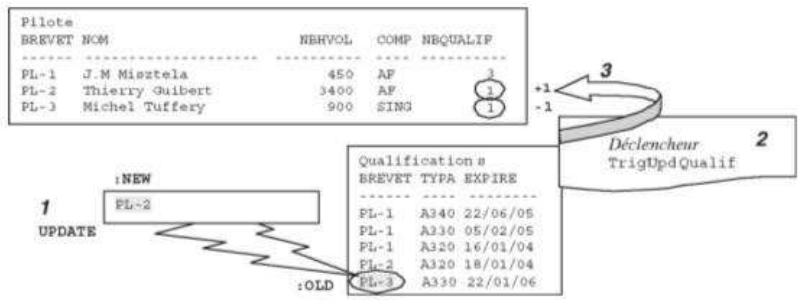
Pour tester le fait que l'instruction UPDATE n'affecte aucune ligne, il faudrait utiliser un curseur implicite (SQL%FOUND) et une erreur utilisateur (voir le paragraphe « Utilisation du curseur implicite » dans la section « Exceptions »).

Quand utiliser à la fois les directives :NEW et :OLD ?

Seuls les déclencheurs de type UPDATE FOR EACH ROW permettent de manipuler à la fois les directives :NEW et :OLD. En effet, la mise à jour d'une ligne dans une table fait intervenir une nouvelle donnée qui en remplace une ancienne. L'accès aux anciennes valeurs se fera par la notation pointée du pseudo-enregistrement :OLD. L'accès aux nouvelles valeurs se fera par :NEW.

La figure suivante illustre ce mécanisme dans le cas de la modification de la colonne brevet du dernier enregistrement de la table Qualifications. Le déclencheur doit programmer deux mises à jour dans la table Pilote.

Figure 7-13 *Principe du déclencheur* TrigUpdQualif



L'événement déclencheur est ici AFTER UPDATE car il faudra s'assurer que la suppression n'est pas entravée par d'éventuelles contraintes référentielles. Le code minimal de ce déclencheur (on ne prend pas en compte le fait qu'un pilote n'ait pas pu être mis à jour) est décrit dans le tableau 7-33.

En considérant les données présentées à la figure précédente, le test de ce déclencheur sous SQL*Plus est présenté dans le tableau 7-34.

Synthèse à propos de :NEW et :OLD

Le tableau 7-35 résume les valeurs contenues dans les pseudo-enregistrements :OLD et :NEW pour les déclencheurs FOR EACH ROW. Retenez que seuls les déclencheurs UPDATE peuvent manipuler à bon escient les deux types de directives.

Attention, Oracle ne vous prévient pas à la compilation que vous utilisez une variable :OLD dans un déclencheur INSERT (ou :NEW dans un déclencheur DELETE), et qui sera toujours nulle.

Tableau 7-36 Déclencheur après modification

Code PL/SQL	Commentaires

`CREATE TRIGGER TrigUpdQualif`
`AFTER UPDATE OF brevet ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`DECLARE`
`v_compteur Pilote.nbHVol%TYPE;`
`v_nom Pilote.nom%TYPE;`
	Déclaration des variables locales.

`BEGIN`
`SELECT nbQualif, nom INTO v_compteur, v_nom`
`FROM Pilote WHERE brevet = :NEW.brevet;`
	Corps du déclencheur.

`IF v_compteur < 3 THEN`
`UPDATE Pilote SET nbQualif = nbQualif + 1`
`WHERE brevet = :NEW.brevet;`
`UPDATE Pilote SET nbQualif = nbQualif - 1`
`WHERE brevet = :OLD.brevet;`
	Mise à jour des pilotes concernés par la modification de la qualification.

`ELSE`
`RAISE_APPLICATION_ERROR(-20100, 'Le pilote ' ||`
`:NEW.brevet || ' a déjà 3 qualifications!');`
	Renvoi d'une erreur utilisateur.

`END IF;`
`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`RAISE_APPLICATION_ERROR(-20101, 'Pas de pilote`
`de code brevet ' || :NEW.brevet);`
	Renvoi d'une erreur utilisateur.

`WHEN OTHERS THEN` `RAISE;`	Retour de l'erreur courante.
`END;` `/`	

Tableau 7-37 Test du déclencheur

Événement déclencheur	Sortie SQL*Plus

`SQL> UPDATE Qualifications`
`SET brevet = 'PL-2'`
`WHERE brevet = 'PL-3'`
`AND typa = 'A330';`
	`1 ligne mise à jour.`
	`SQL> SELECT * FROM Pilote;`
	`BREVET NOM NBHVOL COMP NBQUALIF`
	`------ --------------- ------ ---- --------`
	`PL-1 J.M Misztela 450 AF 3`
	`PL-2` `Thierry Guibert 3400 AF` **`2`**
	`PL-3` `Michel Tuffery 900 SING` **`0`**

Condition dans un déclencheur (WHEN)

Il est possible de restreindre l'exécution d'un déclencheur en amont du code de ce dernier. La clause WHEN, placée avant le corps du déclencheur, permet de programmer cette condition. Si celle-ci est réalisée pour l'enregistrement concerné par l'événement, le déclencheur s'exécute. Dans le cas inverse, le déclencheur n'a aucun effet.

Tableau 7-38 Valeurs de :OLD et :NEW

Nature de l'événement	:OLD.*colonne*	:NEW.*colonne*
INSERT	NULL	Nouvelle valeur.
UPDATE	Ancienne valeur.	Nouvelle valeur.
DELETE	Ancienne valeur.	NULL

La condition contenue dans la clause WHEN doit être une expression SQL, et ne peut inclure de requêtes ni de fonctions PL/SQL.

Restreignons par exemple la règle de gestion que nous avons programmée jusqu'à présent – *tout pilote ne peut être qualifié sur plus de trois types d'appareils* – aux appareils de type 'A320', 'A330' ou 'A340'. Il suffira de modifier les en-têtes des trois déclencheurs de la manière suivante (exemple pour le déclencheur d'insertion). Notez que dans la condition WHEN, les « pseudo-enregistrements » NEW et OLD s'écrivent sans le symbole :.

Tableau 7-39 Déclencheur conditionnel

Code PL/SQL	Commentaires

`CREATE OR REPLACE TRIGGER TrigInsQualif`
` BEFORE INSERT ON Qualifications`
` FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`WHEN (NEW.typa = 'A320' OR NEW.typa = 'A340'` `OR NEW.typa = 'A330')`	Condition de déclenchement.

`DECLARE`
` …`
`BEGIN`
` …`
`END;`
`/`
	Corps du déclencheur.

Le tableau suivant présente un jeu de test pour ce déclencheur.

Tableau 7-40 Test du déclencheur

Événement déclencheur	Événement non déclencheur
`INSERT INTO Qualifications` ` VALUES ('PL-2','A340','20-06-2006');`	`INSERT INTO Qualifications` ` VALUES ('PL-2','A380','20-06-2006');`

Corrélation de noms (REFERENCING)

La clause REFERENCING permet de mettre en corrélation les noms des pseudo-enregistrements (:OLD et :NEW) avec des noms de variables. La directive PARENT concerne les déclencheurs

portant sur des collections *nested tables* (extension objet). La condition écrite dans la directive WHEN peut utiliser les noms de variables corrélées.

Utilisons cette clause sur le précédent déclencheur pour renommer le pseudo-enregistrement :NEW par la variable nouveau. Cet enregistrement est opérationnel dans la clause WHEN et dans le corps du déclencheur.

Tableau 7-41 Corrélation de noms

Code PL/SQL	Commentaires

`CREATE OR REPLACE TRIGGER TrigInsQualif`
`  BEFORE INSERT ON Qualifications`
`  REFERENCING NEW AS nouveau`
`  FOR EACH ROW`
`  WHEN (nouveau.typa = 'A320' OR`
`  nouveau.typa='A340' OR nouveau.typa='A330')`
`DECLARE`
	Événement déclencheur.
	Renomme :NEW en nouveau.

`DECLARE`
`  …`
`BEGIN`
`  …`
`  …WHERE brevet = :nouveau.brevet;`
`  …`
`END;`
	Corps du déclencheur.

Regroupements d'événements

Des événements (INSERT, UPDATE ou DELETE) peuvent être regroupés au sein d'un même déclencheur s'ils sont de même type (BEFORE ou AFTER). Ainsi, un seul déclencheur est à coder et des instructions dans le corps du déclencheur permettent de retrouver la nature de l'événement déclencheur :

- IF (INSERTING) THEN.. exécute un bloc dans le cas d'une insertion ;
- IF (UPDATING('*colonne*')) THEN... exécute un bloc dans le cas de la modification d'une colonne ;
- IF (DELETING) THEN... exécute un bloc en cas d'une suppression.

Utilisons cette fonctionnalité pour regrouper les déclencheurs de type AFTER que nous avons programmés.

Si vous regroupez ainsi plusieurs déclencheurs mono-événements en un déclencheur multi-événements, pensez à supprimer les déclencheurs mono-événements (DROP TRIGGER...) pour ne pas programmer involontairement plusieurs fois la même action par l'intermédiaire des différents déclencheurs existants.

Tableau 7-42 Regroupement d'événements

Code PL/SQL	Commentaires

`CREATE OR REPLACE TRIGGER TrigDelUpdQualif`
`AFTER DELETE OR UPDATE OF brevet ON Qualifications`
`FOR EACH ROW`
	Regroupement de deux événements déclencheurs.

`DECLARE`
`…`
`BEGIN`
`IF (DELETING) THEN`
`…`
	Bloc exécuté en cas de DELETE.

`ELSIF (UPDATING('brevet')) THEN`
`…`
`END IF;`
`END;`
	Bloc exécuté en cas de UPDATE de la colonne brevet.

Déclencheurs d'état (statement triggers)

Un déclencheur d'état est déclaré sans la directive `FOR EACH ROW`. Il n'est pas possible d'avoir accès aux valeurs des lignes mises à jour par l'événement. Le raisonnement de tels déclencheurs porte donc sur la globalité de la table et non sur chaque enregistrement particulier.

Dans le cadre de notre exemple, programmons le déclencheur `périodeOKQualifs` qui interdit toute mise à jour sur la table `Qualifications` pendant les week-ends. Quel que soit le nombre de lignes concernées par un événement, le déclencheur s'exécutera une seule fois avant chaque événement sur la table `Qualifications`.

Tableau 7-43 Déclencheur d'état

Code PL/SQL	Commentaires

`CREATE TRIGGER périodeOKQualifs`
`BEFORE DELETE OR UPDATE`
`OR INSERT ON Qualifications`
	Déclaration des événements déclencheurs.

`BEGIN`
`IF TO_CHAR(SYSDATE,'DAY') IN`
`('SAMEDI', 'DIMANCHE') THEN`
`RAISE_APPLICATION_ERROR(-20102, 'Désolé pas`
`de mises à jour des qualifs le week-end.');`
`END IF ;`
`END;`
`/`
	Bloc exécuté avant chaque mise à jour de la table Qualifications.

Pour chaque actualisation de la table, le déclencheur renvoie le résultat suivant sous SQL*Plus (ça tombe bien, j'ai écrit ce code un dimanche…).

Tableau 7-44 Test du déclencheur

Événements déclencheurs	Sortie SQL*Plus
UPDATE Qualifications SET …	ERREUR à la ligne 1 : ORA-20102: Désolé pas de mises à jour des qualifs le week-end.
INSERT INTO Qualifications VALUES…	ORA-06512: à "SOUTOU.PÉRIODEOKQUALIFS", ligne 3
DELETE FROM Qualifications…	ORA-04088: erreur lors d'exécution du déclencheur 'SOUTOU.PÉRIODEOKQUALIFS'

Déclencheurs INSTEAD OF

Un déclencheur INSTEAD OF permet de mettre à jour une vue multitable qui ne pouvait être modifiée directement par INSERT, UPDATE ou DELETE (voir chapitre 5). Nous verrons que seulement certaines vues multitables peuvent être modifiables par l'intermédiaire de ce type de déclencheur. L'expression *instead of* est explicite : le déclencheur programmera des actions *au lieu* d'insérer, de modifier ou de supprimer une vue.

La version 7 d'Oracle n'offrait pas cette possibilité. Ce mécanisme intéresse particulièrement les bases de données réparties par liens (*database links*). Il est désormais plus facile de modifier des informations provenant de différentes tables par ce type de déclencheur.

Caractéristiques

Les déclencheurs INSTEAD OF :

- font intervenir la clause FOR EACH ROW ;
- ne s'utilisent que sur des vues ;
- ne font pas intervenir les options BEFORE et AFTER.

L'option de contrôle (WITH CHECK OPTION) d'une vue n'est pas vérifiée lors d'un événement (ajout, modification ou suppression) si un déclencheur INSTEAD OF est programmé sur cet événement. Le corps du déclencheur doit donc explicitement prendre en compte la contrainte.

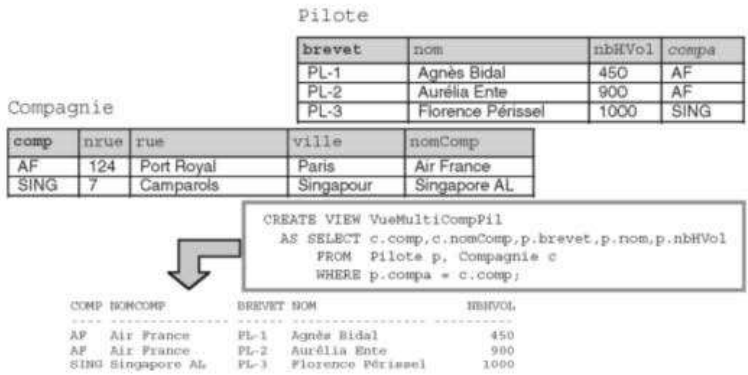
Il n'est pas possible de spécifier une liste de colonnes dans un déclencheur INSTEAD OF UPDATE, le déclencheur s'exécutera quelle que soit la colonne modifiée.

Il n'est pas possible d'utiliser la clause WHEN dans un déclencheur INSTEAD OF.

Exemple

Considérons la vue VueMultiCompPil résultant d'une jointure entre les tables Compagnie et Pilote. Nous avons vu au chapitre 5 que cette vue n'était pas modifiable sous SQL. Nous allons programmer un déclencheur INSTEAD OF qui va permettre de la changer de manière transparente.

Figure 7-14 *Vue multitable à modifier*



Le déclencheur qui gère les insertions dans la vue est chargé d'insérer, à chaque nouvel ajout, un enregistrement dans chacune des deux tables.

Tableau 7-45 Déclencheur INSTEAD OF

Code PL/SQL	Commentaires

`CREATE TRIGGER TrigAulieuInsererVue`
`  INSTEAD OF INSERT ON VueMultiCompPil`
`  FOR EACH ROW`
`DECLARE`
`  v_comp NUMBER := 0;`
`  v_pil  NUMBER := 0;`
	Déclaration de la substitution de l'événement déclencheur.

`BEGIN`
`  SELECT COUNT(*) INTO v_pil  FROM Pilote`
`  WHERE brevet = :NEW.brevet;`
`  SELECT COUNT(*) INTO v_comp FROM Compagnie`
`  WHERE comp = :NEW.comp;`
`  IF v_pil > 0 AND v_comp > 0 THEN`
	Corps du déclencheur.

`  RAISE_APPLICATION_ERROR(-20102, 'Le pilote et la compagnie existent déjà!');` `  ELSE`	Cas d'erreur.

`  IF v_comp = 0 THEN`
`  INSERT INTO Compagnie VALUES`
`  (:NEW.comp,NULL,NULL,NULL,:NEW.nomComp);`
`  END IF;`
	Ajout dans la table `Compagnie`.

`  IF v_pil = 0 THEN`
`  INSERT INTO Pilote VALUES`
`  (:NEW.brevet,:NEW.nom,:NEW.nbHVol,:NEW.comp);`
`  END IF;`
`  END IF;`
`END;`
`/`
	Ajout dans la table `Pilote`.

Pour chaque mise à jour de la vue, le déclencheur insérera un pilote, une compagnie ou les deux, suivant l'existence du pilote et de la compagnie. L'erreur programmée dans le déclencheur concerne le cas pour lequel le pilote et la compagnie existent déjà dans la base. Le tableau suivant décrit une trace de test de ce déclencheur :

Tableau 7-46 Test du déclencheur

Événement déclencheur	Vérification sous SQL*Plus
INSERT INTO VueMultiCompPil VALUES ('AERI', 'Aéris Toulouse', 'PL-4', 'Pascal Larrazet', 5600); 1 ligne créée.	SQL> SELECT * FROM Pilote; BREVET NOM NBHVOL COMP ------ ---------- ---------- ---- PL-1 Agnès Bidal 450 AF … PL-4 Pascal Larrazet 5600 AERI SQL> SELECT * FROM Compagnie; COMP NRUE RUE VILLE NOMCOMP ---- ---- ---------- --------- -------------- - SING 7 Camparols Singapour Singapore AL AF 124 Port Royal Paris Air France AERI Aéris Toulouse

```
SQL> SELECT * FROM VueMultiCompPil;
COMP NOMCOMP          BREVET NOM                  NBHVOL
---- ---------------- ------ ---------- ----------
AF   Air France       PL-1   Agnès Bidal            450
AF   Air France       PL-2   Aurélia Ente           900
SING Singapore AL     PL-3   Florence Périssel     1000
AERI Aéris Toulouse   PL-4   Pascal Larrazet       5600
```

Transactions autonomes

Un déclencheur peut former une transaction (utilisation possible de COMMIT, ROLLBACK et SAVEPOINT) si la directive PRAGMA AUTONOMOUS_TRANSACTION est employée dans la partie déclarative (voir figure 7-1). Une fois démarrée, une telle transaction est autonome et indépendante (voir le début de ce chapitre). Elle ne partage aucun verrou ou ressource, et ne dépend d'aucune transaction principale. Ces déclencheurs autonomes peuvent en outre exécuter des instructions du LDD (CREATE, DROP ou ALTER) en utilisant des fonctions natives de PL/SQL pour le SQL dynamique (voir la section suivante).

Les modifications faites lors d'une transaction autonome deviennent visibles par les autres transactions quand la transaction autonome se termine. Une transaction autonome doit se terminer explicitement par une validation ou une invalidation. Si une exception n'est pas traitée en sortie, la transaction est invalidée.

Déclencheurs LDD

Étudions à présent les déclencheurs gérant les événements liés à la modification de la structure de la base et non plus à la modification des données de la base. Les options BEFORE et AFTER sont disponibles comme le montre la syntaxe générale suivante. La directive DATABASE précise que le déclencheur peut s'exécuter pour quiconque lance l'événement. La directive SCHEMA indique que le déclencheur ne peut s'exécuter que dans le schéma courant.

```
CREATE [OR REPLACE] TRIGGER [schéma.] nomDéclencheur
   BEFORE | AFTER { actionStructureBase [OR actionStructureBase]… }
   ON { [schéma.] SCHEMA | DATABASE } }
   Bloc PL/SQL (variables BEGIN instructions END ; )
   | CALL nomSousProgramme(paramètres) }
```

Les principales actions sur la structure de la base prise en compte sont :

- ALTER pour déclencher en cas de modification d'un objet du dictionnaire (table, index, séquence, etc.).
- COMMENT pour déclencher en cas d'ajout d'un commentaire.
- CREATE pour déclencher en cas d'ajout d'un objet du dictionnaire.
- DROP pour déclencher en cas de suppression d'un objet du dictionnaire.
- GRANT pour déclencher en cas d'affectation de privilège à un autre utilisateur ou rôle.
- RENAME pour déclencher en cas de changement de nom d'un objet du dictionnaire.
- REVOKE pour déclencher en cas de révocation de privilège d'un autre utilisateur ou rôle.

Le déclencheur suivant interdit toute suppression d'objet, dans le schéma *soutou*, se produisant un lundi ou un vendredi.

Tableau 7-47 Déclencheur LDD

Code PL/SQL	Commentaires
`CREATE TRIGGER surveilleDROPSoutou` `  BEFORE DROP ON soutou.SCHEMA`	Événement déclencheur LDD.

`BEGIN`
` IF TO_CHAR(SYSDATE,'DAY') IN`
`  ('LUNDI  ','VENDREDI') THEN`
`  RAISE_APPLICATION_ERROR(-20104,'Désolé pas de destruction ce jour..') ;`
` END IF ;`
`END;`
`/`
	Corps du déclencheur.
	
	Retour d'une erreur.

Déclencheurs d'instances

Le démarrage ou l'arrêt de la base (*startup* ou *shutdown*), une erreur spécifique (NO_DATA_FOUND, DUP_VAL_ON_INDEX, etc.), une connexion ou une déconnexion d'un utilisateur

peuvent être autant d'événements pris en compte par un déclencheur d'instances. Les événements précités sont programmés à l'aide des mots-clés STARTUP, SHUTDOWN, SUSPEND, SERVERERROR, LOGON, LOGOFF, dans la syntaxe suivante :

```
CREATE [OR REPLACE] TRIGGER [schéma.] nomDéclencheur
   BEFORE | AFTER { événementBase [OR événementBase]… }
   ON { [schéma.] SCHEMA | DATABASE } }
   Bloc PL/SQL (variables BEGIN instructions END ; )
   | CALL nomSousProgramme(paramètres) }
```

Les restrictions régissant ces déclencheurs sont les suivantes :

- Seule l'option AFTER est valable pour LOGON, STARTUP, SERVERERROR, et SUSPEND.
- Seule l'option BEFORE est valable pour LOGOFF et SHUTDOWN.
- Les options AFTER STARTUP et BEFORE SHUTDOWN s'appliquent seulement sur les déclencheurs de type DATABASE.

Les erreurs ORA-01403, ORA-01422, ORA-01423, ORA-01034 et ORA-04030 ne sont pas prises en compte par l'événement SERVERERROR.

Le déclencheur suivant insère une ligne dans une table qui indique l'utilisateur et l'heure de déconnexion (sous SQL*Plus, via un programme d'application, etc.). On suppose la table Trace(événement VARCHAR2(100)) créée.

Tableau 7-48 Déclencheurs d'instances

Code PL/SQL	Commentaires
`CREATE TRIGGER espionDéconnexion` `  BEFORE LOGOFF ON DATABASE`	Événement déclencheur.

`BEGIN`
`  INSERT INTO Trace VALUES (USER ||`
`  ' déconnexion le ' ||`
`  TO_CHAR(SYSDATE,'DD-MM-YYYY HH24:MI:SS'));`
`END;`
`/`
	Corps du déclencheur exécuté à chaque déconnexion.

Appels de sous-programmes

Un déclencheur peut appeler directement par CALL (ou dans son corps) un sous-programme PL/SQL ou une procédure externe écrite en C, C++ ou Java. Le tableau suivant décrit quelques appels de sous-programmes qu'il est possible de coder dans un déclencheur (quel que soit son type). On suppose la procédure PL/SQL suivante existante.

```
CREATE PROCEDURE sousProgDéclencheur(param IN VARCHAR2) IS
```

```
BEGIN
   INSERT INTO Trace VALUES ('sousProgDéclencheur (' || param || ')' );
END sousProgDéclencheur;
```

Tableau 7-49 Appels de sous-programmes dans un déclencheur

Code PL/SQL	Commentaires

`CREATE TRIGGER espionConnexion`
`AFTER LOGON ON DATABASE`
`CALL soutou.sousProgDéclencheur(SYSDATE)`
`/`
	Appel direct d'une procédure PL/SQL.

`CREATE TRIGGER TrigDelTrace`
`AFTER SERVERERROR ON soutou.SCHEMA`
`BEGIN`
`sousProgDéclencheur('Une erreur s''est produite');`
`END;`
`/`
	Appel dans le corps du déclencheur d'une procédure PL/SQL.

`CREATE TRIGGER Ex_trig_Java`
`AFTER DELETE ON Compagnie FOR EACH ROW`
`BEGIN`
`DeuxièmeExemple_affiche(:OLD.nomcomp);`
`END;`
`/`
	Appel dans le corps du déclencheur d'un sous-programme Java (voir chapitre 11).

Gestion des déclencheurs

Un déclencheur est actif, comme une contrainte, dès sa création. Il est possible de le désactiver, de le supprimer ou de le réactiver à la demande grâce aux instructions `ALTER TRIGGER` (pour agir sur un déclencheur en particulier) ou `ALTER TABLE` (pour agir sur tous les déclencheurs d'une table en même temps). Le tableau suivant résume les commandes SQL nécessaires à la gestion des déclencheurs :

Tableau 7-50 Gestion des déclencheurs

SQL	Commentaires
ALTER TRIGGER *nomDéclencheur* **COMPILE**;	Recompilation d'un déclencheur.
ALTER TRIGGER *nomDéclencheur* **DISABLE**;	Désactivation d'un déclencheur.
ALTER TABLE *nomTable* **DISABLE ALL TRIGGERS**;	Désactivation de tous les déclencheurs d'une table.
ALTER TRIGGER *nomDéclencheur* **ENABLE**;	Réactivation d'un déclencheur.
ALTER TABLE *nomTable* **ENABLE ALL TRIGGERS**;	Réactivation de tous les déclencheurs d'une table.
DROP TRIGGER *nomDéclencheur*;	Suppression d'un déclencheur.

Ordre d'exécution

La séquence d'exécution des déclencheurs est théoriquement la suivante. En pratique certaines exécutions peuvent ne pas suivre cet ordre !

- tous les déclencheurs d'état BEFORE ;
- analyse de toutes les lignes affectées par l'instruction SQL ;
- tous les déclencheurs de lignes BEFORE ;
- verrouillage, modification et vérification des contraintes d'intégrité ;
- tous les déclencheurs de lignes AFTER ;
- vérification des contraintes différées ;
- tous les déclencheurs d'état AFTER.

Tables mutantes

Il est, en principe, interdit de manipuler la table sur laquelle se porte le déclencheur dans le corps du déclencheur lui-même. Oracle parle de *mutating tables* (erreur : `ORA-04091: table ... en mutation, déclencheur/fonction ne peut la voir`).

Cette restriction concerne les déclencheurs de lignes (`FOR EACH ROW`), et les déclencheurs d'état qui sont exécutés via la directive `DELETE CASCADE`. Les vues modifiables par des déclencheurs `INSTEAD OF` ne sont pas considérées comme des tables mutantes.

L'exemple suivant décrit la programmation d'un déclencheur qui compte les lignes d'une table après chaque nouvelle insertion. L'erreur n'est pas soulignée à la compilation mais est levée dès la première insertion.

Tableau 7-51 Déclencheur (table mutante)

Code PL/SQL	Trace SQL*Plus

`CREATE OR REPLACE TRIGGER TrigMutant1`
`  AFTER INSERT ON Trace FOR EACH ROW`
`DECLARE`
`  v_nombre NUMBER;`
`BEGIN`
`  SELECT COUNT(*) INTO v_nombre`
`  FROM Trace;`
`  DBMS_OUTPUT.PUT_LINE`
`  ('Nombre de traces : '`
`  || v_nombre);`
`END;`
`/`
	`INSERT INTO Trace VALUES ('Insertion`
	`le ' || TO_CHAR(SYSDATE,'DD-MM-YYYY`
	`HH24:MI:SS'));`
	`ERREUR à la ligne 1 :`
	`ORA-04091: table SOUTOU.TRACE en`
	`mutation, déclencheur/fonction ne`
	`peut la voir`
	`ORA-06512: à "SOUTOU.TRIGMUTANT1",`
	`ligne 4`
	`ORA-04088: erreur lors d'exécution du`
	`déclencheur 'SOUTOU.TRIGMUTANT1'`

Une solution consiste, dans ce cas, à programmer le même code dans un déclencheur d'état (il suffit d'enlever la clause FOR EACH ROW). Pour des cas plus complexes, il fallait programmer (avant la version 11*g*) plusieurs déclencheurs dont le code et les variables sont définis dans un paquetage à part.

Activation et désactivation

Bien qu'il soit possible de désactiver ou réactiver un déclencheur par ALTER TRIGGER ou ALTER TABLE, tout déclencheur créé était de fait actif (*enable*) avant la version 11*g*. Depuis la version 12, un déclencheur peut être désactivé dès sa création. En l'absence de la directive DISABLE ou en présence de la directive ENABLE (qui est appliquée par défaut), tout déclencheur sera actif dès sa création. La syntaxe simplifiée qui permet la déclaration d'un tel déclencheur est la suivante :

```
CREATE [ OR REPLACE ] TRIGGER [schéma.] nomTrigger
  ... { ENABLE | DISABLE }
  BEGIN
  ...
END;
/
```

Ordre d'exécution (FOLLOWS)

Bien qu'Oracle permette que plusieurs déclencheurs soient programmés pour le même événement, il n'était pas possible de connaître l'ordre dans lequel les déclencheurs s'exécutaient. Depuis la version 11*g*, la directive FOLLOWS précise cet ordre. La syntaxe simplifiée qui permet la déclaration d'un tel déclencheur est la suivante :

```
CREATE [ OR REPLACE ] TRIGGER [schéma.] nomTrigger
  ... FOLLOWS [schéma.] nomTriggerQuiSexecuteAvant ...
BEGIN
  ...
END;
/
```

L'exemple suivant déclare deux déclencheurs portant sur le même événement (avant chaque insertion de la table TypeAvion).

```
CREATE OR REPLACE TRIGGER Trig_follows_1
BEFORE INSERT ON TypeAvion FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('Trig_follows_1 en exécution');
END;
/
```

```
CREATE OR REPLACE TRIGGER Trig_follows_2
BEFORE INSERT ON TypeAvion FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('Trig_follows_2 en exécution');
END;
/
```

Si on désire que le premier déclencheur se lance toujours après le deuxième, il faut recompiler ce dernier de la manière suivante (il n'est pas possible de faire une référence avant, à savoir déclarer un déclencheur référençant un déclencheur inexistant) :

```
CREATE OR REPLACE TRIGGER Trig_follows_1
BEFORE INSERT ON TypeAvion FOR EACH ROW
FOLLOWS Trig_follows_2
BEGIN
  DBMS_OUTPUT.put_line('Trig_follows_1 en exécution');
END;
/
```

Déclencheur composé

Un déclencheur composé (*compound trigger*) permet de programmer plusieurs blocs pour différents événements. Cette technique est particulièrement utile pour pallier le problème des tables mutantes.

Le corps d'un déclencheur composé est constitué d'une éventuelle section de variables globales et d'au moins un (jusqu'à quatre) blocs PL/SQL correspondant à la chronologie des événements au niveau de la ligne ou de l'état. Les blocs peuvent contenir des variables locales. Chaque section peut utiliser les directives INSERTING, UPDATING et DELETING. La syntaxe simplifiée qui permet la déclaration d'un tel déclencheur est la suivante :

```
CREATE [ OR REPLACE ] TRIGGER [schéma.] nomTrigger
  FOR  { DELETE | INSERT | UPDATE
     [ OF col1 [, col2 ]... ] }
  [ OR { DELETE | INSERT | UPDATE [ OF col1 [, col2]... ] ]...
   ON { [ schéma. ] nomTable  |  [ schéma. ] nomVue }
 COMPOUND TRIGGER
  -- Variables globales
  BEFORE STATEMENT IS
  BEGIN
    ...
  END BEFORE STATEMENT;
  AFTER STATEMENT IS
  BEGIN
```

```
    ...
  END AFTER STATEMENT;
  BEFORE EACH ROW IS
  BEGIN
    ...
  END BEFORE EACH ROW;
  AFTER EACH ROW IS
  BEGIN
    ...
  END AFTER EACH ROW;
END nomTrigger;
/
```

Le déclencheur suivant traque les ajouts et les suppressions dans la table `TypeAvionBis`. Un tableau fait office de variable globale et permet de tracer le code après une insertion multiple et une suppression collective.

```
CREATE TRIGGER TrigCompose FOR DELETE OR INSERT ON TypeAvionBis
 COMPOUND TRIGGER
 TYPE typav_tytab IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
 tab  typav_tytab;
 i    NUMBER := 0;
  BEFORE STATEMENT IS
  BEGIN
      i := i+1;
      CASE
      WHEN INSERTING THEN tab(i) := 'Avant insertion STATEMENT';
      WHEN DELETING THEN tab(i) := 'Avant suppression STATEMENT';
      END CASE;
  END BEFORE STATEMENT;
  AFTER STATEMENT IS
  BEGIN
      i := i+1;
      tab(i) := 'Après STATEMENT';
      FOR i IN 1 .. tab.last LOOP
         DBMS_OUTPUT.PUT_LINE(tab(i));
      END LOOP;
  END AFTER STATEMENT;
  BEFORE EACH ROW IS
  BEGIN
     i := i+1;
     tab(i) := 'Avant événement niveau ligne';
  END BEFORE EACH ROW;
  AFTER EACH ROW IS
```

```
  BEGIN
     i := i+1;
     CASE
      WHEN INSERTING THEN tab(i) := :NEW.typa||' inséré';
      WHEN DELETING THEN tab(i) := :NEW.typa||' supprimé';
     END CASE;
  END AFTER EACH ROW;
END TrigCompose;
/
```

En considérant les tables et les données suivantes :

```
CREATE TABLE TypeAvion    (typa VARCHAR2(5),nomtype VARCHAR2(30));
CREATE TABLE TypeAvionBis (typa VARCHAR2(5),nomtype VARCHAR2(30));
INSERT INTO TypeAvion VALUES ('A320','Biréacteur Airbus 320');
INSERT INTO TypeAvion VALUES ('A340','Quadriréacteur Airbus 340');
```

La trace de l'insertion multiple dans la table concernée par le déclencheur décrit la chronologie des actions.

```
SQL> INSERT INTO TypeAvionBis SELECT * FROM TypeAvion;
Avant insertion STATEMENT
Avant événement niveau ligne
A320 inséré
Avant événement niveau ligne
A340 inséré
Après STATEMENT
2 ligne(s) créée(s).
```

Les principales restrictions régissant ce type de déclencheurs sont les suivantes :

- Seuls les déclencheurs LMD peuvent être composés.
- Il n'est possible de déclarer un bloc d'exceptions que dans une section particulière (aucune exception globale n'est permise).
- Seule la section `BEFORE EACH ROW` peut modifier une valeur de type `NEW`.

Résolution au problème des tables mutantes

Le déclencheur composé convient parfaitement pour résoudre le problème des tables mutantes. Les sections `BEFORE STATEMENT` et `AFTER STATEMENT` permettent de manipuler la table concernée par le déclencheur comme le montre l'exemple de la section précédente et sur les données

courantes. L'événement déclencheur à programmer était AFTER INSERT qui se traduit avec le déclencheur composé de type FOR INSERT contenant les sections BEFORE STATEMENT pour interroger la table et AFTER EACH ROW pour définir l'action.

Tableau 7-52 Déclencheur composé pour résoudre une table mutante

Code PL/SQL	Trace SQL*Plus
CREATE OR REPLACE TRIGGER TrigMutant1 **FOR INSERT** ON Trace **COMPOUND TRIGGER** v_nombre NUMBER; BEFORE STATEMENT IS BEGIN SELECT COUNT(*) INTO v_nombre FROM Trace; END BEFORE STATEMENT; AFTER EACH ROW IS BEGIN DBMS_OUTPUT.PUT_LINE ('Nombre de traces : '||v_nombre); END AFTER EACH ROW; END; /	SQL>INSERT INTO Trace SELECT nomtype FROM TypeAvion; Nombre de traces : 0 Nombre de traces : 0 2 ligne(s) créée(s). SQL>INSERT INTO Trace VALUES ('Insertion le '||TO_CHAR(SYSDATE,'DD- MM-YYYY HH24:MI:SS')); Nombre de traces : 2 1 ligne créée. SQL>SELECT * FROM Trace; EVENEMENT ---------- Biréacteur Airbus 320 Quadriréacteur Airbus 340 Insertion le 23-11-2007 17:52:27

SQL dynamique

PL/SQL inclut un aspect dynamique : en plus des directives SQL (LMD, LID), il est possible de construire automatiquement des instructions SQL du LDD et du LCD (CREATE, DROP, GRANT et REVOKE) ainsi que des instructions relatives aux sessions (ALTER SESSION, SET ROLE, etc.).

L'utilisation de SQL dynamique dans un sous-programme PL/SQL permet de paramétrer des instructions SQL au niveau de l'organisation même de la commande. Par exemple, il sera possible de créer une table dont le nom passera en paramètre et ayant un nombre variable de colonnes. Il sera aussi permis de construire automatiquement une requête SQL en fonction des choix d'un utilisateur. En plus des ordres simples, on pourra également paramétrer une suite d'instructions dans un bloc PL/SQL ou l'appel d'un sous-programme.

Une instruction SQL dynamique est stockée en tant que chaîne de caractères qui sera évaluée à l'exécution et non à la compilation (en opposition aux instructions SQL statiques qui peuplent la majorité des sous-programmes).

Les instructions suivantes ne peuvent pas être prises en compte par un ordre SQL dynamique : CLOSE, DECLARE, DESCRIBE, EXECUTE, FETCH, OPEN, PREPARE, SET, WHENEVER.

Comme il n'y a pas de phase préalable de compilation, il n'y a pas de vérification des privilèges sur les objets avant l'exécution des instructions SQL qui sont construites dynamiquement.

Classification

Les ordres SQL dynamiques peuvent être classifiés en quatre familles :

- instructions SQL (sauf les requêtes) sans variables hôtes ;
- instructions SQL, (sauf les requêtes) avec un nombre connu de variables hôtes ;
- instructions SQL (et requêtes) avec un nombre connu de colonnes (dans le SELECT) et de variables hôtes ;
- instructions SQL (et requêtes) avec un nombre inconnu de colonnes (dans le SELECT) et de variables hôtes.

Ces familles d'instructions s'incluent entre elles : la famille 2 comprend la famille 1 ; la famille 3 comprend la famille 2 ; la famille 4 comprend la famille 3. Le tableau suivant décrit des exemples d'instructions en classifiant ces dernières.

Tableau 7-53 Instructions SQL dynamique sous PL/SQL

Instruction	Famille
`'DELETE FROM Avion WHERE nbHVol > 1000'` `'GRANT SELECT ON Avion TO teste, soutou'`	1. Utilisation de EXECUTE IMMEDIATE.

`'INSERT INTO Avion`
`VALUES (:variable, :variable, …)'`
`'DELETE FROM Avion WHERE immat = :variable`
`'`
	2. Utilisation de EXECUTE IMMEDIATE avec USING.

`'SELECT comp, MAX(nbHVol) FROM Pilote`
`GROUP BY comp'`
`'SELECT brevet, nbHVol FROM Pilote`
`WHERE comp = :variable '`
	3. Utilisation de EXECUTE IMMEDIATE avec USING et INTO.

`'INSERT INTO Avion ( Inconnu )`
`VALUES ( Inconnu )'`
`'SELECT Inconnu FROM Pilote`
`WHERE comp = :variable '`
	4. Utilisation d'un curseur variable avec OPEN, FETCH, et CLOSE.

Utilisez au maximum les variables hôtes (*bind variables*) pour vous prémunir d'éventuelles injections de code SQL et ne pas dégrader les performances (dues à la régénération systématique de plans d'exécution pour différentes valeurs en paramètres).

Il existe deux méthodes pour construire des instructions : le paquetage DBMS_SQL et la méthode native (EXECUTE IMMEDIATE). Avec ces deux approches, vous pourrez utiliser tout type de donnée, même les collections et les objets (*user define*). Depuis la version 11*g*, les ordres SQL peuvent être construits à l'aide de CLOB (jusqu'à 32 Ko).

Utilisation de EXECUTE IMMEDIATE

La syntaxe de l'instruction PL/SQL EXECUTE IMMEDIATE, qui permet d'exécuter des ordres SQL dynamiques des trois premières classifications, est la suivante.

```
EXECUTE IMMEDIATE chaîneCaractères
[INTO { variable [,variable]… | typeRecord} ]
[USING [IN | OUT | IN OUT] paramètre
    [, [IN | OUT | IN OUT] paramètre]…]
[{RETURNING | RETURN} INTO paramètre [,paramètre]…];
```

Le tableau ci-après décrit des exemples d'utilisations réunis dans un bloc PL/SQL :

Tableau 7-54 Utilisations de EXECUTE IMMEDIATE

Code PL/SQL	Commentaires

DECLARE
ordreSQLdynamique VARCHAR2(200);
pilote_record Pilote%ROWTYPE;
blocPLSQL VARCHAR2(500);
v_immat VARCHAR2(6) := 'F-WTSS';
v_immat2 VARCHAR2(6) := 'F-WTFG';
v_typeAv CHAR(8) := 'Concorde';
v_nbHVol NUMBER(7,2) := 3650.70;
v_comp VARCHAR2(4) := 'AF';
p_immat VARCHAR2(6) := 'F-WTSS';
v_brevet VARCHAR2(6) := 'PL-2';
	Déclaration des variables.

BEGIN EXECUTE IMMEDIATE 'DELETE FROM Avion WHERE nbHVol > 1000 '; EXECUTE IMMEDIATE 'CREATE TABLE AvionChasse (immat VARCHAR2(6), prixEuros NUMBER)';	Famille 1. Sans paramètre.

Tableau 7-54 Utilisations de EXECUTE IMMEDIATE *(suite)*

Code PL/SQL	Commentaires

`ordreSQLdynamique := 'INSERT INTO Avion VALUES (:1, :2, :3, :4)';`
`EXECUTE IMMEDIATE ordreSQLdynamique USING v_immat, v_typeAv, v_nbHVol, v_comp;`
`blocPLSQL := 'BEGIN sousProg(:p1); END;';`
`EXECUTE IMMEDIATE blocPLSQL USING v_immat;`
	Famille 2.
	Insertion paramétrée.
	
	Appel du sous-programme sousProg.

`ordreSQLdynamique := 'SELECT * FROM Pilote WHERE brevet= :v1';`
`EXECUTE IMMEDIATE ordreSQLdynamique INTO pilote_record USING v_brevet;`
`END;`
`/`
	Famille 3.
	Extraction monoligne paramétrée.

Il est bien sûr possible d'utiliser EXECUTE IMMEDIATE dans un sous-programme ou dans un programme d'application (C, C++, Java) en utilisant l'API du langage traduisant cette instruction. Par ailleurs, il est possible d'employer une section exception pour récupérer des éventuelles erreurs d'exécution.

Utilisation d'une variable curseur

Les variables curseurs (REF CURSOR), décrites dans ce chapitre, permettent de programmer les instructions SQL dynamiques les plus complexes (extraction paramétrée renvoyant plusieurs lignes par exemple). On va retrouver l'utilisation des directives OPEN, FETCH et CLOSE pour manipuler le curseur. La directive OPEN d'une variable curseur permettant de construire une instruction SQL est la suivante :

```
OPEN {variableCurseur | :variableCurseurHôte }
    FOR chaîneCaractères
   [USING paramètre[,paramètre]…];
```

Le tableau ci-après décrit la construction automatique d'une requête qui extrait plusieurs lignes. L'ouverture de la variable curseur déclenche le passage des paramètres au niveau du SELECT et dans la clause WHERE.

Le bloc suivant affiche le nom des pilotes de la compagnie de code 'AF'.

Tableau 7-55 Utilisation d'une variable curseur

Code PL/SQL	Commentaires

`DECLARE`
`  TYPE piloteCurs_type IS REF CURSOR;`
`  refCursPilote piloteCurs_type;`
`  ordreSQLdynamique VARCHAR2(200);`
`  v_1 CHAR(6) := 'brevet';`
`  v_2 CHAR(3) := 'nom';`
`  v_3 CHAR(6) := 'nbHVol';`
`  v_4 CHAR(2) := 'AF';`
`  v_brevet VARCHAR2(6);`
`  v_nom VARCHAR(20);`
`  v_nbHVol NUMBER(7,2);`
	Déclaration de la variable curseur et des autres variables.

`BEGIN`
`ordreSQLdynamique := 'SELECT ';`
`OPEN refCursPilote FOR ordreSQLdynamique`
`  || v_1 || ',' || v_2 || ',' || v_3 ||`
`  ' FROM Pilote WHERE comp = :v4' USING v_4;`
`LOOP`
`FETCH refCursPilote`
`  INTO v_brevet,v_nom,v_nbHVol;`
`EXIT WHEN refCursPilote%NOTFOUND;`
`DBMS_OUTPUT.PUT_LINE ('Pilote : ' ||v_nom);`
`END LOOP;`
`CLOSE refCursPilote ;`
`END;`
`/`
	Famille 4, requête multiligne paramétrée. Passage du paramètre.
	
	Parcours du curseur.

Nouveautés de la version 12c

Retourner des jeux de résultats (implict statement results)

Avant la version 12*c*, PL/SQL ne permettait pas de retourner simplement un jeu de résultats avec des entrées variables sans l'écriture d'une requête, puis d'itérer en utilisant `DBMS_OUTPUT.PUT_LINE` pour afficher le résultat dans l'interface SQL*Plus.

Depuis la version 12*c*, la procédure `RETURN_RESULT` du paquetage `DBMS_SQL` permet de s'affranchir d'un paramètre de sortie de type `REF CURSOR`. La procédure `GET_NEXT_RESULT` du même paquetage sert à parcourir plusieurs jeux de résultats produits par `RETURN_RESULT`.

Le tableau 7-56 présente une procédure qui compose deux jeux de résultats à retourner. L'appel de cette procédure dans l'interface SQL*Plus est également décrit.

Tableau 7-56 Retour de plusieurs jeux de résultats

Procédure	Appel dans SQL*Plus

`CREATE OR REPLACE PROCEDURE liste_adh(`
`  p1 IN adherent.civilite%TYPE,`
`  p2 IN pratique.spid%TYPE) IS`
`v_cursor  SYS_REFCURSOR;`
`BEGIN`
`  OPEN v_cursor FOR SELECT prenom, nom`
`  FROM adherent`
`  WHERE civilite = p1 ORDER BY nom;`
`  DBMS_SQL.RETURN_RESULT(v_cursor);`
`  OPEN v_cursor FOR SELECT a.prenom,a.nom`
`  FROM  pratique p, adherent a`
`  WHERE  p.adhid = a.adhid`
`  AND  p.spid = p2;`
`  DBMS_SQL.RETURN_RESULT(v_cursor);`
`END liste_adh;`
`/`
	`SQL> EXEC liste_adh('Mr.', 12);`
	`Procédure PL/SQL terminée avec succès.`
	
	`Ensemble de résultats #1`
	`PRENOM  NOM`
	`------------------------------ ----------`
	`YVES  RENOUF`
	`THIBAULT  JOUANNE`
	`CLAUDE  MARIE`
	`…`
	
	`Ensemble de résultats #2`
	`PRENOM  NOM`
	`------------------------------ ----------`
	`ROMUALD  BELLIN`
	`STEPHANE  DESLOGES`
	`GUY  SPITZA`
	`…`

Pour présenter la procédure `GET_NEXT_RESULT`, considérons le bloc suivant qui appelle la procédure et dépile les deux jeux de résultats à l'aide d'une boucle.

Tableau 7-57 Utilisation de GET_NEXT_RESULT

Bloc	Appel dans SQL*Plus

`DECLARE`
`  v_cur  PLS_INTEGER;`
`  v_refcur  SYS_REFCURSOR;`
`  v_ret  PLS_INTEGER;`
`  v_col1  VARCHAR2(30);`
`  v_col2  VARCHAR2(30);`
`BEGIN`
` v_cur := DBMS_SQL.OPEN_CURSOR`
`  (treat_as_client_for_results => TRUE);`
` DBMS_SQL.PARSE (c => v_cur,`
`  statement => 'BEGIN liste_adh(''Mr.'',12); END;',`
`  language_flag => DBMS_SQL.NATIVE);`
` v_ret := DBMS_SQL.EXECUTE(v_cur);`
` LOOP`
`  BEGIN`
`  DBMS_SQL.GET_NEXT_RESULT(v_cur, v_refcur);`
`  EXCEPTION`
`  WHEN NO_DATA_FOUND THEN EXIT;`
`  END;`
`  DBMS_OUTPUT.PUT_LINE('- jeu de resultats -');`
`  LOOP`
`  FETCH v_refcur INTO v_col1, v_col2;`
`  EXIT  WHEN v_refcur%NOTFOUND;`
`  END LOOP;`
`  DBMS_OUTPUT.PUT_LINE('le dernier ' || v_col1 ||`
`' ' || v_col2);`
`  CLOSE v_refcur;`
`  END LOOP;`
`END;`
`/`
	`- jeu de résultats -`
	`le dernier MARC ZUNIGA`
	
	`- jeu de résultats -`
	`le dernier JESSICA TILLAUT`
	
	`Procédure PL/SQL terminée avec succès.`

Accessibilité

Depuis la version 12*c*, un sous-programme n'est pas forcément accessible par un objet du même schéma (fonction, procédure, déclencheur ou paquetage). Ainsi, la clause `ACCESSIBLE BY` qui peut être ajoutée à un objet ou à un type constitue une liste d'autorisation.

Pour présenter ce mécanisme, considérons la procédure suivante qui ne peut être invoquée que par une fonction ou une procédure. L'appel direct provoque une erreur, mais en passant par une unité de programme autorisée, l'appel est valide.

Tableau 7-58 Utilisation

Déclaration de la liste blanche

```
SQL> CREATE OR REPLACE PROCEDURE p_tel_adh
          (p1 IN adherent.adhid%TYPE)
  2  ACCESSIBLE BY (PROCEDURE p_tel, FUNCTION f_tel)
  3  IS
  4    result   VARCHAR2(20);
  5  BEGIN
  6   SELECT tel INTO result FROM adherent WHERE adhid = p1;
  7    DBMS_OUTPUT.PUT_LINE('Tel : ' || result);
  8  EXCEPTION
  9  WHEN NO_DATA_FOUND THEN
 10    DBMS_OUTPUT.PUT_LINE('Non trouvé');
 11  END;
 12  /
Procédure créée.

SQL> EXEC p_tel_adh(27486);
ORA-06550: Ligne 1, colonne 7 :PLS-00904: privilège
insuffisant pour accéder à l'objet P_TEL_ADH
```

Appel correct

```
SQL> CREATE OR REPLACE
PROCEDURE
   p_tel(p1 IN adherent.
         adhid%TYPE) IS
  2    BEGIN
  3      p_tel_adh(p1);
  4  END;
  5  /

Procédure créée.

SQL> EXEC p_tel(27486);
Non trouvé.

Procédure PL/SQL terminée
avec succès.
```

Pour les paquetages, la liste blanche s'applique au niveau de la spécification (et pas au niveau d'une procédure, fonction ou d'un type de paquetage)

```
CREATE PACKAGE nom_paq
               ACCESSIBLE BY (PROCEDURE nom_proc,…) AS
      …
END;
```

Exercices

L'objectif de ces exercices est d'écrire des déclencheurs et des sous-programmes PL/SQL manipulant des curseurs et gérant des exceptions sur les bases de données *Parc informatique* et *Chantiers*.

Exercice 7.1 Curseur

On désire connaître, pour chaque logiciel installé, le temps (nombre de jours) passé entre l'achat et l'installation. Ce calcul devra renseigner la colonne `delai` de la table `Installer` pour l'instant nulle. Les résultats devront être affichés (par `DBMS_OUTPUT.PUT_LINE`) ainsi que les incohérences (date d'installation antérieure à la date d'achat, date d'installation ou date d'achat inconnue).

Écrivez la procédure `calculTemps` pour programmer ce processus. Un exemple d'état de sortie est présenté ci-après :

```
Logiciel Oracle 6 sur Poste 2, attente 2924 jour(s).
Logiciel Oracle 8 sur Poste 2, attente 1463 jour(s).
Date d'achat inconnue pour le logiciel SQL*Net sur Poste 2
Pas de date d'installation pour le logiciel WinDev sur Poste 4
…
Logiciel I. I. S. installé sur Poste 7 11 jour(s) avant d'être acheté!
…
```

Exercice 7.2 Transaction

Écrivez la procédure transactionnelle `installLogSeg` permettant d'effectuer une installation groupée sur tous les postes d'un même segment d'un nouveau logiciel. La transaction doit enregistrer dans un premier temps le nouveau logiciel puis, les différentes installations sur tous les postes du segment de même type que celui du logiciel acheté. L'installation se fera à la date du jour.

Ne pas encore tenir compte des éventuelles exceptions et tracer les insertions. Utiliser les paramètres suivants pour tester votre procédure :

```
SQL> EXECUTE installLogSeg('130.120.80', 'log99','Blaster', '05-09-
2003', '9.9', 'PCWS', 999.9 )
Blaster stocké dans la table Logiciel
Installation sur Poste 4 dans Salle 2
Installation sur Poste 5 dans Salle 2

Procédure PL/SQL terminée avec succès.
```

Exercice 7.3 Exceptions

Modifiez la procédure `installLogSeg` afin de prendre en compte les exceptions potentielles :

- numéro de segment inconnu (erreur prédéfinie `NO_DATA_FOUND`) ;

- numéro de logiciel déjà présent (erreur prédéfinie `DUP_VAL_ON_INDEX`) ;
- date d'achat plus grande que celle du jour (erreur utilisateur `date_fausse`) ;
- type de logiciel inconnu (erreur non prédéfinie de code Oracle -2291) ;
- aucune installation réalisée, car pas de poste de travail de ce type (erreur utilisateur `pas_install_possible`).

Testez chacun de ces cas avec les valeurs suivantes :

```
--Mauvais segment
EXECUTE installLogSeg('130.120.87', … )
--Logiciel déjà présent
EXECUTE installLogSeg('130.120.80', 'log6',….)
--date > jour
EXECUTE installLogSeg('130.120.80', 'log66', 'Test', '05-09-3000', …)
--Type de logiciel inconnu
EXECUTE installLogSeg('130.120.80','log66','Test','05-09-2003','9.9',
'APPL', …)
--Aucune install
EXECUTE installLogSeg('130.120.81', 'log55', '…', '…', '…', 'PCWS', …)
--Bonne installation
EXECUTE installLogSeg('130.120.80', 'log66','Eudora6', '10-09-2003',
'6.0', 'PCWS', 66)
```

Exercice 7.4 Déclencheurs

Mises à jour de colonnes

Écrivez le déclencheur `Trig_Après_DI_Installer` sur la table `Installer` permettant de faire la mise à jour automatique des colonnes `nbLog` de la table `Poste`, et `nbInstall` de la table `Logiciel`. Prévoir les cas de désinstallation d'un logiciel sur un poste, et d'installation d'un logiciel sur un autre.

Écrivez le déclencheur `Trig_Après_DI_Poste` sur la table `Poste` permettant de mettre à jour la colonne `nbPoste` de la table `Salle` à chaque ajout ou suppression d'un nouveau poste.

Écrivez le déclencheur `Trig_Après_U_Salle` sur la table `Salle` qui met à jour automatiquement la colonne `nbPoste` de la table `Segment` après la modification de la colonne `nbPoste`.

Ces deux derniers déclencheurs vont s'enchaîner : l'ajout ou la suppression d'un poste entraînera l'actualisation de la colonne `nbPoste` de la table `Salle` qui conduira à la mise à jour de la colonne `nbPoste` de la table `Segment`. Ajouter un poste pour vérifier le rafraîchissement des deux tables (`Salle` et `Segment`). Supprimer ce poste puis vérifier à nouveau la cohérence des deux tables.

Programmation de contraintes

Écrivez le déclencheur `Trig_Avant_UI_Installer` sur la table `Installer` permettant de contrôler, à chaque installation d'un logiciel sur un poste, que le type du logiciel correspond au type du poste, et que la date d'installation est soit nulle soit postérieure à la date d'achat.

Exercice 7.5 Transaction de la base *Chantiers*

Écrivez la procédure `finAnnee` permettant de rajouter à chaque véhicule les kilométrages faits lors des visites de l'année. Vous utiliserez un seul curseur pour parcourir tous les véhicules. Il faudra ensuite supprimer toutes les missions de l'année (visites et détails des trajets des employés transportés).

Exercice 7.6 Déclencheurs de la base *Chantiers*

Déclencheur ligne

Écrivez le déclencheur `TrigPassagerConducteur` sur la table `transporter` permettant de vérifier qu'à chaque nouveau transport, le passager déclaré n'est pas déjà enregistré en tant que conducteur le même jour.

Déclencheur composé

Écrivez le déclencheur composé `TrigcapaciteVehicule` sur la table `transporter` permettant de contrôler, qu'à chaque nouveau transport, la capacité du véhicule n'est pas dépassée.

Vous éviterez le problème des tables mutantes en :

- déclarant dans la zone de définition commune un tableau recensant le nombre de personnes transportées par visite ;
- déclarant dans cette même zone un curseur qui va parcourir toutes les visites ;
- chargeant le tableau dans la section `BEFORE STATEMENT` ;
- examinant le tableau dans la section `BEFORE EACH ROW` et en le comparant avec les données à insérer.

Les messages à afficher pour tracer et rendre plus lisible ce déclencheur sont :

- dans la section `BEFORE EACH ROW` : "Enregistrement du transport de *nom*" puis éventuellement " Premier trajet de la visite" ;
- dans la section `AFTER EACH ROW` : "Transport de *nom* bien enregistré" puis "Il ne reste plus que *x* place(s) disponible(s)" ;
- dans la section `AFTER STATEMENT` : "Nombre de trajet(s) traité(s) : *nombre*" ;

Les messages d'erreur à produire le cas échéant sont les suivants :

- "Capacité max atteinte *n* pour la visite *chantier* du *date*, pour le véhicule *v*" ;
- "BASE INCORRECTE : Capacité dépassée *n* pour la visite *chantier* du *date*, pour le véhicule *v*".

Partie III

SQL avancé

Chapitre 8

Le précompilateur Pro*C/C++

Oracle fournit plusieurs précompilateurs permettant d'inclure des instructions SQL au sein de programmes écrits dans des langages procéduraux (Cobol, Fortran, PL/1, C et C++). Les précompilateurs s'appellent ainsi : Pro*COBOL, Pro*FORTRAN, Pro*PL/I et Pro*C/C++ que nous étudions dans ce chapitre. Nous employons seulement une syntaxe C, mais les mécanismes décrits dans ce chapitre valent également dans le cas d'une syntaxe C++. Il existe un autre mécanisme d'interfaçage (que nous n'étudierons pas) qui consiste à utiliser des primitives de bas niveau OCI (*Oracle Call Interface*).

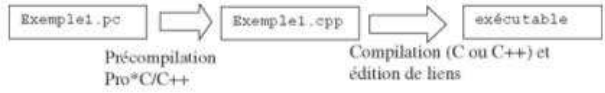
Il est possible d'intégrer le précompilateur Pro*C/C++ dans Microsoft Visual C++, de manière à précompiler, à compiler et à exécuter dans le même environnement de développement. La dernière section de ce chapitre traite de la configuration à mettre en œuvre.

Pour tester ces exemples, il faudra installer Pro*C/C++ qui n'est pas inclus dans la version `Personal Edition`. Il faut exécuter une installation personnalisée et choisir d'installer `Oracle Call Interface`.

Généralités

La précompilation est une technique qui permet d'incorporer dans un programme procédural (dit « hôte ») des commandes SQL dont la syntaxe est presque identique à celle de la forme interactive. Le préprocesseur traduit ces commandes automatiquement en appels OCI.

Figure 8-1 *Précompilation*



Ordres SQL intégrés

Les ordres SQL sont dits « intégrés » car ils apparaissent au même niveau que des instructions du langage (dont la syntaxe n'a rien à voir avec Oracle). Ces ordres sont déclaratifs ou exécutables.

Les ordres déclaratifs permettent de déclarer des objets (au sens Oracle, variables, curseurs, types, etc.) et des zones de communication (nommées SQLCA) entre le programme et la base. Le tableau suivant décrit les instructions qui appartiennent à ce type d'ordres :

Tableau 8-1 Ordres SQL intégrés déclaratifs

Code Pro*C	Commentaires
`BEGIN DECLARE SECTION ... END DECLARE SECTION`	Déclaration des variables hôtes (scalaires ou tableaux).
`DECLARE ...`	Déclaration d'objets.
`INCLUDE ...`	Inclusion de fichiers.
`WHENEVER ...`	Capture des exceptions.

Les ordres exécutables SQL sont, d'une part, les instructions interactives qu'on connaît (CREATE, SELECT, INSERT...). D'autre part, il existe aussi des ordres non interactifs dont quelques-uns sont résumés dans le tableau suivant :

Tableau 8-2 Ordres SQL intégrés non interactifs

Code Pro*C	Commentaires
`CLOSE ...`	Fermeture d'un curseur.
`CONNECT ...`	Connexion à une base.
`FETCH ...`	Lecture dans un curseur.
`OPEN ...`	Ouverture d'un curseur.

Pour inclure tout ordre SQL (dit « intégré ») dans un programme hôte, il faut le faire précéder de la directive `EXEC SQL`.

Variables

Les variables hôtes (scalaires ou tableaux) permettent d'interagir avec la base. Elles peuvent se trouver en paramètre d'un ordre SQL ou en tant que zone de réception d'une extraction (SELECT ou FETCH). Dans tout ordre SQL intégré, une variable hôte est préfixée du symbole : comme le montre le tableau suivant.

Tableau 8-3 Variables hôtes

Code Pro*C	Commentaires

`EXEC SQL BEGIN DECLARE SECTION;`
`  float   nbHeuresVol;`
`  int     budgetMax;`
`  VARCHAR codecomp[20];`
`  VARCHAR tabNomcomp [15] [20];`
`EXEC SQL END DECLARE SECTION;`
	Déclaration de quatre variables (trois scalaires et un tableau).

…
`EXEC SQL SELECT nbHVol, comp`
`INTO :nbHeuresVol, :codecomp`
`FROM Avion WHERE immat = 'F-WTSS';`
	Extraction de données dans deux zones de réception.

…
`EXEC SQL DECLARE CURSOR curs FOR`
`SELECT nomComp FROM Compagnie`
`WHERE budget < :budgetMax;`
…
`EXEC SQL FETCH curs INTO :tabNomcomp;`
	Curseur paramétré.
	Chargement d'une partie du tableau (détail plus loin).

Variable indicatrice

Il est possible d'associer à toute variable un indicateur (de type `smallint`), bien utile pour tester le bon fonctionnement du transfert de données entre la base et le langage hôte. Dans le cas d'une requête, sa valeur permet de détecter une erreur : 0, tout va bien ; -1, aucune valeur n'a été renvoyée ; >0, la valeur renvoyée a été tronquée, l'indicateur contient la longueur de la chaîne avant l'opération. Dans le cas d'une mise à jour, l'indicateur permet d'attribuer la valeur nulle à une colonne (valeur de l'indicateur de la variable correspondant à la colonne positionnée à -1). Tout indicateur est préfixé du symbole : dans un ordre SQL intégré, comme le montre le tableau suivant :

Tableau 8-4 Indicateur de variables

Code Pro*C	Commentaires

`EXEC SQL BEGIN DECLARE SECTION;`
`  float    nbHeuresVol;`
`  smallint indicnbHVol;`
`  int      budgetMax;`
`  VARCHAR  codecomp[20];`
`EXEC SQL END DECLARE SECTION;`
	Déclaration de l'indicateur d'une variable.

…
`EXEC SQL SELECT nbHVol, comp`
`INTO :nbHeuresVol :indicnbHVol, :codecomp`
`FROM Avion WHERE immat = 'F-WTSS';`
`if (:indicnbHVol == -1)`
`/* la colonne nbHVol est NULL */`
…
	Extraction de l'indicateur de la colonne `nbHVol`.

…
`indicnbHVol = -1;`
`EXEC SQL INSERT INTO Avion`
`VALUES ('F-GLDX', 'DR400',`
`:nbHeuresVol :indicnbHVol, 'AF');`
	Insertion d'une valeur `NULL` dans la colonne `nbHVol` de la table `Avion`.

Cas du VARCHAR

VARCHAR est considéré comme un « pseudo-type » au niveau du précompilateur, car il intervient au même niveau que les types primitifs `int`, `float`, `char`, etc. Chaque variable VARCHAR déclarée (aussi valable pour les tableaux de chaînes) doit être manipulée à l'aide de la structure (`struct`) C/C++ automatiquement générée à la précompilation. Souvenez-vous de l'antislash zéro, des fonctions de chaînes `strcpy`, etc. Héritages de la pénible gestion des entrées-sorties de ces langages (Java a heureusement simplifié la situation).

Le tableau suivant décrit comment manipuler une variable VARCHAR par sa structure C/C++ dans le programme :

Tableau 8-5 Correspondance VARCHAR / struct

Code Pro*C	Commentaire
`VARCHAR nomCompa[20];`	`struct { unsigned short len; unsigned char arr[20]; } nomCompa;`
`EXEC SQL SELECT nomComp INTO :nomCompa FROM Compagnie WHERE comp = 'AF';`	Chargement de la structure.
`…`	
`nomCompa.arr[nomCompa.len]='\0';`	Définition de la fin de chaîne.
`printf("Compagnie: %s", nomCompa.arr);`	Affichage de la chaîne.

Penser à rajouter un caractère dans la déclaration à chaque variable hôte correspondant à une colonne VARCHAR, VARCHAR2 ou CHAR de la base (pour pouvoir stocker le « `\0` »).

Zone de communication (SQLCA)

Il est indispensable d'inclure la zone de communication SQLCA (*SQL Communication Area*), comme on inclut une bibliothèque, à l'aide de l'instruction SQL intégrée `INCLUDE`. La syntaxe à composer est la suivante : « `EXEC SQL INCLUDE sqlca.h;` ».

À chaque ordre SQL intégré exécuté, cette zone est mise à jour et il est possible ainsi de tester le code retour d'Oracle pour chaque instruction. On retrouve les variables `sqlcode` et `sqlerrm` étudiées avec PL/SQL. La variable est une structure composée de deux champs :

```
struct { unsigned short sqlerrml;
         char           sqlerrmc[70];} sqlerrm;
```

Le champ `sqlerrml` indique le nombre de caractères du message d'erreur : le champ `sqlerrmc` contient le message d'erreur lui-même.

Connexion à une base

La connexion à une base Oracle se réalise par l'ordre SQL intégré CONNECT qui comporte trois variables hôtes de type VARCHAR (nom d'utilisateur, mot de passe et le descripteur de la connexion). La syntaxe est la suivante :

```
EXEC SQL CONNECT :utilisateur IDENTIFIED BY :pwd USING :descripteur;
```

Gestion des exceptions

Il existe deux mécanismes pour gérer les erreurs :

- l'exploitation de la zone SQLCA après chaque instruction SQL intégrée (test de la variable `sqlca.sqlcode` et affichage de la variable `sqlca.sqlerrm.sqlerrmc`) ;
- l'utilisation de la directive WHENEVER qui met en œuvre des étiquettes pour dérouter le traitement en fonction de la nature de l'exception. Ce type de programmation est plus rigoureux et facilite la maintenance.

Notons que ces mécanismes peuvent cohabiter (voir le premier exemple). Les tableaux suivants décrivent les possibilités de l'instruction :

```
EXEC SQL WHENEVER événement action;
```

Tableau 8-6 Événements pris en compte par WHENEVER

Événement	Commentaires
SQLERROR	Toute exception.
SQLWARNING	Anomalie (*warning*) signalée par Oracle.
NOT FOUND	Donnée non trouvée (ORA-01403).

Tableau 8-7 Actions prises en compte par WHENEVER

Action	Commentaires
STOP	Arrêt du programme (annulation de la transaction en cours).
CONTINUE	Force le programme à continuer en séquences malgré l'erreur retournée par Oracle.
GOTO *étiquette*	Branchement à l'étiquette indiquée par son identificateur.
DO *function*([*parameters*])	Appel de la fonction C/C++ en passant d'éventuels paramètres.

La portée de l'instruction WHENEVER est dictée par sa position (si elle se trouve dans le *main* elle reste valable dans tout le bloc principal). L'action spécifiée reste valable jusqu'à la fin du bloc ou jusqu'à l'exécution d'une autre instruction WHENEVER portant sur le même événement.

Transactions

Il est tout à fait possible de programmer des transactions comme le montrent les instructions SQL intégrées du tableau suivant :

Tableau 8-8 Instructions pour les transactions

Action	Commentaires
`EXEC SQL COMMIT WORK [RELEASE];`	Validation de la transaction. L'option `RELEASE` libère les éventuels verrous.
`EXEC SQL ROLLBACK WORK [RELEASE];`	Invalidation de la transaction. Idem pour `RELEASE`.
`EXEC SQL SAVEPOINT` *`nomPoint`*`;`	Pose d'un point de validation.
`EXEC SQL ROLLBACK TO SAVEPOINT` *`nomPoint`*`;`	Invalidation d'une partie de la transaction.

Extraction d'un enregistrement

L'exemple suivant (`proC1.pc`) extrait d'un enregistrement de la table `Avion` à partir du schéma `scott/tiger` et du descripteur de connexion `CXBDSOUTOU`. Si aucune donnée n'est trouvée, le traitement se déroute vers l'étiquette `pasTrouve`. Dans le cas d'une autre erreur, le traitement se déroute vers l'étiquette `probleme` où est testée la possibilité que la requête ramène plusieurs enregistrements.

Tableau 8-9 Extraction d'un enregistrement

Code Pro*C	Commentaires

`#include <stdio.h>`
`#include <ctype.h>`
`#include <string.h>`
`void afficheErreur(void);`
	Inclusion des bibliothèques C.
	Déclaration de la fonction qui affiche les messages d'erreur.

`EXEC SQL BEGIN DECLARE SECTION;`
`VARCHAR utilisateur[30];`
`VARCHAR pwd[10];`
`VARCHAR descripteur[10];`
`VARCHAR immat[6];`
`VARCHAR typeav[15];`
`int capacite;`
`VARCHAR codecomp[4];`
`EXEC SQL END DECLARE SECTION;`
	Déclaration des variables hôtes.

`EXEC SQL INCLUDE sqlca.h;`	Inclusion de la zone de communication.

Tableau 8-9 Extraction d'un enregistrement *(suite)*

Code Pro*C	Commentaires

`void main()`
`{ strcpy((char *) utilisateur.arr,"SCOTT");`
`  utilisateur.len =`
`  (int) strlen((char *) utilisateur.arr);`
`  strcpy((char *)pwd.arr,"TIGER");`
`  pwd.len = (int) strlen((char *) pwd.arr);`
`  strcpy((char *) descripteur.arr,"CXBDSOUTOU");`
`  descripteur.len =`
`  (int) strlen((char *)descripteur.arr);`
	Initialisation des paramètres de connexion à la base.

`EXEC SQL WHENEVER SQLERROR GOTO probleme;`
`EXEC SQL CONNECT :utilisateur`
`  IDENTIFIED BY :pwd USING :descripteur;`
	Préparation du déroutement en cas d'erreur.
	Connexion à la base.

`EXEC SQL WHENEVER NOT FOUND GOTO pasTrouve;`
`EXEC SQL SELECT typeAvion, cap, comp`
`  INTO :typeav, :capacite, :codecomp`
`  FROM Avion WHERE immat = 'F-WTSS';`
`typeav.arr[typeav.len] = '\0';`
`codecomp.arr[codecomp.len] = '\0';`
	Gestion de l'exception.
	Extraction d'un enregistrement.
	
	Ajout du caractère `\0` en fin de chaînes.

`printf("Détails de l'avion : %s %d %s\n",`
`  typeav.arr, capacite, codecomp.arr);`
`return;`
	Affichage des résultats.

`probleme:`
`if (sqlca.sqlcode == -2112)`
`  printf ("Trop de lignes ramenées!");`
`else`
`  afficheErreur();`
`return;`
`pasTrouve:`
`  printf ("Aucun avion de ce code…\n");`
`}`
	Gestion des erreurs.

`void afficheErreur()`
`  {printf("%s (%d)\n", sqlca.sqlerrm.sqlerrmc,`
`  -sqlca.sqlcode);}`
	Affichage des messages d'erreur.

Dans les exemples qui suivent, nous ne réécrivons pas les parties d'inclusion (des bibliothèques et de la zone de communication) de la connexion à la base, et la fonction (`afficheErreur`) d'affichage des messages d'erreur.

Mises à jour

L'exemple suivant (proC2.pc) insère un enregistrement dans la table Compagnie.

Tableau 8-10 Mise à jour de la base

Code Pro*C	Commentaires
`void main()` `{ …`	Initialisation.

`EXEC SQL WHENEVER SQLERROR GOTO sortie;`
`EXEC SQL CONNECT :utilisateur`
`IDENTIFIED BY :pwd USING :descripteur;`
	Connexion.

`EXEC SQL WHENEVER SQLERROR GOTO probleme;`
`EXEC SQL INSERT INTO Compagnie`
`VALUES ('BAW', 'British Airways');`
	Insertion.

`EXEC SQL COMMIT WORK;`	Validation.
`return;`	

`sortie:`
`printf ("Problème de connexion!");`
`afficheErreur();`
`return;`
	Traitement d'une erreur à la connexion.

`probleme:`
`afficheErreur();`
`EXEC SQL WHENEVER SQLERROR CONTINUE;`
`EXEC SQL ROLLBACK WORK;`
`}`
	Traitement d'une erreur lors de l'insertion avec invalidation de la transaction.

Utilisation de curseurs

Dès qu'une requête retourne plusieurs enregistrements, il faut utiliser un curseur pour traiter les résultats extraits. Le mécanisme des curseurs s'apparente à celui étudié au chapitre 7. Il comporte quatre étapes chronologiques : déclaration, ouverture, parcours et fermeture.

Par ailleurs, à l'inverse de PL/SQL qui ne supporte que des variables scalaires (ou RECORD) dans le type de retour, le précompilateur Pro*C/C++ permet de récupérer un ensemble de lignes résultats dans un tableau (par paquets de données de la taille du tableau). Étudions à présent ces deux techniques.

Variables scalaires

L'exemple suivant (proC3.pc) programme un curseur qui alimente des variables scalaires. Il s'agit d'afficher les caractéristiques des avions appartenant à une compagnie de nom saisi au clavier (via la fonction C saisieChaine qui convient mieux que scanf).

Tableau 8-11 Curseur avec variables scalaires

Code Pro*C	Commentaires
`int saisieChaine(char *,char *);`	Déclaration de la fonction.

`void main()`
`{…`
`EXEC SQL DECLARE curs CURSOR FOR`
`SELECT a.immat, a.typeAvion, a.cap`
`FROM Compagnie c, Avion a`
`WHERE c.comp = a.comp AND nomcomp = :nomcomplu;`
	Déclaration du curseur.

`nomcomplu.len = saisieChaine`
`("Nom de la compagnie (ou fin) : ",nomcomplu.arr);`
`nomcomplu.arr[nomcomplu.len]='\0';`
	Saisie du nom de la compagnie.

`EXEC SQL WHENEVER SQLERROR GOTO erreur;`
`EXEC SQL OPEN curs;`
`EXEC SQL WHENEVER NOT FOUND GOTO finBoucle;`
	Gestion de l'ouverture du curseur.

`printf("\nFlotte de %s\n",nomcomplu.arr);`
`while(1)`
`{ EXEC SQL FETCH curs`
`INTO :immat,:typeav,:capacite;`
`immat.arr[immat.len] = '\0';`
`typeav.arr[typeav.len] = '\0';`
`printf("%s %s %d\n", immat.arr, typeav.arr,`
`capacite); }`
	Parcours du curseur.
	
	Affichage de l'enregistrement courant.

`finBoucle:`
`EXEC SQL CLOSE curs;`
`return;`
	Fin de curseur.

`erreur:`
`afficheErreur();`
`return;`
`sortie:`
`printf ("Probleme à la connexion!");`
`afficheErreur();`
`}`
	Gestion des erreurs.

`int saisieChaine(char texte[], char variable[])`
`{printf(texte);`
`fflush(stdout);`
`return (gets(variable) ==`
`(char *)0 ? EOF : strlen(variable));`
`}`
	Fonction de saisie d'une chaîne.

Variables tableaux

L'utilisation de tableaux comme types de retour d'un curseur évite de nombreux échanges de données entre la base et le programme. En effet, alors qu'il fallait une lecture (FETCH) du curseur pour chaque enregistrement extrait (voir l'exemple précédent), la lecture d'un curseur dans un tableau chargera un paquet d'enregistrements (d'un nombre égal à la taille du tableau). Notons qu'il est aussi possible d'insérer par paquets (tableaux C initialisés qu'on utilise comme paramètres d'une instruction SQL intégrée INSERT).

La variable `sqlerrd[2]` de la zone SQLCA contient après chaque lecture dans le curseur (exécution de `FETCH`), le nombre cumulé de lignes extraites.

L'exemple suivant (`proC4.pc`) met en œuvre un curseur qui charge à chaque lecture quatre tableaux de trois enregistrements. Il s'agit d'afficher les caractéristiques de tous les avions. Dès que la fin de curseur est atteinte, le programme se déroute à l'étiquette `finBoucle`. S'il reste des lignes à traiter (moins de trois enregistrements ont été extraits), le calcul du nombre de lignes à traiter permet d'afficher le reste des tableaux. Par exemple, supposons que 7 enregistrements soient à extraire et que la taille des tableaux est 3. Deux tours de boucle chargent 6 enregistrements, le dernier est traité par l'intermédiaire de l'étiquette.

Tableau 8-12 Extraction dans des tableaux

Code Pro*C	Commentaires
`#define TAILLE 3` `void affiche(int);`	Déclaration de la fonction.

`EXEC SQL BEGIN DECLARE SECTION;`
`…`
`VARCHAR tabimmat [3] [7];`
`VARCHAR tabtypeav [3] [16];`
`int tabcapacite [3];`
`VARCHAR tabnomcomp [3] [26];`
`int nbpaquets;`
`int ligne_restante;`
`EXEC SQL END DECLARE SECTION;`
	Déclaration des tableaux.

`void main()`
`{ …`
`EXEC SQL DECLARE curs CURSOR FOR`
`SELECT a.immat, a.typeAvion, a.cap, c.nomcomp`
`FROM Compagnie c, Avion a`
`WHERE c.comp = a.comp;`
`EXEC SQL WHENEVER SQLERROR GOTO probleme;`
`EXEC SQL OPEN curs;`
`EXEC SQL WHENEVER NOT FOUND GOTO finBoucle;`
	Déclaration du curseur.

`nbpaquets = 0;`
`printf("Flotte");`
`while(1)`
`{EXEC SQL FETCH curs INTO :tabimmat, :tabtypeav,`
`:tabcapacite, :tabnomcomp;`
`affiche(TAILLE);`
`nbpaquets ++;}`
`finBoucle:`
`ligne_restante =`
`sqlca.sqlerrd[2] - (nbpaquets * TAILLE);`
`affiche(ligne_restante);`
`printf ("\n----------\n");;`
`EXEC SQL CLOSE curs;`
`return;`
`probleme:`
`afficheErreur();`
`return;`
`sortie:`
`printf("Probleme à la connexion!");`
`afficheErreur();`
`}`
	Déclaration du curseur.
	Affichage du paquet.
	
	Affichage du reste.

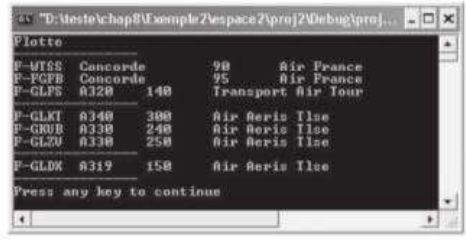
Tableau 8-12 Extraction dans des tableaux *(suite)*

Code Pro*C	Commentaires

`void affiche(int n)`
`{ int i;`
`printf ("\n---------------");`
`for (i=0; i<n; i++)`
`{ tabimmat[i].arr[tabimmat[i].len] = '\0';`
`tabtypeav[i].arr[tabtypeav[i].len] = '\0';`
`tabnomcomp[i].arr[tabnomcomp[i].len] = '\0';`
`printf("\n%s\t%s\t%d\t%s",tabimmat[i].arr,`
`tabtypeav[i].arr, tabcapacite[i],`
`tabnomcomp[i].arr ); }`
`}`
	Affichage du tableau chargé par le curseur.

Le résultat de ce programme est le suivant :

Figure 8-2 *Résultats à l'écran*



Utilisation de Microsoft Visual C++

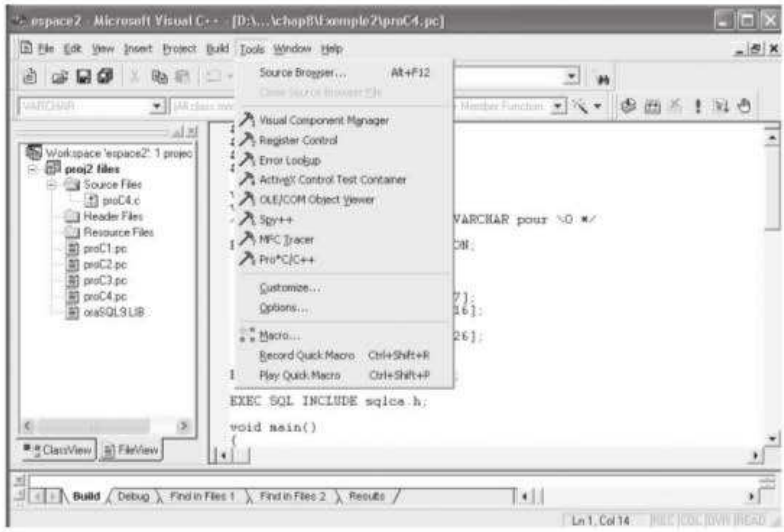
Afin de travailler avec Microsoft Visual C++, vous devez éventuellement avoir installé le précompilateur Pro*C/C++ en lançant à nouveau une installation à partir des extensions d'Oracle. La documentation à consulter est *Pro*C/C++ Precompiler Getting Started for Windows*, chapitre « Integrating Pro*C/C++ into Microsoft Visual C++ ». Elle est assez claire, et plusieurs étapes sont à respecter, nous les résumons ici pour Oracle9*i* :

- Spécifier la localisation des exécutables (en général `C:\oracle\ora92\bin`).
- Spécifier la localisation des sources à précompiler (en général `C:\oracle\ora92\precomp\public`).
- Ajouter les sources (extensions `.pc`) à précompiler dans le projet.
- Ajouter la librairie Pro*C/C++ (`C:\oracle\ora92\precomp\lib\msvc`) au projet.

- Spécifier les options de compilation.
- Ajouter l'entrée vers Pro*C/C++ à la barre de menu, pour lancer une précompilation (`C:\oracle\ora92\bin\procui.exe`) à partir de l'environnement.

Une fois tout mis en place, vous pouvez précompiler, compiler, créer des liaisons et exécuter un programme C ou C++ à travers l'interface suivante :

Figure 8-3 *Développement sous MS Visual C++*



Chapitre 9

L'interface JDBC

La technologie JDBC (*Java DataBase Connectivity*), conçue initialement par Sun Microsystems et propriété d'Oracle depuis 2009, permet à des applications Java d'accéder à des sources de données compatibles SQL (tables relationnelles en général, mais aussi données issues d'un fichier texte ou d'un classeur Excel, par exemple).

Cette spécification est considérée comme une API virtuelle ; tout éditeur de logiciels peut donc l'implémenter et proposer un pilote (*driver*) afin de permettre le dialogue avec du code Java. C'est le cas de tous les SGBD qui disposent d'un ou de plusieurs pilotes JDBC. Inclus dans le JDK depuis 1997, le paquetage `java.sql` regroupe les fonctionnalités principales qui sont détaillées dans ce chapitre. Certains aspects du paquetage `javax.sql`, apparu avec JDBC 2, seront aussi étudiés (`DataSource` et `RowSet`).

Depuis JDK 1.5, la version de JDBC est la 4.0. La prochaine version 4.1 devrait apparaître avec la version 7 du langage Java. Il y a relativement peu d'apports depuis JDBC 3, apparu avec JDK 1.4 : citons la prise en compte de XML, du type `ROWID`, des *National Character* (`NCHAR`, `NVARCHAR`...) et plus de possibilités pour les *BLOB*.

Généralités

La technologie JDBC est conforme au niveau d'entrée de la norme ANSI SQL2 (*entry level*) et de la spécification SQLX/OPEN CLI (*Call Level Interface*), compatible ODBC (Microsoft) et avec d'autres API propriétaires. JDBC supporte la programmation *multithread*. La communication est réalisée en mode client-serveur déconnecté et s'effectue en plusieurs étapes :

- connexion à la base de données ;
- émissions d'instructions SQL et exploitation des résultats provenant de la base de données ;
- déconnexion de la base.

Le spectre de JDBC est large, car l'application Java connectée à la base peut être une classe ou une *applet*, une *servlet*, un EJB (*Enterprise Java Beans*) ou une procédure cataloguée. Par ailleurs, tous les outils de *mapping* objet-relationnels génèrent en interne du code JDBC.

Classification des pilotes (drivers)

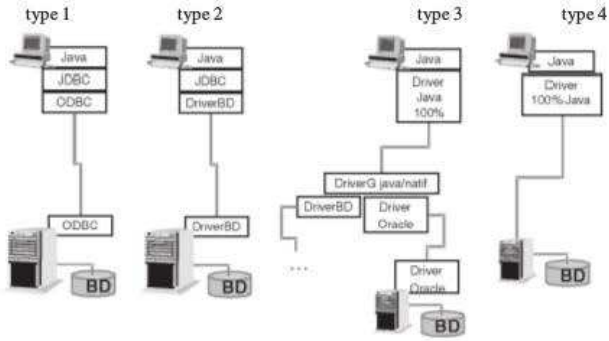
Un pilote (*driver*) JDBC est la couche logicielle qui est chargée d'assurer la liaison entre l'application Java cliente et le SGBD serveur. On parle de *middlware*. Il est disponible sur le site des éditeurs, le plus souvent gratuitement, sous la forme d'un fichier (en général, `.jar`, `.zip`, `.tar` ou `.gz`). Vous devrez placer ce fichier, après décompression, dans le *classpath* de votre environnement de compilation.

On distingue quatre types de pilotes JDBC.

- Les pilotes de type 1 (*JDBC-ODBC Bridge*) utilisent la couche logicielle de Microsoft appelée ODBC (*Open DataBase Connectivity*). Le client est dit « épais » puisque le pilote JDBC convertit les appels Java en appels ODBC avant de les exécuter. Cette approche convient bien pour des sources de données Windows ou si l'interface cliente est écrite dans un langage natif de Microsoft.
- Les pilotes de type 2 (*Native-API Partly-Java Driver*) utilisent un pilote fourni par le constructeur de la base de données (natif). Le pilote n'étant pas développé en Java, le client est aussi dit « épais » pour cette approche. En effet, les commandes JDBC sont toutes converties en appels natifs au SGBD considéré. Cette approche convient pour les applications qui manipulent des sources de données uniques (tout Oracle ou IBM, etc.).
- Les pilotes de type 3 (*Net Protocol All-Java Driver*) utilisent un pilote générique natif écrit en Java. Le client est plus « léger » car les appels JDBC sont transformés par un protocole indépendant du SGBD. Cette approche convient pour des sources de données hétérogènes.
- Les pilotes de type 4 (*Native Protocol All-Java Driver*) sont écrits en Java. Le client est léger car il ne nécessite d'aucune autre couche logicielle supplémentaire. Les appels JDBC sont traduits en *sockets* exploités par le SGBD. Cette approche est la plus simple mais pas forcément la plus puissante, elle convient pour tous types d'architectures.

La figure suivante schématise le principe mis en œuvre au travers des quatre types de pilote JDBC :

Figure 9-1 *Types de pilotes JDBC*



Le choix du pilote n'a pas d'influence majeure sur la programmation. Seules les phases de chargement du pilote et de connexion aux bases sont spécifiques, les autres instructions sont indépendantes du pilote. En d'autres termes, si vous avez une application déjà écrite et que vous décidez de changer le type du pilote – soit que la source de données migre d'Access à Oracle ou à SQL Server par exemple, soit que vous optiez pour un autre pilote en conservant votre source de données –, seules quelques instructions devront être réécrites.

Les paquetages

Les différentes spécifications JDBC se composent d'interfaces, classes, énumérations et exceptions situées dans le paquetage `java.sql`. Selon la version de JDBC, il existe aussi des paquetages additionnels de type `javax.sql.rowset`. Oracle propose des API propriétaires qui implémentent les API de référence de Sun (qui désormais lui appartiennent).

Le paquetage `oracle.jdbc.driver` devra être importé pour utiliser un pilote de connexion d'Oracle. Le paquetage `oracle.sql` devra être importé pour pouvoir manipuler des types spécifiques à Oracle (`BFILE`, `ROWID`, extensions objets, etc.).

Le tableau suivant présente les éléments principaux du paquetage `java.sql` de l'API JDBC (versions 3 et ultérieures).

Tableau 9-1 Éléments principaux de l'API JDBC

Classe/interface	Description
`java.sql.Driver` `java.sql.Connection`	Pilotes JDBC pour les connexions aux sources de données SQL.

`java.sql.Statement`
`java.sql.PreparedStatement`
`java.sql.CallableStatement`
	Construction d'ordres SQL.

`java.sql.ResultSet`	Gestion des résultats des requêtes SQL.
`java.sql.DriverManager`	Gestion des pilotes de connexion.
`java.sql.SQLException`	Gestion des erreurs SQL.
`java.sql.DatabaseMetaData` `java.sql.ResultSetMetaData`	Gestion des méta-informations (description de la base de données, des tables...).
`java.sql.SavePoint`	Gestion des transactions et sous-transactions.

Les pilotes d'Oracle supportent historiquement différentes versions de JDBC. Depuis la version 11g R2, les pilotes sont compatibles avec JDBC 4.0. Les implémentations sont fournies au travers des paquetages `oracle.jdbc` (gestion des accès à la base) et `oracle.sql` (types et manipulations), compatibles avec les JDK récents (1.5 et 1.6).

Le tableau suivant présente les éléments principaux du paquetage `oracle.sql`.

Tableau 9-2 Éléments principaux de l'API JDBC Oracle

Classe/interface	Description
`oracle.sql.OracleDriver` `oracle.sql.OracleConnection`	Connexions aux bases de données (pilotes JDBC OCI et léger).

`oracle.sql.OracleStatement`
`oracle.sql.OraclePreparedStatement`
`oracle.sql.OracleCallableStatement`
	Construction d'ordres SQL.

`oracle.sql.OracleResultSet`	Gestion des résultats des requêtes SQL.
`oracle.sql.OracleDriverManager`	Gestion des pilotes de connexion.
`oracle.sql.OracleSQLException`	Gestion des erreurs SQL.
`oracle.sql.OracleSavePoint`	Gestion des transactions et des sous-transactions.

Les exemples de ce chapitre utilisent le pilote Oracle pour assurer la connexion (`oracle.jdbc`) et l'API de référence (`java.sql`).

Structure d'un programme

La structure d'un programme Java utilisant JDBC pour Oracle comprend successivement les phases :

- d'importation de paquetages ;
- de chargement d'un pilote ;
- de création d'une ou plusieurs connexions ;
- de création d'un ou de plusieurs états ;
- d'émission d'instructions SQL sur ces états ;
- de fermeture des objets créés.

Le code suivant (`JDBCTest.java`) décrit la syntaxe du plus simple programme JDBC. Nous inscrivons toutes les phases dans un même bloc mais elles peuvent se trouver dans différents blocs ou dans plusieurs méthodes de diverses classes.

Tableau 9-3 Programme JDBC

Code Java	Commentaires
`import java.sql.*;` `import oracle.jdbc.driver.*;`	Importation de paquetages.

`class JDBCTest`
`{ public static void main (String args [])`
`throws SQLException`
`{try{`
	Classe ayant une méthode `main`.

`DriverManager.registerDriver`
`(new oracle.jdbc.driver.OracleDriver());`
`Connection conn = DriverManager.getConnection`
`("jdbc:oracle:thin:@CAMPAROLS:1521:BDSoutou",`
`"scott", "tiger");`
	Chargement d'un pilote JDBC Oracle et création d'une connexion.

`Statement stmt = conn.createStatement();`
…
`//Ordres SQL, voir plus loin`
…
	Création d'un état de connexion et exécutions d'instructions SQL.

`} catch(SQLException ex){` `System.err.println("Erreur : "+ex);} } }`	Gestion des erreurs.

Le dernier bloc permet de récupérer les erreurs renvoyées par le SGBD. Nous détaillerons en fin de chapitre le traitement des exceptions.

Variables d'environnement

L'environnement JDBC sous Oracle nécessite la configuration d'un certain nombre de variables.

- La variable `PATH` doit contenir le chemin où se trouvent les exécutables `javac.exe` et `java.exe` (généralement `C:\Program Files\Java\version_jdk\bin`).
- La variable `CLASSPATH` doit inclure le paquetage Oracle à utiliser en fonction du pilote JDBC choisi (suivant, généralement, un chemin de la forme `Oracle_Home\jdbc\lib\paquetage`). Le tableau ci-après rappelle la configuration à mettre en œuvre.

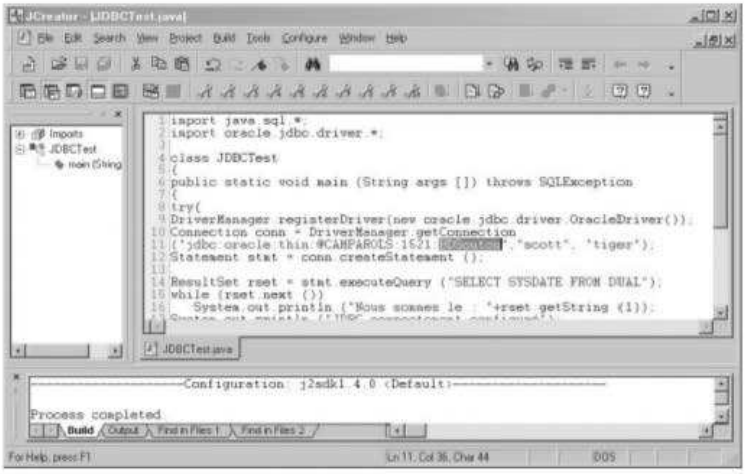
Tableau 9-4 Paquetages Oracle JDBC

Version du JDK utilisé	Paquetage JDBC Oracle
JDK 1.1	`classes111.jar` (1 038 135 octets).
JDK 1.2 et JDK 1.3	`classes12.jar` (1 202 911 octets).
JDK 1.4	`ojdbc14.jar` (1 181 679 octets).
JDK 1.5	`ojdbc5.jar` (2 030 460 octets).
JDK 1.6	`ojdbc6.jar` (2 152 137 octets).

Test de votre configuration

Vous pouvez tester votre environnement en utilisant le fichier `JDBCTest.java`. Si vous utilisez l'outil *JCreator*, configurez la variable `CLASSPATH` de la manière suivante : `Configure/Options/JDK Profiles`, clic sur la version du JDK, puis `Edit`, onglet `Classes`, faire `Add Archive` et choisir le paquetage adéquat (par exemple, `ojdbc6.jar` pour une version 11*g* XE), situé dans mon environnement : `C:\oraclexe\app\oracle\product\11.2.0\server\jdbc\lib`.

Figure 9-2 *Interface JCreator*



Cet exemple décrit le code nécessaire à la connexion à votre base (il faudra modifier le nom de la base, le nom et le mot de passe de l'utilisateur dans l'instruction surlignée de la figure précédente) et doit renvoyer les messages suivants :

```
Nous sommes le : 2003-07-27 13:49:55.0 (date et heure de l'exécution)
JDBC correctement configuré
```

Connexion à une base

La connexion à une base de données est rendue possible par l'utilisation de la classe `DriverManager` et de l'interface `Connection`.

Deux étapes sont nécessaires pour qu'un programme se connecte à une base :

- Le chargement du pilote par l'appel de la méthode `java.lang.Class.forName` pour les pilotes de type 1 ou la création d'un objet de la classe `DriverManager` pour les autres types de pilotes Oracle.
- L'établissement de la connexion en appelant un objet (ici `cx`) de l'interface `Connection` par l'instruction suivante :

```
cx = DriverManager.getConnection(chaîneConnexion, login, password);
```

Depuis la version 4 de JDBC, le pilote `java.sql.Driver` se charge automatiquement. Il est alors inutile d'invoquer `Class.forName` ; la première étape devient alors optionnelle.

Le paramètre *`chaîneConnexion`* représente une variable de type *«`protocole:sousProtocole:infoConnexion`»* permettant de désigner le protocole du pilote et d'identifier la base de données cible.

- *`protocole`* prend la valeur « `jdbc` » pour une connexion JDBC.
- *`sousProtocole`* indique la nature du pilote (« `odbc` » pour un pilote de type 1, « `oracle:thin` » pour un pilote Oracle de type 4, « `oracle:oci` » pour un pilote Oracle de type 2).
- *`infoConnexion`* donne les paramètres qui localisent et identifient la base de données cible.

Base Access

Étudions brièvement l'établissement de la connexion d'un pilote de type 1 pour se mettre en rapport avec une base Access via une source de données ODBC. La figure suivante illustre les parties du panneau de configuration Windows qui permettent de désigner une base Access. Dans notre exemple, la source (`BaseGTR.MDB`) est située dans le répertoire `D:\...\SQL-Oracle9i\Java` et désignée par le DSN (*Data Source Name*) `sourcebaseGTR` :

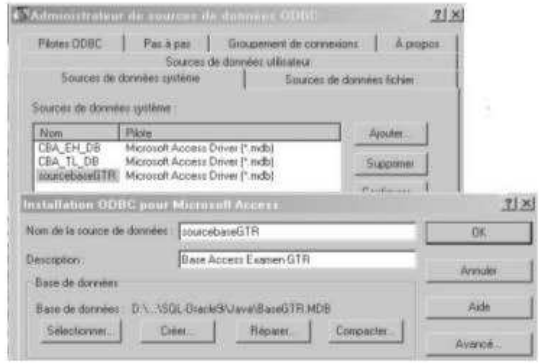


Figure 9-3 *Source de données ODBC*

Le code suivant (`TestJDBCODBC.java`) charge le pilote de type 1 et se connecte à la source ODBC précitée (base Access, donc inutile de préciser le nom et le mot de passe de l'utilisateur). Le DSN est noté en gras dans le script.

Tableau 9-5 Programme JDBC

Code Java	Commentaires
`import java.sql.*;`	Importation.

`class TestJDBCODBC`
`{ public static void main (String args [])`
`throws SQLException`
	Classe ayant une méthode `main`.

`{try {Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");}`
`catch (ClassNotFoundException ex)`
`{ System.out.println ("Problème au chargement"); }`
`try {Connection conn = DriverManager.getConnection`
`("jdbc:odbc:**sourcebaseGTR**","","");`
… }
	Chargement d'un pilote JDBC/ODBC.
	Connexion à la base Access.

`catch(SQLException ex){ … } } }`	Gestion des erreurs.

Base Oracle

Seules les phases de chargement de pilote et de création de la connexion changent. Afin de charger un pilote Oracle, il faut utiliser la classe `DriverManager` de l'API Oracle comme le montre le code suivant. Nous étudierons ensuite les différentes connexions qu'il est possible d'établir. La connexion s'effectue par la méthode `getConnection` de l'interface `DriverManager`.

Tableau 9-6 Chargement du pilote Oracle

Code Java	Commentaires
`import java.sql.*;` `import oracle.jdbc.driver.*;`	Importation.

`public class testoracleSimple`
`{ public static void main (String args [])`
`throws SQLException`
`{ try { DriverManager.registerDriver(new`
`oracle.jdbc.driver.OracleDriver());`
`Connection conn = DriverManager.getConnection(…);`
…
	Classe ayant une méthode `main`.
	Chargement d'un pilote Oracle.
	Déclaration d'une connexion.

`} catch(SQLException ex){ … } } }`	Gestion des erreurs.

Oracle fournit en standard deux types de pilotes : les pilotes OCI (*Oracle Call Interface*) qui sont de type 2 selon la classification étudiée précédemment, et les pilotes légers (*thin*) de type 4.

Connexions OCI

Ces types de connexions conviennent pour les applications utilisant des fonctionnalités du *middleware* OracleNet (couches 5-6-7 ISO), pour des besoins de grandes architectures faisant intervenir des bases de données réparties ou répliquées.

L'exemple suivant (`JDBCOCI.java`) réalise la connexion OCI de l'utilisateur `scott` à une base de données identifiée par le descripteur de connexion `CXBDSOUTOU` (entrée du fichier `tnsnames.ora`, voir « Introduction »).

Tableau 9-7 Connexion OCI

Code Java	Commentaires
`String chaineCx = "jdbc:oracle:oci:@CXBDSOUTOU";` `try{`	Description de la connexion.

`DriverManager.registerDriver`
`(new oracle.jdbc.driver.OracleDriver());`
`Connection conn =`
`DriverManager.getConnection(chaineCx,"scott", "tiger");`
…
	Chargement d'un pilote Oracle OCI.
	Déclaration d'une connexion.

`} catch(SQLException ex){ … } } }`	Gestion des erreurs.

Il est possible d'établir une connexion en utilisant une autre forme de la méthode `getConnection` (avec un seul paramètre).

```
String lienBD = "jdbc:oracle:oci:scott/tiger@CXBDSOUTOU";
…
Connection conn = DriverManager.getConnection(lienBD);
```

Connexion thin

Ces types de connexions conviennent pour les applications qui n'ont pas besoin, côté client, de fonctionnalités du *middleware* OracleNet. C'est la solution qui nécessite le moins de configuration sur les postes clients.

Si vous avez noté, lors de l'installation, le port UDP d'écoute du *listener* (en général 1521), le nom du service (nom de votre base), et si vous connaissez le nom du serveur, vous pouvez vous connecter sans problème a priori.

Le code suivant (`JDBCThin.java`) présente les quatre écritures possibles d'une connexion de type 4, pour l'utilisateur `soutou`, à la base de données `BDSoutou` localisée sur le serveur `CAMPAROLS` sur le port 1521.

Tableau 9-8 Connexion thin

Code Java	Commentaires
`String lienBD = "jdbc:oracle:thin:@CAMPAROLS:1521:BDSoutou"; Connection cx1, cx2, cx3, cx4;`	Définition de 4 connexions.
`try{ DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());`	Chargement du pilote Oracle.
`cx1 = DriverManager.getConnection ("jdbc:oracle:thin:@CAMPAROLS:1521:BDSoutou", "soutou","iut");`	Connexion explicite.
`cx2 = DriverManager.getConnection(lienBD,"soutou","iut");`	Connexion implicite.
`cx3 = DriverManager.getConnection ("jdbc:oracle:thin:@192.168.4.118:1521:BDSoutou", "soutou","iut");`	Connexion avec adresse IP.
`cx4 = DriverManager.getConnection ("jdbc:oracle:thin:@camparols:1521:BDSoutou", "soutou","iut");`	Connexion avec nom du serveur.
`… } catch(SQLException ex){ … } } }`	Gestion des erreurs.

Base MySQL

Les pilotes de MySQL sont disponibles sur le site de l'éditeur (*http://dev.mysql.com/downloads/connector/j/*). Vous y trouverez *Connector/J*, pilote JDBC de type 4.

Le code suivant (`JDBCMySQL.java`) présente une connexion à la base MySQL `BDsoutou`, hébergée sur la machine cliente pour l'utilisateur `soutou`. Suivant la version de votre pilote, le premier `try` est optionnel.

Tableau 9-9 Connexion MySQL

Code Java	Commentaires
`try { Class.forName("com.mysql.jdbc.Driver").newInstance(); } catch (ClassNotFoundException ex) { System.out.println("Problème au chargement"+ex.toString());}`	Chargement du pilote MySQL.
`try { Connection cx = DriverManager.getConnection ("jdbc:mysql://localhost/bdsoutou?user=soutou&password=iut"); … }`	Déclaration de la connexion.
`catch(SQLException ex) { … }`	Gestion des erreurs.

Déconnexion

Appliquez la méthode `close()` à tous les objets `Connection` ouverts avant de terminer vos programmes.

Interface Connection

Le tableau suivant présente les principales méthodes disponibles de l'interface `Connection`. Nous détaillerons l'invocation de certaines de ces méthodes à l'aide des exemples des sections suivantes.

Tableau 9-10 Méthodes de l'interface Connection

Méthode	Description
`createStatement()`	Création d'un objet destiné à recevoir un ordre SQL statique non paramétré.
`prepareStatement(String)`	Précompile un ordre SQL acceptant des paramètres et pouvant être exécuté plusieurs fois.
`prepareCall(String)`	Appel d'une procédure cataloguée (certains pilotes attendent `execute` ou ne supportent pas `prepareCall`).
`void setAutoCommit(boolean)`	Positionne ou non le *commit* automatique.
`void commit()`	Valide la transaction.
`void rollback()`	Invalide la transaction.
`void close()`	Ferme la connexion.

Sources de données

Apparue avec JDBC 3, l'interface `javax.sql.DataSource` permet de créer une connexion sans charger dynamiquement un pilote (et donc, sans utiliser l'interface `DriverManager`). Oracle implémente cette interface à travers la classe `OracleDataSource`, située dans le paquetage `oracle.jdbc.pool`.

Il est préférable d'utiliser ce mécanisme de connexion, car il est plus souple de modifier les propriétés d'une source de données une fois créée (par exemple, le nom de la base, son emplacement physique ou encore le type du pilote). De plus, les instances de l'interface `Connection`, selon l'implémentation fournie par des DataSource, permettent de nouvelles fonctionnalités telles qu'un pool de connexion ou une transaction distribuée. Enfin, la connexion à un annuaire de type JNDI (*Java Naming and Directory Interface*) est facilitée.

Sans traiter de ces aspects avancés, intéressons-nous à l'implémentation basique d'une source de données. Le code suivant (`DataSourceExemple.java`) décrit l'initialisation de chacun des six éléments constituant une connexion.

Tableau 9-11 Connexion à l'aide d'une source de données

Code Java	Commentaires
`try {`	
`oracle.jdbc.pool.OracleDataSource ds;` `ds = new oracle.jdbc.pool.OracleDataSource();`	Création de la source.

`ds.setDriverType("thin");`
`ds.setServerName("camparols");`
`ds.setNetworkProtocol("tcp");`
`ds.setDatabaseName("bd10gR2");`
`ds.setPortNumber(1521);`
	Initialisation des éléments.

`ds.setUser("soutou");`
`ds.setPassword("iut");`
`Connection conn = ds.getConnection();`
`… }`
	Création de la connexion.

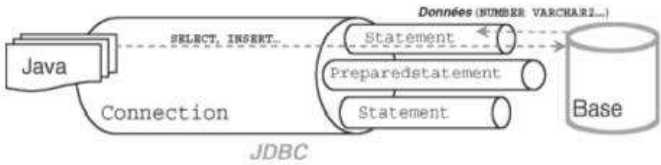
`catch(SQLException ex) { … }`	Gestion des erreurs.

Notez l'existence de la méthode `setURL("jdbc:oracle:thin:@//camparols:1521/bd10gR2")` qui permet de regrouper certains éléments. Par ailleurs, il est possible d'utiliser la méthode `getConnection("soutou","iut")` sur l'objet source de données avant de créer la connexion.

États d'une connexion

Une fois la connexion établie, il est nécessaire de définir des états qui permettront l'encapsulation d'instructions SQL dans du code Java. Un état permet de faire passer plusieurs instructions SQL sur le réseau. On peut affecter à un état une (ou plusieurs) instruction SQL. Si on désire exécuter plusieurs fois la même instruction, il est intéressant de réserver l'utilisation d'un état à cet effet.

Figure 9-4 Connexion et états



Interfaces disponibles

Différentes interfaces sont prévues à cet effet :

- `Statement` pour les ordres SQL statiques. Ces états sont construits par la méthode `createStatement` appliquée à la connexion.
- `PreparedStatement` pour les ordres SQL paramétrés. Ces états sont construits par la méthode `prepareStatement` appliquée à la connexion.

- `CallableStatement` pour les procédures ou fonctions cataloguées (PL/SQL, C, Java, etc.). Ces états sont construits par la méthode `prepareCall` appliquée à la connexion.

S'il ne doit plus être utilisé dans la suite du code Java, chaque objet de type `Statement`, `PreparedStatement` ou `CallableStatement` devra être fermé à l'aide de la méthode `close`.

Méthodes génériques pour les paramètres

Une fois qu'un état est créé, il est possible de lui passer des paramètres par des méthodes génériques (étudiées plus en détail par la suite) :

- `setxxx` où *xxx* désigne le type de la variable (exemple : `setString` ou `setInt`) du sens Java vers Oracle (*setter methods*). Il s'agit ici de paramétrer un ordre SQL (instruction ou appel d'un sous-programme) ;
- `getxxx` (exemple : `getString` ou `getInt`) du sens Oracle vers Java. Il s'agit ici d'extraire des données de la base dans des variables hôtes Java via un curseur Java (*getter methods*) ;
- `updatexxx` (exemple : `updateString` ou `upadateInt`) du sens Java vers Oracle. Il s'agit ici de mettre à jour des données de la base via un curseur Java (*updater methods*). Ces méthodes sont disponibles seulement depuis la version 2 de JDBC (SDK 1.2).

États simples (interface Statement)

Nous décrivons ici l'utilisation d'un état simple (interface `Statement`). Nous étudierons par la suite les instructions paramétrées (interface `PreparedStatement`) et appels de sous-programmes (interface `CallableStatement`). Le tableau suivant décrit les principales méthodes de l'interface `Statement`.

Tableau 9-12 Méthodes de l'interface Statement

Méthode	Description
`ResultSet executeQuery(String)`	Exécute une requête et retourne un ensemble de lignes (objet `ResultSet`).
`int executeUpdate(String)`	Exécute une instruction SQL et retourne le nombre de lignes traitées (`INSERT`, `UPDATE` ou `DELETE`) ou 0 pour les instructions ne retournant aucun résultat (LDD).
`boolean execute(String)`	Exécute une instruction SQL et renvoie `true` si c'est une instruction `SELECT`, `false` sinon (instructions LMD ou plusieurs résultats `ResultSet`).
`Connection getConnection()`	Retourne l'objet de la connexion.
`void setMaxRows(int)`	Positionne la limite du nombre d'enregistrements à extraire par toute requête issue de cet état.
`int getUpdateCount()`	Nombre de lignes traitées par l'instruction SQL (-1 si c'est une requête ou si l'instruction n'affecte aucune ligne).
`void close()`	Ferme l'état.

Le code suivant (Etats.java) présente quelques exemples d'utilisation de ces méthodes sur un état (objet etatSimple). Nous supposons qu'un pilote JDBC est chargé et que la connexion cx a été créée. Nous verrons en fin de chapitre comment traiter proprement les exceptions.

Tableau 9-13 États simples

Code Java	Commentaires
`Statement etatSimple = cx.createStatement();`	Création de l'état.
`etatSimple.execute("CREATE TABLE Compagnie(comp VARCHAR(4), nomComp VARCHAR(30), CONSTRAINT pk_Compagnie PRIMARY KEY(comp))");`	Ordre LDD.
`int j = etatSimple.executeUpdate("CREATE TABLE Avion (immat VARCHAR(6),typeAvion VARCHAR(15), cap NUMBER(3), compa VARCHAR(4), CONSTRAINT pk_Avion PRIMARY KEY(immat), CONSTRAINT fk_Avion_comp_Compagnie FOREIGN KEY(compa) REFERENCES Compagnie(comp))");`	Ordre LDD (autre écriture), *j* contient 0 (aucune ligne n'est concernée).
`int k = etatSimple.executeUpdate("INSERT INTO Compagnie VALUES ('AF','Air France')");`	Ordre LMD, *k* contient 1 (une ligne est concernée).
`etatSimple.execute("INSERT INTO Avion VALUES ('F-WTSS','Concorde',90,'AF')");` `etatSimple.execute("INSERT INTO Avion VALUES ('F-FGFB','A320',148,'AF')");`	Ordres LMD (autres écritures).
`etatSimple.setMaxRows(10);`	Pas plus de 10 lignes retournées.
`ResultSet curseurJava = etatSimple.executeQuery("SELECT * FROM Avion");`	Chargement d'un curseur Java.
`etatSimple.execute("DELETE FROM Avion");` `int l = etatSimple.getUpdateCount();`	Ordre LMD, *l* contient 2 (avions supprimés).

Méthodes à utiliser

Le tableau suivant indique la méthode préférentielle à utiliser sur l'état courant (objet Statement) en fonction de l'instruction SQL à émettre :

Tableau 9-14 Méthodes Java pour les ordres SQL

Instruction SQL	Méthode	Type de retour
CREATE ALTER DROP	executeUpdate	int
INSERT UPDATE DELETE	executeUpdate	int
SELECT	executeQuery	ResultSet

Correspondances de types

Les échanges de données entre variables Java et colonnes des tables Oracle impliquent de prévoir des conversions de types. Les tableaux suivants présentent les principales correspondances existantes :

Tableau 9-15 Correspondances entre les types (JDBC 1.0)

Types SQL	Types JDBC `java.sql.Types`	Types Java standards	Extensions Oracle des types Java `oracle.sql`
CHAR	CHAR	lang.String	CHAR
VARCHAR2	VARCHAR	lang.String	CHAR
LONG	LONGVARCHAR	lang.String	CHAR
NUMBER	NUMERIC	math.BigDecimal	NUMBER
NUMBER	DECIMAL	math.BigDecimal	NUMBER
NUMBER	BIT	boolean	NUMBER
NUMBER	TINYINT	byte	NUMBER
NUMBER	SMALLINT	short	NUMBER
NUMBER	INTEGER	int	NUMBER
NUMBER	BIGINT	long	NUMBER
NUMBER	REAL	float	NUMBER
NUMBER	FLOAT	double	NUMBER
NUMBER	DOUBLE	double	NUMBER
RAW	BINARY	byte	RAW
RAW	VARBINARY	byte	RAW
LONGRAW	LONGVARBINARY	byte	RAW
DATE	DATE	java.sql.Date	DATE
DATE	TIME	java.sql.Time	DATE
DATE	TIMESTAMP	java.sql.Timestamp	DATE

Tableau 9-16 Correspondances entre les types (JDBC 2.0)

Types SQL	Types JDBC `java.sql.Types`	Types Java standards	Extensions Oracle des types Java `oracle.sql`
BLOB	BLOB	java.sql.Blob	oracle.sql.BLOB
CLOB	CLOB	java.sql.Clob	oracle.sql.CLOB
Type objet	STRUCT	java.sql.Struct	oracle.sql.STRUCT
Référence	REF	java.sql.Ref	oracle.sql.REF
Collection	ARRAY	java.sql.Array	oracle.sql.ARRAY

Tableau 9-17 Correspondances entre les types (JDBC Oracle)

Types SQL	Types JDBC `oracle.jdbc`	Types Java standards	Extensions Oracle des types Java `oracle.jbdc.`
`BFILE`	`OracleTypes.BFILE`	Néant	`BFILE`
`ROWID`	`OracleTypes.ROWID`	Néant	`ROWID`
`REF CURSOR`	`OracleTypes.CURSOR`	`ResultSet`	`oracle.jdbc.OracleResultSet`

Interactions avec la base

Détaillons à présent les différents scénarios que l'on peut rencontrer lors d'une manipulation de la base de données par un programme Java. Les tableaux suivants répertorient les conséquences les plus fréquentes. Les autres cas (relatifs aux contraintes référentielles et aux problèmes de syntaxe) seront étudiés dans la section « Traitement des exceptions ».

Suppression de données

Tableau 9-18 Enregistrements présents dans la table

Code Java	Résultat
`etat.executeUpdate("DELETE FROM Avion");`	Fait la suppression et passe en séquence.
`j = etat.executeUpdate("DELETE FROM Avion");`	Fait la suppression, affecte à `j` le nombre d'enregistrements supprimés et passe en séquence.

Tableau 9-19 Aucun enregistrement dans la table

Code Java	Résultat
`etat.executeUpdate("DELETE FROM Avion");`	Aucune action sur la base et passe en séquence.
`j = etat.executeUpdate("DELETE FROM Avion");`	Aucune action sur la base, affecte à `j` la valeur `0` et passe en séquence.

Ajout d'enregistrements

Tableau 9-20 Différentes écritures d'un INSERT

Code Java	Résultat
`etat.executeUpdate("INSERT INTO Compagnie VALUES ('TAF','Toulouse Air Free')");`	Fait l'insertion et passe en séquence.
`int j= etat.executeUpdate("INSERT INTO Compagnie VALUES ('TAF','Toulouse Air Free')");`	Fait l'insertion, affecte à `j` le nombre `1` et passe en séquence.

Modification d'enregistrements

Tableau 9-21 Différentes écritures d'un UPDATE

Code Java	Résultat
`etat.executeUpdate("UPDATE Compagnie SET nomComp = 'Air France Compagny' WHERE comp = 'AF'");`	Fait la modification et passe en séquence. Si aucun enregistrement n'est concerné, aucune exception n'est levée.
`int j= etat.executeUpdate("UPDATE Avion SET capacite=capacite*1.2");`	Fait la (les) modification(s), affecte à `j` le nombre d'enregistrements modifiés et passe en séquence (`0` si aucun enregistrement n'est modifié).

Extraction de données

Étudions ici la gestion des résultats d'une instruction `SELECT`.

Le résultat d'une requête est affecté dans un objet de l'interface `ResultSet` qui s'apparente à un curseur Java.

Le tableau suivant présente les principales méthodes disponibles de l'interface `ResultSet`. Les méthodes relatives aux curseurs navigables seront étudiées par la suite. Le parcours de ce curseur s'opère par la méthode `next`. Initialement (après création et chargement du curseur), on est positionné avant la première ligne. Bien qu'un objet de l'interface `ResultSet` soit automatiquement fermé quand son état est fermé ou recréé, il est préférable de le fermer explicitement par la méthode `close` s'il ne doit pas être réutilisé.

Tableau 9-22 Méthodes principales de l'interface ResultSet

Méthode	Description
`boolean next()`	Charge l'enregistrement suivant en retournant `true`, retourne `false` lorsqu'il n'y a plus d'enregistrement suivant.
`void close()`	Ferme le curseur.
`getxxx(int)`	Récupère, au niveau de l'enregistrement, la valeur de la colonne numérotée de type *xxx*. Exemple : `getInt`(1), `getString`(1), `getDate`(1), etc. pour récupérer la valeur de la première colonne.
`updatexxx(…)`	Modifie, au niveau de l'enregistrement, la valeur de la colonne numérotée de type *xxx*. Exemple : `updateInt`(1,*i*), `updateString`(1,`nom`), etc.
`ResultSetMetaData getMetaData()`	Retourne un objet `ResultSetMetaData` correspondant au curseur.

Distinguons l'instruction `SELECT` qui génère un curseur statique (objet `Resultset` utilisé sans option particulière) de celle qui produit un curseur navigable ou modifiable (objet `Resultset` employé avec des options disponibles depuis la version 2 de JDBC).

Curseurs statiques

Le code suivant (`SELECTstatique.java`) extrait les avions de la compagnie 'Air France' par l'intermédiaire du curseur `curseurJava`. Notez l'utilisation des différentes méthodes `get` pour récupérer des valeurs issues de colonnes.

Tableau 9-23 Extraction de données dans un curseur statique

Code Java	Commentaires
`try {…` `Statement etatSimple = cx.createStatement();`	Création de l'état.
`ResultSet curseurJava =` `  etatSimple.executeQuery("SELECT immat, cap FROM Avion WHERE comp = (SELECT comp FROM Compagnie WHERE nomComp='Air France')");`	Création et chargement du curseur.

`float moyenneCapacité =0;`
`int nbAvions = 0;`
`while (curseurJava.next())`
`{System.out.print("Immat : "+curseurJava.getString(1));`
`System.out.println("Capacité : "+curseurJava.getInt(2));`
`moyenneCapacité += curseurJava.getInt(2);`
`nbAvions ++; }`
`moyenneCapacité /= nbAvions;`
`System.out.println("Capacité moy : "+moyenneCapacité);`
`curseurJava.close();`
	Parcours du curseur.
	Extraction de colonnes.
	
	Fermeture du curseur.

`} catch(SQLException ex) { … }`	Gestion des erreurs.

Curseurs navigables

Un curseur `ResultSet` déclaré sans option n'est ni navigable ni modifiable. Seul un déplacement du début vers la fin (par la méthode `next`) est permis. Il est possible de rendre un curseur navigable en permettant de le parcourir en avant ou en arrière, et en rendant possible l'accès direct à un enregistrement d'une manière absolue (en partant du début ou de la fin du curseur) ou relative (en partant de la position courante du curseur). Il est aussi possible de rendre un curseur modifiable (la base pourra être changée par l'intermédiaire du curseur).

Dès l'instant où on déclare un curseur navigable, il faut aussi statuer sur le fait qu'il soit modifiable ou pas (section suivante). La nature du curseur est explicitée à l'aide d'options de la méthode `createStatement` :

```
Statement createStatement(int typeCurseur, int modifCurseur)
```

Constantes

Les valeurs permises du premier paramètre (*`typeCurseur`*), et qui concernent le sens de parcours, sont présentées dans le tableau suivant :

Tableau 9-24 Constantes de navigation d'un curseur

Constante	Explication
`ResultSet.TYPE_FORWARD_ONLY`	Le parcours du curseur s'opère invariablement du début à la fin (non navigable).
`ResultSet.TYPE_SCROLL_INSENSITIVE`	Le curseur est navigable mais pas sensible aux modifications.
`ResultSet.TYPE_SCROLL_SENSITIVE`	Le curseur est navigable et sensible aux modifications.

Un curseur est sensible dès que des mises à jour de la table sont automatiquement répercutées au niveau du curseur durant la transaction. Lorsque le curseur est déclaré insensible, les modifications de la table ne sont pas répercutées dans le curseur.

Méthodes

Les principales méthodes que l'on peut appliquer à un curseur navigable sont les suivantes. Les deux premières sont aussi des méthodes de l'interface `Statement` qui affectent et précisent le sens de parcours pour tous les curseurs de l'état donné.

Tableau 9-25 Méthodes de navigation dans un curseur

Méthode	Fonction
`void setFetchDirection(int)`	Affecte la direction du parcours : `ResultSet.FETCH_FORWARD` (1000), `ResultSet.FETCH_REVERSE` (1001) ou `ResultSet.FETCH_UNKNOWN` (1002).
`int getFetchDirection()`	Extrait la direction courante (une des trois valeurs ci-dessus).
`boolean isBeforeFirst()`	Indique si le curseur est positionné avant le premier enregistrement (`false` si aucun enregistrement n'existe).
`void beforeFirst()`	Positionne le curseur avant le premier enregistrement (aucun effet si le curseur est vide).
`boolean isFirst()`	Indique si le curseur est positionné sur le premier enregistrement (`false` si aucun enregistrement n'existe).
`boolean isLast()`	Indique si le curseur est positionné sur le dernier enregistrement (`false` si aucun enregistrement n'existe).
`boolean isAfterLast()`	Indique si le curseur est positionné après le dernier enregistrement (`false` si aucun enregistrement n'existe).
`void afterLast()`	Positionne le curseur après le dernier enregistrement (aucun effet si le curseur est vide).
`boolean first()`	Positionne le curseur sur le premier enregistrement (`false` si aucun enregistrement n'existe).
`boolean previous()`	Positionne le curseur sur l'enregistrement précédent (`false` si aucun enregistrement ne précède).
`boolean last()`	Positionne le curseur sur le dernier enregistrement (`false` si aucun enregistrement n'existe).
`boolean absolute(int)`	Positionne le curseur sur le n-ièmeenregistrement (en partant du début si *n* positif, ou de la fin si *n* négatif, `false` si aucun enregistrement n'existe à cet indice).
`boolean relative(int)`	Positionne le curseur sur le n-ième enregistrement en partant de la position courante (en avant si *n* positif, ou en arrière si *n* négatif, `false` si aucun enregistrement n'existe à cet indice).

Oracle ne permet pas encore de changer le sens de parcours d'un curseur au niveau de l'état et du curseur lui-même (seule la constante `ResultSet.FETCH_FORWARD` est interprétée). Aucune erreur n'a lieu à l'exécution si vous modifiez le sens de parcours d'un curseur, la direction restera simplement inchangée.

Ainsi, pour parcourir un curseur à l'envers, il faudra utiliser des indices négatifs (dans les méthodes `absolute` et `relative`) ou la méthode `previous` en partant de la fin du curseur.

Parcours

Le code suivant (`SELECTnavigable.java`) présente une utilisation du curseur navigable `curseurNaviJava`. Le deuxième test renvoie `false`, car, après l'ouverture, le curseur n'est

pas positionné sur le premier enregistrement, et la méthode next le place selon le sens du parcours du curseur.

Tableau 9-26 Parcours d'un curseur navigable

Code Java	Commentaires

`try {…`
`Statement etatSimple =createStatement`
`(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);`
	Création de l'état.

`ResultSet curseurNaviJava = etatSimple.execute-` `Query("SELECT immat,typeAvion,cap FROM Avion");`	Création et chargement du curseur.
`if ( curseurNaviJava.isBeforeFirst() )` `System.out.println("Curseur positionné au début");`	Test renvoyant `true`.
`if ( curseurNaviJava.isFirst() )` `System.out.println("Curseur positionné sur le 1er déjà");`	Test renvoyant `false`.

`while( curseurNaviJava.next() )`
`{if ( curseurNaviJava.isFirst() )`
`System.out.println("1er avion : ");`
`if ( curseurNaviJava.isLast() )`
`System.out.println("Dernier avion : ");`
`System.out.print("Immat: "+curseurNaviJava.getString(1));`
`System.out.println(" type : "+curseurNaviJava.getString(2));}`
	Parcours du curseur en affichant les premier et dernier enregistrements.

`if ( curseurNaviJava.isAfterLast() )System.out.println("Curseur positionné après la fin");`	Test renvoyant `true`.

`if ( curseurNaviJava.previous() )`
`if ( curseurNaviJava.previous() )`
`{System.out.prinln("Avant dernier avion : "+`
`curseurNaviJava.getString(1));}`
	Affiche l'avant-dernier enregistrement.

`if ( curseurNaviJava.first() )`
`{System.out.println("First avion : "+`
`curseurNaviJava.getString(1));}`
	Affiche le premier enregistrement.

`if ( curseurNaviJava.last() )`
`{System.out.println("Last avion : "+`
`curseurNaviJava.getString(1));}`
	Affiche le dernier enregistrement.

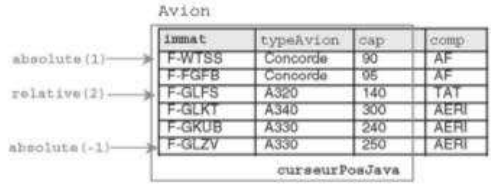
`curseurNaviJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Créez des curseurs non navigables quand vous voulez rapatrier de très gros volumes de données (taille du cache limitative côté client). Fragmentez vos requêtes quand vous voulez manipuler des curseurs navigables. Les prochaines versions d'Oracle verront une gestion côté serveur des curseurs navigables.

Positionnements

Des méthodes assurent l'accès direct à un curseur navigable. Notez que `absolute(1)` équivaut à `first()`, de même `absolute(-1)` équivaut à `last()`. Concernant la méthode `relative`, il faut l'utiliser dans un test pour s'assurer qu'elle s'applique à un enregistrement existant, par ailleurs `relative(0)` n'a aucun effet. Considérons la table suivante qui est interrogée au niveau des trois premières colonnes par le curseur navigable `curseurPosJava` :

Figure 9-5 *Curseur navigable*



Le code suivant (`SELECTPositions.java`) présente les méthodes qui permettent d'accéder directement à des enregistrements de ce curseur :

Tableau 9-27 Positionnements dans un curseur navigable

Code Java	Commentaires

`try {…`
`Statement etatSimple =createStatement`
`(ResultSet.TYPE_SCROLL_INSENSITIVE,`
`ResultSet.CONCUR_READ_ONLY);`
	Création de l'état avec curseurs insensibles et non modifiables.

`ResultSet curseurPosJava =`
`etatSimple.executeQuery("SELECT immat,typeAvion,cap`
`FROM Avion");`
	Création et chargement du curseur.

`curseurPosJava.absolute(1);`	Curseur sur le premier avion.

`if (curseurPosJava.relative(2))`
`System.out.println("relative(2): "+`
`curseurPosJava.getString(1));`
`else`
`System.out.println("Pas de 3ème avion!");`
	Accès au troisième avion.

`if (curseurPosJava.relative(-2))`
`System.out.println("relative(-2) : "+`
`curseurPosJava.getString(1));`
`else`
`System.out.println("Pas retour -2 possible !");`
	Retour au premier avion.

`if (curseurPosJava.absolute(-2)`
`System.out.println("absolute(-2) : "+`
`curseurPosJava.getString(1));`
`else`
`System.out.println("Pas d'avant dernier avion");`
	Accès à l'avant-dernier enregistrement.

Tableau 9-27 Positionnements dans un curseur navigable *(suite)*

Code Java	Commentaires
`curseurPosJava.afterLast();` `while(curseurPosJava.previous()) { … }`	Parcours du curseur en sens inverse.
`curseurPosJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Pour définir un curseur navigable :

- Une requête ne doit pas contenir de jointure.
- Écrivez (mais évitez) « `SELECT a.* FROM` *`table`* `a…` » à la place de « `SELECT * FROM` *`table`*`…` ».

Curseurs modifiables

Un curseur modifiable permet de mettre à jour la base de données : modification de colonnes, suppressions et insertions d'enregistrements.

Les valeurs permises du deuxième paramètre (*`modifCurseur`*) de la méthode `createStatement`, définie à la section précédente, sont présentées dans le tableau suivant :

Tableau 9-28 Constantes de modification d'un curseur

Constante	Explication
`ResultSet.CONCUR_READ_ONLY`	Le curseur ne peut être modifié.
`ResultSet.CONCUR_UPDATABLE`	Le curseur peut être modifié.

Le caractère modifiable d'un curseur est indépendant de sa navigabilité. Néanmoins, il est courant qu'un curseur modifiable soit également navigable (pour pouvoir se positionner à la demande sur un enregistrement avant d'effectuer sa mise à jour).

La gestion des accès concurrents n'est pas totalement assurée par les pilotes JDBC : aucune pose de verrou n'est automatiquement opérée à l'ouverture d'un curseur (il n'est pas possible de définir un curseur par une requête de type `SELECT… FOR UPDATE`).

Pour composer un curseur de nature `CONCUR_UPDATABLE` :

- Une requête ne doit pas contenir de jointure ni de regroupement.
- Écrivez (mais évitez) « `SELECT a.* FROM` *`table`* `a…` » à la place de « `SELECT * FROM` *`table`*`…` ».
- Une requête doit seulement extraire des colonnes (les fonctions monolignes et multilignes sont interdites).

Les principales méthodes relatives aux curseurs modifiables sont les suivantes :

Tableau 9-29 Méthodes de navigation dans un curseur

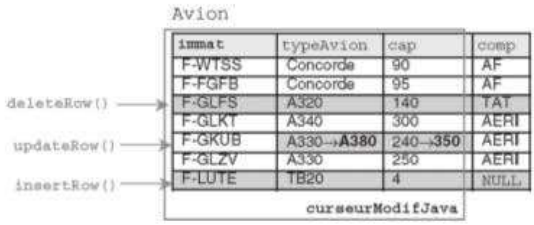
Méthode	Fonction
`int getResultSetType()`	Renvoie le caractère navigable des curseurs d'un état donné (`ResultSet.TYPE_FORWARD_ONLY...`).
`int getResultSetConcurrency()`	Renvoie le caractère modifiable des curseurs d'un état donné (`ResultSet.CONCUR_READ_ONLY` ou `ResultSet.CONCUR_UPDATABLE`).
`int getType()`	Renvoie le caractère navigable d'un curseur donné.
`int getConcurrency()`	Renvoie le caractère modifiable d'un curseur donné.
`void deleteRow()`	Supprime l'enregistrement courant.
`void updateRow()`	Modifie la table avec l'enregistrement courant.
`void cancelRowUpdates()`	Annule les modifications faites sur l'enregistrement courant.
`void moveToInsertRow()`	Déplace le curseur vers un nouvel enregistrement.
`void insertRow()`	Insère dans la table l'enregistrement courant.
`void moveToCurrentRow()`	Retour vers l'enregistrement courant (à utiliser éventuellement après `moveToInsertRow`).

Les opérations de modification et d'insertion (`UPDATE` et `INSERT`) à travers un curseur se réalisent en deux temps : mise à jour du curseur puis propagation à la table de la base de données. Il suffit ainsi de ne pas exécuter la deuxième étape pour ne pas opérer la mise à jour de la base.

La suppression d'enregistrements (`DELETE`) à travers un curseur s'opère en une seule instruction qui n'est pas forcément validée par la suite : il faudra programmer explicitement le `COMMIT` ou laisser le paramètre d'autocommit à `true` (par défaut).

La figure suivante illustre les modifications opérées sur la table `Avion` par l'intermédiaire du curseur `CurseurModifJava` utilisé par les trois programmes Java suivants :

Figure 9-6 Mises à jour d'un curseur



Suppressions

Le code suivant (ResultDELETE.java) supprime le troisième enregistrement du curseur et répercute la mise à jour au niveau de la table Avion du schéma connecté. Nous déclarons ici ce curseur « navigable » :

Tableau 9-30 Suppression d'un enregistrement

Code Java	Commentaires
`try {… Statement etatSimple = cx.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE); cx.setAutoCommit(false);`	Création de l'état et désactivation de la validation automatique.
`ResultSet curseurModifJava = etatSimple.executeQuery ("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`if (curseurModifJava.absolute(3)) { curseurModifJava.deleteRow(); cx.commit(); } else System.out.println("Pas de 3ème avion!");`	Accès direct au troisième avion, suppression de l'enregistrement.
`curseurModifJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Le code suivant (ResultDELETE2.java) supprime le même enregistrement en supposant son indice a priori inconnu. Nous déclarons ici ce curseur « non navigable ». Notez l'utilisation de la méthode equals pour comparer deux chaînes de caractères :

Tableau 9-31 Suppression d'un enregistrement

Code Java	Commentaires
`try {… Statement etatSimple = cx.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE); cx.setAutoCommit(false);`	Création de l'état et désactivation de la validation automatique.
`ResultSet curseurModifJava = etatSimple.executeQuery ("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`String p_immat = "F-GLFS"; while(curseurModifJava.next()) {if (curseurModifJava.getString(1).equals(p_immat)) {curseurModifJava.deleteRow(); cx.commit(); } }`	Accès à l'enregistrement et suppression.
`curseurModifJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Modifications

La modification de colonnes d'un enregistrement au niveau de la base de données s'opère en deux étapes : mise à jour du curseur par les méthodes `updatexxx` (*updater methods*) puis propagation des mises à jour dans la table par la méthode `updateRow()`.

Les méthodes `updatexxx` ont chacune deux signatures. Par exemple, la méthode de modification d'une chaîne de caractères (valable pour les colonnes `CHAR`, `VARCHAR` et `VARCHAR2`) est disponible en raisonnant en fonction soit de la position soit du nom de la colonne du curseur :

```
void updateString(int positionColonne, String chaîne)
void updateString(String nomColonne, String chaîne)
```

Le code suivant (`ResultUPDATE.java`) modifie, au niveau de la table `Avion`, deux colonnes du cinquième enregistrement du curseur. Nous déclarons ici ce curseur « sensible » pour pouvoir éventuellement visualiser la modification réalisée dans le même programme.

Tableau 9-32 Modifications d'un enregistrement

Code Java	Commentaires

`try {…`
`Statement etatSimple =`
`cx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,`
`ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseurModifJava = etatSimple.executeQuery` `("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.

`if (curseurModifJava.absolute(5))`
`{ curseurModifJava.updateString(2,"A380");`
`curseurModifJava.updateInt(3,350);`
 	
`curseurModifJava.updateRow();`
`cx.commit(); }`
`else`
`System.out.println("Pas de 5ème avion!");`
	Accès à l'enregistrement.
	Première étape.
	
	Deuxième étape.
	Validation.

`curseurModifJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Insertions

L'insertion d'un enregistrement au niveau de la base de données s'opère en trois étapes : préparation à l'insertion dans le curseur par la méthode `moveToInsertRow`, mise à jour du curseur par les méthodes `updatexxx`, puis propagation des mises à jour dans la table par la

méthode `insertRow`. L'éventuel retour à l'enregistrement courant se programme à l'aide de la méthode `moveToCurrentRow`.

Le code suivant (`ResultINSERT.java`) insère un nouvel enregistrement au niveau de la table `Avion`. La quatrième colonne de la table n'est pas indiquée dans le curseur, elle est donc passée à `NULL` au niveau de la table en l'absence de valeur par défaut définie dans la colonne.

Tableau 9-33 Insertion d'un enregistrement

Code Java	Commentaires

`try {…`
`Statement etatSimple =`
`cx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,`
`ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseurModifJava = etatSimple.executeQuery` `("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`curseurModifJava.moveToInsertRow();`	Première étape.

`curseurModifJava.updateString(1,"F-LUTE");`
`curseurModifJava.updateString(2,"TB20");`
`curseurModifJava.updateInt(3,4);`
	Deuxième étape.

`curseurModifJava.insertRow();` `cx.commit();`	Troisième étape. Validation.
`curseurModifJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Restrictions

Les limitations d'Oracle sont, pour l'heure, les suivantes :

En travaillant avec des curseurs navigables, il n'est pas possible de se positionner sur un enregistrement avec les méthodes `beforeFirst` ou `afterLast` avant de supprimer, modifier ou d'insérer un enregistrement.

On ne peut avoir accès en lecture à un nouvel enregistrement inséré au sein du même programme Java (que le curseur soit sensible ou pas).

Ensembles de lignes (RowSet)

Introduit avec JDBC 2.0, le concept de *RowSet* est natif dans JDK 5. Un *RowSet* est un objet qui encapsule un ensemble de lignes (de type *ResultSet* ou d'une source de données tabulaire), qui permet un mode de développement s'apparentant aux *Java Beans*, incluant un ensemble de propriétés et un mécanisme de notifications. Un *RowSet* peut être mis à jour et tout mouvement d'un curseur est permis (même si la base de données ou le pilote ne fournit pas nativement ces fonctionnalités).

- L'interface `RowSet` (qui hérite de `ResultSet`) du paquetage `javax.sql` autorise la configuration d'un ensemble de lignes (nom de l'utilisateur, URL de la connexion ou instruction SQL), grâce à des méthodes de type `setXXX`. Il n'est donc plus nécessaire d'implémenter explicitement une `Connection` ou un `Statement`.
- L'interface `RowSetListener` permet la gestion des événements relatifs aux *RowSets*.

Deux catégories de *RowSets* se distinguent.

- **Les *RowSets* connectés.** Ils fonctionnent de la même manière que les *ResultSets* et gardent une connexion au SGBD durant leur cycle de vie.
- **Les *RowSets* déconnectés.** Ils sont capables d'interrompre la connexion à la base, d'opérer des modifications, puis de se reconnecter en transmettant les mises à jour, tout en gérant d'éventuels conflits.

Présentes dans le paquetage `javax.sql.rowset`, les interfaces suivantes héritent toutes de *RowSet*, mais elles implémentent chacune un type différent de *RowSets*.

- `CachedRowSet` : *RowSet* déconnecté particulièrement adapté aux clients légers (PDA ou smartphones) et à un volume restreint de données. Pour contourner cette limitation, optez pour l'interface `OracleCachedRowSet` présente dans le paquetage `oracle.jdbc.rowset`.
- `WebRowSet` : *RowSet* dérivé du `CachedRowSet` particulièrement adapté aux applications Web et aux flux de données XML. L'interface implémentée par Oracle est `OracleCachedRowSet` (paquetage `oracle.jdbc.rowset`).
- `FilteredRowSet` : *RowSet* dérivé du `WebRowSet`. Il permet de filtrer des données grâce à l'interface `Predicate`. Pour ceux qui ne savent pas écrire des conditions SQL.
- `JoinRowSet` : *RowSet* dérivé du `WebRowSet`. Il est adapté aux jointures de plusieurs *RowSets*. Pour ceux qui ne savent pas écrire des jointures SQL.
- `JdbcRowSet` : *RowSet* connecté. Il simule un `ResultSet` sous la forme d'un *Java Bean*.

Avant de manipuler un *RowSet*, il faut suivre trois étapes.

1. La première implémente une des interfaces pour obtenir une instance du *RowSet*.
2. Par la suite, vous devrez spécifier les propriétés de cette instance.
3. Enfin, il faudra peupler ce *RowSet* par les données désirées.

RowSet sans connexion

Le code suivant (`RowSet1.java`) illustre plusieurs avantages d'un *RowSet* : il n'est plus nécessaire de créer explicitement une connexion et un état (`Statement`). De plus, une instruction peut être paramétrée (à la manière d'un `PreparedStatement`).

Vous devrez importer le paquetage `oracle.jdbc.rowset.OracleCachedRowSet`.

Tableau 9-34 RowSet déconnecté et paramétré

Code Java	Commentaires
`OracleCachedRowSet rowset = new OracleCachedRowSet();`	Création du *RowSet*.

`rowset.setUrl("jdbc:oracle:thin:@//soutou-PC-W7:1521/XE");`
`rowset.setUsername("soutou");`
`rowset.setPassword("iut");`
`rowset.setCommand("SELECT immat,cap,typeavion`
`FROM Avion WHERE comp=?");`
`rowset.setString(1, "AERI");`
`rowset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);`
`rowset.setConcurrency(ResultSet.CONCUR_UPDATABLE);`
	Spécification des propriétés du *RowSet*.

`rowset.execute();`
`while (rowset.next())`
`System.out.println(rowset.getString(1)+`
`"-"+rowset.getInt(2)+"-"+rowset.getString(3));`
	Manipulation du *RowSet*.

RowSet avec ResultSet

Le code suivant (`RowSet2.java`) présente un autre avantage d'un *RowSet* : pouvoir manipuler les données extraites d'un *ResultSet* après que la connexion soit fermée. Notez la méthode `populate` qui initialise un *RowSet* à partir d'un *ResulSet*.

Tableau 9-35 RowSet déconnecté et peuplé par un ResultSet

Code Java	Commentaires

`oracle.jdbc.pool.OracleDataSource ds;`
`ds = new oracle.jdbc.pool.OracleDataSource();`
`ds.setURL("jdbc:oracle:thin:@//soutou-PC-W7:1521/XE");`
`Connection cx = ds.getConnection("soutou","iut");`
`Statement stmt = cx.createStatement();`
`ResultSet rset = stmt.executeQuery`
`("SELECT immat,cap,typeavion,comp FROM Avion");`
	Création de la connexion, de l'état et d'un *ResultSet*

`OracleCachedRowSet rowset = new OracleCachedRowSet();`
`rowset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);`
`rowset.setConcurrency(ResultSet.CONCUR_UPDATABLE);`
	Création et spécification des propriétés du *RowSet*.

`rowset.populate(rset);`	Chargement du *RowSet*.

Tableau 9-35 RowSet déconnecté et peuplé par un ResultSet *(suite)*

Code Java	Commentaires

`cx.close();`
`rowset.afterLast();`
`if (rowset.previous())`
`  System.out.println(rowset.getString(1)+`
`  "-"+rowset.getInt(2)+"-"+rowset.getString(3));`
	Après fermeture, lecture de la dernière ligne du *RowSet*.

RowSet pour XML

Le code suivant (`RowSet3.java`) présente la génération d'un fichier XML par l'intermédiaire d'un *RowSet* de type `WebRowSet`. On suppose la connexion et l'état créés. Notez l'utilisation de la méthode `writeXml` qui génère en une passe le document XML.

Tableau 9-36 RowSet pour générer du XML

Code Java	Commentaires
`ResultSet rset = stmt.executeQuery` `  ("SELECT immat,cap,typeavion,comp FROM Avion");`	Création du *ResultSet*.

`OracleWebRowSet wset = new OracleWebRowSet();`
`wset.populate(rset);`
`try`
`{ FileWriter out = new FileWriter("avions-base.xml");`
`  wset.writeXml(out);`
`}`
`catch (IOException exc)`
`  { System.out.println("Problème avec FileWriter");,}`
	Création et chargement du *RowSet*.
	
	Génération du fichier XML.

Le fichier XML généré contient les données sous l'élément `data`. Les premières balises (`properties` et `metadata`) renseignent, d'une part, la connexion et, d'autre part, la structure du résultat.

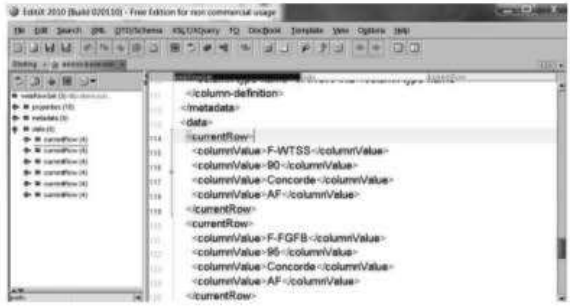


Figure 9-7 *Fichier XML généré*

À l'inverse, la méthode `readXml` charge un nouveau *RowSet* à partir d'un document XML passé en paramètre (sous réserve qu'il vérifie la grammaire attendue).

Mises à jour d'un RowSet

Un *RowSet* n'est pas continuellement connecté à la source (à part les JDBC *RowSets*), sa mise à jour nécessite donc l'appel de la méthode `acceptChanges` qui transmet les modifications à la source. La méthode `commit` est également nécessaire si on n'est pas en mode *autocommit*.

Il est à noter que la connexion (ou reconnexion) à la source s'opère d'une manière transparente à l'invocation des méthodes `execute` ou `acceptChanges` (sous réserve que les propriétés *user*, *password* et *URL* soient correctement initialisées).

Le code suivant (`RowSet4.java`) présente deux mises à jour de la table `Avion` (une modification et un ajout) par l'intermédiaire d'un *RowSet* de type `CachedRowSet`. On suppose ce *RowSet* créé d'une manière identique au premier exemple (`RowSet1.java`).

Tableau 9-37 Mises à jour d'un RowSet

Code Java	Commentaires

`rowset.setCommand("SELECT immat,cap,typeavion,comp FROM Avion");`
`rowset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);`
`rowset.setConcurrency(ResultSet.CONCUR_UPDATABLE);`
	Création du *RowSet*.

`rowset.execute();`	Chargement du *RowSet*.
`if (rowset.first())` `{ rowset.updateInt(2,92);rowset.updateRow(); }`	Modification d'une colonne.

`rowset.moveToInsertRow();`
`rowset.updateString("IMMAT", "F-GVFP");`
`rowset.updateString("TYPEAVION", "AT01");`
`rowset.updateInt("CAP", 16);`
`rowset.updateString("COMP", "AF");`
`rowset.insertRow();`
	Ajout d'une ligne.

`rowset.`**`acceptChanges`**`();` `rowset.`**`commit`**`();`	Validation.

Notifications pour un RowSet

Il est possible d'intervenir lors de toutes les mises à jour d'un *RowSet* par un processus d'écoute implémenté par l'interface `RowSetListener`, disponible dans le paquetage `javax.sql`. En implémentant cette interface à l'aide des méthodes suivantes.

Tableau 9-38 Méthodes de l'interface RowSetListener

Code Java	Notifications
`void cursorMoved(RowSetEvent event)`	Dès que le curseur est en mouvement.
`void rowChanged(RowSetEvent event)`	Dès qu'une ligne du curseur est modifiée.
`void rowSetChanged(RowSetEvent event)`	Dès que le curseur est modifié.

Le code suivant (`EcouteRowSet.java`) implémente l'interface et permettra de tracer les événements vécus par le *RowSet*.

Tableau 9-39 Implémentation de l'interface RowSetListener

Code Java	Commentaires

`import javax.sql.*;`
`public class EcouteRowSet implements RowSetListener`
`{public void cursorMoved(RowSetEvent event)`
`{ System.out.println("Le curseur bouge"); }`
`public void rowChanged(RowSetEvent event)`
`{ System.out.println("Une ligne du curseur change"); }`
`public void rowSetChanged(RowSetEvent event)`
`{ System.out.println("Le curseur change"); } }`
	Création d'une classe.
	
	Surcharge des trois méthodes.

Le code suivant (`RowSet5.java`) attache ce processus d'écoute à un *RowSet* par la méthode `addRowSetListener`. On suppose ce *RowSet* créé et initialisé d'une manière identique au précédent exemple (`RowSet4.java`). Pour détacher un processus d'écoute, vous devrez utiliser par analogie la méthode `removeRowSetListener`.

Tableau 9-40 Attachement d'un processus d'écoute à un RowSet

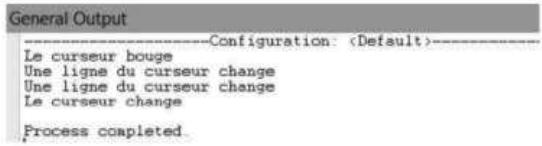
Code Java	Commentaires

`// idem début de RowSet4.java`
`…`
`EcouteRowSet ecoute = new EcouteRowSet();`
`rowset.`**`addRowSetListener`**`(ecoute);`
`if (rowset.first())`
	Création du *RowSet*.
	
	Affectation au *RowSet* d'un processus d'écoute.

`… rowset.updateInt(…)`
`…`
`rowset.moveToInsertRow();`
`…`
	Modifications du *RowSet*.

Les mises à jour produisent le résultat suivant. Deux lignes du *RowSet* sont bien mises à jour (la première concerne deux modifications, la seconde une insertion). La validation entraîne la mise à jour du *RowSet* dans son intégralité.

Figure 9-8 Trace des notifications d'un RowSet



Interface ResultSetMetaData

L'interface `ResultSetMetaData` est utile pour retrouver dynamiquement des propriétés des tables qui sont manipulées par des curseurs `ResultSet`. Cette interface est intéressante pour programmer dynamiquement des requêtes ou d'autres instructions SQL. Ces fonctions vont extraire de manière transparente des informations par l'intermédiaire du dictionnaire des données.

Une fois un curseur `ResultSet` programmé, il suffit de lui appliquer la méthode `getMetaData()` pour disposer d'un objet `ResultSetMetaData`. Le tableau suivant présente les principales méthodes disponibles de l'interface `ResultSetMetaData` :

Tableau 9-41 Méthodes principales de l'interface ResultSetMetaData

Méthode	Description
`int getColumnCount()`	Retourne le nombre de colonnes du curseur.
`String getColumnName(int)`	Retourne le nom de la colonne d'un indice donné du curseur.
`int getColumnType(int)`	Retourne le code du type (selon la classification de `java.sql.Types`) de la colonne d'un indice donné du curseur.
`String getColumnTypeName(int)`	Retourne le nom du type SQL de la colonne d'un indice donné du curseur.
`int isNullable(int)`	Indique si la colonne d'un indice donné du curseur peut être nulle (constantes retournées : `ResultSetMetaData.columnNoNulls`, `ResultSetMetaData.columnNullable` ou `ResultSetMetaData.columnNullableUnknown`).
`int getPrecision(int)`	Nombre de chiffres avant la virgule de la colonne désignée.
`int getScale(int)`	Nombre de décimales de la colonne désignée.
`String getSchemaName(int)`	Nom du schéma propriétaire de la colonne.
`String getTableName(int)`	Nom de la table de la colonne.

Oracle n'emploie pas encore les méthodes `getSchemaName()` et `getTableName()`.

Le code suivant (`ResulSetMeta.java`) utilise des méthodes de l'interface `ResultSetMetaData` sur la base de la requête extrayant trois colonnes dans la table `Avion`.

Tableau 9-42 Extraction de méta-informations au niveau d'un curseur

Code Java	Commentaires

`try { …`
`ResultSet curseurJava=etatSimple.executeQuery`
`("SELECT immat, typeAvion, cap FROM Avion");`
	Création du curseur.

`ResultSetMetaData rsmd =` `curseurJava.getMetaData();`	Création d'un objet `ResultSetMetaData`.
`int nbCol = rsmd.getColumnCount();`	`nbCol` contient 3.
`String nom2emeCol = rsmd.getColumnName(2);`	`nom2emeCol` contient `TYPEAVION`.
`String type2emeCol = rsmd.getColumnTypeName(2);`	`type2emeCol` contient `VARCHAR2`.
`int codeType2emeCol = rsmd.getColumnType(2);`	`codeType2emeCol` contient `12` (code pour `VARCHAR2`).

`if (rsmd.isNullable(1) ==`
`ResultSetMetaData.columnNoNulls)`
`…`
	Test renvoyant vrai (la première colonne est la clé primaire).

`curseurJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Interface DatabaseMetaData

L'interface `DatabaseMetaData` est utile pour connaître des aspects plus généraux de la base de données cible (version, éditeur, si les transactions sont supportées…) ou des informations sur la structure de la base (structures des tables et vues, prérogatives…).

Plus de quarante méthodes sont proposées par l'interface `DatabaseMetaData`. Le tableau suivant en présente quelques-unes. Consultez la documentation du JDK pour en savoir plus.

Tableau 9-43 Méthodes principales de l'interface ResultSetMetaData

Méthode	Description
`ResultSet getColumns(String, String, String, String)`	Description de toutes les colonnes d'une table d'un schéma donné.
`String getDatabaseProductName()`	Nom de l'éditeur de la base de données utilisée.
`String getDatabaseProductVersion()`	Numéro de la version de la base utilisée.
`ResultSet getTables(String, String, String, String[])`	Description des tables d'un schéma donné.
`String getUserName()`	Nom de l'utilisateur connecté (schéma courant).
`boolean supportsSavepoints()`	Renvoie `true` si la base supporte les points de validation.
`boolean supportsTransactions()`	Renvoie `true` si la base supporte les transactions.

Le code suivant (`MetaData.java`) utilise ces méthodes pour extraire des informations à propos de la base cible et des objets (tables, vues, séquences…) du schéma courant.

Tableau 9-44 Extraction de méta-informations au niveau d'un schéma

Code Java	Commentaires

`try {`
`…`
`DatabaseMetaData infoBase = cx.getMetaData();`
	Création d'un objet `DatabaseMetaData`.

`ResultSet toutesLesTables = infoBase.getTables("",` `infoBase.getUserName(), null, null);`	Création d'un objet `ResultSet` contenant les caractéristiques du schéma courant.

`while (toutesLesTables.next())`
`{ System.out.print("Nom de l'objet: "+`
`toutesLesTables.getString(3));`
`System.out.println("Type : "+`
`toutesLesTables.getString(4)); }`
	Parcours du curseur en affichant quelques caractéristiques.

`System.out.println("Nom base : "+` `infoBase.getDatabaseProductName());`	Affiche le nom de la base.
`System.out.println("Version base : "+` `infoBase.getDatabaseProductVersion());`	Affiche la version de la base.
`if (infoBase.supportsTransactions())` `System.out.println("Supporte les Transactions");`	Transactions supportées ou pas.
`toutesLesTables.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

La trace de ce programme est la suivante (dans notre jeu d'exemple) :

```
Objets du schéma SOUTOU
Nom de l'objet: AVION Type : TABLE
Nom de l'objet: COMPAGNIE Type : TABLE
…
Nom base : Oracle
Version base : Personal Oracle9i Release 9.2.0.3.0 - Production With the
Partitioning, OLAP and Oracle Data Mining options JServer Release
9.2.0.3.0 - Production
Supporte les Transactions
```

Instructions paramétrées (PreparedStatement)

L'interface `PreparedStatement` hérite de l'interface `Statement`, et la spécialise en permettant de paramétrer des objets (états préparés) représentant des instructions SQL précompilées. Ces états sont créés par la méthode `prepareStatement` de l'interface `Connection` décrite ci-après. La chaîne de caractères contient l'ordre SQL dont les paramètres, s'il en possède, doivent être indiqués par le symbole « ? ».

```
PreparedStatement prepareStatement(String)
```

Une fois créés, ces objets peuvent être aisément réutilisés pour exécuter à la demande l'instruction SQL, en modifiant éventuellement les valeurs des paramètres d'entrée à l'aide des méthodes `setxxx` (*setter methods*). Le tableau suivant décrit les principales méthodes de l'interface `PreparedStatement` :

Tableau 9-45 Méthodes de l'interface PreparedStatement

Méthode	Description
`ResultSet executeQuery()`	Exécute la requête et retourne un curseur ni navigable, ni modifiable par défaut.
`int executeUpdate()`	Exécute une instruction LMD (`INSERT`, `UPDATE` ou `DELETE`) et retourne le nombre de lignes traitées ou 0 pour les instructions SQL ne retournant aucun résultat (LDD).
`boolean execute()`	Exécute une instruction SQL et renvoie `true`, si c'est une instruction `SELECT`, `false` sinon.
`void setNull(int, int)`	Affecte la valeur `NULL` au paramètre de numéro et de type (classification `java.sql.Types`) spécifiés.
`void close()`	Ferme l'état.

Décrivons à présent un exemple d'appel pour chaque méthode de compilation d'un ordre paramétré. On suppose la connexion `cx` créée :

Extraction de données (executeQuery)

Le code suivant (PrepareSELECT.java) illustre l'utilisation de la méthode executeQuery pour extraire les enregistrements de la table Avion.

Tableau 9-46 Extraction de données par un ordre préparé

Code Java	Commentaires

`try { …`
`String ordreSQL =`
`"SELECT immat, typeAvion, cap FROM Avion";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
	Création d'un état préparé.

`ResultSet curseurJava =étatPréparé.executeQuery();`
 	
`while(curseurJava.next())`
`{ … }`
	Création du curseur résultant de la compilation de l'état.
	Parcours du curseur.

`curseurJava.close();`	Ferme le curseur.
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Mises à jour (executeUpdate)

Le code suivant (PrepareINSERT.java) illustre l'utilisation de la méthode executeUpdate pour insérer l'enregistrement (F-NEW, A319, 178, AF) dans la table Avion composée de quatre colonnes : CHAR(6), VARCHAR2(15), NUMBER(3) et VARCHAR2(4) :

Tableau 9-47 Insertion d'un enregistrement par un ordre préparé

Code Java	Commentaires

`try { …`
`String ordreSQL =`
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
	Création d'un état préparé.

`étatPréparé.setString(1, "F-NEW");`
`étatPréparé.setString(2, "A319");`
`étatPréparé.setInt(3, 178);`
`étatPréparé.setString(4, "AF");`
	Passage des paramètres.

`System.out.println(étatPréparé.executeUpdate()` `+ " avion inséré.");`	Exécution de l'instruction.
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Instruction LDD (execute)

Le code suivant (`PrepareDELETE.java`) illustre l'utilisation de la méthode `execute` pour supprimer un avion dont l'immatriculation passe en paramètre :

Tableau 9-48 Suppression d'un enregistrement par un ordre préparé

Code Java	Commentaires

`try { …`
`String ordreSQL =`
`"DELETE FROM Avion WHERE immat = ?";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
	Création d'un état préparé.

`étatPréparé.setString(1, "F-NEW ");`	Passage du paramètre.

`if (! étatPréparé.execute() )`
`{ System.out.println("Enregistrement supprimé");`
`cx.commit(); }`
	Exécution de l'instruction.

`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Il n'est pas possible de paramétrer des instructions SQL du LDD (`CREATE`, `ALTER`...). Pour résoudre ce problème, il faut construire dynamiquement la chaîne (`String`) qui contient l'instruction à l'aide de l'opérateur de concaténation Java (+). Cette chaîne sera ensuite l'unique paramètre de la méthode `prepareStatement`.

Appels de sous-programmes

L'interface `CallableStatement` permet d'appeler des sous-programmes (fonctions ou procédures cataloguées écrites en PL/SQL, Java...), en passant d'éventuels paramètres en entrée et en en récupérant en sortie. L'interface `CallableStatement` spécialise l'interface `PreparedStatement`. Les paramètres d'entrée sont affectés par les méthodes `setxxx`. Les paramètres de sortie (définis `OUT` au niveau du sous-programme) sont extraits à l'aide des méthodes `getxxx`.

Ces états qui permettent d'appeler des sous-programmes sont créés par la méthode `prepareCall` de l'interface `Connection`, décrite ci-après :

```
CallableStatement prepareCall(String)
```

Le tableau suivant décrit le paramètre de cette méthode (deux écritures sont possibles). Chaque paramètre est indiqué par un symbole ? :

Tableau 9-49 Paramètre de prepareCall

Type du sous-programme	Paramètre
Fonction	`{? = call nomFonction( [?, ?, …] ) }`
Procédure	`{call nomProcédure( [?, ?, …] ) }`

Une fois l'état créé, il faut répertorier le type des paramètres de sortie (méthode `registerOutParameter`), passer les valeurs des paramètres d'entrée, appeler le sous-programme et analyser les résultats. Le tableau suivant décrit les principales méthodes de l'interface `CallableStatement` :

Tableau 9-50 Méthodes de l'interface CallableStatement

Méthode	Description
`ResultSet executeQuery()`	Idem `PreparedStatement`.
`int executeUpdate()`	Idem `PreparedStatement`.
`boolean execute()`	Idem `PreparedStatement`.
`void registerOutParameter (int, int)`	Transfère un paramètre de sortie à un indice donné d'un type Java (classification `java.sql.Types`).
`boolean wasNull()`	Détermine si le dernier paramètre de sortie extrait est à `NULL`. Cette méthode doit être seulement invoquée après une méthode de type `getXxx`.

Appel d'une fonction

Le programme JDBC suivant (`CallableFonction.java`) décrit l'appel de la fonction `LeNomCompagnieEst` qui renvoie le nom de la compagnie d'un avion dont l'immatriculation passe en paramètre :

```
CREATE FUNCTION LeNomCompagnieEst(p_immat IN VARCHAR) RETURN VARCHAR IS
   résultat Compagnie.nomComp%TYPE;
BEGIN
   SELECT nomComp INTO résultat
   FROM Compagnie WHERE comp = (SELECT comp FROM Avion WHERE immat = p_
   immat);
   RETURN résultat;
EXCEPTION
   WHEN NO_DATA_FOUND THEN RETURN NULL;
END;
```

Nous appelons cette fonction pour l'avion d'immatriculation 'F-GLFS'.

Tableau 9-51 Appel d'une fonction

Code Java	Commentaires

`try { …`
`  String ordreSQL =`
`  "{? = call LeNomCompagnieEst(?)}";`
`CallableStatement étatAppelable =`
`  cx.prepareCall(ordreSQL);`
	Création d'un état appelable.

`étatAppelable.registerOutParameter` `  (1,java.sql.Types.VARCHAR);`	Déclaration du paramètre de sortie.
`étatAppelable.setString(2,"F-GLFS");`	Passage du paramètre d'entrée.
`étatAppelable.execute();`	Exécution de la fonction.
`System.out.print("Compagnie de F-GLFS : "+` `  étatAppelable.getString(1));`	Extraction du résultat.
`étatAppelable.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Appel d'une procédure

Le programme JDBC suivant (`CallableProcedure.java`) décrit l'appel de la procédure `AugmenteCapacité` (ayant deux paramètres) qui augmente la capacité d'un avion dont l'immatriculation passe en paramètre.

```
CREATE PROCEDURE AugmenteCapacité(p_immat IN VARCHAR,
p_n IN NUMBER) IS
BEGIN
   UPDATE Avion SET cap = cap + p_n WHERE immat = p_immat;
END;
```

Nous augmentons la capacité de l'avion 'F-GLFS' de 50 places :

Tableau 9-52 Appel d'une procédure

Code Java	Commentaires

`try { …`
`String ordreSQL = "{call AugmenteCapacité(?,?)}";`
`CallableStatement étatAppelable =`
`  cx.prepareCall(ordreSQL);`
	Création d'un état appelable.

`étatAppelable.setString(1,"F-GLFS");` `étatAppelable.setInt(2,50);`	Passage des paramètres d'entrée.
`étatAppelable.execute();`	Exécution de la procédure.
`étatAppelable.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Transactions

JDBC supporte le mode transactionnel qui consiste à valider tout ou une partie d'un ensemble d'instructions. Nous avons déjà décrit à la section « Interface Connection » les méthodes qui permettent à un programme Java de coder des transactions (`setAutoCommit`, `commit` et `rollback`).

Par défaut, chaque instruction SQL est validée (on parle *d'autocommit*). Lorsque ce mode est désactivé, il faut gérer manuellement les transactions avec `commit` ou `rollback`.

Quand le mode *autocommit* est désactivé :

- La déconnexion d'un objet `Connection` (par la méthode `close`) valide implicitement la transaction (même si `commit` n'a pas été invoqué avant la déconnexion).
- Chaque instruction du LDD (`CREATE`, `ALTER`, `DROP`) valide implicitement la transaction.

Points de validation

Depuis la version 3.0 de JDBC (JDK 1.4), on peut inclure des points de validation et affiner ainsi la programmation des transactions. Les interfaces `Connection` et `Savepoint` rendent possible cette programmation.

Interface Connection

Le tableau suivant présente les méthodes de l'interface `Connection` qui sont relatives au principe des points de validation :

Tableau 9-53 Méthodes concernant les points de validation de l'interface Connection

Méthode	Description
`Savepoint setSavepoint()`	Positionne un point de validation anonyme et retourne un objet `Savepoint`.
`Savepoint setSavepoint(String)`	Positionne un point de validation nommé et retourne un objet `Savepoint`.
`void releaseSavepoint(Savepoint)`	Supprime le point de validation de la transaction courante.
`void rollback(Savepoint)`	Invalide la transaction à partir du point de validation.

Oracle ne supporte pas encore la méthode `releaseSavepoint`.

Interface Savepoint

Les points de validation sont anonymes (identifiés toutefois par un entier) ou nommés. Le tableau suivant présente les deux seules méthodes de l'interface `Savepoint` :

Tableau 9-54 Méthodes de l'interface Savepoint

Méthode	Description
`int getSavepointId()`	Retourne l'identifiant du point de validation de l'objet `Savepoint`.
`String getSavepointName()`	Retourne le nom du point de validation de l'objet `Savepoint`.

Le code suivant (`Transaction2.java`) illustre une transaction découpée en deux phases par deux points de validation. Dans notre exemple, nous validons seulement la première partie. On suppose la connexion `cx` créée.

Tableau 9-55 Points de validation

Code Java	Commentaires

`try { …`
`cx.setAutoCommit(false);`
`String ordreSQL =`
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
	Désactivation de l'*autocommit*.
	Création d'un état appelable.

`Savepoint p1 = cx.setSavepoint("P1");`	Création du point de validation P1.

`étatPréparé.setString(1, "F-NEW2");`
`…`
`if (! étatPréparé.execute() )`
`System.out.println("F-NEW2 inséré");`
	Passage de paramètres et première insertion.

`Savepoint p2 = cx.setSavepoint("P2");`	Création du point de validation P2.

`étatPréparé.setString(1, "F-NEW3");`
`…`
`if (! étatPréparé.execute() )`
`System.out.println("F-NEW3 inséré");`
	Passage de paramètres et deuxième insertion.

`cx.rollback(p2);`	Annulation de la deuxième partie.
`cx.commit();`	Validation de la première partie.
`cx.close();`	Fermeture de la connexion.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Traitement des exceptions

Les exceptions qui ne sont pas traitées dans les sous-programmes appelés, ou celles que les sous-programmes ou déclencheurs peuvent retourner doivent être prises en compte au niveau du code Java (dans un bloc `try... catch...`). Le bloc d'exceptions permet de programmer des traitements en fonction des codes d'erreur renvoyés par la base Oracle. Plusieurs blocs d'exceptions peuvent être imbriqués dans un programme JDBC.

Afin de gérer les erreurs renvoyées par le SGBD, JDBC propose la classe `SQLException` qui hérite de la classe `Exception`. Chaque objet (automatiquement créé dès la première erreur) de cette classe dispose des méthodes suivantes :

Tableau 9-56 Méthodes de la classe SQLException

Méthode	Description
`String getMessage()`	Message décrivant l'erreur.
`String getSQLState()`	Code erreur SQL Standard (XOPEN ou SQL99).
`int getErrorCode()`	Code erreur SQL de la base.
`SQLException getNextException()`	Chaînage à l'exception suivante (si une erreur renvoie plusieurs messages).

Affichage des erreurs

Le code suivant illustre une manière d'afficher explicitement toutes les erreurs sans effectuer d'autres instructions :

Tableau 9-57 Affichage des erreurs

Code Java	Commentaires

`import java.sql.*;`
`import oracle.jdbc.driver.*;`
`class Exceptions1`
`{public static void main(String args [])`
`throws SQLException`
	Classe principale.

`{try{`
`DriverManager.registerDriver(…);`
`Connection cx = DriverManager.getConnection(…);`
`…}`
	Instructions.

`catch(SQLException ex)`
`{System.err.println("Erreur");`
`while ((ex != null))`
`{System.err.println("Statut : "+ ex.getSQLState());`
`System.err.println("Message : "+ ex.getMessage());`
`System.err.println("Code base : "+ ex.getErrorCode());`
`ex = ex.getNextException();}`
`} } }`
	Gestion des erreurs.

Traitement des erreurs

Il est possible d'associer des traitements à chaque erreur répertoriée avant l'exécution du programme. On peut appeler des méthodes de la classe principale ou coder directement dans le bloc des exceptions.

Le code suivant (`Exceptions2.java`) insère un enregistrement dans la table `Avion` en gérant un certain nombre d'exceptions possibles. Le premier bloc des exceptions permet d'afficher un message personnalisé pour chaque type d'erreur préalablement répertorié (duplication de clé primaire, mauvais nombre ou type de colonnes…). Si l'avion à insérer n'est pas rattaché à une compagnie existante (contrainte référentielle), on décide de créer la compagnie et l'avion à nouveau à l'aide de l'exception 2291 (`touche parent introuvable`). Le dernier bloc d'exceptions affiche l'éventuelle erreur qui pourrait se produire lors de ces deux insertions.

Tableau 9-58 Traitement des exceptions

Code SQLJ	Commentaires

`String ordreSQL =`
`"INSERT INTO Avion VALUES ('F-A0','A319', 148, 'NEW')";`
`try`
	Importation des paquetages.

`{ DriverManager.registerDriver`
`(new oracle.jdbc.driver.OracleDriver());`
`Connection cx = DriverManager.getConnection(……);`
`cx.setAutoCommit(false);`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
`System.out.println(étatPréparé.executeUpdate() +`
`" avion inséré.");`
`cx.commit();`
`cx.close(); }`
	Validation.

Tableau 9-58 Traitement des exceptions *(suite)*

Code SQLJ	Commentaires

`catch(SQLException ex)`
`{ if (ex.getErrorCode() == 1)`
`  System.out.println("Avion déjà existant!");`
`else if (ex.getErrorCode() == 913)`
`  System.out.println("Trop de valeurs!");`
`else if (ex.getErrorCode() == 942)`
`  System.out.println("Nom de table inconnue!");`
`else if (ex.getErrorCode() == 947)`
`  System.out.println("Manque de valeurs!");`
`else if (ex.getErrorCode() == 1401)`
`  System.out.println("Valeur trop longue!");`
`else if (ex.getErrorCode() == 1438)`
`  System.out.println("Valeur trop importante!");`
	Gestion des erreurs.

`else if (ex.getErrorCode() == 2291)`	Clé étrangère absente.

`try {Connection cx =`
`  DriverManager.getConnection(……);`
`  cx.setAutoCommit(false);`	

`  String ordreSQL2 = "INSERT INTO Compagnie` `  VALUES ('NEW','Nouvelle Compagnie')";`	Insertion d'une compagnie.

`  PreparedStatement étatPréparé2 =`
`  cx.prepareStatement(ordreSQL2);`
`  System.out.println(étatPréparé2.executeUpdate() +`
`  " compagnie insérée.");`	

`  étatPréparé2 = cx.prepareStatement(ordreSQL);`	Insertion d'un avion.
`  System.out.println(étatPréparé2.executeUpdate() +` `  " avion inséré.");`	
`  cx.commit();`	Validation.

`  cx.close(); }`
`catch (SQLException e)`
`  {System.err.println("Erreur : " + e); }`
	Gestion des erreurs.

`}`	

À l'aide de la méthode `getErrorCode` (en testant sur le numéro de l'erreur Oracle ou applicative), il est possible de récupérer des exceptions retournées par un sous-programme ou par un déclencheur.

Exercices

L'objectif de ces exercices est de développer des méthodes de la classe Java ExoJDBC pour extraire et mettre à jour certaines de vos tables.

Exercice 9.1 Curseur statique

Écrire les méthodes :

- `ArrayList getSalles()` qui retourne sous la forme d'une liste les enregistrements de la table `Salle`.
- `main` qui se connecte à la base, appelle la méthode `getSalles` et affiche les résultats (exemple donné ci-dessous) :

```
nSalle  nomSalle        nbPoste indIP
-----------------------------------------
s01     Salle 1         3       130.120.80
s02     Salle 2         2       130.120.80
...
```

Ajoutez une nouvelle salle dans la table `Salle` sous SQL*Plus, validez et lancez à nouveau le programme pour vérifier.

Exercice 9.2 Curseur modifiable

Écrire la méthode `void deleteSalle(int)` qui supprime de la table `Salle` l'enregistrement de rang passé en paramètre. Vous utiliserez la méthode `deleteRow` appliquée à un curseur modifiable.

Appeler cette méthode pour supprimer l'enregistrement de la table `Salle` que vous avez ajouté précédemment.

Exercice 9.3 Appel d'un sous-programme

Compiler dans votre schéma la fonction PL/SQL `supprimeSalle(VARCHAR2)` qui se trouve sur le Web et qui supprime une salle dont le numéro est passé en paramètre. La fonction retourne :

- 0 si la suppression s'est déroulée correctement ;
- -1 si le code de la salle est inconnu ;
- -2 si la suppression est impossible (contraintes référentielles).

Écrire la méthode `int deleteSallePL(String)` qui appelle la fonction `supprimeSalle`. Ajouter une nouvelle salle dans la table `Salle` sous SQL*Plus, valider. Appeler la méthode `deleteSallePL` dans le `main` pour supprimer la dernière salle créée. Essayer les différents cas d'erreurs en appelant cette méthode avec un numéro de salle référencé par un poste de travail et un numéro de salle inexistant.

Chapitre 10

Oracle et PHP

Ce chapitre détaille les moyens de faire interagir un programme PHP avec une base Oracle en présentant les fonctionnalités principales des API OC8 et PDO. Vous trouverez dans les compléments (disponibles à l'adresse *www.editions-eyrolles.com*, sur la fiche de l'ouvrage) d'autres mécanismes un peu plus datés ; il s'agit de PL/SQL Web Toolkit et de PL/SQL Server Pages.

Configuration adoptée

De nombreuses configurations sont possibles en fonction des versions PHP, Apache et Oracle que vous utiliserez. Si votre machine héberge plusieurs instances, la mise en place PHP et Apache nécessite un environnement béton (fichiers de configuration, variables d'environnement et chemins vers les répertoires). Évitez autant que possible d'utiliser Windows en version 64 bits, car il n'existe pas, pour l'heure, de version PHP adéquate et vous ajouterez une difficulté à la mise en œuvre de votre maquette (vous devrez installer un *instant client* Oracle et faire, par la suite, de nombreuses manipulations).

La configuration adoptée ici est Apache 2.2/PHP 5.2 avec une base 10*g* R2. À l'époque d'Oracle9*i*, Apache était inclus et il fallait simplement modifier les fichiers de configuration. Je décris ici une procédure minimale sans plus d'explications, car vous trouverez sur le Web de nombreuses ressources à ce sujet (*http://www.oracle.com/technetwork/topics/php/*).

Les logiciels

Téléchargez la dernière version stable d'Apache disponible sur *http://httpd.apache.org/download.cgi*. Pour Windows, optez pour le fichier `apache_x.x.xx-win32-x86-no_ssl.msi` ; pour Linux, choisissez `httpd-2.2.xx.tar.bz2`. Après l'installation, testez le service dans un navigateur (`http://localhost` dans la plupart des cas, `http://camparols` dans mon cas).

Téléchargez la dernière version *thread safe* de PHP au format archive : pour Windows, extension `.zip` (*http://windows.php.net/download*) ; pour Linux, au format `tar.bz2` ou `tar.gz` (*http://www.php.net/downloads.php*). Si vous utilisez Apache, optez pour la version VC6 (les versions VC9 étant dédiées à IIS). Décompressez l'archive dans un de vos répertoires (`C:\PHP` dans mon cas).

Les fichiers de configuration

Concernant Apache, éditez le fichier `httpd.conf` (situé par défaut sous Windows dans `C:\Program Files\Apache Software Foundation\Apache2.2`), puis ajoutez les lignes suivantes (# désigne un commentaire). Notez que les chemins de répertoires Windows doivent être écrits avec le symbole / et non le \.

```
# écoute sur le port 9999
Listen 9999
…
LoadModule php5_module "C:/php/php5apache2_2.dll"
AddHandler application/x-httpd-php .php
PHPIniDir "C:/php"
…
# répertoires des sources php (pas d'accent dans le nom du répertoire)
DocumentRoot "C:/Donnees/dev/PHP-Oracle"
…
# This should be changed to whatever you set DocumentRoot to…
<Directory "C:/Donnees/dev/PHP-Oracle">
..
# dans la section <IfModule mime_module> :
 AddType application/x-httpd-php .php
```

Concernant PHP, renommez le fichier `php.ini-development` en `php.ini`, puis éditez-le pour :

- modifier la ligne `extension_dir` en indiquant le répertoire contenant les extensions de PHP (et, notamment, la librairie `php_oci8.dll`), dans mon cas : `extension_dir="C:/php/ext"` ;
- décommenter la ligne `extension=php_oci8.dll`.

Les librairies Windows `php_oci8.dll` et `php_pdo_oci.dll` conviennent aux bases 10*g*, tandis que l'utilisation de bases 11*g* nécessite `php_oci8_11g.dll`.

Test d'Apache et de PHP

Écrire le programme suivant (`index.php`) et disposez le dans le répertoire contenant les sources PHP (`D:/dev/PHP-Oracle` dans mon cas).

```
html> <head> <title>test Apache et PHP</title> </head>
<body> Test de la configuration Apache - PHP
<?php
phpinfo();
?>
</body> </html>
```

Pour tester votre serveur, redémarrez le service Apache et inscrivez dans le navigateur l'adresse du serveur (`http://localhost:9999/index.php`, dans mon cas). En fonction de la configuration choisie, vous devrez voir le message « Test de la configuration Apache – PHP », suivi de la configuration actuelle de PHP (résultat de la fonction `phpinfo`).

Test d'Apache, de PHP et d'Oracle

Vérifiez que les services *Listener* et l'instance Oracle sont démarrés. Le programme suivant (`cx1.php`) doit se trouver dans le répertoire contenant les sources PHP (dans mon cas, `C:\Donnees\dev\PHP-Oracle`). Renseignez le nom d'utilisateur, le mot de passe et la description de l'instance (consultez le fichier `tnsnames.ora` situé dans `ORACLE_HOME\product\xx.x.x\server\network\ADMIN`; dans la dernière version d'Oracle *Express*, `ORACLE_HOME` référence `C:\oraclexe\app\oracle`).

Notez que vous pouvez aussi travailler sur votre base locale (installée par défaut) sans utiliser la description du service et en utilisant l'instruction : `$cx= oci_connect($utilisateur,$mdp)`.

```
<?php
$service = "(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP) (HOST =
localhost) (PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED)
(SERVICE_NAME = bdcs10g)))";
$utilisateur = "soutou";
$mdp = "iut";
print "Avant la connexion <BR>";
$cx = oci_connect($utilisateur ,$mdp, $service );
print "La connexion <B>passe </B>avec la description complète du
service! ";
oci_close($cx);
?>
```

Tester votre programme dans le navigateur (`http://camparols:9999/cx1.php` dans mon cas). Vous devez obtenir le résultat suivant.

Figure 10-1 *Test d'une connexion*

API de PHP pour Oracle (OCI)

Les extensions Oracle (fonctions préfixées par `ora`) sont désormais obsolètes (elles étaient valables avec des bases 7 et 8.0 et concernaient les premières versions de PHP). À partir de la version 8*i* sont apparues des fonctions PHP dont les noms étaient préfixés par `oci8`. Ces fonctions ont permis la gestion des LOB, descripteurs de fichiers, objets, collections et ROWID. Par ailleurs, la manipulation des métadonnées (vues du dictionnaire des données) était possible.

Depuis PHP 5, certaines de ces fonctions se sont standardisées ; le préfixe a été modifié en `oci_`. Ainsi, `OCILogin` et `OCIParse` sont devenus, respectivement, `oci_connect` et `oci_parse`, etc. Notez que, bien que les anciens noms demeurent en tant qu'alias, il est plus prudent de ne plus les utiliser. Vous trouverez dans le livre gratuit *The Underground PHP and Oracle Manual* (*http://www.oracle.com/technetwork/topics/php/underground-php-oracle-manual-098250.html*) toutes ces correspondances.

Les principales API pour accéder à Oracle via un programme PHP sont désormais OCI8 (*Oracle Call Interface*) et PDO (*PHP Data Objects*). Ces extensions sont écrites en C et sont inclues en tant que librairie de PHP (présente avec Windows dans `php_oci8.dll` et `php_pdo.dll`). Si vous désirez rendre votre code davantage portable (vers une autre base qu'Oracle), vous devez adopter PDO. Les extensions OCI8 vous permettront davantage de fonctionnalités (options de connexion, LOB, etc.) et continueront à faire partie du langage PHP.

Avant de présenter la technologie PDO, étudions les principales fonctions OCI qui s'intègrent à un programme PHP.

Connexions

La fonction `oci_connect` retourne un identifiant de connexion utilisé par la majorité des appels à la base (*oci calls*). Cette fonction ne rétablit pas une nouvelle connexion si une autre avait été ouverte auparavant avec les mêmes paramètres. Dans ce cas, `oci_connect` retourne l'identifiant de la précédente connexion ouverte. Il n'est donc pas possible d'utiliser cette fonction pour programmer des transactions séparées. Il faudra utiliser à cet effet la fonction `oci_new_connect` (ayant même signature que `oci_connect`).

Pour les bases Oracle d'une version supérieure à 9.2, il est possible d'utiliser un paramètre désignant le jeu de caractère à considérer lors de la connexion. `oci_connect` et `oci_new_connect` retournent `FALSE` si une erreur survient. La fonction `oci_close` retourne `TRUE` en cas de succès, `FALSE` en cas d'erreur. Ces genres de connexions se ferment implicitement en fin de programme PHP même si elle n'ont pas été cloturées avec `oci_close`.

Tableau 10-1 Fonction de connexion et déconnexion

Nom de la fonction	Paramètres
`oci_connect(string utilisateur, string password [, string bd [, string charset]] )`	Utilisateur, mot de passe, nom de la base locale ou description du service (en l'absence de ce paramètre PHP5 utilise la variable `ORACLE_SID`). Le dernier paramètre désigne éventuellement le jeu de caractères à considérer.
`oci_close($connexion)`	Ferme la connexion dont l'identifiant passe en paramètre.

Les connexions persistantes sont des liens qui ne se ferment implicitement pas à la fin du programme PHP. Quand une connexion persistante est invoquée, PHP vérifie son existence ou en créé une nouvelle (identique au niveau du serveur, de l'utilisateur et du mot de passe) en cas d'absence. Cela permet de passer en paramètre un identifiant de connexion entre plusieurs programmes PHP.

Ce ne sont pas ce type de connexions que l'on peut assimiler à des sessions. Les connexions persistantes n'offrent pas de fonctionalités additionnelles en terme de transaction que les connexions non persistantes. La fonction `oci_pconnect` retourne un identifiant persistant de connexion.

Tableau 10-2 Fonction oci_pconnect

Nom de la fonction	Paramètres
`oci_pconnect(string utilisateur, string password [, string bd [, string charset]] )`	Mêmes paramètres que `oci_connect`.

Constantes prédéfinies

Les constantes suivantes permettent de positionner des indicateurs jouant le rôle de paramètres systèmes (modes d'exécution) au sein d'instruction SQL. Nous verrons au long de nos exemples l'utilisation de certaines de ces constantes.

Tableau 10-3 Constantes prédéfinies

Constante	Commentaires
`OCI_DEFAULT`	Mode par défaut d'exécution des ordres SQL (pas de validation automatique).
`OCI_COMMIT_ON_SUCCESS`	Validation automatique (après appel à `oci_execute`).
`OCI_FETCHSTATEMENT_BY_COLUMN`	Mode par défaut de l'instruction `oci_fetch_all`.
`OCI_FETCHSTATEMENT_BY_ROW`	Mode alternatif de l'instruction `oci_fetch_all`.

Tableau 10-3 Constantes prédéfinies *(suite)*

Constante	Commentaires
`OCI_ASSOC`	Utilisé par `oci_fetch_all` et `oci_fetch_array` afin d'extraire un *associative array* comme résultat.
`OCI_NUM`	Utilisé par `oci_fetch_all` et `oci_fetch_array` afin d'extraire un *enumerated array* comme résultat.
`OCI_BOTH`	Utilisé par `oci_fetch_all` et `oci_fetch_array` afin d'extraire un *array* supportant à la fois le mode associatif et le mode numérique en indices.
`OCI_RETURN_NULLS`	Utilisé par `oci_fetch_array` afin d'extraire des lignes même si des colonnes sont valuées à `NULL`.

Interactions avec la base

La majorité des traitements SQL, lorsqu'ils incluent des paramètres, s'effectuent comme suit : connexion (*connect*), préparation de l'ordre (*parse*), association des paramètres à l'ordre SQL (*bind*), exécution dudit ordre (*execute*), lecture des lignes (pour les `SELECT`, *fetch*) et libération des ressources (*free* et *close*) après validation ou annulation de la transaction courante (*commit* et *rollback*).

La fontion `oci_parse` prépare l'ordre SQL puis retourne un identifiant d'état qui peut être utilisé notamment par les fonctions `oci_bind_by_name` et `oci_execute`. La fonction `oci_parse` retourne `FALSE` dans le cas d'une erreur mais ne valide ni sémantiquement ni syntaxiquement l'ordre SQL. Il faudra attendre pour cela son exécution par `oci_execute`.

La fonction `oci_execute` exécute un ordre SQL préparé. Le mode par défaut est `OCI_COMMIT_ON_SUCCESS` (*auto-commit*). Pour la programmation de transactions, préférez le mode `OCI_DEFAULT` puis validez explicitement par `oci_commit`. La fonction `oci_execute` retourne `TRUE` en cas de succès, `FALSE` sinon.

Tableau 10-4 Fonctions d'analyse et d'exécution

Nom de la fonction	Paramètres
`ressource oci_parse(ressource` *`connexion`*`, string` *`ordreSQL`*`)`	Le premier paramètre désigne l'identifiant de la connexion. Le second contient l'ordre SQL à analyser (`SELECT, INSERT, UPDATE, DELETE, CREATE...`)
`boolean oci_execute(ressource` *`ordreSQL`* `[,int` *`mode`*`])`	Le premier paramètre désigne l'ordre SQL à exécuter. Le deuxième paramètre est optionnel, il définit le mode de validation à l'aide d'une constante prédéfinie.

Mises à jour

Les fonctions `oci_commit` et `oci_rollback` permettent de gérer des transactions, elles retournent `TRUE` en cas de succès, sinon `FALSE`.

Tableau 10-5 Fonctions de validation et d'annulation

Nom de la fonction	Paramètres
`boolean` **`oci_commit`**`(ressource` *`connexion`*`)`	Valide la transaction de la connexion en cours.
`boolean` **`oci_rollback`**`(ressource` *`connexion`*`)`	Annule la transaction de la connexion en cours.

Le code suivant (partie du programme `insert1.php`) décrit l'insertion d'une nouvelle compagnie (en supposant qu'aucune erreur n'est retournée de la part de la base). Notez que les lignes d'exécution et de validation auraient pu être remplacées par l'instruction « `oci_execute($ordre, OCI_COMMIT_ON_SUCCESS)` ». Ce mode de programmation est également valable pour les modifications de colonnes (`UPDATE`) et suppression d'enregistrements (`DELETE`). Nous étudierons plus loin comment récupérer au niveau de PHP les erreurs renvoyées par Oracle.

Tableau 10-6 Insertion d'un enregistrement

Code PHP	Commentaires
`$insert1 = "INSERT INTO Compagnie VALUES ('AL', 'Air Lib')";`	Création de l'instruction.
`$ordre = oci_parse($cx, $insert1);`	Prépare l'insertion.
`oci_execute($ordre);`	Exécute l'insertion.
`oci_commit($cx);`	Validation.
`oci_free_statement($ordre);`	Libère les ressources.
`oci_close($cx);`	

Si vous souhaitez connaître le nombre de lignes affectées par l'ordre SQL, utilisez « **`oci_num_rows`**`($ordre)` » (voir la section « Métadonnées »).

Extractions simples

Les fonctions suivantes permettent d'extraire des données *via* un curseur que la documentation de PHP appelle *tableau*. Il est à noter qu'Oracle retourne les noms de colonnes toujours en majuscules. Cette remarque intéressera les habitués des tableaux à accès associatifs (exemple : `$tab['PRENOM']`, `PRENOM` étant une colonne extraite d'une table).

Tableau 10-7 Fonctions d'extraction

Nom de la fonction	Paramètres
`int` **`oci_fetch_all`**`(ressource` *`ordreSQL`*`, array &tableau [, int saut [, int maxrows [, int param]]] )`	Extrait les lignes dans un tableau. Retourne le nombre de lignes extraites ou `FALSE` en cas d'erreur. *`saut`* désigne le nombre de lignes à ignorer (par défaut 0). *`maxrows`* désigne le nombre de lignes à lire (par défaut -1 qui signifie toutes les lignes) en démarrant de l'indice *`saut`*. *`param`* peut être une combinaison de : • `OCI_FETCHSTATEMENT_BY_ROW` • `OCI_FETCHSTATEMENT_BY_COLUMN` (par défaut) • `OCI_NUM` • `OCI_ASSOC`
`array` **`oci_fetch_array`**`(ressource` *`ordreSQL`* `[, int param] )`	Retourne un tableau qui contient la ligne du curseur suivante ou `FALSE` en cas d'erreur ou en fin de curseur. Le tableau est accessible de manière associative ou numérique suivant le paramètre *`param`* qui peut être une combinaison de : • `OCI_BOTH` (par défaut, identique à `OCI_ASSOC` + `OCI_NUM`). • `OCI_ASSOC` pour un tableau à accès associatif (comme `oci_fetch_assoc`). • `OCI_NUM` pour un tableau à accès numérique (comme `oci_fetch_row`). • `OCI_RETURN_NULLS` prend en compte les valeurs `NULL` retournées par Oracle.
`array` **`oci_fetch_assoc`**`(ressource` *`ordreSQL`*`)`	Retourne la ligne du curseur suivante dans un tableau associatif ou `FALSE` en cas d'erreur ou en fin de curseur.
`object` **`oci_fetch_object`**`(ressource` *`ordreSQL`*`)`	Retourne la ligne du curseur suivante dans un objet PHP ou `FALSE` en cas d'erreur ou en fin de curseur.
`array` **`oci_fetch_row`**`(ressource` *`ordreSQL`*`)`	Retourne la ligne du curseur suivante dans un tableau numérique ou `FALSE` en cas d'erreur ou en fin de curseur.
`boolean` **`oci_set_prefetch`**`(ressource` *`ordreSQL`* `[, int nbLignes] )`	Limite le nombre de lignes à extraire à la suite d'un appel à `oci_execute`. Par défaut le deuxième paramètre vaut 1. Retourne `TRUE` en cas de succès, `FALSE` dans le cas inverse.
`boolean` **`oci_cancel`**`(ressource` *`ordreSQL`*`)`	Invalide le curseur libérant les ressources. Retourne `TRUE` en cas de succès, `FALSE` sinon.
`boolean` **`oci_free_statement`**`(res-source` *`ordreSQL`*`)`	Libère les ressources associées aux curseurs occupées après `oci_parse`. Retourne `TRUE` en cas de succès, `FALSE` dans le cas inverse.

Illustrons à partir d'exemples certaines utilisations de quelques-unes de ces fonctions.

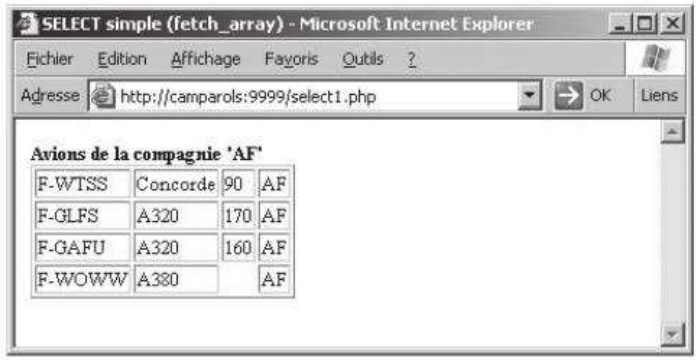
Le code suivant (partie du programme `select1.php`) décrit l'extraction des avions de la compagnie de code 'AF' avec la fonction `oci_fetch_array`. On suppose ici et dans les programmes suivants que la connexion à la base est réalisée et se nomme `$cx`. Le curseur obtenu est nommé `ligne`, il prend en compte les valeurs nulles éventuelles. La fonction `oci_num_fields` renvoie le nombre de colonnes de la requête et sa signature est détaillée à la section « Métadonnées ».

Tableau 10-8 Fonction oci_fetch_array

Code PHP	Commentaires
`$requete = "SELECT immat,typeavion,`	Création de la requête.
`capacite,compa FROM Avion`	
`WHERE compa = 'AF'";`	Exécution de la requête.
`$ordre = oci_parse($cx, $requete);`	Obtention du nombre de colonnes.
`oci_execute ($ordre);`	Exécution.
`$ncols = oci_num_fields($ordre);`	Obtention.
`print "<B>Avions de la compagnie 'AF'</B>";`	Chargement et parcours du curseur.
`print "<TABLE BORDER=1> ";`	Parcours des colonnes.
`while ($ligne = oci_fetch_array($ordre,`	Chargement.
`OCI_NUM + OCI_RETURN_NULLS))`	
`{print "<TR> ";`	
`for ( $i=0;$i < $ncols; $i++)`	Parcours.
`{print "<TD> $ligne[$i] </TD>" ;}`	Affichage.
`print "</TR> "; }`	
`print "</TABLE> ";`	

Le résultat est le suivant (en supposant que la compagnie 'AF' dispose de 4 avions dont un est affecté d'une capacité nulle).

Figure 10-2 Exemple avec oci_fetch_array



Le code suivant (partie du programme `select2.php`) décrit l'extraction de tous les avions à l'exception des deux premiers (grâce au troisième paramètre de la fonction `oci_fetch_all`). Le curseur obtenu est nommé `tabresults`. L'instruction PHP `reset` replace le pointeur au premier élément de la ligne courante du curseur. L'instruction PHP `each` retourne la paire (clé, valeur) et avance le curseur d'une ligne.

Tableau 10-9 Fonction oci_fetch_all

Code PHP	Commentaires
`$requete = "SELECT immat, typeavion, capacite FROM Avion";`	Création de la requête.

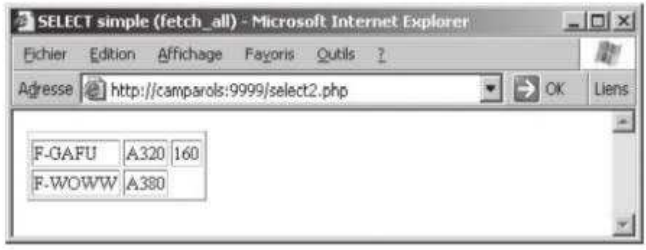
`$ordre = oci_parse ($cx, $requete);`
`oci_execute ($ordre);`
`$nblignes = oci_fetch_all($ordre, $tabresults, 2);`
`if ($nblignes > 0) {`
`print "<table border=1>\n";`
`print"<tr>\n";`
	Exécution de la requête.
	Obtention du nombre de lignes et chargement.

`for ($i = 0; $i < $nblignes; $i++) {`
`reset($tabresults);`
`print "<tr>\n";`
`while ($col = each($tabresults)) {`
`$donnée = $col['value'];`
`print "<td>$donnée[$i]</td>\n";}`
`print "</tr>\n"; }`
`print "</table>\n";`
`}`
	Parcours du curseur.
	Parcours des colonnes.
	Affichage des colonnes.

`else` `print "Pas de données  \n"; }`	

Le résultat est le suivant (en supposant toujours que la base ne stocke que 4 avions).

Figure 10-3 *Exemple avec oci_fetch_all*



Passage de paramètres

Les fonctions `oci_define_by_name` et `oci_bind_by_name` permettent d'associer à des colonnes Oracle (toujours notées en majuscules) des variables PHP, et inversement. Ces fonctions retournent `TRUE` en cas de succès, sinon `FALSE`.

Afin de vous prémunir d'éventuelles attaques par injection de code SQL, évitez de construire dynamiquement une requête et préférez l'utilisation de paramètres de liens (*bind variables*).

Tableau 10-10 Fonctions de passage de paramètres

Nom de la fonction	Paramètres
`boolean` **`oci_define_by_name`**`(ressource` *`ordreSQL`*`, string` *`nomColonne`*`, mixed` *`&variable`* `[, int` *`type`*`])`	Définition d'une variable PHP de réception pour la colonne. Le paramètre optionnel *`type`* concerne la gestion des LOBs par des descripteurs.
`boolean` **`oci_bind_by_name`**`(ressource` *`ordreSQL`*`, string` *`":colOracle"`*`, mixed` *`&variable`* `[, int` *`longueur`* `[, int` *`type`*`]] )`	Association d'une variable PHP à une colonne Oracle dans une instruction de manipulation de données de SQL (`INSERT`, `UPDATE` et `DELETE`). Le paramètre *`longueur`* ajuste la taille en octets de la valeur passée en paramètre. Si il vaut -1, `oci_bind_by_name` utilisera la taille courante de la variable. Même signification pour le paramètre optionnel *`type`* que précédemment.

Le code suivant (partie du programme `select3.php`) extrait l'immatriculation et le type de tous les avions en utilisant la fonction `oci_define_by_name` pour sélectionner les colonnes.

Les deux variables PHP sont définies avant d'exécuter l'ordre.

Tableau 10-11 Fonction oci_define_by_name

Code PHP	Commentaires
`$requete = "SELECT immat,typeavion,capacite FROM Avion";` `$ordre = oci_parse ($cx, $requete);`	Création de la requête.
`oci_define_by_name``($ordre, "IMMAT", $immatriculation);` **`oci_define_by_name`**`($ordre, "TYPEAVION", $typav);` `oci_execute ($ordre);`	Définition des variables PHP. Exécution de la requête.

`print "<B>Liste des avions</B>";`
`print "<TABLE BORDER=1> ";`
`while (oci_fetch_array($ordre))`
`{print "<TR> <TD> $immatriculation </TD>" ;`
`print " <TD> $typav </TD> </TR> "; }`
`print "</TABLE> ";`
	
	Chargement et parcours du curseur.
	Affichage des colonnes.

Le code suivant (partie du programme `insert2.php`) décrit l'insertion paramétrée d'une nouvelle compagnie. La fonction `oci_bind_by_name` permet de faire passer deux paramètres à l'instruction SQL.

Tableau 10-12 Fonction oci_bind_by_name

Code PHP	Commentaires

`$codeComp = "CAST";`
`$nomComp = "Castanet Air";`
`$insert2 = "INSERT INTO Compagnie VALUES (:v1, :v2) ";`
`$ordre = oci_parse ($cx, $insert2);`
	Affectation des variables PHP.
	Définition de l'ordre paramétré.

`oci_bind_by_name``($ordre, ":v1", $codeComp, -1);` **`oci_bind_by_name`**`($ordre, ":v2", $nomComp, -1);`	Association avec les variables PHP.
`oci_execute($ordre);` `oci_commit($cx);`	Exécution de l'ordre.

Traitements des erreurs

Les fonctions `oci_error` et `oci_internal_debug` permettent de gérer les erreurs retournées par Oracle. Le tableau associatif retourné par `oci_error` contient le code erreur Oracle (colonne `code`), le libellé du message (colonne `message`), le texte de l'instruction (colonne `sqltext`) et le déplacement (débutant à l'indice 0) dans le texte de l'instruction indiquant l'erreur (colonne `offset`).

Tableau 10-13 Fonctions pour la gestion des erreurs Oracle

Nom de la fonction	Paramètres
`array` **`oci_error`**`([ressource` *`source`*`])`	Dans la plupart des cas le paramètre *`source`* désigne l'ordre SQL. Pour les erreurs de connexion (`oci_connect`, `oci_new_connect` ou `oci_pconnect`), il ne faut pas indiquer de paramètre.
`void` **`oci_internal_debug`**`(int` *`valeur`*`)`	Active ou désactive le deboggage interne par le paramètre *`valeur`* (0 pour le désactiver, 1 pour l'activer). Par défaut le déboggage est désactivé.

Il faudra utiliser le préfixe `@` devant la fonction pour laquelle vous souhaiter lever une éventuelle exception. Ce préfixe entraîne l'annulation du rapport d'erreur de cette expression tout en conservant les messages d'erreur dues aux erreurs d'analyse.

Le code suivant (programme `erreur1.php`) décrit l'affichage d'une erreur de connexion en utilisant la fonction `oci_error` sans paramètre. Dans cet exemple, la connexion ne se déroule pas correctement du fait d'un nom erroné du serveur.

Tableau 10-14 Fonction oci_error (sans paramètre)

Code PHP	Commentaires
`<html> <head> <title>Erreur connexion </title> </ head>`	
`<body>`	
`<?php`	Début du code PHP.
`$service = "(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP) (HOST = toto) (PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = bdcs10g)))";`	
`$utilisateur = "soutou";`	
`$mdp = "iut";`	
`$cx = @oci_connect($utilisateur ,$mdp, $service);`	Connexion.
`if (!$cx)`	Connexion.
`{print "L'utilisateur <B>$utilisateur</B> n'a pu se connecter à la base  ";`	
`$taberr = oci_error();`	Récupération de l'erreur.
`print "<B>Message : </B>" . $taberr['message'];`	Affichage
`print " <B>Code : </B>" . $taberr['code'];`	de l'erreur.
`}`	
`else`	
`{`	
`// début de la transaction …`	Codage de la transaction.
`oci_close($cx);`	Fermeture de la connexion.
`}`	
`?>`	
`</body> </html>`	

Le résultat est le suivant.

Figure 10-4 *Problème de connexion décelé à l'aide de oci_error*

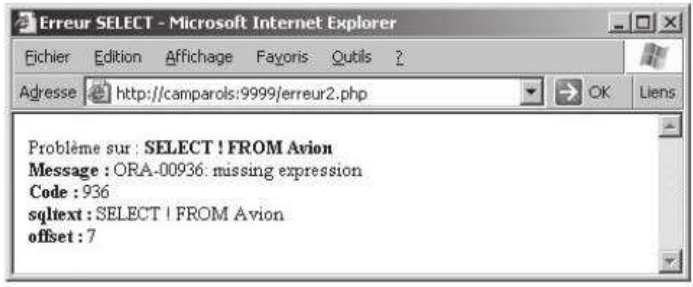
Le code suivant (programme `erreur2.php`) décrit l'affichage détaillé d'une erreur au sein d'une instruction SQL en utilisant la fonction `oci_error` avec un paramètre. La position de l'erreur est donnée pas la valeur du déplacement (`offset`) dans l'instruction (ici l'erreur est située en 8e position).

Tableau 10-15 Fonction oci_error (avec paramètre)

Code PHP	Commentaires
`$requetePB = "SELECT ! FROM Avion";` `$ordre = oci_parse($cx,$requetePB);` `if (!@oci_execute($ordre))` `{` `  print "Problème sur : <B> ". $requetePB . "</B> ";` `  $taberr = oci_error($ordre);` `  print "<B>Message : </B>" . $taberr['message'];` `  print " <B>Code  : </B>" . $taberr['code'];` `  print " <B>sqltext  : </B>" . $taberr['sqltext'];` `  print " <B>offset  : </B>" . $taberr['offset'];}`	Analyse et exécution de l'ordre (erroné). Récupération de l'erreur. Affichage détaillé du tableau associatif contenant le résultat de l'erreur.

Le résultat est le suivant.

Figure 10-5 Erreur de syntaxe SQL décelé à l'aide oci_error



En activant le déboggage interne (appel à la fonction `oci_internal_debug`), on obtient la séquence suivante.

```
// active le deboggage
   oci_internal_debug(1);
   $cx = @oci_connect($utilisateur ,$mdp, $service);
   …
```

Le résultat détaille les différents appels aux fonctions *OCI* d'Oracle.

Figure 10-6 *Déboggage interne à l'aide de oci_internal_debug*

Procédures cataloguées

Comme dans tout autre langage hôte, PHP permet d'invoquer des procédures cataloguées situées côté serveur. Supposons que nous disposions de la procédure suivante qui augmente la capacité (premier paramètre) des avions d'une compagnie donnée (deuxième paramètre).

```
CREATE PROCEDURE augmenteCap(nbre IN NUMBER, compag IN CHAR) AS
BEGIN
   UPDATE Avion SET capacite = capacite + nbre WHERE compa = compag;
   COMMIT;
 END;
/
```

Le code suivant (programme `procedureCat.php`) décrit l'appel de la procédure qui augmente la capacité des avions de la compagnie de code 'AF' d'une valeur de 50 places. Notez l'utilisation de deux espaces lors de l'initialisation de la variable PHP `$comp` car la colonne `compa` de la table `Avion` est dimensionnée en `CHAR(4)`. L'utilisation de « -1 » lors des *bind* indique que c'est la longueur des variables PHP qui sera considérée dans la procédure cataloguée.

Tableau 10-16 Appel d'une procédure cataloguée

Code PHP	Commentaires
`$procedure = "BEGIN augmenteCap(:nbre,:compag); END;";` `$ordre = oci_parse($cx, $procedure);`	Déclaration et analyse de la procédure.
`$nb   = 50;` `$comp = 'AF  ';`	Initialisation des variables de liens PHP.
`oci_bind_by_name ($ordre, ":nbre"  , $nb,   -1);` `oci_bind_by_name ($ordre, ":compag", $comp, -1);`	Liaison des variables PHP à l'instruction Oracle.

`oci_execute($ordre);`
`print "Procédure réalisée correctement.";`
`oci_free_statement ($ordre);`
`oci_close($cx);`
	Appel de la procédure.

Métadonnées

Les fonctions suivantes permettent d'extraire des informations en provenance du dictionnaire des données.

Tableau 10-17 Fonctions pour gérer les métadonnées

Nom de la fonction	Paramètres
`string` **`oci_server_version`**`(ressource` *`connexion`*`)`	Retourne une chaîne décrivant la version du noyau Oracle utilisé par la connexion passée en paramètre. La fonction retourne `FALSE` en cas d'erreur.
`boolean` **`oci_field_is_null`**`(ressource` *`ordreSQL`*`, mixed` *`colonne`*`)`	Retourne `TRUE` si la colonne désignée (paramètre *`colonne`* noté en majuscules) est `NULL`. La fonction retourne `FALSE` sinon.
`int` **`oci_num_fields`**`(ressource` *`ordreSQL`*`)`	Retourne le nombre de colonnes du resultat de l'ordre SQL.
`string` **`oci_field_name`**`(ressource` *`ordreSQL`*`, int` *`pos`*`)`	Retourne le nom de la colonne de l'ordre SQL correspondant à la position *`pos`* (débutant à 1 pour la première colonne).
`int` **`oci_field_precision`**`(ressource` *`ordreSQL`*`, int` *`pos`*`)`	Retourne la précision de la colonne de l'ordre SQL correspondant à la position *pos*. Pour les `FLOAT`, la precision vaut -127. Si la précision est égale à 0, il s'agit d'un `NUMBER`. Sinon il s'agit de la précision d'une colonne `NUMBER` (*`precision`*, *`scale`*).
`int` **`oci_field_scale`**`(ressource` *`ordreSQL`*`, int` *`pos`*`)`	Retourne l'echelle (nombre de décimales) de la colonne de l'ordre SQL correspondant à la position *`pos`*. Même règle que pour `oci_field_precision`. Si aucune échelle n'existe, la valeur `FALSE` est retournée.
`int` **`oci_field_size`**`(ressource` *`ordreSQL`*`, mixed` *`field`*`)`	Retourne la taille de la colonne de l'ordre SQL correspondant à la position *`field`* ou au nom « *field* ».

Tableau 10-17 Fonctions pour gérer les métadonnées *(suite)*

Nom de la fonction	Paramètres
`string` **`oci_statement_type`**`(ressource` *ordreSQL*`)`	Retourne le type de l'ordre SQL provenant de `oci_parse` (1 pour `SELECT`, 2 pour `UPDATE`, 3 pour `DELETE`, 4 pour `INSERT`, 5 pour `CREATE`, 6 pour `DROP`, 7 pour `ALTER`, 8 pour `BEGIN`, 9 pour `DECLARE`, 10 pour `UNKNOWN`).
`mixed` **`oci_field_type`**`(ressource` *ordreSQL*`, int` *pos*`)`	Retourne le type de la colonne de l'ordre SQL correspondant à la position *pos*.
`int` **`oci_num_rows`**`(ressource` *ordreSQL*`)`	Retourne le nombre de lignes affectées par un ordre SQL (LMD ou LCD). Cette fonction ne ramène pas le nombre de lignes extraites par un `SELECT`. Pour cela utilisez `COUNT`.
`boolean` **`oci_password_change`** `(ressource` *connexion*`, string` *utilisateur*`, string` *ancienMDP*`, string` *nouveauMDP*`)`	Change le mot de passe de l'utilisateur passé en paramètre. Retourne `TRUE` si le changment est effectif, `FALSE` sinon.

Illustrons à partir d'exemples certaines de ces fonctions.

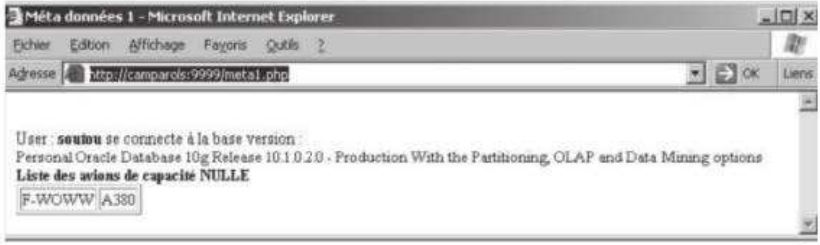
Le code suivant (programme `meta1.php`) décrit l'extraction des avions de capacité nulle en utilisant la fonction `oci_field_is_null`.

Le résultat est le suivant.

Tableau 10-18 Affichage de la version et test de nullité

Code PHP	Commentaires
`$cx = oci_connect($utilisateur ,$mdp, $service);` `print " User : <B>$utilisateur</B> se connecte à la base version :  ";` `print` **`oci_server_version`**`($cx);`	Affichage de la version de la base utilisée.
`$requete = "SELECT * FROM Avion";` `$ordre = oci_parse ($cx, $requete);` `oci_define_by_name($ordre, "IMMAT", $immatriculation);` `oci_define_by_name($ordre, "TYPEAVION", $typav);` `oci_define_by_name($ordre, "CAPACITE", $cap);` `oci_execute ($ordre);` `print " <B>Liste des avions de capacité NULLE</B>";` `print "<TABLE BORDER=1> ";` `while (oci_fetch_array($ordre))` `{if (`**`oci_field_is_null`**`($ordre,"CAPACITE"))` `{print "<TR> <TD> $immatriculation </TD>";` `print " <TD> $typav </TD> </TR>" ;}` `}` `print "</TABLE> ";`	Test de la nullité de la colonne `capacite`. Affichage des données extraites.

Figure 10-7 Affichage de la version de la base et test de nullité d'une colonne



Le code suivant (programme `meta2.php`) décrit l'extraction de la structure complète (en termes de colonnes) d'une table en utilisant les fonctions `oci_num_fields`, `oci_field_name`, `oci_field_type` et `oci_field_size`.

Tableau 10-19 Extraction de la structure d'une table

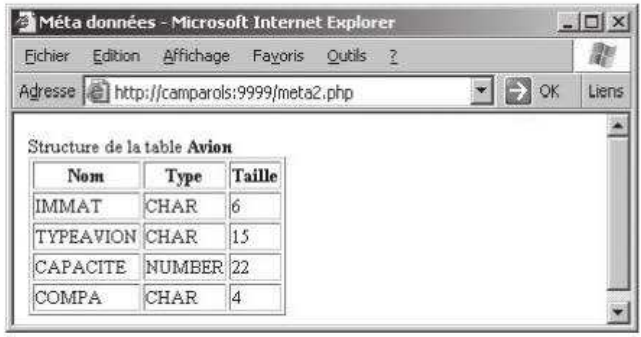
Code PHP	Commentaires
`$ordre = oci_parse($cx, "SELECT * FROM Avion");` `oci_execute($ordre);`	Extraction des avions de la base.

`print "Structure de la table <B>Avion</B>";`
`print "<table border=1>";`
`print "<tr><th>Nom</th>";`
`print "<th>Type</th>";`
`print "<th>Taille</th></tr>";`
`$ncols = oci_num_fields($ordre);`
	Extraction du nombre de colonnes.

`for ($i = 1; $i <= $ncols; $i++)`
`($col_nom = oci_field_name($ordre, $i);`
`$col_type = oci_field_type($ordre, $i);`
`$col_size = oci_field_size($ordre, $i);`
`print "<tr><td>$col_nom</td>";`
`print "<td>$col_type</td>";`
`print "<td>$col_size</td></tr>";`
`}`
`print "</table>\n";`
	Nom, type et taille de la colonne extraite.
	Affichage des informations extraites.

Le résultat est le suivant.

Figure 10-8 *Extraction de la structure d'une table*



API Objet PHP pour Oracle (PDO)

PDO (*PHP Data Objects*) est une API objet qui permet de traduire et de transmettre les instructions SQL au SGBD. Indépendante de l'éditeur, la couche PDO est dite « abstraite », car elle permet de séparer les traitements de la base de données et s'adapte tout aussi bien à Oracle qu'à MySQL, PostgreSQL, etc. Comme pour la technologie JDBC, chaque éditeur de SGBD dispose d'un ou plusieurs pilotes PDO qu'il faudra inclure dans les librairies de PHP. Il existe aussi des pilotes en provenance de la communauté PHP.

Le fait d'utiliser PDO ne rend pas vos requêtes compatibles avec n'importe quelle base de données, mais assure que les fonctions d'accès seront universelles (mises à jour, parcours d'un résultat, etc.), sous réserve que le code SQL soit le plus standard possible. Ainsi, le jour où vous décidez de migrer vers une autre base de données, il suffira en principe de modifier le fichier de configuration de la connexion sans avoir à réécrire totalement votre code. La documentation officielle se trouve sur le site *http://www.php.net/manual/fr/book.pdo.php*.

Au préalable, vous devrez décommenter les lignes `extension=php_pdo_oci8.dll` et `extension=php_pdo.dll` du fichier `php.ini`, puis relancer le service d'Apache. Pour des bases 11*g*, vous devrez agir sur la ligne concernant `php_oci8_11g.dll`.

Connexions

Une connexion s'établit à la création d'une instance de la classe `PDO`. Le constructeur accepte plusieurs paramètres. Le premier est appelé « DSN » (*Data Source Name*), les autres correspon-

dent à l'utilisateur, au mot de passe et à d'éventuelles options de connexion. Comme avec Java, il est possible de router toute erreur par l'intermédiaire d'un objet `PDOException`.

Le code suivant (programme `pdo1.php`) décrit la connexion à une base. La déconnexion s'opère par la suppression de l'objet (affectation à `null` de la référence).

Tableau 10-20 Connexion et déconnexion

Code PHP	Commentaires

`$service = "(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)`
`(HOST = camparols)(PORT = 1521))`
`(CONNECT_DATA = (SERVER = DEDICATED)`
`(SERVICE_NAME = bd10gR2)))";`
`$dsn = 'oci:dbname='.$service;`
`$utilisateur = "soutou";`
`$mdp = "soutou";`
`try`	

`{$cx = new PDO($dsn,$utilisateur,$mdp);`	Connexion.
`// traitement…`	
`$cx = null;`	Déconnexion.
`}`	
`catch (PDOException $e)`	Gestion des erreurs.
`{ print "Problème connexion : ".$e->getMessage(); }`	

Afin de créer une connexion persistante (c'est-à-dire qui n'est pas fermée à la fin du script, mais mise en cache et réutilisable), il suffit d'ajouter le paramètre `array(PDO::ATTR_PERSISTENT => true)` en quatrième position du constructeur de la connexion.

Mises à jour

Le code suivant (programme `pdo-insert1.php`) décrit une transaction insérant une ligne dans la table Avion.

- La méthode `setAttribute` permet de gérer les exceptions dues à une erreur côté serveur lors de la transaction.
- L'instruction paramétrée utilise les *place holders* (symbole `?`) qu'il faudra associer à une variable (ou valeur) par l'indice à l'aide de la méthode `bindValue`. Les constantes prédéfinies du type `PDO::PARAM_xxx` renseignent le type de la colonne. Les méthodes `prepare` et `execute` sont classiques dans ce genre de programmation.
- La validation et l'invalidation d'une transaction s'opère traditionnellement à l'aide des méthodes `commit` et `rollBack`.

Si vous souhaitez connaître le nombre de lignes affectées par l'instruction SQL, utilisez la méthode `rowCount` au niveau de l'état `prep->rowCount()`.

Tableau 10-21 Instruction paramétrée avec bindValue

Code PHP	Commentaires

`try`
`{`
`$cx = new PDO($dsn,$utilisateur,$mdp);`
`$cx->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);`
	Création de la connexion.

`$cx->beginTransaction();`
`$insert1 = "INSERT INTO Avion VALUES (?,?,?,?) ";`
`$prep = $cx->prepare($insert1);`
	Début de la transaction. Préparation de l'instruction.

`$prep->bindValue(1,'F-HAAB', PDO::PARAM_STR);`
`$prep->bindValue(2,'AT01', PDO::PARAM_STR);`
`$prep->bindValue(3,10, PDO::PARAM_INT);`
`$prep->bindValue(4,'SING', PDO::PARAM_STR);`
	Passage des paramètres.

`$prep->execute();`
`$cx->commit();`
`$cx = null;`
	Exécution et validation.

`catch (PDOException $e)`
`{`
`$cx->rollBack();`
`print "Problème : ".$e->getMessage();`
`}`
	Gestion des erreurs et invalidation.

Le code suivant (programme `pdo-update1.php`) décrit une transaction modifiant tous les avions (augmentation de la capacité, premier paramètre) d'une compagnie de code (dont la valeur passe en deuxième paramètre). La phase de connexion n'est pas décrite pour alléger le code.

La méthode `bindParam` affecte à un *place holder* (ici, `:v1` et `:v2`) une variable. Avant d'exécuter l'instruction, il convient d'affecter une valeur à chaque variable définissant un paramètre (ici, `$v1` et `$v2`).

Tableau 10-22 Instruction paramétrée avec bindParam

Code PHP	Commentaires

`$cx->beginTransaction();`
`$update1 = "UPDATE Avion SET capacite=capacite+ :v1`
`WHERE compa= :v2";`
`$prep = $cx->prepare($update1);`
	Début de la transaction et préparation de l'instruction.

`$prep->bindParam(':v1',$v1);` `$prep->bindParam(':v2',$v2);`	Mise en place des paramètres.

`$v1=60;`
`$v2='AF ';`
`$prep->execute();`
	Affectation des paramètres et exécution.

`$cx->commit();`
`print "Mises à jour OK.".$prep->rowCount()." ligne(s) modifiée(s)";`
`$cx = null;`
	Validation et fermeture de la connexion.

Extractions

Le code suivant (programme `pdo-select1.php`) extrait les avions d'une compagnie de code passant en paramètre. La phase de connexion n'est pas décrite pour alléger le code.

La phase de préparation, passage des paramètres, est similaire à l'exemple précédent. La méthode `fetch` d'un objet de la classe `PDOStatement` permet d'extraire une ligne ramenée par la requête à chaque itération de la boucle. Chaque ligne est un tableau associatif qu'il suffit de parcourir en utilisant le nom des colonnes.

Tableau 10-23 Requête paramétrée

Code PHP	Commentaires

`$select1 = "SELECT immat,typeavion,capacite FROM Avion WHERE compa= :v1";`
`$prep = $cx->prepare($select1);`
`$prep->bindParam(':v1',$v1);`
	Préparation de l'instruction.

`$v1='AF ';` `if ($prep->execute())`	Mise en place du paramètre et exécution.

`{print "<table border=1>";`
`print "<tr><th>IMMAT</th><th>TYPEAVION</th></thr>";`
`while ($row = $prep->fetch())`
`{ print "<tr><td>".$row['IMMAT']."</td><td>".`
`$row['TYPEAVION']."</td></tr>"; }`
`print "</table>";`
`}`
`else print "Aucune ligne...";`
	Parcours du résultat.

`$cx = null;`	Fermeture de la connexion.

Le résultat obtenu est le suivant.

Figure 10-9 *Extraction avec PDO*

Procédures cataloguées

PDO permet l'appel de procédures cataloguées. Considérons la procédure suivante qui dispose de trois paramètres. Le premier correspond au nombre de places ajoutées pour un avion, le deuxième permet de désigner un code compagnie, et le dernier paramètre (de retour) renvoie une chaîne de caractères.

```
CREATE OR REPLACE PROCEDURE augmenteCap2
 (nbre IN NUMBER, compag IN CHAR, retour OUT VARCHAR)
AS
BEGIN
   UPDATE Avion SET capacite = capacite + nbre
          WHERE compa = compag;
   COMMIT;
   retour := 'OK';
 END;
```

Le code suivant (programme `pdo-proc1.php`) augmente d'une valeur de 20 places la capacité des avions de la compagnie 'SING'. Le code retour est affiché (`La procédure a retourné : OK`).

Notez l'utilisation du paramètre `PARAM_xxx|PDO::PARAM_INPUT_OUTPUT PARAM` pour typer le paramètre et désigner qu'il s'agit d'un paramètre d'entrée ou de sortie. La taille (dernier paramètre de `bindParam`) est également requise.

Tableau 10-24 Appel d'une procédure cataloguée

Code PHP	Commentaires
`$proc = "BEGIN augmenteCap2(?,?,?); END;";` `$stmt = $cx->prepare($proc);`	Préparation de l'appel.

`$stmt->bindParam(1,$v1,PDO::PARAM_INT|PDO::PARAM_INPUT_OUTPUT,1000);`
`$stmt->bindParam(2,$v2,PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT,4);`
`$stmt->bindParam(3,$v3,PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT,2);`
	Mise en place des paramètres.

`$v1 = 20;`
`$v2 = 'SING';`
`$stmt->execute();`
	Affectation des valeurs et appel de la procédure.

`print "La procédure a retourné : ".$v3;` `$cx = null;`	Affichage d'un retour et fermeture de la connexion.

Chapitre 11

Oracle XML DB

Généralités

XML DB est le nom de la technologie d'Oracle qui permet de gérer du contenu XML (stockage, mises à jour et extractions). Alors que certains systèmes ne permettent que la persistance, XML DB offre le contrôle des transactions, l'intégrité, la réplication, l'indexation, la sauvegarde, l'exportation, etc.

Il existe des alternatives à XML DB pour gérer du contenu XML ; citons le XDK (*XML Development Kit*) pour C, C++ et Java. Ces techniques permettent d'analyser le document (*parsing*), les documents XML à l'extérieur de la base et de les stocker dans des types conventionnels (`CLOB`, `BLOB`, `BFILE`, ou `VARCHAR2`). En faisant cela, vous ne pourrez pas bénéficier de toutes les fonctionnalités précitées.

Historique

XML a été pris en compte il y a une dizaine d'années par Oracle 8*i* avec l'apparition de plusieurs paquetages PL/SQL (dont `DBMS_XMLSAVE` et `DBMS_XMLQUERY` qui composaient l'offre XSU : *XML SQL Utility*). Depuis la *Release 1* de la version 9*i*, le type de donnée XMLType est dédié à la gestion de contenus XML. Avec la version 9*i* R2, il est possible d'y associer des grammaires *XML Schema* et de travailler avec *XML Repository*. La version 10*g* a fait évoluer les grammaires *XML Schema*. La version 11*g* a introduit le mode de stockage *binary XML*, un accès par *Web Services*. Depuis la version 12*c*, XML DB est inclus nativement dans la base, le langage XQuery est adopté pour les mises à jour, des fonctionnalités d'indexation textuelles apparaissent et le type `CLOB` a été abandonné en tant que mode de stockage.

Beaucoup (trop) de choses ont évolué depuis le début de cette technologie complexe : fonctions d'extractions, mode de stockage, options par défaut, préconisations, etc. Espérons que les versions à venir ne compromettront pas trop les fonctionnalités présentes.

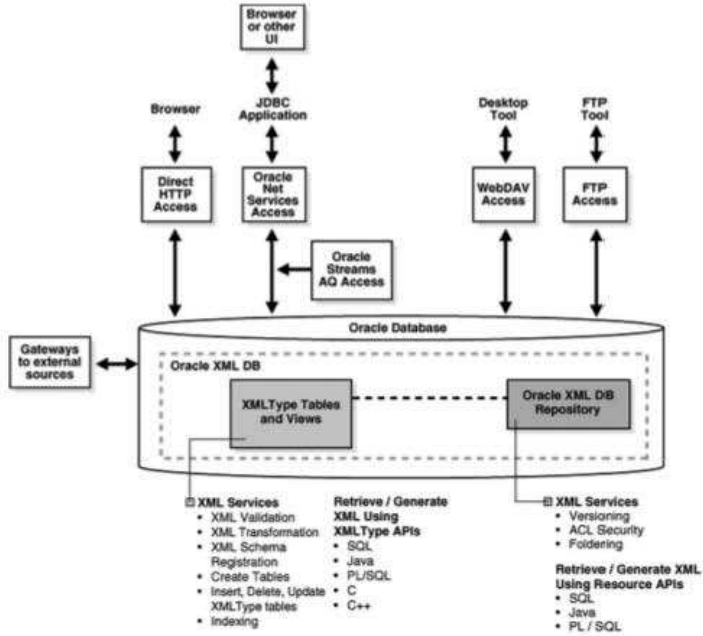
Pour contrôler la présence de l'environnement XML DB, vérifiez que votre instance est bien associée à un service (dans le système d'exploitation, la commande `lsnrctl status` doit retourner « Le service "…XDB" comporte 1 instance(s)… ».

Ce chapitre présente les principales fonctionnalités de XML DB, qui est documenté dans la section « Application Development » du livre *Oracle XML DB Developer's Guide*.

Architecture générale

Les deux principales caractéristiques de XML DB sont, d'une part, l'interopérabilité entre SQL et XML (documents et grammaires) et, d'autre part, la gestion de ressources XML dans un contexte multi-utilisateur avec *XML Repository*.

Figure 11-1 *Architecture de XML DB (© doc. Oracle)*



XML DB est en principe opérationnel si vous avez choisi les options par défaut lors de l'installation. Dans le doute, vous pouvez interroger le dictionnaire des données pour constater

la présence de l'utilisateur XDB (le compte doit être déverrouillé) ou de la vue RESOURCE_VIEW.

Répertoire logique

Si vous n'utilisez pas l'environnement *XML DB Repository*, la création d'un répertoire logique associé à celui qui contiendra vos documents XML est nécessaire. Pensez également à positionner certaines variables d'environnement dans SQL*Plus (SET LONG 10000 et SET PAGESIZE 100) pour ne pas tronquer vos résultats.

```
CREATE DIRECTORY repxml AS 'C:\dev\XML';
```

Les modes de stockage

Suivant la nature du contenu XML que vous aurez à stocker, il vous faut choisir un mode de stockage. Vous devez statuer sur la nature de vos documents XML entre :

- orienté « données » (*data-centric*) où la structure des données est relativement régulière à granularité fine (la plus petite information est située au niveau d'un élément terminal ou d'un attribut) et ne contenant peu ou pas du tout de contenus mixtes. Le mode préconisé est *object-relational* (et son indexation *B-tree* conventionnelle) ;
- orienté « document » (*document-centric*), structure moins régulière des données avec une granularité importante et des contenus mixtes et un ordre des éléments très significatif (page HTML, par exemple). Le mode préconisé est *binary XML* (et son indexation spécifique).

Figure 11-2 *Préconisations du mode de stockage (© doc. Oracle)*

	Data-Centric	Document-Centric	
Use Case	XML schema-based data, with little variation and little structural change over time	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Binary XML	
Indexing	B-tree index	· **XMLIndex** index with structured and unstructured components · XML Full-Text index	· **XMLIndex** index with unstructured component · XML Full-Text index

Selon le mode de stockage choisi, vous disposerez des mêmes fonctions d'extraction mais de mécanismes de validation, de mises à jour et d'indexation différents.

Depuis la version 12*c*, le type CLOB est abandonné en tant que mode de stockage ; il convenait auparavant pour les contenus XML non structurés.

Le tableau 11-1 résume les points forts de chaque mode de stockage. Choisissez le mode *binary XML* si le contenu XML ne doit pas être associé à une grammaire.

Tableau 11-1 Comparatif des modes de stockage

	Object-relational	Binary XML
Extraction	++ (index *B-tree*)	+
Mises à jour	++	+
Espace disque	++	+
Flexibilité des données	– (conformité à une grammaire)	+
Flexibilité des grammaires *XML Schema*	– (une seule grammaire)	++ (une ou plusieurs grammaires)
Validation après insertion	Partielle	+ Totale (si grammaire)
Partitionnement	+	(colonne virtuelle)
Compression	++	+

Le type XMLType

Le type de données XMLType fournit de nombreuses fonctionnalités, la plupart relatives à XML (validation de schéma XML et transformation XSL), ainsi que d'autres qui concernent SQL :

- définition d'une colonne d'une table (jouant le rôle d'un type de donnée) ;
- définition d'une table (jouant le rôle d'un type d'objet) ;
- déclaration de variables PL/SQL ;
- appel de méthodes (procédures PL/SQL).

Le mode de stockage se choisit à l'aide de la clause XMLTYPE résumée dans la syntaxe suivante :

```
XMLTYPE [STORE AS
      { OBJECT RELATIONAL | BINARY XML }
      [ [ XMLSCHEMA nomXMLSchema ]
      ELEMENT { élément | nomXMLSchema # élément …} ]
      [ VIRTUAL COLUMNS ( colonne1 AS (expression1),...) ] ];
```

- La clause `OBJECT RELATIONAL` devra être associée à l'option `XMLSCHEMA` pour associer une grammaire *XML Schema*.
- La clause `VIRTUAL COLUMNS` est réservée au mode `BINARY XML` pour construire des index ou des contraintes (définies avec `ALTER TABLE`).
- En l'absence de clause `STORE AS`, le mode de stockage par défaut était *object-relational* jusqu'à la version 11gR2, depuis c'est devenu *binary XML*.

Le tableau 11-2 présente différentes utilisations de types XML en mode *binary XML*.

Tableau 11-2 Table, colonne et variable XMLType en mode binary XML

Utilisation	Code SQL
Table	`CREATE TABLE t_documents_xml OF XMLType XMLTYPE STORE AS BINARY XML;`
Colonne	`CREATE TABLE t_col_xml (nom_doc VARCHAR2(30)PRIMARY KEY, col_xml XMLType) XMLTYPE col_xml STORE AS BINARY XML;`
Variable PL/SQL	`DECLARE var_xml XMLTYPE; -- par défaut : Binary depuis 11gR2 BEGIN …`

Un type XMLType peut contenir tout contenu XML à condition qu'il soit bien formé (sinon on obtient les erreurs `ORA-64464: Erreur d'événement XML` et `ORA-19202: Une erreur s'est produite lors du traitement la fonction XML…`).

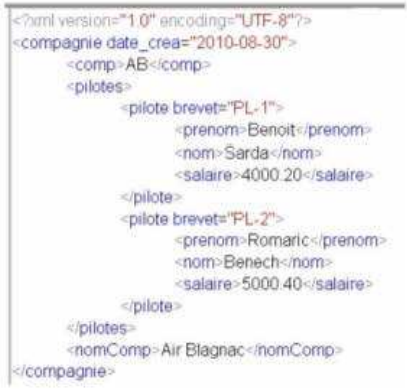
Le contenu peut être contraint selon une grammaire *XML Schema* avec les avantages suivants.

- Le SGBD s'assure de la validité du document XML avant de le stocker dans une ligne (ou colonne) d'une table.
- Comme le contenu d'une table (ou colonne) est conforme à une structure bien connue, XML DB peut optimiser les requêtes et mises à jour.

Insertion d'un document

Le document XML non contraint par une grammaire qui sera stocké (`compagnies.xml`) est le suivant :

Figure 11-3 Document XML non contraint



Le tableau 11-3 présente différentes insertions de ce même document (que vous aurez disposé dans le répertoire associé au répertoire logique précédemment créé) avec le constructeur qui convient au paramètre `BFILE`. Notez l'utilisation des majuscules pour désigner le répertoire logique dans la fonction `BFILENAME` (comme tout objet de schéma au niveau du dictionnaire des données).

Tableau 11-3 Insertion d'un contenu XML bien formé

Utilisation	Code SQL

Table	`INSERT INTO t_documents_xml VALUES`
	`(XMLType`
	`(xmlData => BFILENAME('REPXML','compagnie.xml'),`
	`csid => NLS_CHARSET_ID('AL32UTF8')));`

Colonne	`INSERT INTO t_col_xml (nom_doc,col_xml) VALUES`
	`('compagnie.xml',`
	`(XMLType`
	`(xmlData => BFILENAME('REPXML','compagnie.xml'),`
	`csid => NLS_CHARSET_ID('AL32UTF8'))));`

Variable PL/SQL	`DECLARE`
	`var_xml XMLTYPE;`
	`BEGIN`
	`var_xml := (XMLType`
	`(xmlData => BFILENAME('REPXML','compagnie.xml'),`
	`csid => NLS_CHARSET_ID('AL32UTF8')));`
	`...`

Grammaire XML Schema

Voyons à présent comment valider un contenu XML à l'aide d'une grammaire qui sera enregistrée au préalable dans l'environnement *XML DB Repository*. Considérons une simple grammaire (`compagnies.xsd`) au document XML précédent (`compagnies.xml`).

Figure 11-4 *Exemple de grammaire XML Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="compagnie">
        <xs:complexType>
            <xs:sequence>
                <xs:element type="xs:string" name="comp"/>
                <xs:element name="pilotes">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="pilote" maxOccurs="500" minOccurs="0">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element type="xs:string" name="prenom"/>
                                        <xs:element type="xs:string" name="nom"/>
                                        <xs:element type="xs:float" name="salaire"/>
                                    </xs:sequence>
                                    <xs:attribute type="xs:string" name="brevet" use="optional"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element type="xs:string" name="nomComp"/>
            </xs:sequence>
            <xs:attribute type="xs:date" name="date_crea"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Enregistrement de la grammaire

Pour enregistrer votre grammaire en base (dans le *repository*), vous devez utiliser la procédure `registerschema` du paquetage `DBMS_XMLSCHEMA`. Si la grammaire existe déjà, la procédure `deleteschema` se chargera de la supprimer.

```
BEGIN
--
  DBMS_XMLSCHEMA.deleteschema(
     schemaurl     => 'http://www.actmp.fr/compagnies.xsd',
     delete_option => DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);
--
```

```
  DBMS_XMLSCHEMA.registerschema(
    schemaurl => 'http://www.actmp.fr/compagnies.xsd',
    schemadoc => BFILENAME('REPXML','compagnies.xsd'),
    local     => TRUE,
    gentypes  => FALSE,
    gentables => FALSE,
    options   => DBMS_XMLSCHEMA.REGISTER_BINARYXML,
    csid      => NLS_CHARSET_ID('AL32UTF8'));
END;
/
```

- `schemaurl` spécifie une URI identifiant votre grammaire.
- `delete_option` choisit la politique de suppression (ici, les types, tables et instances conformes au schéma sont éventuellement détruits).
- `schemadoc` référence le fichier grammaire (extension `.xsd`).
- `local` précise que la grammaire est locale (enregistrement dans le répertoire `/sys/schemas/username/...` de *XML DB Repository*). Dans le cas contraire, la grammaire serait globale et se trouverait dans le répertoire `/sys/schemas/PUBLIC/...`
- `gentypes` génère des types objet.
- `gentables` génère une table associée.
- `options` précise la nature de la grammaire (ici, l'encodage des contenus sera identique à celui de leur grammaire).
- `csid` indique le jeu de caractères (AL32UTF8 convient au type de donnée XMLType et équivaut au standard UTF-8).

Le mode de stockage *binary XML* associé à une grammaire fournit une validation complète (*full compliant*)

Validation totale

Pour bénéficier de cette validation, il faut créer une table contenant une colonne de type `XMLType` en mode *binary XML* avec l'option `XMLSCHEMA` et en indiquant l'élément concerné (ici, la racine).

Tableau 11-4 Tables XMLType pour la validation de schéma

Utilisation	Code SQL

Table	`CREATE TABLE t_documents_xml OF XMLType`
	`XMLTYPE STORE AS BINARY XML`
	`XMLSCHEMA "http://www.actmp.fr/compagnies.xsd"`
	`ELEMENT "compagnie";`

Colonne	`CREATE TABLE t_col_xml`
	`(nom_doc VARCHAR2(30) PRIMARY KEY, col_xml XMLType)`
	`XMLTYPE col_xml STORE AS BINARY XML`
	`XMLSCHEMA "http://www.actmp.fr/compagnies.xsd"`
	`ELEMENT "compagnie";`

Tout contenu XML non conforme à la grammaire indiquée lors de la création de la table sera rejeté (`ORA-31000: La ressource … n'est pas un document de schéma XDB`).

Contraintes

Bien que les grammaires *XML Schema* permettent de contraindre du contenu XML considéré individuellement, une contrainte SQL étend une restriction à plusieurs documents (table ou colonne XMLType). De même, la technologie *XML Schema* n'est pas capable d'assurer l'unicité d'une valeur parmi plusieurs documents ou l'existence d'une référence à l'extérieur du document XML.

Si vous désirez bénéficier de contraintes SQL (`UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY` ou `CHECK`), il est nécessaire de définir une ou plusieurs colonnes virtuelles. Une colonne virtuelle est basée sur une expression *XPath* (qui retourne une valeur scalaire provenant d'un élément ou d'un attribut). Des déclencheurs peuvent être programmés pour les règles de gestion plus complexes.

Attention, il n'est pas possible de définir une colonne virtuelle *a posteriori* à l'aide de l'instruction `ALTER TABLE`.

Par ailleurs, ces mécanismes ne sont pas disponibles pour le stockage en mode `CLOB`.

Le tableau 11-5 définit trois colonnes virtuelles (sur le code, le nom et la date de création de la compagnie). Notez l'utilisation du caractère `@` pour désigner un attribut et le nom de la colonne (`doc_xml`) dans la seconde table à l'opposé de `OBJECT_VALUE` pour la table XMLType.

Tableau 11-5 Tables XMLType pour la validation de schéma avec colonnes virtuelles

Utilisation	Code SQL
Table	CREATE TABLE t_documents_xml OF XMLType XMLTYPE STORE AS BINARY XML XMLSCHEMA "http://www.actmp.fr/compagnies.xsd" ELEMENT "compagnie" **VIRTUAL COLUMNS** (c_comp AS (XMLCast(XMLQuery('/compagnie/comp' PASSING **OBJECT_VALUE** RETURNING CONTENT) AS VARCHAR2(6))), c_nomcomp AS (XMLCast(XMLQuery('/compagnie/nomComp' PASSING **OBJECT_VALUE** RETURNING CONTENT) AS VARCHAR2(30))), c_date_crea AS (XMLCast(XMLQuery('/compagnie/@date_crea' PASSING **OBJECT_VALUE** RETURNING CONTENT) AS DATE)));
Colonne	CREATE TABLE t_col_xml (nom_doc VARCHAR2(30) PRIMARY KEY, col_xml XMLType) XMLTYPE col_xml STORE AS BINARY XML XMLSCHEMA "http://www.actmp.fr/compagnies.xsd" ELEMENT "compagnie" **VIRTUAL COLUMNS** (c_comp AS (XMLCast(XMLQuery('/compagnie/comp' PASSING **col_xml** RETURNING CONTENT) AS VARCHAR2(6))), c_nomcomp AS (XMLCast(XMLQuery('/compagnie/nomComp' PASSING **col_xml** RETURNING CONTENT) AS VARCHAR2(30))), c_date_crea AS (XMLCast(XMLQuery('/compagnie/@date_crea' PASSING col_xml RETURNING CONTENT) AS DATE)));

Le tableau 11-6 présente toutes les contraintes qu'il est possible de programmer sur une colonne ou sur une table XMLType en mode *binary XML*.

Tableau 11-6 Déclaration de contraintes SQL

Contraintes	Instructions SQL
Unicité du nom de la compagnie	ALTER TABLE t_col_xml ADD **CONSTRAINT** un_nomcomp_col_bin **UNIQUE**(c_nomcomp);
Clé primaire sur le code compagnie	ALTER TABLE t_documents_xml ADD **CONSTRAINT** pk_t_documents_xml **PRIMARY KEY**(c_comp);

Tableau 11-6 Déclaration de contraintes SQL *(suite)*

Contraintes	Instructions SQL

Vérification de valeurs	`ALTER TABLE t_col_xml`
	`ADD CONSTRAINT ck_date_crea`
	`CHECK (c_date_crea <`
	`TO_DATE('01/01/2015','DD/MM/YYYY')`
	`AND c_date_crea IS NOT NULL);`

Intégrité référentielle vers une table classique qui contient une clé avec un code analogue	`ALTER TABLE t_col_xml`
	`ADD CONSTRAINT fk_comp2`
	`FOREIGN KEY(c_comp)`
	`REFERENCES table_ref(comp_ref);`

Tout contenu XML non conforme aux contraintes sera rejeté. En général, c'est l'erreur `ORA-31000: La ressource … n'est pas un document de schéma XDB` qui est retournée.

Stockage en mode object-relational

Le mode *object-relational* convient pour les documents XML fortement structurés qu'il sera possible de contraindre avec SQL. Vous devrez associer une grammaire *XML Schema* à votre table (ou colonne) de type XMLType pour ne pas obtenir l'erreur `ORA-19002: URL XMLSchema absent`.

Annotation de la grammaire

Il est intéressant d'annoter la grammaire pour mieux faire correspondre le modèle de document XML (éléments et attributs) avec les colonnes du SGBD (nom et type) :

- spécifier les tables qui stockeront le contenu XML ;
- surcharger le *mapping* entre les types *XML Schema* et les types SQL ;
- nommer les colonnes qui seront générées.

Préfixés par `xdb` (indiquant l'espace de nom imposé par Oracle *http://xmlns.oracle.com/xdb*), de nombreux attributs permettent d'enrichir la grammaire. La figure 11-5 présente les principaux attributs.

Figure 11-5 *Attributs d'annotation*

`xdb:defaultTable`	Nom de la table par défaut générée automatiquement et exploitable avec *XML DB Repository*.
`xdb:defaultTableSchema`	Nom du schéma Oracle.
`xdb:SQLName`	Nom d'une colonne donné à un élément ou un attribut XML.
`xdb:SQLType`	Nom du type Oracle.
`xdb:SQLCollType`	Nom du type de la collection.
`xdb:storeVarrayAsTable`	`true` par défaut ; la collection est stockée comme un ensemble de lignes d'une table (*ordered collection table* : OCT). Si `false`, la collection est sérialisée et stockée dans une colonne *LOB*.
`xdb:columnProps`	Précise les caractéristiques des colonnes de la table par défaut. Utile pour déclarer une clé primaire, une clé étrangère ou une contrainte de vérification.
`xdb:tableProps`	Indique les caractéristiques de stockage de la table par défaut.

Par défaut, l'enregistrement de la grammaire génère une table pour chaque élément global du *XML Schema* (pour empêcher cela : `xdb:defaultTable=""`).

Cette technique d'annotation peut aussi être utilisée avec le mode de stockage *binary XML* mais elle permet moins de fonctionnalités.

Considérons les annotations suivantes :

- les types et colonnes sont notés en majuscules pour mieux les différencier des éléments et attributs XML, mais aussi car c'est ainsi qu'Oracle les stocke en interne ;
- les éléments code et nom de la compagnie sont obligatoires (`minOccurs="1"`) ;
- si une collection de pilotes existe, elle n'est pas vide ; notez l'utilisation de l'attribut `SQLCollType`.

L'élément de la collection est décrit par la figure 11-7, et les types de données des éléments inclus sont précisés ainsi que le type de l'attribut (ici, le numéro de brevet sur 4 caractères au maximum).

Enfin, les tailles des colonnes sont précisées dans la grammaire (voir figure 11-8). Ici, le code de la compagnie ne dépasse pas 6 caractères, son nom 40 caractères, les prénom et nom seront limités à 30 caractères et le salaire sera codé en `NUMBER(9,2)`.

Figure 11-6 *Exemple de grammaire annotée*

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xdb="http://xmlns.oracle.com/xdb">
    <xsd:element name="compagnie" type="compagnieType" xdb:defaultTable=""/>
    <xsd:complexType name="compagnieType" xdb:SQLType="COMPAGNIE_TYPE">
        <xsd:sequence>
            <xsd:element name="comp" type="compType" minOccurs="1" xdb:SQLName="COMP"/>
            <xsd:element name="pilotes" type="pilotesType" xdb:SQLName="PILOTES"/>
            <xsd:element name="nomComp" type="nomCompType" minOccurs="1" xdb:SQLName="NOM_COMP"/>
        </xsd:sequence>
        <xsd:attribute type="xsd:date" name="date_crea" xdb:SQLName="DATE_CREA" xdb:SQLType="DATE"/>
    </xsd:complexType>
    <xsd:complexType name="pilotesType" xdb:SQLType="PILOTES_TYPE">
        <xsd:sequence>
            <xsd:element name="pilote" type="piloteType" minOccurs="1" maxOccurs="500"
            xdb:SQLName="PILOTE" xdb:SQLCollType="PILOTE_VRY_TYPE"/>
        </xsd:sequence>
    </xsd:complexType>
    voir la suite plus loin...
</xsd:schema>
```

Figure 11-7 *Suite de la grammaire annotée*

```
<xsd:complexType name="piloteType" xdb:SQLType="PILOTE_TYPE">
    <xsd:sequence>
        <xsd:element name="prenom" type="nomType" xdb:SQLName="PRENOM" xdb:SQLType="VARCHAR2"/>
        <xsd:element name="nom" type="nomType" xdb:SQLName="NOM" xdb:SQLType="VARCHAR2"/>
        <xsd:element name="salaire" type="salaireType" minOccurs="0" xdb:SQLName="SALAIRE" xdb:SQLType="NUMBER"/>
    </xsd:sequence>
    <xsd:attribute name="brevet" use="required" xdb:SQLName="BREVET" xdb:SQLType="VARCHAR2">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:minLength value="1"/>
                <xsd:maxLength value="4"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>
voir la suite plus loin...
```

Figure 11-8 *Suite et fin de la grammaire annotée*

```
<xsd:simpleType name="compType">
    <xsd:restriction base="xsd:string">
        <xsd:minLength value="1"/>
        <xsd:maxLength value="6"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="nomCompType">
    <xsd:restriction base="xsd:string">
        <xsd:minLength value="1"/>
        <xsd:maxLength value="40"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="nomType">
    <xsd:restriction base="xsd:string">
        <xsd:minLength value="1"/>
        <xsd:maxLength value="30"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="salaireType">
    <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="7"/>
        <xsd:fractionDigits value="2"/>
    </xsd:restriction>
</xsd:simpleType>
```

Vous devrez enregistrer cette grammaire avec un nouvel URI (`schemaurl`) et sans utiliser le paramètre `options` qui pourrait désigner une grammaire *binary XML*. Notez ici l'utilisation du paramètre `gentypes` pour générer les types objet qui sont nommés dans la grammaire.

```
BEGIN
  DBMS_XMLSCHEMA.DELETESCHEMA(
    schemaurl     => 'http://www.actmp.fr/compagniesannote.xsd',
    delete_option => DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);
--
  DBMS_XMLSCHEMA.REGISTERSCHEMA(
    schemaurl => 'http://www.actmp.fr/compagniesannote.xsd",
    schemadoc => BFILENAME('REPXML','compagniesannote.xsd'),
    local     => TRUE,
    gentypes  => TRUE,
    gentables => FALSE,
    csid      => NLS_CHARSET_ID('AL32UTF8'));
END;
/
```

En général, chaque collection (élément XML disposant d'un attribut `maxOccurs` >1) est sérialisée et donc mal adaptée pour les modifications de contenu. La clause `VARRAY` (voir plus loin) définira une table de stockage pour chaque collection afin de faciliter les mises à jour.

Création d'une table (ou colonne) object-relational

Le tableau 11-7 présente les deux possibilités pour stocker du contenu XML en mode *object-relational*.

Tableau 11-7 Modes de stockage object-relational

Utilisation	Code SQL

Table	`CREATE TABLE t_documents_xml OF XMLType`
	`XMLTYPE STORE AS OBJECT RELATIONAL`
	`XMLSCHEMA "http://www.actmp.fr/compagniesannote.xsd"`
	`ELEMENT "compagnie";`

Colonne	`CREATE TABLE t_col_xml`
	`(nom_doc VARCHAR2(30) PRIMARY KEY, col_xml XMLType)`
	`XMLTYPE col_xml STORE AS OBJECT RELATIONAL`
	`XMLSCHEMA "http://www.actmp.fr/compagniesannote.xsd"`
	`ELEMENT "compagnie";`

Validation partielle

Par défaut, le mode de stockage *object-relational* n'offre qu'une validation partielle même si la grammaire associée est enrichie.

Dans notre exemple, il sera possible d'insérer un document dans la table (ou la colonne) contenant une compagnie sans pilote, sans nom ou code. Par ailleurs, l'ordre des éléments dans la séquence ne sera pas respecté (le prénom peut se trouver après le salaire et le nom précéder le prénom). De plus, le salaire n'est même pas obligatoire, ce qui est en opposition avec la grammaire enregistrée ! En revanche, vous ne pourrez pas insérer du contenu XML contenant des éléments ou attributs supplémentaires (qui ne sont pas prévus dans la grammaire) ou dont l'élément racine n'est pas celui de la grammaire (ici, compagnie).

Les erreurs retournées seront les mêmes que pour la validation totale du mode *binary XML*, à savoir `ORA-31000: La ressource … n'est pas un document de schéma XDB`. Pour les incohérences dues aux types de données (taille d'une chaîne, précision d'un numérique ou format de date), les messages peuvent être plus clairs (voir le tableau 11-8).

Tableau 11-8 Erreur de typage

Fragment XML concerné	Erreur Oracle

`<?xml version="1.0" encoding="UTF-8"?>`
`<compagnie date_crea="2010-08-35">`
`<comp>AB</comp> …`
	`ORA-01847: le jour du mois doit être compris entre 1 et le dernier jour du mois.`

`<?xml version="1.0" encoding="UTF-8"?>`
`<compagnie date_crea="2010-08-30">`
`<comp>ABC-FTRG</comp> …`
	`ORA-30951: L'élément ou l'attribut (Xpath ABC-FTRG) dépasse la longueur maximale.`

Validation totale

Pour que la validation soit totale comme en mode *binary XML* (*full compliant*), vous devrez ajouter à une table (ou colonne) XMLType en question soit une contrainte `CHECK` avec la fonction `XMLISVALID`, soit un déclencheur `BEFORE INSERT`.

Assurez-vous que la table ne contienne pas des enregistrements ne respectant pas la grammaire sinon Oracle retournera l'erreur `ORA-02293: impossible de valider (…) - violation d'une contrainte de contrôle`.

Le tableau 11-9 décrit la mise en place de la validation totale pour une table ou une colonne XMLType avec un mode de stockage objet.

Tableau 11-9 Contraintes de vérification

Au niveau table	Au niveau colonne

`ALTER TABLE t_documents_xml`
`ADD CONSTRAINT valide_comp_t`
`CHECK (XMLIsValid(OBJECT_VALUE) = 1);`
	`ALTER TABLE t_col_xml`
	`ADD CONSTRAINT valide_comp_col`
	`CHECK (XMLIsValid(col_xml) = 1);`

Une fois cette contrainte activée, seuls les documents conformes à la grammaire pourront être stockés. L'erreur retournée, le cas échéant, sera invariablement `ORA-02290: violation de contraintes (...) de vérification`.

Si vous optez pour un déclencheur `BEFORE INSERT`, vous devrez utiliser la fonction `SCHEMAVALIDATE` qui retournera une exception en cas de contenu non conforme.

Tableau 11-10 Déclencheurs de validation

Au niveau table	Au niveau colonne

`CREATE TRIGGER trig_valide_comp_t`
`BEFORE INSERT OR UPDATE ON t_documents_xml`
`FOR EACH ROW`
`BEGIN`
`IF (:NEW.OBJECT_VALUE IS NOT NULL) THEN`
`:NEW.OBJECT_VALUE.SchemaValidate();`
`END IF;`
`END;`
`/`
	`CREATE TRIGGER trig_valide_comp_col`
	`BEFORE INSERT OR UPDATE ON t_col_xml`
	`FOR EACH ROW`
	`BEGIN`
	`IF (:NEW.col_xml IS NOT NULL) THEN`
	`:NEW.col_xml.SchemaValidate();`
	`END IF;`
	`END;`
	`/`

En général, le déclencheur aura l'avantage de renseigner davantage les erreurs comme l'illustre le code suivant dans lequel se trouve un problème d'ordonnancement entre les éléments composant la balise `pilote`.

Tableau 11-11 Insertion de contenu non conforme

Fragment XML concerné	Erreur Oracle

`<?xml version="1.0" encoding="UTF-8"?>`
`<compagnie date_crea="2010-08-30">`
`<comp>AB</comp>`
`<pilotes>`
`<pilote brevet="PL-1">`
`<nom>Sarda</nom>`
`<salaire>4000.20</salaire>`
`<prenom>Benoit</prenom>`
`</pilote> …`
	`ORA-31154: document XML non valide`
	`ORA-19202: Une erreur s'est produite lors du traitement de la fonction XML (LSX-00213: seulement 0 occurrences de l'élément "prenom" ; minimum : 1)…`
	`ORA-04088: erreur lors de l'exécution du déclencheur 'OXM.TRIG_VALIDE_COMP_T'`

Contraintes

Deux mécanismes peuvent être mis en œuvre pour définir une contrainte SQL. La pseudo-colonne `XMLDATA` qui indique un chemin dans une arborescence permettra de contraindre un élément (ou un attribut). Pour disposer des contraintes sur des éléments (ou attributs) présents dans une collection, vous devrez utiliser la table de stockage définie dans la directive `VARRAY` (voir plus loin).

Éléments et attributs hors collection

Le tableau 11-12 présente toutes les contraintes qu'il est possible de programmer sur une colonne ou sur une table XMLType en mode objet.

Tableau 11-12 Déclaration de contraintes SQL

Contraintes	Instructions SQL

Unicité du nom de la compagnie	`ALTER TABLE t_documents_xml`
	`ADD CONSTRAINT un_nomcomp_bin`
	`  UNIQUE(XMLDATA."NOM_COMP");`
	
	`ALTER TABLE t_col_xml`
	`ADD CONSTRAINT un_nomcomp_col_bin`
	`  UNIQUE(col_xml.XMLDATA."NOM_COMP");`

Clé primaire sur le code compagnie	`ALTER TABLE t_documents_xml`
	`ADD CONSTRAINT pk_t_documents_xml`
	`  PRIMARY KEY(XMLDATA."COMP");`

Vérification de valeurs	`ALTER TABLE t_documents_xml`
	`ADD  CONSTRAINT ck_date_crea`
	`  CHECK (XMLDATA."DATE_CREA" <`
	`  TO_DATE('01/01/2015','DD/MM/YYYY')`
	`  AND XMLDATA."DATE_CREA" IS NOT NULL);`
	
	`ALTER TABLE t_col_xml`
	`ADD  CONSTRAINT ck_col_date_crea`
	`  CHECK (col_xml.XMLDATA."DATE_CREA" <`
	`  TO_DATE('01/01/2015','DD/MM/YYYY')`
	`  AND col_xml.XMLDATA."DATE_CREA"`
	`  IS NOT NULL);`

Intégrité référentielle vers une table classique qui contient une clé avec un code analogue	`ALTER TABLE  t_documents_xml`
	`ADD CONSTRAINT fk_comp1`
	`  FOREIGN KEY(XMLDATA."COMP")`
	`  REFERENCES compagnie_ref (comp_ref);`
	
	`ALTER TABLE  t_col_xml`
	`ADD CONSTRAINT fk_comp2`
	`  FOREIGN KEY(col_xml.XMLDATA."COMP")`
	`  REFERENCES compagnie_ref (comp_ref);`

Tout contenu XML non conforme aux contraintes sera rejeté avec des messages d'erreur plus parlants que ceux du mode de stockage *binary XML* (qui retourne le plus souvent `ORA-31000:`). Le tableau 11-13 présente les messages d'erreur potentiels (ici, `oxm` est le nom de l'utilisateur).

Tableau 11-13 Vérification des contraintes SQL

Contraintes	Erreur Oracle
Unicité du nom de la compagnie	`ORA-00001: violation de contrainte unique (OXM.UN_NOMCOMP_BIN)`
Clé primaire sur le code compagnie	`ORA-00001: violation de contrainte unique (OXM.PK_T_DOCUMENTS_XML)`
Vérification de valeurs	`ORA-02290: violation de contraintes (OXM.CK_COL_DATE_CREA) de vérification`
Intégrité référentielle	`ORA-02291: violation de contrainte d'intégrité (OXM.FK_COMP2) - clé parent introuvable`

Éléments et attributs dans une collection

Pour manipuler efficacement des éléments (ou attributs) se trouvant dans une collection XML, vous devrez créer une table associée (appelée *varray*). Cette table imbriquée va stocker l'union de toutes les collections des contenus XML. Pour contrôler le nom des types générés pour cette collection, vous pouvez annoter la grammaire (voir l'attribut `xdb:SQLCollType` dans l'exemple précédent).

Le nom de la table de stockage est défini dans la clause `STORE AS` de la directive `VARRAY`. En interne et au niveau de chaque contenu XML, la colonne `NESTED_TABLE_ID` est l'identifiant de la table de stockage. Avec le nom de la table stockage, il est possible de définir des contraintes SQL sur des éléments (ou attributs) de la collection.

Le tableau 11-14 présente la création de tables imbriquées visant à manipuler des collections d'éléments XML. La syntaxe est présentée pour une table XMLType et une colonne XMLType de mode objet. Notez que chaque table imbriquée est nommée pour pouvoir définir par la suite des contraintes.

Tableau 11-14 Stockage de collections pour le mode object-relational

Utilisation	Code SQL

Table	`CREATE TABLE t_documents_xml OF XMLType`
	`XMLTYPE STORE AS OBJECT RELATIONAL`
	`XMLSCHEMA "http://www.actmp.fr/compagniesannote.xsd"`
	`ELEMENT "compagnie"`
	`VARRAY "XMLDATA"."PILOTES"."PILOTE"`
	`STORE AS TABLE pilote_table`
	`((PRIMARY KEY (NESTED_TABLE_ID,`
	`SYS_NC_ARRAY_INDEX$)));`

Colonne	`CREATE TABLE t_col_xml`
	`(nom_doc VARCHAR2(30) PRIMARY KEY, col_xml XMLType)`
	`XMLTYPE col_xml STORE AS OBJECT RELATIONAL`
	`XMLSCHEMA "http://www.actmp.fr/compagniesannote.xsd"`
	`ELEMENT "compagnie"`
	`VARRAY col_xml."XMLDATA"."PILOTES"."PILOTE"`
	`STORE AS TABLE pilote_col_table`
	`((PRIMARY KEY (NESTED_TABLE_ID,`
	`SYS_NC_ARRAY_INDEX$)));`

Il existe d'autres moyens de définir des collections par annotation (`xdb:storeVarrayAsTables` et `xdb:maintainOrder`) ou par indicateur lors de l'enregistrement de la grammaire (par exemple, `REGISTER_NT_AS_IOT` dans le paramètre `options`). Cependant, les valeurs par défaut de ces paramètres ont déjà évolué avec les versions d'Oracle.

Tableau 11-15 Création de contraintes SQL sur une collection

Contraintes	Code SQL

Unicité du code d'un brevet pour chaque compagnie	`ALTER TABLE pilote_table`
	`ADD CONSTRAINT un_pilote_table_doc_brevet`
	`UNIQUE(NESTED_TABLE_ID, "BREVET");`
	
	`ALTER TABLE pilote_col_table`
	`ADD CONSTRAINT un_pilcol_table_doc_brevet`
	`UNIQUE(NESTED_TABLE_ID, "BREVET");`

Unicité du code d'un brevet pour toutes les compagnies (tous les contenus XML seront concernés)	`ALTER TABLE pilote_table`
	`ADD CONSTRAINT un_pilote_table_docs_brevet`
	`UNIQUE (brevet);`
	
	`ALTER TABLE pilote_col_table`
	`ADD CONSTRAINT un_pilcol_table_docs_brevet`
	`UNIQUE (brevet);`

Tableau 11-15 Création de contraintes SQL sur une collection *(suite)*

Contraintes	Code SQL
Vérification de valeurs (ici, sur le salaire et le prénom). Même syntaxe pour la table XMLType	`ALTER TABLE pilote_col_table ADD CONSTRAINT ck_pilcol_table_salaire CHECK ((salaire IS NULL OR salaire > 600) AND prenom = UPPER(prenom));`
Intégrité référentielle	`Pas possible : ORA-30730: contrainte référentielle interdite sur une colonne de table imbriquée`

Tout contenu XML non conforme à ces nouvelles contraintes sera rejeté avec des messages d'erreur correspondants aux exceptions. Le tableau 11-16 présente les messages d'erreur potentiels.

Tableau 11-16 Vérification des contraintes SQL sur une collection

Contraintes	Erreur Oracle
Unicité du code d'un brevet pour chaque compagnie	`ORA-00001: violation de contrainte unique (OXM.UN_PILCOL_TABLE_DOC_BREVET)`
Unicité du code d'un brevet pour toutes les compagnies	`ORA-00001: violation de contrainte unique (OXM.UN_PILCOL_TABLE_DOCS_BREVET)`
Vérification de valeurs	`ORA-02290: violation de contraintes (OXM.CK_PILCOL_TABLE_SALAIRE) de vérification`

Extractions

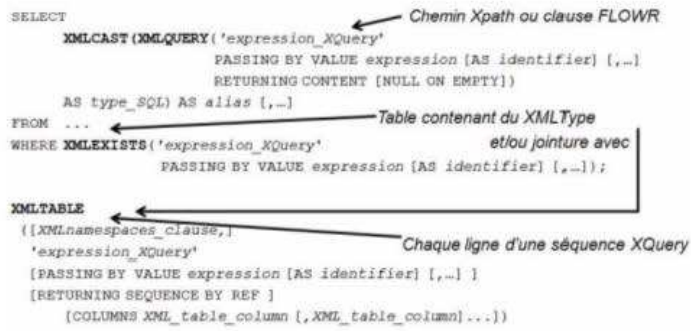
Oracle fournit différentes fonctions SQL pour XML qui manipulent ou retournent des fragments XML. Les paramètres de ces fonctions ne sont pas définis dans les normes SQL ANSI/ISO/IEC mais sont explicités dans des spécifications du W3C (notamment celles concernant XPath, XQuery et les *namespaces*).

Figure 11-9 *Fonctions SQL pour XML*

APPENDCHILDXML	SYS_DBURIGEN	XMLAGG	XMLPATCH
DELETEXML	SYS_XMLAGG	XMLCAST	XMLPI
DEPTH	SYS_XMLGEN	XMLCDATA	XMLQUERY
EXISTSNODE	UPDATEXML	XMLCOLATTVAL	XMLROOT
EXTRACT (XML)		XMLCOMMENT	XMLSEQUENCE
EXTRACTVALUE		XMLCONCAT	XMLSERIALIZE
INSERTCHILDXML		XMLDIFF	XMLTABLE
INSERTCHILDXMLAFTER		XMLELEMENT	XMLTRANSFORM
INSERTCHILDXMLBEFORE		XMLEXISTS	
INSERTXMLAFTER		XMLFOREST	
INSERTXMLBEFORE		XMLISVALID	
PATH		XMLPARSE	

Sans entrer dans les détails de tous les paramètres de chaque fonction, vous aurez besoin de connaître `XMLQUERY`, `XMLTABLE`, `XMLEXISTS`, et `XMLCAST` pour interroger efficacement vos contenus XML. Les requêtes utilisant ces fonctions devront s'organiser de la manière suivante.

Figure 11-10 *Architecture générale d'une requête Oracle SQL/XML*



Ces fonctions respectent les normes SQL/XML:2011 et permettent :

- de générer du contenu XML à partir de tables/vues conventionnelles ;
- d'extraire des lignes (*relational data*) à partir de contenu XML.

Pour éviter que les résultats de vos requêtes soient tronqués dans la console SQL*Plus (SQL Developer est beaucoup moins stable pour cela), positionnez la variable d'environnement LONG à un chiffre suffisamment important (par exemple, SET LONG 10000).

Les fonctions EXTRACT et EXTRACTVALUE sont déclarées obsolètes depuis la version 11*g*R2.

La fonction XMLQuery

La fonction XMLQuery retourne une séquence d'instances XMLType (ou NULL) et sa syntaxe simplifiée est la suivante :

```
XMLQuery
 (expression_XQuery
  [PASSING [BY VALUE] expression [AS identifiant] ...]
   RETURNING CONTENT [NULL ON EMPTY]))
```

- PASSING désigne une ou plusieurs expressions (colonnes, *bind variable* ou PL/SQL). Chacune de ces expressions doit retourner un type XMLType ou un type SQL (qui ne doit être ni objet ni collection). Le terme OBJECT_VALUE (dénoté dans certaines versions SYS_NC_ROWINFO$) permet d'adresser une table XMLType.
- NULL ON EMPTY (par défaut) retourne NULL si aucun résultat ne peut être extrait.

Le tableau 11-17 présente quelques extractions avec la fonction XMLQuery ; trois documents XML ont été insérés dans les tables présentées précédemment (t_documents_xml et t_col_xml), et peu importe le mode de stockage, qu'il soit *binary XML* ou objet et qu'une grammaire *XML Schema* soit associée ou pas. En remplaçant col_xml par OBJECT_VALUE dans la clause PASSING BY VALUE, vous obtiendrez l'expression XQuery pour la table XMLType t_documents_xml.

Tableau 11-17 Utilisation de la fonction XMLQuery

Code SQL

```
-- Un élément
SELECT nom_doc,
       XMLQuery('/compagnie/comp'
        PASSING BY VALUE col_xml
        RETURNING CONTENT)
       AS "/compagnie/comp"
FROM  t_col_xml;
```

Résultats

```
NOM_DOC         /compagnie/comp
--------------- ------------------
compagnie.xml   <comp>AB</comp>
compagnie2.xml  <comp>AC</comp>
compagnie3.xml  <comp>TA</comp>
```

Code SQL

```
-- Plusieurs éléments
SELECT XMLQuery('/compagnie/pilotes'
        PASSING BY VALUE col_xml
        RETURNING CONTENT)
       AS "/compagnie/pilotes"
FROM  t_col_xml
WHERE nom_doc = 'compagnie3.xml';
```

Résultats

```
<pilotes>
  <pilote brevet="PL-25">
    <prenom>Jean-Phi</prenom
    <nom>Ferrage</nom>
    <salaire>5000</salaire>
  </pilote>
  <pilote brevet="PL-62">
    <prenom>Joel</prenom>
    <nom>Hartman</nom>
    <salaire>5000</salaire>
  </pilote>
</pilotes>
```

Code SQL

```
-- Un attribut
SELECT nom_doc,
      XMLQuery('/compagnie/@date_crea'
       PASSING BY VALUE col_xml
       RETURNING CONTENT)
      AS "/compagnie/@date_crea"
FROM  t_col_xml;
```

Résultats

```
NOM_DOC         /compagnie/@date_crea
--------------- ----------------------
compagnie.xml   2010-08-30
compagnie2.xml  2012-09-01
compagnie3.xml  2013-04-01
```

Code SQL

```
-- prédicat XPath
SELECT nom_doc,
    XMLQuery('/compagnie/pilotes/pilote
              [salaire<5000]/nom'
     PASSING BY VALUE col_xml
     RETURNING CONTENT)
    AS "plus de 5000"
FROM  t_col_xml;
```

Résultats

```
NOM_DOC         plus de 5000
--------------- --------------------
compagnie.xml   <nom>Sarda</nom>
compagnie2.xml  <nom>Giaccone</nom>
                <nom>Calac</nom>
compagnie3.xml
```

La fonction XMLCast

La fonction XMLCast transforme une expression en un type SQL (NUMBER, VARCHAR2, CHAR, CLOB, BLOB, REF XMLTYPE ou DATE). Il n'est pas possible de transformer un type XML en un autre type XML, ou un type SQL en un type XML. La syntaxe simplifiée de cette fonction est la suivante :

```
SELECT XMLCast(XMLQUERY(...) AS type_SQL)
FROM   ...
```

Le tableau 11-18 présente quelques extractions avec la fonction XMLCast. En remplaçant col_xml par OBJECT_VALUE dans la clause PASSING, vous obtiendrez l'expression XQuery pour la table XMLType t_documents_xml.

Tableau 11-18 Utilisation de la fonction XMLCast

Code SQL	Résultats

```
-- Un élément
SELECT nom_doc,
  XMLCAST(
    XMLQUERY('/compagnie/nomComp'
    PASSING col_xml RETURNING CONTENT)
  AS VARCHAR2(30)) AS nom_comp
FROM  t_col_xml;
```

```
NOM_DOC         NOM_COMP
--------------- --------------------
compagnie.xml   Air Blagnac
compagnie2.xml  Air Castanet
compagnie3.xml  Toulouse Air
```

```
-- Un attribut
SELECT nom_doc,
  XMLCAST(
    XMLQUERY('/compagnie/@date_crea'
    PASSING col_xml RETURNING CONTENT)
  AS DATE) AS date_crea
FROM  t_col_xml;
```

```
NOM_DOC         DATE_CREA
--------------- ----------------
compagnie.xml   30/08/10
compagnie2.xml  01/09/12
compagnie3.xml  01/04/13
```

```
-- prédicat XPath
SELECT nom_doc,
  XMLCAST(
    XMLQUERY('/compagnie/pilotes/pilote
            [@brevet="PL-25"]/nom'
    PASSING col_xml RETURNING CONTENT)
  AS VARCHAR2(20)) AS nom
FROM  t_col_xml;
```

```
NOM_DOC         NOM
--------------- ---------------
compagnie.xml
compagnie2.xml
compagnie3.xml  Ferrage
```

```
-- plusieurs items retournés
SELECT XMLCAST(XMLQUERY(
    '/compagnie/pilotes/pilote/nom'
    PASSING col_xml RETURNING CONTENT)
  AS VARCHAR2(80)) AS resultat
FROM  t_col_xml
WHERE nom_doc = 'compagnie2.xml';
```

```
RESULTAT
-------------------------
GiacconeCalac
```

```
-- élément non terminal
SELECT XMLCAST(
       XMLQUERY('/compagnie/pilotes'
      PASSING col_xml RETURNING CONTENT)
  AS VARCHAR2(80)) AS resultat
FROM  t_col_xml
WHERE nom_doc = 'compagnie2.xml';
```

```
RESULTAT
------------------------------
AimeGiaccone4200PierreCalac3000
```

Cette fonction convient pour l'extraction de valeurs scalaires (éléments terminaux ou attributs). Si l'élément est complexe, le résultat est la sérialisation de tout son contenu. Si plusieurs items sont retournés, vous devrez utiliser conjointement la fonction XMLTABLE.

La fonction XMLTable

La fonction XMLTable transforme une expression en un type SQL (NUMBER, VARCHAR2, CHAR, CLOB, BLOB ou DATE). Il n'est pas possible de transformer un type XML en un autre type XML ou un type SQL en un type XML. La syntaxe simplifiée de cette fonction est la suivante :

```
SELECT alias1.col | alias2.col | alias2.COLUMN_VALUE …
FROM   nom_table alias1 , XMLTABLE
       (expression_XQuery
       [PASSING expression [AS identifiant]
       [COLUMNS column [PATH string] [,...] ) alias2
```

…

- *expression* indique le nom colonne XMLType (ou OBJECT_VALUE pour une table XMLType).
- COLUMN_VALUE devra être utilisé si aucune colonne n'est décrite dans la directive COLUMNS.

Le tableau 11-19 présente quelques extractions avec la fonction XMLTable. En remplaçant col_xml par OBJECT_VALUE dans la clause PASSING, vous obtiendrez l'expression XQuery pour la table XMLType t_documents_xml.

Tableau 11-19 Utilisation de la fonction XMLTable

Code SQL	Résultats

`-- Plusieurs éléments non terminaux`
 	
`SELECT a2.COLUMN_VALUE AS pilote`
`FROM t_col_xml a1,`
`XMLTABLE(`
`'/compagnie/pilotes/pilote'`
`PASSING a1.col_xml) a2`
`WHERE a1.nom_doc= 'compagnie2.xml';`
	`PILOTE`
	`--------------------------`
	`<pilote brevet="PL-9">`
	`<prenom>Aime</prenom>`
	`<nom>Giaccone</nom>`
	`<salaire>4200</salaire>`
	`</pilote>`
	`<pilote brevet="PL-6">`
	`<prenom>Pierre</prenom>`
	`<nom>Calac</nom>`
	`<salaire>3000</salaire>`
	`</pilote>`

`-- Plusieurs éléments terminaux`
 	
`SELECT a1.nom_doc,`
`a2.COLUMN_VALUE AS nom`
`FROM t_col_xml a1,`
`XMLTABLE(`
`'/compagnie/pilotes/pilote/nom'`
`PASSING a1.col_xml) a2;`
	`NOM_DOC NOM`
	`--------------- ------------------`
	`compagnie.xml <nom>Sarda</nom>`
	`compagnie.xml <nom>Benech</nom>`
	`compagnie2.xml <nom>Giaccone</nom>`
	`compagnie2.xml <nom>Calac</nom>`
	`compagnie3.xml <nom>Gazagnes</nom>`
	`compagnie3.xml <nom>Ferrage</nom>`
	`compagnie3.xml <nom>Hartman</nom>`

Tableau 11-19 Utilisation de la fonction XMLTable *(suite)*

Code SQL

```
-- Idem avec COLUMNS
SELECT   a1.nom_doc,a2.brevetp, a2.nomp
FROM     t_col_xml a1,
 XMLTABLE(
 '/compagnie/pilotes/pilote'
 PASSING a1.col_xml
 COLUMNS nomp    VARCHAR2(20) PATH 'nom',
         brevetp VARCHAR2(10)
                 PATH '@brevet') a2
ORDER BY a2.nomp;
```

Résultats

```
NOM_DOC          BREVETP    NOMP
---------------- ---------- -----------
-
compagnie.xml    PL-2       Benech
compagnie2.xml   PL-6       Calac
compagnie3.xml   PL-25      Ferrage
compagnie3.xml   PL-15      Gazagnes
compagnie2.xml   PL-9       Giaccone
compagnie3.xml   PL-62      Hartman
compagnie.xml    PL-1       Sarda
```

```
-- Plusieurs XMLTABLE avec « jointure »

SELECT a1.nom_doc, a2.nomc,
     SUM(TO_NUMBER(a3.sal)) AS salaires
FROM  t_col_xml a1,
 XMLTABLE('/compagnie' PASSING a1.col_xml
   COLUMNS nomc VARCHAR2(20) PATH
                'nomComp',
   pils XMLType PATH 'pilotes/pilote') a2,
 XMLTABLE('pilote' PASSING a2.pils
    COLUMNS sal NUMBER PATH 'salaire') a3
GROUP BY a1.nom_doc, a2.nomc;
```

```
NOM_DOC         NOMC          SALAIRES
--------------- ------------ ----------
compagnie3.xml  Toulouse Air      17200
compagnie.xml   Air Blagnac     9000.6
compagnie2.xml  Air Castanet       7200
```

La fonction XMLExists

La fonction `XMLExists` teste une expression XQuery et retourne `TRUE` si un fragment est non vide (sinon `FALSE`). La syntaxe simplifiée de cette fonction est la suivante :

```
XMLExists(expression_XQuery
                [PASSING expression [AS identifiant] ])
```

L'utilisation de la fonction `XMLExists` est limitée au `WHERE` et au `CASE`.

Par ailleurs, la fonction antérieure `existsNode` est obsolète depuis la version 11*g*R2.

Le tableau 11-20 présente quelques extractions avec la fonction `XMLExists`. En remplaçant `col_xml` par `OBJECT_VALUE` dans la clause `PASSING`, vous obtiendrez l'expression XQuery pour la table XMLType `t_documents_xml`.

Vous découvrirez qu'il est souvent nécessaire d'utiliser conjointement la fonction `XMLTable`. En effet, bien que les expressions *XQuery* qui vont se trouver dans le `SELECT` et dans le `WHERE` utilisent le même paramètre (dans `PASSING`), elles restent toutefois décorrélées.

Tableau 11-20 Utilisation de la fonction XMLExists

Code SQL	Résultats

`-- Test sur une date`
`SELECT XMLCAST(`
`  XMLQUERY('/compagnie/nomComp'`
`  PASSING col_xml RETURNING CONTENT)`
`  AS VARCHAR2(30)) AS nom_comp`
`FROM t_col_xml`
`WHERE **XMLEXISTS**('/compagnie`
`  [@date_crea=xs:date("2013-04-01")]'`
`  PASSING col_xml);`
	`NOM_COMP`
	`--------------------`
	`Toulouse Air`

`-- Prédicats XPAth à 2 endroits`
`SELECT XMLCAST(`
`  XMLQUERY('/compagnie`
`  [nomComp="Air Castanet"]/comp'`
`  PASSING col_xml RETURNING CONTENT)`
`  AS VARCHAR2(6)) AS comp`
`FROM t_col_xml`
`WHERE **XMLEXISTS**('/compagnie`
`  [@date_crea=xs:date("2012-09-01")]'`
`  PASSING col_xml);`
	`COMP`
	`------`
	`AC`

`-- Ne convient pas (décorrélation)`
`SELECT XMLCAST(`
`  XMLQUERY(`
`  '/compagnie/pilotes/pilote/nom'`
`  PASSING col_xml RETURNING CONTENT)`
`  AS VARCHAR2(30)) AS resultat`
`FROM t_col_xml`
`WHERE **XMLEXISTS**(`
`  '/compagnie/pilotes/pilote`
`  [@brevet="PL-25"]' PASSING col_xml)`
`AND **XMLEXISTS**(`
`  '/compagnie/pilotes/pilote/salaire`
`  [number()=5000]' PASSING col_xml);`
	`RESULTAT`
	`--------------------------`
	`GazagnesFerrageHartman`

`SELECT a3.nomp`
`FROM t_col_xml a1,`
`  XMLTABLE('/compagnie' PASSING a1.col_xml`
`  COLUMNS nomc VARCHAR2(20) PATH 'nomComp',`
`  pils XMLType PATH 'pilotes/pilote')`
`a2,`
`  XMLTABLE('pilote' PASSING a2.pils`
`  COLUMNS brev VARCHAR2(6) PATH '@brevet',`
`  nomp VARCHAR2(20) PATH 'nom',`
`  sal NUMBER PATH 'salaire') a3`
`WHERE a3.sal = 5000`
`AND a3.brev = 'PL-25';`
	`NOMP`
	`---------------`
	`Ferrage`

La fonction isSchemaValid

La méthode `isSchemaValid` est associée à un type XMLType et permet de statuer sur la validation d'un document par rapport à une grammaire enregistrée. La syntaxe de cette fonction est la suivante :

```
FUNCTION isSchemaValid(schurl IN VARCHAR2 := NULL,
                       elem IN VARCHAR2   := NULL) RETURN NUMBER
```

Cette fonction retourne 1 si l'objet est valide par rapport à sa grammaire, sinon 0. Le tableau 11-21 présente un cas d'utilisation de cette fonction en supposant que la table contient du contenu XML non contraint par une grammaire. La requête permet de vérifier parmi les documents stockés lesquels sont valides par rapport à la grammaire dont l'URL est passée en paramètre. En remplaçant `col_xml` par `OBJECT_VALUE` dans la clause `SELECT`, vous trouverez aisément les moyens d'appliquer ce raisonnement pour la table XMLType `t_documents_xml`.

Tableau 11-21 Détermination des documents invalides

Code SQL	Résultat

`SELECT t.nom_doc,`
`  t.col_xml.isSchemaValid(`
`  schurl => 'http://www.actmp.fr/compagnies.xsd',`
`  elem  => 'compagnie') AS validation`
`FROM  t_col_xml t;`
	`NOM_DOC  VALIDATION`
	`---------- ----------`
	`compagnie.xml  1`
	`compagnie2.xml  1`
	`compagnie3.xml  1`
	`vol1.xml  0`
	`vol2.xml  0`

Mises à jour

Depuis la version 12*c*, XML DB supporte et préconise l'utilisation de la syntaxe *XQuery Update*. Ainsi, les fonctions XML précédentes de mise à jour sont déclarées obsolètes, notamment `UPDATEXML`, `INSERTCHILDXML`, `INSERTXMLBEFORE`, `APPENDCHILDXML` et `DELETEXML`.

Vous trouverez la syntaxe complète de *XQuery Update* sur le site du W3C (*http://www.w3.org/TR/xquery-update-10/*). Nous allons étudier l'insertion, la suppression et la modification de fragments XML grâce à quelques exemples.

Insertion d'un fragment

Il s'agit ici d'ajouter un élément mail à un pilote en particulier.

Figure 11-11 Fragment XML à insérer

Le tableau suivant présente les deux écritures possibles pour cette mise à jour. En remplaçant `col_xml` par `OBJECT_VALUE` dans les clauses `PASSING` et `SET`, vous pourrez aisément transposer ces instructions à la table XMLType `t_documents_xml`.

Tableau 11-22 Insertion d'un fragment

Fonction SQL (obsolète)	XQuery Update

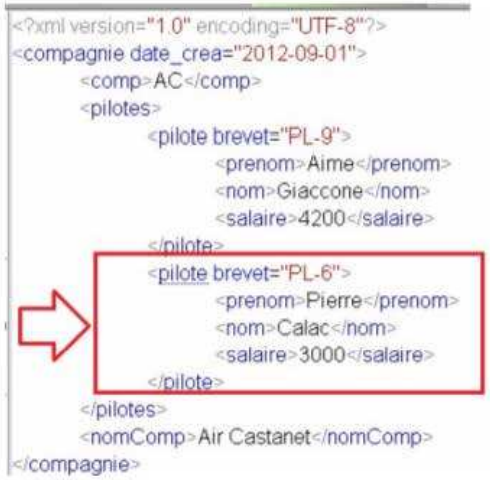
`UPDATE t_col_xml`
`SET col_xml =`
`APPENDCHILDXML(`
`col_xml,`
`'/compagnie/pilotes/pilote`
`[@brevet="PL-2"]',`
`XMLType(`
`'<mail>r.benech@actmp.fr</mail>'))`
`WHERE nom_doc = 'compagnie.xml';`
	`UPDATE t_col_xml`
	`SET col_xml =`
	`XMLQuery('copy $tmp := . modify`
	`(for $i in`
	`$tmp/compagnie/pilotes/pilote`
	`where $i/@brevet = ''PL-2''`
	`return insert node`
	`<mail>r.benech@actmp.fr</mail>`
	`as last into $i)`
	`return $tmp'`
	`PASSING col_xml RETURNING CONTENT)`
	`WHERE nom_doc = 'compagnie.xml';`

Une autre possibilité consiste à insérer un fragment avant ou après une expression XPath donnée. Dans cet exemple, il faudra remplacer `as last` par `before` ou `after` le cas échéant.

Suppression d'un fragment

Dans cet exemple, nous voulons supprimer un élément pilote en particulier (ici, le deuxième).

Figure 11-12 Fragment XML à supprimer



Le tableau 11-23 présente les deux écritures possibles pour cette suppression. En remplaçant `col_xml` par `OBJECT_VALUE` dans les clauses `PASSING` et `SET`, vous pourrez aisément transposer ces instructions à la table XMLType `t_documents_xml`.

Tableau 11-23 Suppression d'un fragment

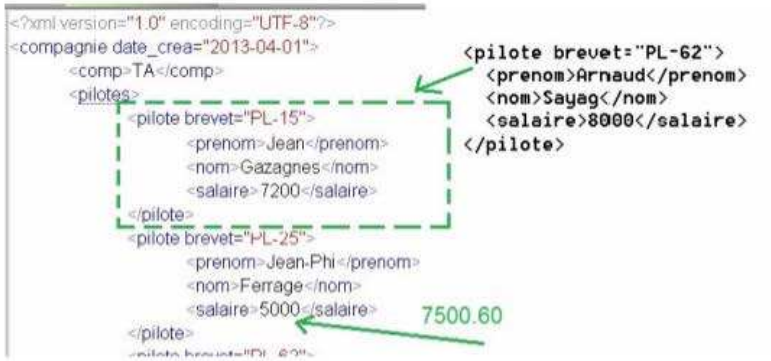
Fonction SQL (obsolète)	XQuery Update

`UPDATE t_col_xml`
`SET col_xml =`
`DELETEXML(`
`col_xml,`
`'/compagnie/pilotes/pilote`
`[position()=2]')`
`WHERE nom_doc = 'compagnie2.xml';`
	`UPDATE t_col_xml`
	`SET col_xml =`
	`XMLQuery('copy $tmp := . modify`
	`(for $i in`
	`$tmp/compagnie/pilotes/pilote`
	`[position()=2]`
	`return delete node $i)`
	`return $tmp'`
	`PASSING col_xml RETURNING CONTENT)`
	`WHERE nom_doc = 'compagnie2.xml';`

Modification d'un fragment

Il s'agit de modifier un élément d'un pilote en particulier (ici, le salaire). Le remplacement d'un fragment (ici, le premier pilote) sera également étudié.

Figure 11-13 *Fragment XML à modifier*



Le tableau 11-24 présente les deux écritures possibles pour la première modification. En remplaçant `col_xml` par `OBJECT_VALUE` dans les clauses `PASSING` et `SET`, vous pourrez facilement transposer ces instructions à la table XMLType `t_documents_xml`.

Tableau 11-24 Modification d'un fragment

Fonction SQL (obsolète)	XQuery Update

`UPDATE t_col_xml`
`SET col_xml =`
`  UPDATEXML(col_xml,`
`  '/compagnie/pilotes/pilote`
`  [@brevet="PL-25"]/salaire/text()',`
`  '7500.60')`
`WHERE nom_doc = 'compagnie3.xml';`
	`UPDATE t_col_xml`
	`SET col_xml =`
	`  XMLQuery('copy $tmp := . modify`
	`  (for $i in`
	`  $tmp/compagnie/pilotes/pilote`
	`  [@brevet="PL-25"]/salaire/text()`
	`  return replace value of node $i`
	`  with ''7500.60'')`
	`  return $tmp'`
	`  PASSING col_xml RETURNING CONTENT)`
	`WHERE nom_doc = 'compagnie3.xml';`

Le code qui permet de remplacer un fragment plus complexe est présenté dans le tableau 11-25. Par la même occasion, vous découvrirez l'utilisation de deux expressions dans la clause `PASSING`. La première désigne le document XML à modifier, la seconde le nouveau fragment.

Tableau 11-25 Remplacement d'un fragment

XQuery Update	Commentaires

`DECLARE`
`var_xml XMLTYPE;`
`BEGIN`
`var_xml :=XMLType('<pilote brevet="PL-62">`
`<prenom>Arnaud</prenom>`
`<nom>Sayag</nom>`
`<salaire>8000</salaire>`
`</pilote>');`
`UPDATE t_col_xml`
`SET col_xml =`
`XMLQuery('copy $tmp := $p1 modify`
`(for $j in $tmp/compagnie/pilotes/pilote`
`where $j/@brevet = ''PL-15''`
`return replace node $j with $p2)`
`return $tmp'`
`PASSING col_xml AS "p1", var_xml AS "p2"`
`RETURNING CONTENT)`
`WHERE nom_doc = 'compagnie3.xml';`
`END;`
	Variable PL qui contient le nouveau fragment.
	
	Remplacement du fragment courant (désigné par `$j`) par le nouveau.

Indexation

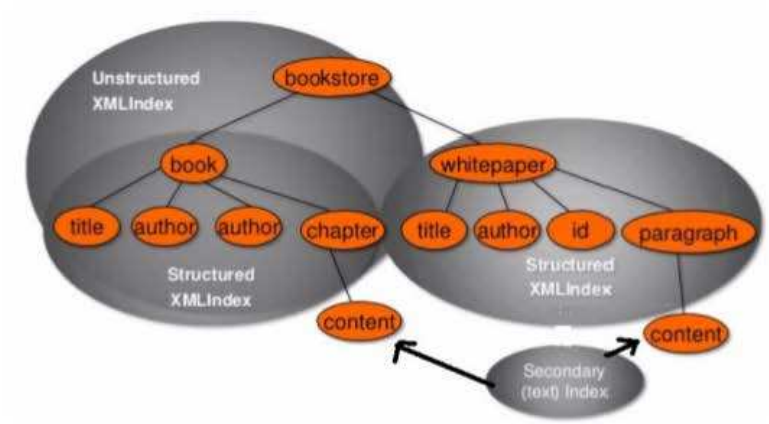
Suivant le mode de stockage choisi, vous avez plusieurs possibilités d'indexation :

- mode de stockage *binary XML* : XMLIndex (*domain index*), index textuels et index *B-tree* (sur des colonnes virtuelles) ;
- mode de stockage objet : index textuels et index *B-tree* (sur toute colonne ou expression).

Depuis la version 12*c*, les index basés sur la fonction `extractValue` (*function based index*) ne doivent plus être utilisés (`CREATE INDEX... ON nom_table(extractValue(…))`).

La figure 11-14 (*http://fr.slideshare.net/MGralike*) résume les possibilités d'indexation de contenu XML disponibles depuis la version 11gR2 d'Oracle XML DB.

Figure 11-14 Modes d'indexation XML



Index B-tree

Pour indexer les éléments du premier niveau `comp`, `nomComp` et l'attribut `date_crea`, trois index seront nécessaires. Pour l'élément `brevet` et l'attribut `nom`, qui sont tous deux du deuxième niveau (celui de la collection nommée `pilotes`), deux index seront nécessaires.

Figure 11-15 Exemple d'indexation

```
<?xml version="1.0" encoding="UTF-8"?>
<compagnie date_crea ="2012-09-01">
        <comp>AC</comp>
        <pilotes>
                <pilote brevet = "PL-9">
                        <prenom>Aime</prenom>
                        <nom>Giaccone</nom>
                        <salaire>4200</salaire>
                </pilote>
                <pilote brevet="PL-6">
                        <prenom>Pierre</prenom>
                        <nom>Calac</nom>
                        <salaire>3000</salaire>
                </pilote>
        </pilotes>
        <nomComp>Air Castanet</nomComp>
</compagnie>
```

Le tableau 11-26 présente la création de ces index, que ce soit pour une table XMLType ou une colonne XMLType.

Tableau 11-26 Création d'index conventionnels

Syntaxe SQL	Commentaires

`CREATE UNIQUE INDEX idx_comp_xml_col`
`  ON t_documents_xml`
`  (CAST ("XMLDATA"."COMP" AS VARCHAR2(6)));`
 	
`CREATE UNIQUE INDEX idx_comp_xml`
`  ON t_col_xml`
`  (CAST (col_xml."XMLDATA"."COMP" AS VARCHAR2(6)));`
	Si vous avez annoté la grammaire, respectez la casse des noms des colonnes.

`CREATE INDEX idx_date_crea_xml ON t_documents_xml`
`(CAST ("XMLDATA"."DATE_CREA" AS DATE));`
 	
`CREATE INDEX idx_date_crea_xml_col ON t_col_xml`
`(CAST (col_xml."XMLDATA"."DATE_CREA" AS DATE));`
	Même écriture, que ce soit un attribut ou un élément, respectez la casse des noms des colonnes de la grammaire.

`CREATE UNIQUE INDEX idx_nomcomp_xml`
`  ON t_documents_xml`
`  (XMLCast(`
`  XMLQuery('/compagnie/nomComp'`
`  PASSING OBJECT_VALUE RETURNING CONTENT)`
`  AS VARCHAR2(20)));`
 	
`CREATE UNIQUE INDEX idx_nomcomp_xml_col`
`  ON t_col_xml`
`  (XMLCast(`
`  XMLQuery('/compagnie/nomComp'`
`  PASSING col_xml RETURNING CONTENT)`
`  AS VARCHAR2(20)));`
	Autre type d'écriture (depuis la version 11 qui n'utilise pas les colonnes annotées).

`CREATE INDEX idx_coll_brevet_xml`
`  ON pilote_col_table p`
`  (p."BREVET", p.NESTED_TABLE_ID);`
 	
`CREATE INDEX idx_coll_nompil_xml`
`  ON pilote_col_table p`
`  (p."NOM", p.NESTED_TABLE_ID);`
	Utilisation du nom de la table de stockage.

Les index des informations de premier niveau seront définis en interne `FUNCTION-BASED NORMAL`. Ceux concernant les données d'une collection le seront en interne `NORMAL`.

Mode non structuré (Unstructured XMLIndex)

Le type d'index présenté ici (*domain index*) convient au mode de stockage *binary XML* et aux contenus XML qui sont orientés document (figure 11-2). Un seul index de cette nature peut être défini sur une table (ou une colonne) XMLType *binary XML*.

Le tableau 11-27 décrit la création d'un XMLIndex non structuré sur une colonne XMLType. Ce type d'index génère une table (*path table*) et des index sur cette table qui auront pour but de référencer les différentes parties des documents XML et les valeurs terminales (éléments et attributs).

Tableau 11-27 Création d'index XMLType non structuré

Syntaxe SQL	Commentaires

`CREATE INDEX compagnie_xmlindex ON t_col_xml(col_xml)`
`INDEXTYPE IS XDB.XMLINDEX`
`PARAMETERS`
`('PATH TABLE compagnie_path_table`
`(TABLESPACE USERS)`
`PATH ID INDEX compagnie_idx`
`(TABLESPACE USERS)`
`ORDER KEY INDEX compagnie_ok_idx`
`(TABLESPACE USERS)`
`VALUE INDEX compagnie_value_idx`
`(TABLESPACE USERS)');`
	La *path table* répertorie les chemins.
	Le *path index* identifie des fragments.
	L'*order index* recense la position hiérarchique des nœuds.
	Le *value index* adresse les valeurs des éléments terminaux et des attributs.

```
SQL> SELECT index_name, index_type, table_name
     FROM   user_indexes;

INDEX_NAME                INDEX_TYPE                TABLE_NAME
------------------------- ------------------------- --------------------
COMPAGNIE_XMLINDEX        FUNCTION-BASED DOMAIN     T_COL_XML
COMPAGNIE_VALUE_IDX       NORMAL                    COMPAGNIE_PATH_TABLE
COMPAGNIE_OK_IDX          NORMAL                    COMPAGNIE_PATH_TABLE
COMPAGNIE_IDX             NORMAL                    COMPAGNIE_PATH_TABLE
```

Des index secondaires additionnels (conventionnels, *function-based* ou textuels) peuvent être ajoutés à la *path table* : `CREATE INDEX ... ON nom_path_table...` Comme pour les tables de stockage des collections, il n'est pas possible d'exécuter une requête directement sur ce type de table.

La clause `PARAMETERS` peut contenir 1 000 caractères au plus. Au-delà, vous devrez utiliser les procédures `registerParameter` et `modifyParameter` du paquetage `DBMS_XMLINDEX`. Grâce à cette clause ou à ces procédures, vous pouvez préciser les fragments à privilégier ou à exclure avec `ALTER INDEX ... INCLUDE` ou `EXCLUDE...` avec `ADD` ou `REMOVE`.

Mode structuré (Structured XMLIndex)

Depuis la version 11*g*R2, il est possible de composer un nouveau type d'index destiné à du contenu XML qui est principalement de type « orienté document », mais dont une partie est toutefois fortement structurée. Un seul index de cette nature peut être défini sur une table (ou une colonne) XMLType *binary XML*. La clause GROUP dans PARAMETERS permet de définir un tel index.

Le tableau 11-28 décrit la création d'un XMLIndex structuré sur une colonne XMLType. Ici, on indexe la collection et on retrouve les directives COLUMNS et PATH de la fonction XMLTable précédemment étudiée. Plusieurs groupes peuvent exister dans un même index mais une seule colonne VIRTUAL est utilisée (pour une seule collection). Enfin, il est possible d'ajouter des groupes ou d'en enlever avec ALTER INDEX... et ADD_GROUP ou DROP_GROUP. Ce type d'index génère autant de tables que nécessaire et d'index qui auront pour but de référencer les différentes parties des documents XML et les valeurs terminales (éléments et attributs).

Tableau 11-28 Création d'index XMLType non structuré

Syntaxe SQL	Commentaires
CREATE INDEX compagnie_xmlindex ON t_col_xml(col_xml) INDEXTYPE IS **XDB.XMLINDEX** **PARAMETERS** ('**GROUP** grp_pilote XMLTable tab_compa_xml (TABLESPACE USERS COMPRESS FOR OLTP) ''/compagnie'' COLUMNS lineitem XMLType PATH ''pilotes/pilote'' VIRTUAL XMLTable tab_comp_pilote_xml (TABLESPACE USERS COMPRESS FOR OLTP) ''/pilote'' PASSING lineitem COLUMNS numpil VARCHAR2(6) PATH ''@brevet'', sal NUMBER(9) PATH ''salaire'', nompil VARCHAR2(20) PATH ''nom'');	Le premier groupe est destiné aux éléments présents dans la collection.
ALTER INDEX compagnie_xmlindex **PARAMETERS**('**ADD_GROUP GROUP** grp_compagnie XMLTable tab_comp_xml ''/compagnie'' COLUMNS date_crea DATE PATH ''@date_crea'', comp VARCHAR2(6) PATH ''comp'', nomc VARCHAR2(30) PATH ''nomComp'');	Le second groupe est destiné aux éléments (et l'attribut) de premier niveau du document.

```
SQL> SELECT index_name, index_type, table_name
    FROM   user_indexes;

INDEX_NAME                INDEX_TYPE                TABLE_NAME
------------------------- ------------------------- ----------------------
COMPAGNIE_XMLINDEX        FUNCTION-BASED DOMAIN     T_COL_XML
SYS93644_93645_KEY_IDX    NORMAL                    TAB_COMPA_XML
SYS93644_93645_RID_IDX    NORMAL                    TAB_COMPA_XML
SYS93644_93648_RID_IDX    NORMAL                    TAB_COMP_PILOTE_XML
SYS93644_93648_PKY_IDX    NORMAL                    TAB_COMP_PILOTE_XML
SYS93644_93651_RID_IDX    NORMAL                    TAB_COMP_XML
```

Mode mixte

Depuis la version 11gR2, il est possible de mixer un index XMLIndex avec des composants structurés et un autre non structuré (un au maximum). Pour supprimer ce dernier, il faut agir au niveau de l'index par `ALTER INDEX... PARAMETERS('DROP PATH TABLE')`. Par défaut, tous les fragments indexés inclus dans la partie structurée de l'index sont aussi indexés dans celle non structurée. Pour exclure des fragments de la partie non structurée, il faut utiliser l'option `PATHS... EXCLUDE` dans la clause `PARAMETERS`.

Le tableau 11-29 décrit la création d'un XMLIndex mixte sur une colonne XMLType. Tout d'abord, la collection est exclue de l'indexation (de même que le nom de la compagnie). Puis des éléments de la collection sont ajoutés à l'index.

Tableau 11-29 Création d'index XMLType mixte

Syntaxe SQL	Commentaires

`CREATE INDEX compagnie_xmlindex`
`ON t_col_xml(col_xml)`
`INDEXTYPE IS XDB.XMLINDEX`
`PARAMETERS ('PATH TABLE compagnie_path_table`
`PATHS (EXCLUDE (`
`/compagnie/pilotes`
`/compagnie/nomComp))');`
	Création de l'index non structuré qui exclut deux fragments (dont la collection).

`BEGIN`
`DBMS_XMLINDEX.registerParameter`
`('param_compagnie',`
`'ADD_GROUP GROUP grp_pilote`
`XMLTable tab_compa_xml ''/compagnie''`
`COLUMNS lineitem XMLType`
`PATH ''pilotes/pilote'' VIRTUAL`
`XMLTable tab_comp_pilotes_xml ''/pilote''`
`PASSING lineitem`
`COLUMNS numpil VARCHAR2(6) PATH ''@brevet'',`
`nompil VARCHAR2(20) PATH ''nom''');`
`END;`
`/`
	Création d'un groupe pour inclure dans l'index deux éléments de la collection.

`ALTER INDEX compagnie_xmlindex` `PARAMETERS('PARAM param_compagnie');`	Ajout de ce groupe structuré à l'index non structuré.

Génération de contenus

Plusieurs mécanismes permettent de générer du contenu XML à partir de données provenant de tables relationnelles.

- Des fonctions SQL/XML (ANSI/ISO). Citons principalement `XMLELEMENT` (pour créer un élément), `XMLATTRIBUTES` (pour ajouter un attribut à un élément), `XMLFOREST` (pour créer un fragment), `XMLAGG` (pour peupler une collection) et `XMLCOMMENT` (pour ajouter un commentaire).
- Des fonctions Oracle. `XMLROOT` (pour générer un prologue), `XMLCOLATTVAL` (pour générer un triplet {élément, attribut, valeur}), `XMLCDATA` (pour générer une section à ne pas *parser*) et `XMLSerialize` pour la mise en forme.
- Le paquetage `DBMS_XMLGEN` qui fournit des fonctions et procédures pour convertir des requêtes SQL en contenu XML préformaté (voir la section « Les paquetages pour PL/SQL »).

Afin de présenter ces différentes fonctionnalités, considérons les données issues des trois tables suivantes.

Figure 11-16 *Données relationnelles*

COMPAGNIE_R

CODEC	NOM_COMP
AF	Air France
EJ	Easy Jet
AB	Air Blagnac

AVION_R

NA	TYPAV	CAPACITE	PROPRIO
F-GODF	A320	170	AB
F-PROG	A318	140	AF
F-HGFT	A319	120	AF
F-WOWW	A380	490	EJ

AFFRETER_R

NA	CODEC	DATE_A	NB_PASSAGERS
F-GODF	EJ	12/08/14	120
F-GODF	AF	12/08/14	150
F-PROG	AF	12/08/14	130
F-PROG	AF	12/09/14	110
F-WOWW	AB	12/09/14	450

Les fonctions SQL/XML

Le code suivant décrit la génération d'une arborescence décrivant les affrètements de la compagnie 'AF' ordonnés par capacité décroissante. Notez la possibilité de trier un agrégat d'éléments construit avec `XMLAGG`. Le résultat brut (figure 11-17) ne contient pas encore de mise en forme (utilisation conjointe de `XMLSerialize`, voir plus loin).

Tableau 11-30 Génération de contenu XML par fonctions SQL ANSI/ISO

Code SQL	Commentaires

`SELECT XMLElement(NAME "flotte",`
`  XMLAttributes(c.codec AS "comp"),`
`  XMLElement(NAME "nomcomp", c.nom_comp),`
`  XMLElement(NAME "avions",`
`  (SELECT XMLAgg(`
`  XMLElement(NAME "avion",`
`  XMLAttributes(av.na AS "immat"),`
`  XMLForest(av.typav AS "type_avi",`
`  av.capacite AS "nb_pax"))`
`  ORDER BY av.capacite DESC)`
`  FROM avion_R av`
`  WHERE av.proprio = c.codec))`
`  )`
`FROM compagnie_R c`
`WHERE c.codec = 'AF';`
	Création d'un élément `flotte` contenant un attribut `comp`, puis composé de deux éléments (`nomcomp` et `avions`).
	
	Peuplement de la collection à l'aide d'une jointure.

Figure 11-17 Sortie brute

```
XMLELEMENT(NAME"FLOTTE",XMLATTRIBUTES(C.CODECAS"COMP"),XMLELEMENT(NAME"NOMCOMP",
--------------------------------------------------------------------------------
<flotte comp="AF"><nomcomp>Air France</nomcomp><avions><avion immat="F-PROG"><ty
pe_avi>A318</type_avi><nb_pax>140</nb_pax></avion><avion immat="F-HGFT"><type_av
i>A319</type_avi><nb_pax>120</nb_pax></avion></avions></flotte>
```

Le résultat remis en forme est présenté à la figure 11-18 ; il ne contient pas encore de prologue (`XMLROOT`), ni de commentaires (utilisation conjointe de `XMLCOMMENT`).

Figure 11-18 Contenu XML mis en forme

```
<flotte comp="AF">
    <nomcomp>Air France</nomcomp>
    <avions>
        <avion immat="F-PROG">
            <type_avi>A318</type_avi>
            <nb_pax>140</nb_pax>
        </avion>
        <avion immat="F-HGFT">
            <type_avi>A319</type_avi>
            <nb_pax>120</nb_pax>
        </avion>
    </avions>
</flotte>
```

Une autre possibilité est d'utiliser un type objet à qui on donne la structure de l'élément souhaité. Notez l'utilisation de la fonction XMLAGG car plusieurs lignes seront produites, et des doubles guillemets sur les colonnes du type objet pour préserver la casse des noms d'éléments et d'attributs (si le caractère @ est mis en préfixe). Le résultat produit n'est pas ordonné et subit l'ordre des lignes et des blocs de la table relationnelle.

Tableau 11-31 Génération de contenu XML à l'aide d'un type

Code SQL	Commentaires

`CREATE OR REPLACE TYPE aircraf_t AS OBJECT`
`  ("@immat"  VARCHAR2(6),`
`   "type_av" VARCHAR2(6),`
`   "nb_p"    NUMBER(3),`
`   "comp"    VARCHAR2(6));`
`/`
	Création d'un type composé de quatre champs, le premier étant prédestiné à devenir un attribut, les trois autres des éléments.

`SELECT XMLElement(NAME "avions",`
`  (SELECT XMLAgg(`
`          XMLForest(`
`             aircraf_t(na,typav,capacite,proprio)`
`          AS "avion")`
`               )`
`    FROM  avion_R av))`
`FROM DUAL;`
	Utilisation du constructeur (nom du type) pour instancier à chaque ligne retournée par la table des avions un élément préstructuré.

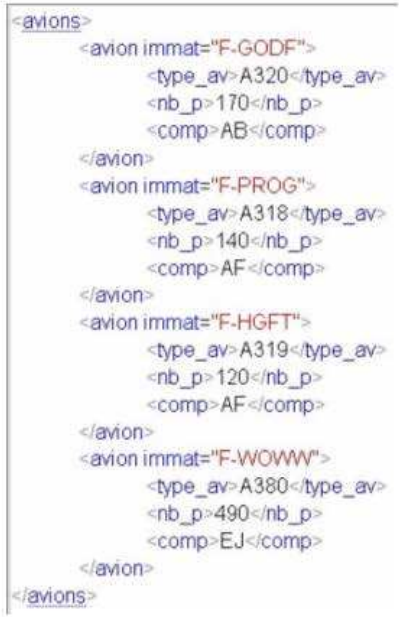
Figure 11-19 Sortie brute

```
XMLELEMENT("AVIONS",(SELECTXMLAGG(XMLFOREST(AIRCRAF_T(NA,TYPAV,CAPACITE,PROPRIO)
--------------------------------------------------------------------------------
<avions><avion immat="F-GODF"><type_av>A320</type_av><nb_p>170</nb_p><comp>AB</c
omp></avion><avion immat="F-PROG"><type_av>A318</type_av><nb_p>140</nb_p><comp>A
F</comp></avion><avion immat="F-HGFT"><type_av>A319</type_av><nb_p>120</nb_p><co
mp>AF</comp></avion><avion immat="F-WOWW"><type_av>A380</type_av><nb_p>490</nb_p
><comp>EJ</comp></avion></avions>
```

Le résultat remis en forme est présenté à la figure 11-20 ; il ne contient ni prologue ni commentaires.

Pour trier des éléments en amont, vous devrez utiliser l'ordonnancement au niveau de XMLAgg et non du SELECT de la table. Ici, vous devrez adopter une écriture de ce type : `XMLAgg(XMLForest(...) ORDER BY...)`.

Figure 11-20 Contenu XML mis en forme



Conversions et analyse

La possibilité de convertir des caractères en contenu XML, et inversement, est intéressante pour la mise en forme de résultats. Nous allons étudier la fonction `XMLSerialize` qui transforme du contenu XML en CLOB, BLOB ou chaîne, et la fonction `XMLParse` qui analyse une chaîne pour retourner une instance XMLType.

Les options de la fonction `XMLSerialize` sont les suivantes :

```
XMLSERIALIZE
  ( { DOCUMENT | CONTENT } expression [ AS type_SQL ]
    [ ENCODING xml_encoding_spec ]
    [ VERSION chaine ]
    [ NO INDENT | { INDENT [ SIZE = nombre ] } ]
    [ { HIDE | SHOW } DEFAULTS ] )
```

- L'option DOCUMENT impose que le contenu de l'expression soit un document XML bien formé. Avec CONTENT, le contenu XML peut ne pas avoir de racine unique, mais il doit être par ailleurs bien formé.
- Le type SQL concerne les caractères (VARCHAR2 ou VARCHAR mais pas NVARCHAR2) ou les binaires (BLOB et CLOB, qui est le type par défaut). Pour les BLOB, la clause ENCODING permet d'enrichir le prologue (encoding="...").
- La clause VERSION concerne le prologue (<?xml version="..." ...?>).
- L'indentation entre les éléments est déterminée par l'option INDENT SIZE = *n*. Si la clause INDENT est présente sans le paramètre SIZE, l'indentation est fixée à 2. SIZE=0 entraîne une séparation de tous les éléments (un par ligne et sans indentation).
- HIDE DEFAULTS et SHOW DEFAULTS s'appliquent seulement à des documents basés sur des grammaires *XML schema* et concernent les éventuelles erreurs.

Le code suivant décrit deux sérialisations. La première extrait une ligne d'une table à partir d'un type et la transforme en chaîne de caractères pour l'état de sortie du document indenté. La seconde extrait une instance XMLType à partir d'une colonne d'une table et la transforme en CLOB pour déterminer la taille en nombre de caractères du document après l'indentation par défaut.

Tableau 11-32 Sérialisations de contenu

Code SQL	Résultat
SELECT XMLSerialize(DOCUMENT XMLElement(NAME "avions", (SELECT XMLAgg(XMLForest(aircraf_t(na,typav,capacite,proprio) AS "avion")) FROM avion_R WHERE capacite > 300)) AS VARCHAR2(400) INDENT SIZE=1) AS resultat FROM DUAL;	RESULTAT ---------- <avions> <avion immat="F-WOWW"> <type_av>A380</type_av> <nb_p>490</nb_p> <comp>EJ</comp> </avion> </avions>
DECLARE var_CLOB CLOB; taille NUMBER; BEGIN SELECT XMLSerialize(DOCUMENT col_xml AS CLOB) INTO var_CLOB FROM t_col_xml WHERE XMLEXISTS('/compagnie/comp[text()="AC"]' PASSING BY VALUE col_xml); taille:= DBMS_LOB.GETLENGTH(var_CLOB); DBMS_OUTPUT.PUT_LINE('taille CLOB :' || taille); END;	taille CLOB :420 Procédure PL/SQL terminée avec succès.

Les options de la fonction XMLParse sont les suivantes :

```
XMLPARSE({DOCUMENT | CONTENT} expression [WELLFORMED])
```

- Les options DOCUMENT et CONTENT ont la même signification que pour XMLSerialize.
- L'option WELLFORMED évite de vérifier la validité du document XML en la garantissant en amont.

Le code suivant décrit trois analyses (*parsing*). La première est correcte du fait d'éléments bien formés. La deuxième retourne une erreur car des éléments sont mal formés. Et la dernière réutilise la mauvaise expression mais évite le contrôle d'Oracle. La fonction getClobVal transforme une instance XMLType en CLOB.

Tableau 11-33 Analyses de contenu

Code SQL	Résultat

`SELECT XMLParse(CONTENT`
`'<vehicule immat="508-BAX-31"/>`
`<km>234567</km><vehicule/>') AS fragment`
`FROM DUAL;`
	`FRAGMENT`
	`--------------------------`
	`<vehicule immat="508-BAX-31"/>`
	`<km>234567</km><vehicule/>`

`SELECT XMLParse(CONTENT`
`'AF<vehicule immat="508-BAX-31"><km>245647</km><vehicule>')`
`FROM DUAL;`
	`ORA-31011: Échec d'analyse XML`
	`ORA-19213: une erreur s'est produite lors du traitement XML aux lignes 1`
	`LPX-00225: la balise de fin d'élément "DummyFragment" ne concorde pas avec balise de début d'élément "vehicule"`

`DECLARE`
` var_string VARCHAR2(300) :=`
`  'AF<vehicule immat="508-BAX-31">`
`  <km>245647</km><vehicule>';`
` var_xml XMLTYPE;`
`BEGIN`
` SELECT XMLParse(`
`  CONTENT var_string WELLFORMED)`
`  INTO var_xml FROM DUAL;`
` DBMS_OUTPUT.PUT_LINE(`
`  'contenu mal formé...');`
`  DBMS_OUTPUT.PUT_LINE(`
`  var_xml.getClobVal());`
`END;`
	`contenu mal formé...`
	`AF<vehicule immat="508-BAX-31">`
	`<km>245647</km><vehicule>`
	
	`Procédure PL/SQL terminée avec succès.`

Les fonctions d'Oracle

Le tableau 11-34 présente les trois fonctions natives qui complètent les fonctions précédentes en enrichissant du contenu XML.

- XMLROOT pour générer le prologue ;
- XMLCOLATTVAL pour générer des triplets (élément, attribut, valeur) ;
- XMLCDATA pour générer une section à ne pas analyser par le *parser*.

Tableau 11-34 Fonctions XML propres à Oracle

Code SQL	Résultat

`SELECT XMLRoot(XMLElement(NAME "avions",`
`  (SELECT XMLAgg(XMLForest`
`    (aircraf_t(na,typav,capacite,proprio)`
`     AS "avion")`
`     ORDER BY capacite DESC)`
`   FROM avion_R WHERE typav = 'A320')),`
`VERSION NO VALUE, STANDALONE YES)`
`FROM DUAL;`
	`<?xml version="1.0" standalone="yes"?>`
	`<avions>`
	`  <avion immat="F-GODF">`
	`  <type_av>A320</type_av>`
	`  <nb_p>170</nb_p>`
	`  <comp>AB</comp>`
	`  </avion>`
	`</avions>`

`SELECT XMLSerialize(DOCUMENT`
`  (SELECT XMLElement(NAME "aircraft",`
`  XMLAttributes(na AS "immat"),`
`  XMLColAttVal(`
`  typav AS "type_aircraft",`
`  capacite AS "pax_num",`
`  proprio AS "compagnie"))`
`  FROM avion_R WHERE typav = 'A318')`
`  AS CLOB INDENT) AS "doc_xml"`
`FROM DUAL;`
	`doc_xml`
	`----------`
	`<aircraft immat="F-PROG">`
	`  <column name="type_aircraft">A318</column>`
	`  <column name="pax_num">140</column>`
	`  <column name="compagnie">AF</column>`
	`</aircraft>`

`SELECT XMLSerialize(DOCUMENT`
`(SELECT XMLELEMENT(NAME "compagnies",`
`  XMLAGG(`
`  XMLELEMENT("compagnie",`
`  XMLFOREST(codec AS "comp",`
`  XMLCDATA(nom_comp || '->'`
`  || 'père & fils.') AS "nom_comp"))`
`  )`
`  ) AS resultat`
`  FROM compagnie_R WHERE codec LIKE 'A%')`
`AS CLOB INDENT) AS "doc_xml"`
`FROM DUAL;`
	`RESULTAT`
	`----------`
	`<compagnies>`
	`  <compagnie>`
	`  <comp>AB</comp>`
	`  <nom_comp><![CDATA[Air Blagnac-`
	`>père & fils.]]></nom_comp>`
	`  </compagnie>`
	`  <compagnie>`
	`  <comp>AF</comp>`
	`  <nom_comp><![CDATA[Air France->`
	`père & fils.]]></nom_comp>`
	`  </compagnie>`
	`</compagnies>`

Les vues

Plusieurs types de vues peuvent être utilisés dans un contexte de manipulation de documents XML.

- Les vues relationnelles fournissent un accès classique (tabulaire) à du contenu qui est stocké en base sous la forme XML.
- Les vues XMLType fournissent un contenu XML à des données conventionnelles stockées dans des tables relationnelles ou à des données plus structurées (tables objet-relationnelles). Ces vues XMLType peuvent être contraintes par une grammaire (enregistrée au préalable).
- Les vues matérialisées (voir le chapitre 12) qui rendent persistant le résultat d'une requête (sans jointure) manipulant une table (ou colonne) XMLType.

De plus, il est possible de bénéficier du mécanisme d'indexation sur les vues relationnelles qui concernent un mode de stockage *binary XML* et sur les vues matérialisées.

Vues relationnelles

Le mécanisme de vue relationnelle permet de présenter sous une forme tabulaire du contenu XML le plus souvent stocké en base (ou provenant de fichiers externes). La requête de définition de ce type de vue fait intervenir la fonction `XMLTable` utilisée conjointement à la clause `COLUMNS` pour définir une correspondance entre les colonnes de la vue et les éléments (ou attributs) des documents XML. La syntaxe simplifiée d'une telle vue est la suivante. Il s'agit d'extraire une ligne par document XML ou d'« aplatir » les collections à l'aide de jointures.

```
CREATE [OR REPLACE] VIEW [nom_schema.]nom_vue[(alias_col[, alias_
col]...)]
AS SELECT ... FROM ... ,
    XMLTABLE('/.../...' PASSING ...
            COLUMNS col type_SQL  PATH 'chemin',...) alias,…
```

Le code suivant déclare une vue tabulaire qui extrait une partie de trois documents XML qui modélisent chacun une compagnie. En remplaçant `col_xml` par `OBJECT_VALUE` dans la clause `PASSING`, vous obtiendrez l'expression XQuery qui convient pour interroger la table XMLType `t_documents_xml` via la vue.

Tableau 11-35 Vue relationnelle de contenus XML

Code SQL	Résultat

`CREATE VIEW comp_master_vue AS`
`SELECT a.*`
`FROM t_col_xml,`
`  XMLTable('/compagnie'`
`  PASSING col_xml`
`  COLUMNS code_c VARCHAR2(6)`
`  PATH 'comp',`
`  nom_c VARCHAR2(30)`
`  PATH 'nomComp',`
`  date_c DATE`
`  PATH '@date_crea') a;`
	`SQL> SELECT * FROM comp_master_vue;`
	
	`CODE_C NOM_C  DATE_C`
	`------ ---------- --------`
	`AB  Air Blagnac  30/08/10`
	`AC  Air Castanet  01/09/12`
	`TA  Toulouse Air  01/04/13`

Le code suivant crée la vue qui est déduite de la collection. La clause FROM référence trois sources : la première concerne les documents XML, la deuxième (virtuelle) compose les colonnes du premier niveau (code de chaque compagnie), la troisième (virtuelle) définit les éléments de chaque collection (pilotes). Une fois ces deux vues créées, les développeurs SQL seront capables d'extraire classiquement tout renseignement issu de ces données désormais normalisées.

Tableau 11-36 Vue tabulaire d'une collection XML

Code SQL	Résultat

```
CREATE VIEW comp_detail_vue AS
SELECT a.comp, b.*
FROM   t_col_xml,
       XMLTABLE('/compagnie'
        PASSING col_xml
        COLUMNS comp VARCHAR2(6)
                PATH 'comp',
        pils XMLType
            PATH 'pilotes/pilote') a,
       XMLTABLE('pilote'
         PASSING a.pils
         COLUMNS brevet VARCHAR2(6)
                 PATH '@brevet',
                 nompil VARCHAR2(20)
                 PATH 'nom',
                 salaire NUMBER
                 PATH 'salaire') b;
```

```
SQL> ALTER SESSION SET
     nls_numeric_characters = ".,";

SQL> SELECT * FROM comp_detail_vue ;

COMP   BREVET NOMPIL           SALAIRE
------ ------ ------------ ----------
AB     PL-1   Sarda              4000.2
AB     PL-2   Benech             5000.4
AC     PL-9   Giaccone             4200
AC     PL-6   Calac                3000
TA     PL-15  Gazagnes             7200
TA     PL-25  Ferrage              5000
TA     PL-62  Hartman              5000
```

Pour bénéficier du mécanisme d'indexation, il faut que le mode de stockage des documents soit *binary XML*. Les deux étapes suivantes sont nécessaires : vous enregistrez tout d'abord la définition de la vue au niveau d'un paramètre d'un index XMLIndex (*domain index*) structuré, puis vous créez l'index sur la table (ou la colonne) XMLType. Le tableau 11-37 décrit l'indexation qui bénéficiera à la vue des compagnies.

Tableau 11-37 Création d'un XMLIndex pour une vue tabulaire

Code SQL	Commentaires

```
BEGIN
 DBMS_XMLINDEX.registerParameter(
    'param_vue_comp_master',
    DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView(
                           'COMP_MASTER_VUE'));
END;
/
CREATE INDEX idx_comp on t_col_xml
     (col_xml) INDEXTYPE IS XDB.XMLIndex
      PARAMETERS ('PARAM param_vue_comp_master');
```

Enregistrement du paramètre pour l'index XMLType structuré.

Création de l'index avec le paramètre de la définition de la vue.

Vues XMLType

Les vues XMLType simulent la gestion de contenu XML de données qui sont stockées dans des tables relationnelles (ou objet-relationnelles). Par ailleurs, il est possible d'associer une grammaire enregistrée à votre vue pour la contraindre davantage. Enfin, comme pour les vues objet, un OID (identifiant chaque ligne de la vue) peut être créé.

Sans grammaire (non-schema based)

Le tableau 11-38 présente la création d'une vue XMLType qui fusionne des données des trois tables relationnelles précédentes. La vue inclut un identifiant objet (ici basé sur le code de la compagnie, par exemple).

Tableau 11-38 Vue XMLType

Création de la vue	Commentaires

`CREATE OR REPLACE VIEW compagnie_vue_xml OF XMLType`
`WITH OBJECT IDENTIFIER`
`  (XMLCast(XMLQuery('/compagnie/@comp'`
`          PASSING OBJECT_VALUE RETURNING CONTENT)`
`   AS VARCHAR2(6)))`
`AS SELECT XMLRoot(`
`       XMLElement`
`       (NAME "compagnie", XMLAttributes(c.codec AS "comp"),`
`        XMLForest(c.nom_comp AS "nom_comp"),`
`         XMLElement( "vols",`
`           (SELECT XMLAGG(XMLElement( "vol",`
`                   XMLForest(TO_CHAR(af.date_a,'DD-MM-YYYY')`
`                                AS "date_vol",`
`                             av.typav AS "avion",`
`                          af.nb_passagers AS "passagers")))`
`              FROM  affreter_R af, avion_R av`
`              WHERE af.codec = c.codec`
`              AND   av.na    = af.na))`
`       ),`
`       VERSION NO VALUE, STANDALONE YES)`
`FROM  compagnie_R c`
`WHERE c.codec = 'AF';`
	Définition de l'identifiant.
	
	Construction de l'arbre des éléments et des attributs.
	
	Jointure pour composer la collection.

Figure 11-21 Vue XMLType

```
<?xml version="1.0" standalone="yes"?>
<compagnie comp="AF">
        <nom_comp>Air France</nom_comp>
        <vols>
                <vol>
                        <date_vol>12-08-2014</date_vol>
                        <avion>A320</avion>
                        <passagers>150</passagers>
                </vol>
                <vol>
                        <date_vol>12-08-2014</date_vol>
                        <avion>A318</avion>
                        <passagers>130</passagers>
                </vol>
                <vol>
                        <date_vol>12-09-2014</date_vol>
                        <avion>A318</avion>
                        <passagers>110</passagers>
                </vol>
        </vols>
</compagnie>
```

Avec grammaire (schema based)

Afin d'associer une grammaire à une vue XMLType, il faut au préalable enregistrer ladite grammaire (sans forcément l'annoter). Considérons la simple grammaire caractérisant l'élément `avioncomp` et définissons une vue XMLType qui sera peuplée à partir de tous les avions de la table relationnelle.

Figure 11-22 Ligne de la vue XMLType

```
<avioncomp
xsi:noNamespaceSchemaLocation="http ://www.actmp.fr/avioncomp.xsd"
immat="[          ]"   immatriculation
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <typeav>[      ]</typeav>     type avion
        <capacite>[  ]</capacite> capacité
        <compav>
                <comp>[ ]</comp> code compagnie
                <nomcomp>[          ]</nomcomp>
        </compav>                      nom
</avioncomp>                           compagnie
```

Le code suivant présente l'enregistrement d'une grammaire qui décrit cette structure de document. Notez l'utilisation du fichier source dans l'instruction (sans passer par le chargement de celui-ci par `BFILENAME`). N'oubliez pas non plus les espaces de noms qui n'induisent des problèmes que lors d'extractions (et non à la création de la vue).

Tableau 11-39 Enregistrement de la grammaire

Code SQL	Commentaires
`BEGIN`	
`  DBMS_XMLSCHEMA.DELETESCHEMA(`	Suppression de la grammaire si elle existait avant.
`    schemaurl     => 'http://www.actmp.fr/avioncomp.xsd',`	
`    delete_option => DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);`	
`--`	
`  DBMS_XMLSCHEMA.REGISTERSCHEMA`	
`  (schemaurl => 'http://www.actmp.fr/avioncomp.xsd',`	
`   schemadoc =>`	URI de la grammaire.
`   '<?xml version="1.0" encoding="UTF-8"?>`	
`    <schema xmlns="http://www.w3.org/2001/XMLSchema"`	Description du *XML Schema*.
`     version="1.0" xmlns:xdb="http://xmlns.oracle.com/xdb">`	
`    <element name="avioncomp">`	
`    <complexType>`	
`      <sequence>`	
`        <element name="typeav" type="string"/>`	
`        <element name="capacite" type="int"/>`	
`        <element name="compav">`	
`          <complexType>`	
`            <sequence>`	
`              <element name="comp" type="string"/>`	
`              <element name="nomcomp" type="string"/>`	
`            </sequence>`	
`          </complexType>`	
`        </element>`	
`      </sequence>`	
`      <attribute name="immat" type="string"/>`	
`    </complexType>`	
`    </element>`	
`  </schema>',`	Grammaire locale et sans génération de types.
`  local     => TRUE,`	
`  gentypes  => FALSE);`	
`END;`	

Le code suivant décrit la création de la vue. Notez l'utilisation des directives `XMLSCHEMA` et `ELEMENT` qui identifient la grammaire et sa racine. L'extraction d'une ligne de la vue est présentée (et mise en forme à la main pour l'occasion). Si vous désirez formater ce résultat, vous devrez utiliser la fonction `XMLSerialize`.

Tableau 11-40 Vue XMLType associée à une grammaire

Code SQL	Interrogation de la vue

`CREATE VIEW avicomp_vue_xml OF XMLType`
`  XMLSCHEMA "http://www.actmp.fr/avioncomp.xsd"`
`  ELEMENT "avioncomp"`
`WITH OBJECT IDENTIFIER`
`(XMLCast(XMLQuery('/avioncomp/@immat'`
`  PASSING OBJECT_VALUE RETURNING CONTENT)`
` AS VARCHAR2(6)))`
`AS`
`SELECT XMLElement(NAME "avioncomp",`
`  XMLAttributes(`
`  'http://www.w3.org/2001/XMLSchema-instance'`
`  AS "xmlns:xsi",`
`  'http://www.actmp.fr/avioncomp.xsd'`
`  AS "xsi:noNamespaceSchemaLocation",`
`  av.na AS "immat"),`
`  XMLForest(av.typav AS "typeav",`
`  av.capacite AS "capacite"),`
`  XMLElement(NAME "compav",`
`  XMLForest(c.codec AS "comp",`
`  c.nom_comp AS "nomcomp")))`
`FROM avion_R av, compagnie_R c`
`WHERE av.proprio = c.codec;`
	`SQL> SELECT OBJECT_VALUE`
	`  2 FROM avicomp_vue_xml`
	`  3 WHERE XMLEXISTS(`
	`  4 '/avioncomp[@immat="F-GODF"]'`
	`  5 PASSING OBJECT_VALUE);`
	
	`OBJECT_VALUE`
	`----------`
	`<avioncomp`
	`xmlns:xsi=`
	`"http://www.w3.org/2001/XMLSchema-instance"`
	`xsi:noNamespaceSchemaLocation=`
	`"http://www.actmp.fr/avioncomp.xsd"`
	`immat="F-GODF">`
	`<typeav>A320</typeav>`
	`<capacite>170</capacite>`
	`<compav>`
	`<comp>AB</comp>`
	`<nomcomp>Air Blagnac</nomcomp>`
	`</compav>`
	`</avioncomp>`

Enfin, il est possible de créer des vues XMLType basées sur une grammaire en utilisant des types objet (ou des vues objet). Le processus consiste à :

- créer le type qui décrira la structure du document final ;
- créer et annoter une grammaire associée pour faire mieux correspondre les noms et types de colonnes souhaités ;
- enregistrer la grammaire (depuis la version 12*c*, la génération automatique n'est plus permise) ;
- créer la vue XMLType dont la requête qui interroge des tables utilisera le constructeur du type pour peupler les documents.

Les paquetages pour PL/SQL

Plusieurs paquetages PL/SQL sont disponibles (voir le tableau 11-41) pour prendre en compte les spécificités de XML DB (pour le type XMLType, pour les grammaires et pour le *repository* étudié à la section suivante).

Tableau 11-41 Principaux paquetages PL/SQL pour XML DB

Paquetage	Fonctionnalités
DBMS_XMLGEN	Générer du contenu XML.
DBMS_XMLSTORE	Mapper du contenu XML.
DBMS_PARSER	Analyser du contenu XML.
DBMS_XSLPROCESSOR	Transformer du contenu XML avec un programme XSLT.
DBMS_XMLDOM	Manipuler du contenu XML.
DBMS_XMLSCHEMA	Enregistrer et gérer des grammaires *XML Schema*.
DBMS_XMLSCHEMA_ANNOTATE	Gérer des annotations opérées après l'enregistrement de la grammaire.
DBMS_XMLSTORAGE_MANAGE	Gérer du stockage après l'enregistrement de la grammaire.
DBMS_XDB_REPOS	Gérer des accès et des ressources dans le *repository*.
DBMS_XDB_CONFIG	Configurer le *repository*.

Le paquetage DBMS_XMLGEN

Ce paquetage offre des méthodes pour générer du contenu XML (retourné CLOB ou XMLType) à partir de requêtes SQL. Bien qu'il offre certaines fonctionnalités analogues à celles du paquetage DBMS_XMLQUERY, il est toutefois problématique dans la construction d'attributs et se limite aux éléments.

Le code suivant présente les principales méthodes de ce paquetage. La procédure setrowsettag définit le nom de la racine (par défaut, ROWSET) et la procédure setrowtag fixe le nom de l'élément qui détermine chaque ligne extraite de la requête (par défaut, ROW). Les sous-éléments sont construits dans la requête à l'aide d'alias de colonnes. Ici, tous les vols sont parcourus et un calcul est effectué pour chacun.

Tableau 11-42 Utilisation du paquetage DBMS_XMLGEN

Code SQL	Commentaires
`DECLARE`	
`  v_xmltype XMLTYPE;`	
`  v_ctx  DBMS_XMLGEN.ctxhandle;`	Variable de contexte.
`BEGIN`	
`  v_ctx := DBMS_XMLGEN.newcontext(`	Création du contexte.
`   'SELECT af.na AS "immat", af.codec AS "comp",`	
`     TO_CHAR(af.date_a,''DD-MM-YYYY'') AS "date_vol",`	
`     (av.capacite - af.nb_passagers)  AS "nb_places"`	
`   FROM  affreter_R af, avion_R av`	
`   WHERE  af.na = av.na`	
`   ORDER BY "date_vol", "immat"');`	
`  DBMS_XMLGEN.setrowsettag(v_ctx, 'vols');`	Définition du nom de la racine.
`  DBMS_XMLGEN.setrowtag(v_ctx, 'vol');`	Définition du nom de chaque élément.
`  v_xmltype := DBMS_XMLGEN.getXmlType(v_ctx) ;`	Génère le document XML et retourne un XMLType.
`  DBMS_OUTPUT.PUT_LINE('nbre lignes : '||`	
`     DBMS_XMLGEN.getnumrowsprocessed(v_ctx));`	Nombre de lignes traitées.
`  DBMS_XMLGEN.closeContext(v_ctx);`	Fermeture du contexte.
`  DBMS_OUTPUT.PUT_LINE(v_xmltype.getClobVal);`	
`END;`	

Tableau 11-42 Utilisation du paquetage DBMS_XMLGEN *(suite)*

Code SQL	Commentaires

```
nbre lignes : 5
<vols>
  <vol>
    <immat>F-GODF</immat>
    <comp>EJ</comp>
         <date_vol>12-08-2014</date_vol>
    <nb_places>50</nb_places>
  </vol>
  <vol>
    <immat>F-GODF</immat>
    <comp>AF</comp>
    <date_vol>12-08-2014</date_vol>
    <nb_places>20</nb_places>
  </vol>
…
</vols>
```

Le paquetage DBMS_XMLSTORE

Ce paquetage succède à DBMS_XMLSAVE et permet de stocker et de manipuler du contenu XML dans des tables conventionnelles en faisant correspondre chaque sous-élément à une ligne d'une table. Les sources de données proviennent de différents types (VARCHAR2, CLOB ou XMLType). Le *mapping* est similaire à celui de DBMS_XMLGEN. Le paquetage DBMS_XMLSTORE est aussi limité à la construction d'éléments et ignore les éventuels attributs.

Le code suivant présente l'insertion de quelques lignes dans la table pils créée à cet effet. La procédure setUpdateColumn ajoute une colonne à la liste de correspondance. Dans le cas d'une insertion, toutes les colonnes de la table seront par défaut mises à jour (attention aux éventuelles contraintes NOT NULL qui pourraient exister). Notez enfin l'importance de la casse au niveau du nom des balises (le premier sous-élément est ignoré de la correspondance).

```
CREATE TABLE pils(num NUMBER(4), nom VARCHAR2(20), date_nais DATE);
```

Tableau 11-43 Utilisation du paquetage DBMS_XMLSTORE

Code SQL	Commentaires
`DECLARE`	
`v_insCtx DBMS_XMLSTORE.ctxType;`	Création du contexte.
`v_rows NUMBER;`	
`v_xml CLOB := '<ROWSET>`	Document XML à faire correspondre.
`<ROW>`	
`<num>101</num><nom>Roche</nom>`	
`<date_nais>31-12-1986</date_nais>`	
`</ROW>`	
`<ROW>`	
`<NUM>401</NUM><NOM>Alquie</NOM>`	
`</ROW>`	
`<ROW>`	
`<NUM mail="asayag@orange.fr">801</NUM>`	
`<NOM>Sayag</NOM>`	
`<DATE_NAIS>30-12-1988</DATE_NAIS>`	
`</ROW>`	
`<ROW>`	
`<NUM>901</NUM>`	
`<NOM>Levade</NOM>`	
`<DATE_NAIS></DATE_NAIS>`	
`</ROW>`	
`</ROWSET>';`	
`BEGIN`	
`v_insCtx := DBMS_XMLSTORE.newContext('pils');`	Définition du nom de la table.
`DBMS_XMLSTORE.setUpdateColumn(v_insCtx, 'NUM');`	Définition de l'ordre des colonnes et des sous-éléments.
`DBMS_XMLSTORE.setUpdateColumn(v_insCtx, 'NOM');`	
`DBMS_XMLSTORE.setUpdateColumn(v_insCtx, 'DATE_NAIS');`	
`v_rows := DBMS_XMLSTORE.insertXML(v_insCtx, v_xml);`	Insertion via le document XML. Retourne le nombre de lignes traitées.
`DBMS_OUTPUT.PUT_LINE('Lignes insérées : ' || v_rows);`	
`DBMS_XMLSTORE.closeContext(v_insCtx);`	Fermeture du contexte.
`END;`	

```
SQL> SELECT ROWNUM, num, nom, date_nais FROM pils;

    ROWNUM        NUM NOM                  DATE_NAIS
---------- ---------- -------------------- ----------
         1
         2        401 Alquie
         3        801 Sayag                30-12-1988
         4        901 Levade
```

Avec ce paquetage, il est également possible de modifier (avec `updateXML`) ou de supprimer (avec `deleteXML`) des lignes dans la table relationnelle à l'aide de la correspondance avec le document XML fournit et le contenu de la table.

Le paquetage DBMS_XMLPARSER

Ce paquetage permet d'analyser différentes sources de données (caractères ou LOB principalement) dans le but de créer un contenu XML (au sens DOM du terme et qui sera traité plus loin avec le paquetage `DBMS_XMLDOM`).

Le code suivant présente une procédure qui a pour but de lire un LOB contenu dans une table relationnelle et d'analyser (avec `parseClob`) ce dernier pour déterminer s'il s'agit d'un document XML bien formé. Dans le cas contraire, une exception est relevée. Dans cet exemple, un document Word et un document XML sont stockés dans une table et sont tous deux analysés.

Tableau 11-44 Utilisation du paquetage DBMS_XMLPARSER

Code SQL	Commentaires
`CREATE PROCEDURE parse_clob(p1 IN NUMBER) AS`	
`  v_clob  CLOB;`	
`  v_nomfic  VARCHAR2(40);`	
`  v_parser  DBMS_XMLPARSER.parser;`	Déclaration d'un objet *parser*.
`  v_domdoc  DBMS_XMLDOM.DOMDocument;`	
`  v_node  DBMS_XMLDOM.DOMNode;`	
`  mauvais_format_XML EXCEPTION;`	
`  PRAGMA EXCEPTION_INIT(mauvais_format_XML,-31011);`	
`BEGIN`	
`  SELECT texte,fic INTO v_clob,v_nomfic`	
`    FROM table_CLOB WHERE num = p1;`	
`  v_parser := DBMS_XMLPARSER.newParser;`	Création d'un objet *parser*.
`  DBMS_XMLPARSER.parseClob(p => v_parser , doc => v_clob);`	Analyse du document LOB.
`  v_domdoc := DBMS_XMLPARSER.getDocument(v_parser);`	
`  v_node := DBMS_XMLDOM.makeNode(v_domdoc);`	
`  DBMS_XMLPARSER.freeParser(p => v_parser);`	Abandon d'un objet *parser*.
`  DBMS_OUTPUT.PUT_LINE('Format XML correct : ' || v_nomfic);`	
`EXCEPTION`	
`  WHEN mauvais_format_XML THEN`	Traitement de l'erreur (document non XML).
`  DBMS_OUTPUT.PUT_LINE('Probleme format XML : ' || v_nomfic);`	
`END;`	

Figure 11-23 *Résultats*

Le paquetage DBMS_XMLDOM

Ce paquetage permet de manipuler des documents XML avec la méthode DOM à partir de différentes sources de données (requête SQL, CLOB ou XMLType). Plus avancé que le constructeur XMLType et le paquetage DBMS_XMLGEN, le paquetage DBMS_XMLDOM est très complet en termes de fonctionnalités et d'exceptions pour ajouter (appendChild, makeNode et createElement), supprimer et renommer des éléments ou des attributs (setAttribute) au sein d'un contenu XML.

Le code suivant construit un document à partir d'une jointure des tables des compagnies et des vols.

Tableau 11-45 Utilisation du paquetage DBMS_XMLDOM

Code SQL	Commentaires

`DECLARE`
`  v_xmltype XMLTYPE;`
`  v_domdoc DBMS_XMLDOM.DOMDocument;`
`  v_root_node DBMS_XMLDOM.DOMNode;`
`  v_vols_element DBMS_XMLDOM.DOMElement;`
`  v_vols_node DBMS_XMLDOM.DOMNode;`
`  v_vol_element DBMS_XMLDOM.DOMElement;`
`  v_vol_node DBMS_XMLDOM.DOMNode;`
`  v_immat_element DBMS_XMLDOM.DOMElement;`
`  v_immat_node DBMS_XMLDOM.DOMNode;`
`  v_immat_text DBMS_XMLDOM.DOMText;`
`  v_immat_textnode DBMS_XMLDOM.DOMNode;`	

`BEGIN` `  v_domdoc := DBMS_XMLDOM.newDomDocument;`	Document vide.
`  v_root_node := DBMS_XMLDOM.makeNode(v_domdoc);`	
`  v_vols_element := DBMS_XMLDOM.createElement(v_domdoc,'vols');`	Racine.
`  v_vols_node := DBMS_XMLDOM.appendChild(v_root_node,` `  DBMS_XMLDOM.makeNode(v_vols_element));`	Nœud racine.

`  FOR c_enreg IN (SELECT c.codec, c.nom_comp, af.na,`
`  af.date_a, af.nb_passagers`
`  FROM compagnie_R c, affreter_R af`
`  WHERE c.codec = af.codec)`
	Requête SQL à traiter ligne par ligne.

`  LOOP`	
`  --`	
`  v_vol_element := DBMS_XMLDOM.createElement(v_domdoc,'vol');`	Pour chaque ligne, ajout d'un élément complexe.
`  DBMS_XMLDOM.setAttribute(v_vol_element,'comp',` `  c_enreg.nom_comp);`	
`  v_vol_node := DBMS_XMLDOM.appendChild(v_vols_node,` `  DBMS_XMLDOM.makeNode(v_vol_element));`	
`  v_immat_element := DBMS_XMLDOM.createElement(` `  v_domdoc,'immat');`	

`  v_immat_node := DBMS_XMLDOM.appendChild(v_vol_node,`
`  DBMS_XMLDOM.makeNode(`
`  v_immat_element));`	

`  v_immat_text := DBMS_XMLDOM.createTextNode(` `  v_domdoc,c_enreg.na);`	

Tableau 11-45 Utilisation du paquetage DBMS_XMLDOM *(suite)*

Code SQL	Commentaires

```
      v_immat_textnode := DBMS_XMLDOM.appendChild(
                          v_immat_node,
                          DBMS_XMLDOM.makeNode(v_immat_text));
      DBMS_XMLDOM.setAttribute(v_vol_element,'date_vol',
                    TO_CHAR(c_enreg.date_a,'YYYY-MM-DD')||'T'||
                    TO_CHAR(c_enreg.date_a,'HH24:MI:SS'));
      DBMS_XMLDOM.setAttribute(v_vol_element,
                    'passagers',c_enreg.nb_passagers);
   END LOOP;
   v_xmltype := DBMS_XMLDOM.getXmlType(v_domdoc);
   DBMS_XMLDOM.freeDocument(v_domdoc);
   DBMS_OUTPUT.PUT_LINE(v_xmltype.getClobVal());
END;
```

Figure 11-24 *Document résultat*

```
<vols>
  <vol comp="Easy Jet" date_vol="2014-08-12T14:30:00" passagers="120">
    <immat>F-GODF</immat>
  </vol>
  <vol comp="Air France" date_vol="2014-08-12T14:45:00" passagers="150">
    <immat>F-GODF</immat>
  </vol>
  <vol comp="Air France" date_vol="2014-08-12T14:45:00" passagers="130">
    <immat>F-PROG</immat>
  </vol>
  <vol comp="Air France" date_vol="2014-09-12T16:30:00" passagers="110">
    <immat>F-PROG</immat>
  </vol>
  <vol comp="Air Blagnac" date_vol="2014-09-12T16:30:00" passagers="450">
    <immat>F-WOWW</immat>
  </vol>
</vols>
```

XML DB Repository

XML DB Repository est un environnement partagé de contenus (XML ou CLOB) basé sur le concept de système de gestion de fichiers (répertoires). L'environnement est compatible avec la norme DAV (*Distributed Authoring and Versioning*), extension du protocole HTTP qui permet un accès multi-utilisateur au contenu d'un dossier. Toutes les informations de cet environnement sont stockées dans le schéma de l'utilisateur `XDB` (à maintenir verrouillé). Toute la documentation se trouve dans la partie *V* ou *VI* (suivant la version de votre base) du livre *Oracle XML DB Repository*.

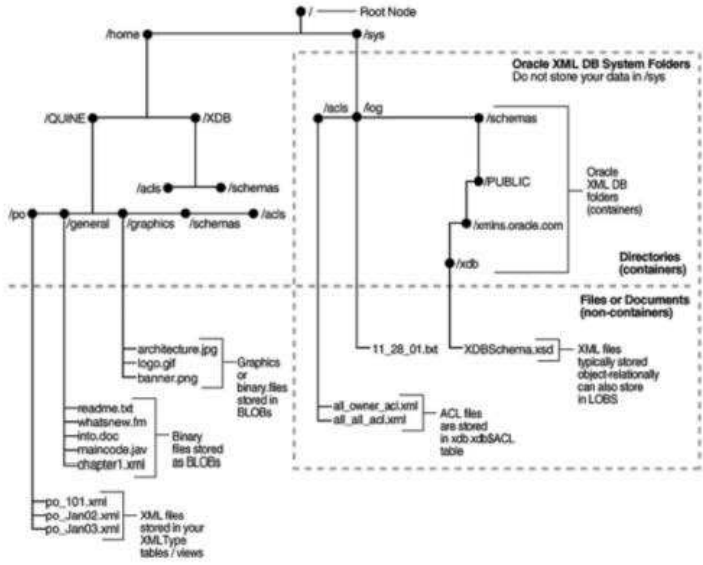
Les moyens d'accéder avec *XML DB Repository* sont les suivants :

- protocoles FTP, WebDAV et HTTP(S) ;
- avec PL/SQL et l'utilisation des paquetages `DBMS_XDB_ADMIN`, `DBMS_XDB_CONFIG`, `DBMS_XDB_REPOS` et `DBML_XDBRESOURCE` ;
- avec SQL par les vues `RESOURCE_VIEW` et `PATH_VIEW` ;
- par l'API Java (XML DB API).

Arborescence

La figure 11-25 décrit l'arborescence qu'Oracle préconise pour travailler avec le système de gestion de fichiers de *XML DB Repository*.

Figure 11-25 *Arborescence du système de gestion de fichiers*



Paquetages DBMS_XBD_REPOS

Le paquetage `DBMS_XDB_REPOS` propose de nombreuses fonctions pour manipuler le système de gestion de fichiers, notamment la création d'un répertoire (`createfolder`), la suppression d'une ressource document ou répertoire (`deleteresource`) ou la vérification de présence d'une ressource (`existsresource`).

Le code du tableau 11-46 crée, d'une part, deux répertoires (`/home/OXM` et `/home/OXM/general`) et supprime, d'autre part, le répertoire `/home/OXM` (et son contenu). Notez l'utilisation du *commit* pour valider la mise à jour. La fonction `createfolder` retourne vrai si le répertoire a été correctement créé.

Tableau 11-46 Gestion de répertoires

Création de répertoires

```
DECLARE
  v_resultat BOOLEAN;
BEGIN
  v_resultat :=
      DBMS_XDB_REPOS.createfolder('/home/OXM');
  IF v_resultat THEN
       DBMS_OUTPUT.PUT_LINE('/home/OXM OK.');
       v_resultat :=
          DBMS_XDB_REPOS.createfolder(
                        '/home/OXM/general');
      IF v_resultat THEN
        DBMS_OUTPUT.PUT_LINE(
                    '/home/OXM/general OK.');
        COMMIT;
      ELSE
         DBMS_OUTPUT.PUT_LINE(
             'bug creation /home/OXM/general');
      END IF;
    ELSE
     DBMS_OUTPUT.PUT_LINE(
           'bug creation /home/OXM');
  END IF;
END;
```

Suppression de répertoires

```
DECLARE
  v_resultat BOOLEAN;
BEGIN
  IF (DBMS_XDB_REPOS.existsresource(
           '/home/OXM')) THEN
    DBMS_XDB_REPOS.deleteresource(
     '/home/OXM',
     DBMS_XDB_REPOS.DELETE_RECURSIVE_FORCE);
   COMMIT;
  END IF;
END;
/
```

Le code du tableau 11-47 crée une ressource (avec `createresource`) sous la forme d'un document XML situé dans un répertoire du système d'exploitation et déposé dans le répertoire `/home/OXM/general` du *repository*. Le *commit* valide la mise à jour et la fonction qui crée la ressource retourne vrai en cas de succès.

Par ailleurs, l'extraction sous forme d'un LOB du document précédemment stocké est présentée (`getContentClob`). La conversion au format XMLType est rendu possible par la méthode `createXML` qui crée une instance qu'on interroge à la fin par une requête SQL/XML.

Tableau 11-47 Création d'une ressource

Chargement d'un document

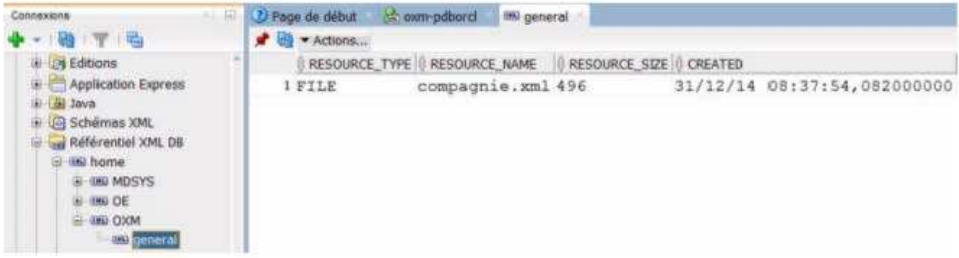
```
DECLARE
  v_resultat BOOLEAN;
BEGIN
  IF NOT (DBMS_XDB_REPOS.existsresource(
     '/home/OXM/general/compagnie.xml')) THEN
   v_resultat :=
    DBMS_XDB_REPOS.createresource(
     abspath =>
          '/home/OXM/general/compagnie.xml',
     data =>
         BFILENAME('REPXML', 'compagnie.xml'),
     csid => NLS_CHARSET_ID('AL32UTF8'));
    IF v_resultat THEN
       COMMIT;
       DBMS_OUTPUT.PUT_LINE('dépot OK.');
    ELSE
       DBMS_OUTPUT.PUT_LINE('bug dépot...');
   END IF;
  END IF;
END;
/
```

Extraction sous la forme d'un CLOB

```
DECLARE
   v_nom    VARCHAR2(15);
   v_xml    XMLTYPE;
   v_clob   CLOB;
   v_path   VARCHAR2(40) :=
      '/home/OXM/general/compagnie.xml';
BEGIN
   v_clob   :=
     DBMS_XDB_REPOS.getContentClob(v_path);
   DBMS_OUTPUT.PUT_LINE('Taille CLOB : ' ||
              DBMS_LOB.GETLENGTH(v_clob));
   v_xml := XMLTYPE.createXML(
               xmlData=>v_clob);
   SELECT XMLCAST(XMLQUERY(
         '/compagnie/pilotes/pilote
             [@brevet="PL-2"]/nom'
          PASSING v_xml RETURNING CONTENT)
         AS VARCHAR2(15)) INTO v_nom
   FROM  DUAL;
   DBMS_OUTPUT.PUT_LINE('nom pilote PL-2 : '
                        || v_nom);
END;
/
Taille CLOB : 426
nom pilote PL-2 : Benech
```

SQL Developer permet d'accéder à ces ressources ; ici, le document déposé apparaît dans son répertoire.

Figure 11-26 Accès par SQL Developer

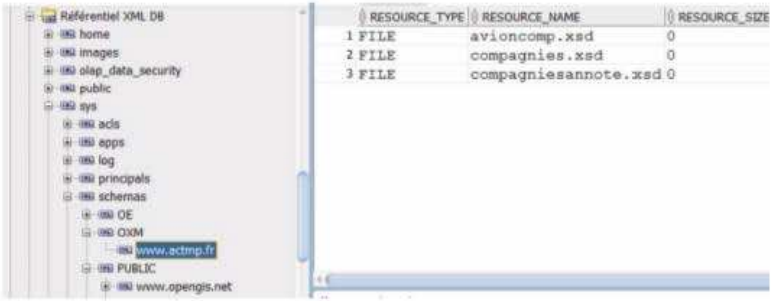


Les grammaires XML Schema

L'enregistrement d'une grammaire avec la procédure `registerschema` du paquetage `DBMS_XMLSCHEMA`, décrite en début de chapitre, a pour conséquence le stockage de ladite grammaire dans le répertoire `/sys/schemas/nom_utilisateur` si la grammaire est locale (sinon, `/sys/schemas/PUBLIC`). Le paramètre `local` (par défaut à `TRUE`) peut être positionné à `FALSE` pour enregistrer une grammaire globale qui pourra être utilisée par tout utilisateur. Vous devez détenir le privilège `XDBADMIN` pour enregistrer une grammaire globale.

SQL Developer permet d'accéder aux grammaires, ici les grammaires locales précédemment enregistrées.

Figure 11-27 *Accès aux grammaires par SQL Developer*



Accès par SQL

Deux vues permettent d'accéder aux ressources de *XML DB Repository* : `RESOURCE_VIEW` et `PATH_VIEW`. Toutes deux possèdent une colonne virtuelle `RES` (de type XMLType) qui rend possible l'accès à du contenu par la notation pointée (alias SQL). Chaque ligne de la vue `RESOURCE_VIEW` concerne un unique chemin dans l'arborescence, tandis que chaque ligne de la vue `PATH_VIEW` concerne une unique ressource.

Vue RESOURCE_VIEW

Cette vue est composée de trois colonnes :

- `RES` (`XMLType`) décrit une ressource d'un répertoire ;
- `ANY_PATH` (`VARCHAR2`) indique un chemin (absolu) d'une ressource ;
- `RESID` (`RAW`) contient l'identifiant d'une ressource.

La grammaire de la colonne `RES` (`XDBResource.xsd`) de la vue `RESOURCE_VIEW` se situe dans l'arborescence `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/`. En considérant

certains éléments de cette grammaire, des requêtes peuvent être composées pour extraire tout ou partie du contenu stocké dans les ressources. Le tableau 11-48 décrit les principaux éléments définis dans la grammaire `XDBResource.xsd`.

Tableau 11-48 Parties de la grammaire de la vue RESOURCE_VIEW

Requêtes SQL	Commentaires

`<Resource`
`  xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"`
`  Container="…"…>`
`<CreationDate> … </CreationDate>`
`<ModificationDate> … </ModificationDate>`
	Répertoire ou fichier.
	Dates de création et de modification de la ressource.

`<DisplayName> … </DisplayName>`	Nom du fichier.

`<Language> … </Language>`
`<CharacterSet> … </CharacterSet>`
`<ContentType> … </ContentType`
	Langage, jeu de caractères et type du contenu.

`<ACL> … </ACL>`	Autorisations (*Acces Control Lists*).

`<Owner> … </Owner>`
`<Creator> … </Creator>`
`<LastModifier> … </LastModifier>`
	Compte Oracle propriétaire de la ressource, créateur et dernier utilisateur ayant modifié la ressource.

`<SchemaElement> … </SchemaElement>`	Élément de la ressource.

`<Contents>`
`  <text> … </text> </Contents>`
`</Resource>`
	Contenu de la ressource.

Plusieurs fonctions SQL sont adaptées à ces vues, à savoir :

- `equals_path` qui teste l'existence d'une ressource ;
- `under_path` qui parcourt les répertoires ;
- `path` et `depth` qui retournent respectivement le chemin et la profondeur d'une ressource, et qui fonctionnent en corrélation avec les fonctions précédentes.

Le tableau 11-49 présente quelques extractions avec cette vue. Notez l'utilisation de `XMLNAMESPACES` pour définir l'espace de noms de la grammaire concernée (ici, `XDBResource.xsd`).

Tableau 11-49 Interrogations avec la vue RESOURCE_VIEW

Requêtes SQL	Commentaires et résultats

`SELECT r.RES.getClobVal()`
`FROM RESOURCE_VIEW r`
`WHERE equals_path(r.RES,`
`'/home/OXM/general/compagnie.xml') = 1;`
	Contenu du document `compagnie.xml` situé dans `/home/OXM/general`. Ce contenu est encapsulé au sein d'un élément `Resource`.

Tableau 11-49 Interrogations avec la vue RESOURCE_VIEW *(suite)*

Requêtes SQL	Commentaires et résultats

`SELECT COUNT(*) AS`
`  "Nombre grammaires www.actmp.fr"`
`FROM  RESOURCE_VIEW rv`
`WHERE  under_path(rv.RES,`
`  '/sys/schemas/OXM/www.actmp.fr') = 1;`
	Nombre de ressources dans un répertoire donné.
	`Nombre grammaires www.actmp.fr`
	`----------------------------`
	`3`

`SELECT a.nom AS "Date création"`
`FROM  RESOURCE_VIEW rv,`
`  XMLTABLE(`
`  XMLNAMESPACES(`
`  'http://xmlns.oracle.com/xdb/XDBResource.xsd'`
`  AS "e"),`
`  '/e:Resource' PASSING rv.RES`
`  COLUMNS nom VARCHAR2(20)`
`  PATH 'e:CreationDate') a`
`WHERE  equals_path(rv.RES,`
`  '/home/OXM/general/compagnie.xml') = 1;`
	Date de création d'une ressource.
	
	`Date création`
	`--------------------`
	`2014-12-31T08:37:54`

`SELECT ANY_PATH`
`FROM  RESOURCE_VIEW rv,`
`  XMLTABLE(`
`  XMLNAMESPACES(`
`  'http://xmlns.oracle.com/xdb/XDBResource.xsd'`
`  AS "e"),`
`  '/e:Resource' PASSING rv.RES`
`  COLUMNS nom VARCHAR2(20) PATH 'e:DisplayName',`
`  proprio VARCHAR2(30) PATH 'e:Owner') a`
`WHERE  a.nom LIKE '%.xml'`
`AND  a.proprio = 'OXM';`
	Chemin des fichiers XML qui sont la propriété de l'utilisateur OXM.
	
	`ANY_PATH`
	`-----------------------------`
	`/home/OXM/general/compagnie.xml`

`SELECT a.brev AS brevet, a.nom`
`FROM  RESOURCE_VIEW rv,`
`  XMLTABLE(`
`  XMLNAMESPACES(`
`  'http://xmlns.oracle.com/xdb/XDBResource.xsd'`
`  AS "r"),`
`  '/r:Resource/r:Contents/compagnie/pilotes/pilote'`
`  PASSING rv.RES`
`  COLUMNS nom VARCHAR2(15) PATH 'nom',`
`  brev VARCHAR2(6) PATH '@brevet') a`
`WHERE equals_path(rv.RES,`
`  '/home/OXM/general/compagnie.xml') = 1;`
	Brevet et noms des pilotes contenus dans le document `compagnie.xml` situé dans `/home/OXM/general/`.
	
	`BREVET NOM`
	`------ ---------------`
	`PL-1  Sarda`
	`PL-2  Benech`

Vue PATH_VIEW

Cette vue est composée de quatre colonnes :

- `PATH` (`VARCHAR2`) indique un chemin (absolu) d'une ressource ;
- `RES` (`XMLType`) présente une ressource du répertoire décrit dans `PATH` ;
- `LINK` (`XMLType`) décrit un lien vers la ressource ;
- `RESID` (`RAW`) contient l'identifiant d'une ressource.

Le tableau 11-50 propose deux extractions de cette vue. La première parcourt les deux niveaux sous le répertoire /home/OXM (la profondeur maximale est donnée par le deuxième paramètre de under_path, ici 3) ; la seconde examine deux répertoires.

Tableau 11-50 Interrogation de la vue PATH_VIEW

Requête SQL	Résultat

```
SELECT path(1)
       AS "path(1)",
       depth(1)
       AS "depth(1)"
FROM   PATH_VIEW
WHERE  under_path(RES,3,
      '/home/OXM',1)=1;
```

```
path(1)                                  depth(1)
---------------------------------------- ----------
general                                           1
general/compagnie.xml                             2
```

```
SELECT PATH,
  depth(1) AS "depth(1)",
 depth(2) AS "depth(2)"
FROM  PATH_VIEW
WHERE (under_path(RES,3,
  '/sys/schemas/OXM',1)=1
OR  under_path(RES,3,
  '/home/OXM',2)=1);
```

```
PATH                                                depth(1) depth(2)
--------------------------------------------------- ------- -------
/sys/schemas/OXM/www.actmp.fr                             1
/sys/schemas/OXM/www.actmp.fr/compagnies.xsd              2
/sys/schemas/OXM/www.actmp.fr/compagniesannote.xsd        2
/sys/schemas/OXM/www.actmp.fr/avioncomp.xsd               2
/home/OXM/general                                                 1
/home/OXM/general/compagnie.xml                                   2
```

Les Access Control Lists (ACL)

Le mécanisme des ACL d'Oracle est identique à celui utilisé par Java, Microsoft, etc. Dans *XML DB Repository*, chaque ressource est protégée par une ACL exprimée sous la forme d'un document XML qui respecte la grammaire /sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd.

Composition

Dans cette grammaire, l'élément racine est acl et la liste des privilèges d'accès est décrite dans chaque sous-élément ace (pour *access control entry*) :

- l'élément grant autorise (avec true) ou interdit (avec false) une liste de privilèges ;
- l'élément principal décrit le bénéficiaire. (ou une liste de bénéficiaires dans un contexte LDAP). Précédé de l'élément invert, l'élément principal désigne tout autre bénéficiaire que celui indiqué ;
- l'élément privilege contient un ou plusieurs droits ;
- les attributs start_date et end_date servent à borner dans le temps une éventuelle période de validité.

Par défaut, les utilisateurs disposant des rôles XDBADMIN et DBA ont un accès total aux ressources. Les utilisateurs basiques (avec le rôle CONNECT) peuvent lire et parcourir tous les répertoires.

Les ACL système se situent dans `/sys/acls`. Vous y trouverez :

- `bootstrap_acl.xml` : lecture pour tout le monde et tous privilèges aux rôles `XDBADMIN` et `DBA` ;
- `all_all_acl.xml` : tous privilèges à tous ;
- `all_owner_acl.xml` : tous privilèges au propriétaire (*owner*) ;
- `ro_all_acl.xml` : lecture à tout le monde.

Interrogation

Enfin, les ACL étant elles-mêmes des ressources, elles sont régies par la grammaire commune à toute ressource, soit `XDBResource.xsd`. Notez l'utilisation de la fonction `getACLDocument` du paquetage `DBMS_XDB_REPOS` pour retrouver l'ACL d'un document donné. Ici, il s'agit de `bootstrap_acl.xml`, appliquée par défaut.

Tableau 11-51 Interrogation des ACL

Requête SQL	Résultat

`SELECT r.RES.getClobVal()`
`FROM RESOURCE_VIEW r`
`WHERE equals_path(r.RES,`
`'/sys/acls/ro_all_acl.xml')=1;`
	…
	<Contents>
	<acl description="Read-Only:Readable by all and writeable by none" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd http://xmlns.oracle.com/xdb/acl.xsd" shared="true">
	<ace>
	<grant>true</grant>
	<principal>PUBLIC</principal>
	<privilege>
	<read-properties/>
	<read-contents/>
	<read-acl/>
	<resolve/>
	</privilege>
	</ace>
	</acl>
	</Contents>
	</Resource>

`SELECT`
`DBMS_XDB_REPOS.getACLDocument(`
`'/home/OXM/general/compagnie.xml')`
`FROM DUAL;`
	<acl description="Protected:Readable by PUBLIC and all privileges to …>
	<ace>
	<grant>true</grant>
	<principal>dav:owner</principal>
	<privilege>
	<all/>
	</privilege>
	</ace>
	…
	</acl>

Affectation

Pour affecter une ACL à une ressource, vous devez utiliser la fonction `setACL` du paquetage `DBMS_XDB_REPOS`. Dans le code suivant, l'ACL système qui permet à tout le monde l'accès en lecture est appliquée au document `compagnie.xml`. Notez la validation qui doit s'en suivre.

Tableau 11-52 Affectation d'une ACL

Code SQL	La nouvelle ACL du document

```
BEGIN
  DBMS_XDB_REPOS.setACL(
   res_path => '/home/OXM/general/compagnie.xml',
   acl_path => '/sys/acls/ro_all_acl.xml');
  COMMIT;
END;
/
```

```
<acl description="Read-Only:Readable by all
 and writeable by none" …>
  <ace>
    <grant>true</grant>
    <principal>PUBLIC</principal>
    <privilege>
      <read-properties/>
      <read-contents/>
      <read-acl/>
      <resolve/>
    </privilege>
  </ace>
</acl>
```

Pour supprimer une ACL, vous devrez modifier toutes les ressources qui en dépendent avec `DBMS_XDB_REPOS.setACL`, puis supprimer la ressource en question avec `DBMS_XDB_REPOS_REPOS.deleteResource`.

Gestion des privilèges

Pour construire vos propres ACL, vous utiliserez nécessairement des privilèges atomiques ou agrégés et deux espaces de noms (`xmlns="http://xmlns.oracle.com/xdb/acl.xsd"` et `xmlns:dav="DAV:"`). Vous trouverez le détail de tous ces privilèges dans le chapitre « Repository Access Control » du livre *XML DB Developer's Guide*.

Dans le code suivant, la fonction `changePrivileges` du paquetage `DBMS_XDB_REPOS` ajoute une entrée qui précise que l'utilisateur OXM2 aura tous les droits sur le document en question. Notez la nécessité de positionner l'élément racine dans l'espace de noms d'Oracle et de déclarer les autres espaces. La validation finale doit s'en suivre.

Tableau 11-53 Ajout d'un privilège

Code SQL	La nouvelle ACL du document

DECLARE
  r PLS_INTEGER;
  v_ace XMLType;
  v_char VARCHAR2(2000);
BEGIN
  v_char :=
'<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
http://xmlns.oracle.com/xdb/acl.xsd
DAV:http://xmlns.oracle.com/xdb/dav.xsd">
<principal>OXM2</principal>
<grant>true</grant>
<privilege><all/></privilege>
</ace>';
  v_ace := XMLType.createXML(v_char);
  r := **DBMS_XDB_REPOS.changePrivileges(**
  res_path=>'/home/OXM/general/compagnie.xml',
  ace => v_ace);
COMMIT;
END;
	<acl …>
	  <ace>
	  <grant>true</grant>
	  <principal>PUBLIC</principal>
	  <privilege>
	  <read-properties/>
	  <read-contents/>
	  <read-acl/>
	  <resolve/>
	  </privilege>
	  </ace>
	  <ace>
	  <grant>true</grant>
	  <principal>OXM2</principal>
	  <privilege>
	  <all/>
	  </privilege>
	  </ace>
	</acl>

Dans le code suivant, une nouvelle ACL (`acl_oxm2.xml`) est créée et déposée dans le répertoire `/home/OXM/acls` (qui devra être préalablement créé). Cette ACL autorise l'utilisateur OXM2 à accéder en lecture à toute ressource et à pouvoir verrouiller ou déverrouiller toute ressource.

Tableau 11-54 Création d'une ACL

Code SQL	Commentaires

`DECLARE`
`  v_resultat BOOLEAN;`
`BEGIN`
`  v_resultat := DBMS_XDB_REPOS.createResource(`
`     abspath => '/home/OXM/acls/acl_oxm2.xml',`
`     data    => '<acl description="exemple acl"`
`  xmlns="http://xmlns.oracle.com/xdb/acl.xsd"`
`  xmlns:dav="DAV:"`
`  xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"`
`  xsi:schemaLocation=`
`        "http://xmlns.oracle.com/xdb/acl.xsd`
`  http://xmlns.oracle.com/xdb/acl.xsd">`
`       <ace>`
`           <grant>true</grant>`
`           <principal>OXM2</principal>`
`           <privilege>`
`           <read-contents/>`
`           <dav:lock/>`
`           <dav:unlock/>`
`           </privilege>`
`       </ace>`
`  </acl>',`
`  schemaurl => 'http://xmlns.oracle.com/xdb/acl.xsd',`
`  elem      => 'acl');`
`  IF v_resultat THEN`
`     DBMS_OUTPUT.PUT_LINE('dépot ACL OK.');`
`     COMMIT;`
`  ELSE`
`     DBMS_OUTPUT.PUT_LINE('Problème au dépot ACL...');`
`   END IF;`
`END;`
	Création d'une ressource.
	Définition de l'entrée.
	Choix de l'espace de noms.
	Vérification du dépôt.

Une fois cette ACL créée, vous pourrez l'affecter à une ressource en particulier avec la méthode `setACL` précédemment étudiée.

Dictionnaire des données

Le dictionnaire des données prend en compte toutes les spécificités relatives à XML DB, notamment au niveau d'un utilisateur :

- les tables et vues XMLType avec les vues `USER_XML_TABLES` et `USER_XML_TAB_COLS` ;
- les grammaires avec la vue `USER_XML_SCHEMAS` ;
- la colonne `XMLSCHEMA` de cette vue contient le code complet de la grammaire ;
- les vues XMLType avec `USER_XML_VIEWS`.

Le code suivant présente quelques extractions de ces vues.

Tableau 11-55 Interrogation du dictionnaire pour XML DB

Code SQL et résultats

```
SQL> SELECT table_name, storage_type FROM USER_XML_TABLES;

TABLE_NAME           STORAGE_TYPE
-------------------- -----------------
T_DOCUMENTS_XML      BINARY
avioncomp651_TAB     CLOB
```

```
SQL> SELECT table_name,xmlschema,element_name FROM USER_XML_TABLES;

TABLE_NAME           XMLSCHEMA                                ELEMENT_NAME
-------------------- ---------------------------------------- ---------------
T_DOCUMENTS_XML
avioncomp651_TAB     http://www.actmp.fr/avioncomp.xsd        avioncomp
```

```
SQL> SELECT column_name, element_name, storage_type
     FROM   USER_XML_TAB_COLS;

COLUMN_NAME          ELEMENT_NAME    STORAGE_TYPE
-------------------- --------------- -----------------
COL_XML                              BINARY
SYS_NC_ROWINFO$      avioncomp       CLOB
```

```
SQL> SELECT schema_url, local, binary FROM USER_XML_SCHEMAS;

SCHEMA_URL                             LOC BIN
-------------------------------------- --- ---
http://www.actmp.fr/compagnies.xsd     YES YES
http://www.actmp.fr/compagniesannote.xsd YES NO
http://www.actmp.fr/avioncomp.xsd      YES NO
```

```
SQL> SELECT view_name, xmlschema, element_name FROM USER_XML_VIEWS;

VIEW_NAME            XMLSCHEMA                                ELEMENT_NAME
-------------------- ---------------------------------------- ---------------
COMPAGNIE_VUE_XML
AVICOMP_VUE_XML      http://www.actmp.fr/avioncomp.xsd        avioncomp
```

Chapitre 12

Optimisations

Ce chapitre traite d'optimisations des requêtes et des schémas relationnels. Plusieurs aspects sont étudiés ; tout d'abord, le fonctionnement de l'optimiseur et l'utilisation de statistiques. Par la suite, quelques outils de mesure de performances sont présentés. Enfin, nous ouvrons la boîte à outils qui servira à optimiser vos applications (contraintes, index, *clusters*, tables organisées en index, partitionnement, vues matérialisées et principes de dénormalisation).

L'optimisation des applications et des serveurs est un domaine de métiers à part entière. En conséquence, vous ne découvrirez pas dans ce chapitre la solution à votre problème particulier. D'abord parce qu'un ouvrage dédié à cela n'y suffirait pas, ensuite parce que chaque problématique est unique et qu'il n'existe pas de recette miracle à adopter en tout cas.

En revanche, vous trouverez une synthèse des mécanismes que vous pouvez mettre en œuvre pour améliorer vos performances. Ne sont pas étudiés ici les aspects système de l'optimisation, tels que les fichiers de trace ou d'alerte, les paramètres d'initialisation (dans `init.ora` ou dans les fichiers *spfile*), les tables et vues dynamiques renseignant l'état de la mémoire (sessions, requêtes, transactions, etc.). La bibliographie référence des ouvrages plus complets à ce sujet.

Cadre général

Commençons par un postulat : si une application ralentit les processus métier, elle doit être optimisée. La problématique des performances concerne toute application et il est normal d'y consacrer du temps même si Oracle peut résoudre seul la majorité des problèmes avec les outils de la 11*g*.

Il y a une dizaine d'années, le développeur écrivait une requête tout en laissant le soin au DBA de la rendre performante. Depuis, les mentalités ont changé, ce qui ne veut pas dire que les rapports entre les personnels représentant ces deux métiers se soient arrangés…

La réalité apporte son lot de désillusions ; bien souvent, par soucis d'économie, le chef de projet réduit les délais ou affecte moins de ressources humaines que prévu, la documentation est rédigée et l'optimisation réfléchie après la mise en production.

Des experts estiment à 60 % le gain potentiel de performances rien que sur l'écriture du code SQL et PL/SQL. Sachant que pour certaines applications, pour des raisons de pseudo-portabilité, l'utilisation de PL/SQL est proscrite ! Tout devant être codé dans les applications… vous

imaginez que la marge de manœuvre est réduite. Par ailleurs, bon nombre de problèmes proviennent du modèle de données (qui n'est pas toujours suffisament normalisé) et il est souvent trop tard pour modifier fortement la structure des tables.

Les performances ne peuvent être considérées sans un contexte : le disque, la mémoire, les processeurs et le réseau sont autant d'éléments qui entrent en compte lors de mesures. Ainsi, la performance n'a souvent de sens qu'associée à une action (on parle alors de *benchmark*) telle qu'une migration vers une version supérieure, migration de données d'un tablespace à l'autre, après ajout de RAM, changement de disque, etc.

Jusqu'à présent, vous avez fait confiance à Oracle (et vous avez bien fait) pour qu'il élabore la meilleure stratégie d'accès à vos données. Dès qu'une requête ou qu'un traitement (qui contient des instructions dont une ou plusieurs requêtes problématiques) va poser problème, votre confiance va s'effriter et vous allez mettre en œuvre des mécanismes pour chercher à les rendre plus efficaces.

Cette dépendance provient du fait que SQL est un langage déclaratif (et non procédural comme les structures de contrôle de PL/SQL) ; le programmeur exprime toujours dans une requête ce qu'il souhaite et non pas le moyen de l'obtenir. Oracle va utiliser son optimiseur afin de produire l'algorithme le plus efficace, selon les données dont il dispose (les statistiques), pour extraire l'information recherchée.

Les acteurs

Plusieurs acteurs influent sur les performances.

- Le concepteur se doit de fournir un modèle conceptuel de qualité, une architecture logicielle raisonnée et une programmation modulaire.
- Le développeur vérifie en principe le modèle relationnel (normalisation et dénormalisation raisonnée), écrit les instructions et requêtes d'une manière concise. Il programme ses transactions en utilisant le plus de procédures cataloguées.
- L'administrateur surveille l'exécution des sessions, organise au mieux les espaces logiques et physiques des bases de données. Il dimensionne la mémoire pour les données et traitements.
- L'utilisateur final qui se fait toujours connaître quand une attente est trop importante. Il convient de le sensibiliser en amont pour éviter parfois des conflits inutiles.

Contexte et objectifs

Idéalement, l'optimisation doit faire partie du cycle de développement et se réaliser ainsi avant la mise en production. Ce n'est pas toujours le cas et cela entraîne un certain nombre de freins.

- De quels droits dispose-t-on pour diagnostiquer et identifier le problème (par exemple, si vous n'avez pas accès aux vues v$ ou aux fichiers de trace, votre premier diagnostic ne peut pas être précis) ?

- Est-il possible de modifier le schéma relationnel, le code SQL ou PL/SQL, le code applicatif, l'organisation des données (index, types de table, etc.), la configuration de l'instance, du réseau et du matériel ?

L'organisation des données peut ne pas nécessiter de recompilation (ajout d'index ou d'une *index organised table*). La solution de changer de prime abord le matériel est souvent une fuite en avant qui peut s'avérer plus pénalisante avec une machine plus puissante.

L'objectif de toute optimisation doit être précis et mesurable (exemple : 90 ms pour extraire la liste des produits d'une commande). En effet, un objectif flou ne sera jamais satisfait et c'est une garantie contre la tentation d'optimisation excessive et contreproductive.

Utiliser un jeu de tests (comparable aux données en production) permet de mesurer objectivement les performances. Se pose le problème de l'accès aux données réelles.

Plus tôt l'optimisation est prise en compte, moins elle va coûter.

Les différents SGBD du marché ne se comportent pas de la même manière (une solution valable pour un SGBD peut se révéler peu performante pour un autre). Cela se vérifie également pour deux versions ou *releases* différentes d'Oracle.

Une solution convenable en mode mono-utilisateur peut s'avérer non opérationnelle en mode multi-utilisateur.

Les causes principales d'une mauvaise optimisation des instructions SQL sont :

- Les statistiques destinées à l'optimiseur sont obsolètes ou non représentatives.
- Des structures d'accès sont inexistantes (index, vues matérialisées ou partitions).
- La sélection de plans d'exécution est non optimale (certains éléments de l'instruction SQL sont mal évalués, par exemple son coût, sa cardinalité ou la sélectivité de son prédicat).
- Les instructions SQL sont mal construites (conditions de jointure manquantes, mauvais prédicats ou opérateurs, etc.).

Des performances médiocres peuvent également être causées par des problèmes matériels (mémoire, entrées-sorties, CPU, disque, etc.).

Présentation du jeu d'exemple

Dans ce jeu d'essai, créé initialement par F. Brouard, des adhérents pratiquent des sports. Deux tables de référence (`Adherent` et `Sport`) et une table d'association (`Pratique`) sont mises en œuvre. Les index seront décrits ultérieurement.

La volumétrie initiale de ces tables est la suivante : 24 033 adhérents, 12 sports et 27 011 lignes dans la table `Participe`. Dans plusieurs sections (notamment *Indexation*, *Cluster, Partition-*

nement et Vues matérialisées), le nombre de participants est porté à plus d'un million (tables `Pratiquebis` et `Adherentbis`) afin d'obtenir des résultats plus démonstratifs.

Tableau 12-1 Jeu d'exemple

Tables du jeu d'essai

```
CREATE TABLE Sport
 (spid NUMBER(5) NOT NULL, splibelle VARCHAR(20) NOT NULL);

CREATE TABLE Adherent
 (adhid NUMBER(5) NOT NULL,   nom VARCHAR(25) NOT NULL,
  prenom VARCHAR(30) NOT NULL,
  civilite VARCHAR(12) NOT NULL, date_nais  DATE NOT NULL,
  tel VARCHAR(15));

CREATE TABLE PRATIQUE
 (adhid NUMBER(5) NOT NULL, spid NUMBER(5) NOT NULL);

ALTER TABLE Sport    ADD CONSTRAINT pk_Sport    PRIMARY KEY (spid);
ALTER TABLE Adherent ADD CONSTRAINT pk_Adherent PRIMARY KEY (adhid);
ALTER TABLE Pratique ADD CONSTRAINT pk_Pratique PRIMARY KEY (adhid,spid);
ALTER TABLE Pratique
      ADD CONSTRAINT fk_Pratique_Sport
                     FOREIGN KEY (spid) REFERENCES Sport(spid);
ALTER TABLE Pratique
      ADD CONSTRAINT fk_Pratique_Adherent
                     FOREIGN KEY (adhid) REFERENCES Adherent(adhid);
```

Les assistants d'Oracle

Avant de présenter quelques mécanismes basiques de contrôle de performances (section *Outils de mesure de performances*) qui sont basés sur les fichiers de trace SQL et les vues de performances, résumons l'offre des outils de *tuning* d'Oracle qui inclut de nombreuses fonctionnalités que ce chapitre ne peut pas détailler.

Depuis la version 10*g*, le référentiel AWR (*Automatic Workload Repository*) permet de collecter et analyser les statistiques (successeur de Statspack). La version 11*g* automatise davantage le *tuning*, en identifiant les instructions SQL problématiques et en exécutant la fonction de conseil STA (*SQL Tuning Advisor*) sur ces instructions. Le moniteur ADDM (*Automatic Database Diagnostic Monitor*) analyse en permanence les informations de performances collectées. Il identifie automatiquement les goulets d'étranglement dans la base et fournit des recommandations sur les options permettant de résoudre ces problèmes.

- La fonction de conseil, *SQL Access Advisor*, analyse une instruction SQL et donne des conseils sur les vues matérialisées, les index, les journaux des vues matérialisées et les partitions.
- La fonction d'analyse des performances, *SQL Performance Analyzer*, automatise l'évaluation de l'impact des modifications (mise à niveau d'une base, ajout d'index), sur la charge globale SQL en identifiant les écarts de performances pour chaque instruction.
- La fonction de surveillance, *SQL Monitoring*, permet de surveiller les performances des instructions SQL pendant leur exécution.
- La fonction de gestion du plan, *SQL Plan Management*, sert à contrôler l'évolution du plan d'exécution.

Les optimiseurs

Dans la version 6, les premières versions de l'optimiseur étaient basées sur les règles (RBO, *Rule-Based Optimizer*). Avec cette technique, un rang était affecté à chaque opération (de 1 pour un accès direct par *rowid* à 15 pour le parcours séquentiel entier d'une table). Ainsi, toute requête était analysée syntaxiquement et pour les différents chemins d'accès aux données, Oracle choisissait celui dont la somme des rangs était minimale.

En version 7, l'optimiseur CBO (*Cost-Based Optimizer*) est apparu et depuis la version 10*g*, seul celui-ci bénéficie du support d'Oracle. Il estime chaque chemin d'accès des tables concernées en fonction des statistiques (situées dans le dictionnaire des données) disponibles. Collecter correctement ces statistiques est donc fondamental (en principe, cela fait partie de la tâche du DBA).

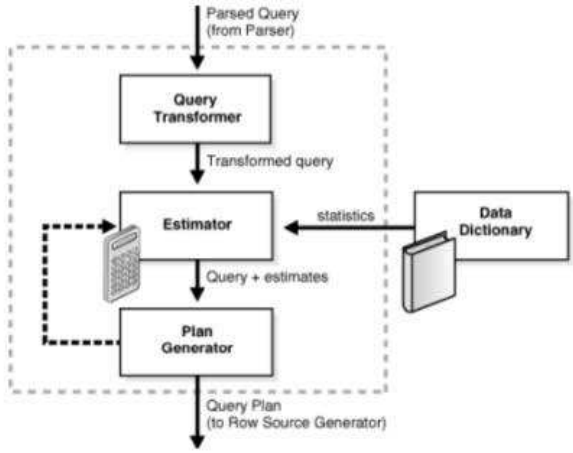
Fonctionnement de CBO

L'optimiseur d'instructions est composé :

- Du transformateur qui dispose en entrée d'une requête *parsée* et la transforme de manière optimale (notamment par expansions). Un jeu de plans potentiels est généré en fonction des chemins d'accès disponibles.
- De l'estimateur qui calcule le coût de chaque plan en fonction des statistiques du dictionnaire de données, pour les caractéristiques des tables en matière de répartition et de stockage des données, ainsi que des index auxquels accède l'instruction SQL.
- Le générateur de plan qui compare les différents plans et sélectionne celui dont le coût est le plus faible.

Parce que la recherche du meilleur plan d'exécution possible pour une interrogation est complexe, l'objectif de l'optimiseur est de trouver un bon plan, généralement appelé « plan au meilleur coût ». L'optimiseur adapte son plan d'exécution si les statistiques changent. À titre d'exemple, si d'après les statistiques il résulte que 80 % des pilotes sont des hommes, le balayage complet de table (*full table scan*) afin d'extraire les pilotes masculins constituera probablement une meilleure solution que l'utilisation d'un index.

Figure 12-1 *Mécanismes de l'optimiseur (© doc. Oracle)*



Expansions

En fonction des index existants, vues matérialisées, partitions et statistiques, et avant d'effectuer un calcul de coût, l'optimiseur peut décider de transformer une requête en une autre équivalente avant de calculer le coût de cette dernière et de l'exécuter. Le principal objectif du transformateur est de déterminer s'il est avantageux de modifier l'instruction afin qu'elle permette la génération du meilleur plan.

Le transformateur utilise plusieurs techniques telles que la transitivité, la fusion de vues, l'inclusion automatique de prédicats, l'extraction de sous-interrogation, la réécriture de requête, la transformation en étoile et l'expansion de l'opérateur `OR`. Le tableau suivant présente quelques équivalences classiques.

Tableau 12-2 Expansions classiques

Expansion	Requête initiale	Requête transformée
Opérateur OR	`SELECT adhid,nom,tel FROM Adherent WHERE civilite = 'Mlle.' OR nom = 'LEBLANC';`	`SELECT adhid,nom,tel FROM Adherent WHERE nom = 'LEBLANC' UNION ALL SELECT adhid,nom,tel FROM Adherent WHERE civilite = 'Mlle.' AND nom <> 'LEBLANC';`
Sous-interrogation	`SELECT adhid FROM Pratique WHERE spid IN (SELECT spid FROM Sport WHERE splibelle='Tennis');`	`SELECT Pratique.adhid FROM Pratique, Sport WHERE Pratique.spid = Sport.spid AND Sport.splibelle='Tennis';`
Fusion de vues	`CREATE VIEW Adherent_miss AS SELECT adhid,prenom, nom,tel,date_nais FROM Adherent WHERE civilite = 'Mlle.'; SELECT prenom,nom,tel FROM Adherent_miss WHERE adhid > 7800;`	`SELECT prenom,nom,tel FROM Adherent WHERE civilite = 'Mlle.' AND adhid > 7800;`
Transitivité	`SELECT p.adhid,s.splibelle FROM Pratique p, Sport s WHERE p.spid = s.spid AND s.spid = 12;`	`SELECT p.adhid,s.splibelle FROM Pratique p, Sport s WHERE p.spid = s.spid AND s.spid = 12 AND p.spid = 12;`

L'estimateur

L'estimateur gère trois types de mesures liées entre elles : la sélectivité, la cardinalité et le coût. La cardinalité est dérivée de la sélectivité, et le coût dépend souvent de la cardinalité.

La sélectivité est une estimation de la proportion des lignes d'un ensemble qui est extraite par un prédicat donné ou une combinaison de prédicats. Le calcul de la sélectivité est basé sur les statistiques. En l'absence de ces dernières, l'optimiseur utilise un mécanisme d'échantillonnage dynamique (paramètre d'initialisation `OPTIMIZER_DYNAMIC_SAMPLING`).

La cardinalité d'une opération du plan d'exécution d'une requête représente l'estimation du nombre de lignes extraites par cette opération. Généralement, la source est une table, une vue, ou le résultat d'une jointure ou d'un opérateur `GROUP BY`. Cette valeur est essentielle pour déterminer le coût des opérations de jointure, de filtre et de tri. Pour chaque colonne, on trouve la relation : *cardinalité* = *sélectivité* × *nbre_total_lignes*. Considérons à titre d'exemple la requête suivante :

```
SELECT adhid, nom FROM Adherent WHERE prenom = 'CELINE';
```

Étant donné que la table contient 24 033 lignes incluant 3 040 prénoms distincts, les indicateurs que l'optimiseur consultera sont les suivants :

```
sélectivité = 1/3040 ⇒ 0,00032
cardinalité = 24033 × 1/3040 ⇒ 7,9
```

Traitement d'une instruction

L'exécution d'une instruction SQL se décompose en différentes phases. Un curseur interne est ouvert (*open*), puis fermé à l'issue du traitement (*close*). Entre ces étapes, les trois phases majeures sont :

- L'analyse syntaxique et sémantique de l'instruction (*parse*), qui vérifie les droits de l'utilisateur, recherche puis élabore éventuellement le plan d'exécution (ensemble d'étapes), charge le plan en mémoire (pour des utilisations ultérieures). L'optimisation est réalisée durant cette phase.
- L'exécution proprement dite de l'instruction (*execute*). Durant cette phase, Oracle applique les étapes du plan précédemment établi.
- L'extraction (*fetch*) d'une (ou de plusieurs) ligne(s).

Pour certaines requêtes de faible ampleur, la phase de *parse* consomme davantage de temps que les deux autres phases. Par ailleurs, si un plan existe déjà en mémoire, il n'est pas regénéré et l'instruction déclenche un *soft parse*. Le cas contraire entraîne un *hard parse* plus coûteux. C'est pour cela qu'exécuter deux fois de suite la même requête entraîne en général un coût inférieur à la seconde exécution.

Deux requêtes identiques (mot à mot) évaluant des valeurs différentes (exemple `age=18` pour la première et `age=20` pour la seconde) peuvent générer deux plans d'exécution distincts. Pour éviter ceci, une variable de lien peut être utilisée dans vos programmes (`age= :v_age` ou `age=v_age` suivant le contexte de programmation) ; voir la section *Variables de lien*. Il est aussi possible d'agir sur le paramètre `CURSOR_SHARING`.

Il est toujours préférable de retourner plusieurs lignes par *fetch*. Pour cela, consulter la section *Comment réaliser des fetchs mutlilignes*.

Configuration de l'optimiseur (les hints)

Un conseil (*hint*) se place dans une instruction sous la forme d'un commentaire (qui n'en est pas un) et impose à l'optimiseur la sélection d'un certain plan d'exécution, en fonction de critères spécifiques.

La séquence de caractères `/*+ indicateur */` indique à l'optimiseur que le commentaire doit être interprété en tant que conseil. Le symbole + doit suivre immédiatement le délimiteur de commentaire sans être précédé d'un espace. La plupart des paramètres des *hints* sont composés du nom des tables (ou alias), de colonnes et d'index.

Le tableau suivant présente l'utilisation de deux *hints* ; le premier (FULL) force le balayage entier de la table et le second (INDEX) impose l'utilisation de l'index associée à la clé primaire (colonne adhid). Dans les deux cas, l'optimiseur avait choisi de lui-même la meilleure stratégie d'accès aux données.

Tableau 12-3 Expansions classiques

Expansion	Opération choisie	Opération forcée
`SELECT a.nom, a.tel FROM Adherentbis a WHERE a.adhid = 20045;`	Index clé primaire.	`SELECT /*+ INDEX(a PK_ADHERENTBIS) */ a.adhid, a.nom, a.tel FROM Adherentbis a WHERE a.tel LIKE '+33%';`
`SELECT a.adhid, a.nom, a.tel FROM Adherentbis a WHERE a.tel LIKE '+33%';`	Parcours entier de la table (*full scan*).	`SELECT /*+ FULL(a) */ a.nom, a.tel FROM Adherentbis a WHERE a.adhid = 20045;`

Il existe un grand nombre de hints, décrits dans le livre *SQL Language Reference* de la documentation officielle. Citons les hints qui privilégient le temps d'exécution, soit global (ALL_ROWS, par défaut), soit au profit des 1, 10, 100 ou 1 000 premières lignes (FIRST_ROWS_*n*).

Vous devez utiliser les conseils avec parcimonie et uniquement après avoir collecté les statistiques et évalué le plan de l'optimiseur sans conseils. Les modifications apportées à la base (structurelles et sur les données) et l'amélioration de performances peuvent les rendre moins pertinents (voire non valides).

Les statistiques destinées à l'optimiseur

Les statistiques qui sont créées par Oracle pour l'optimisation des instructions sont stockées dans le dictionnaire de données (USER_TAB_COLUMNS, USER_TABLES, USER_INDEXES, etc.). Ces statistiques ne doivent pas être confondues avec les statistiques de performances qui se trouvent dans les vues V$.

Les statistiques consignent diverses informations concernant le système (utilisation de la CPU et des entrées-sorties), les tables (volumétrie, taille moyenne des lignes, blocs, etc.), les index (clés, nombre de blocs feuilles, etc.), les colonnes (nombre de valeurs distinctes, nombre de NULL, taille moyenne) et les données de la table (valeurs minimale, maximale et distribution des valeurs).

Toutes ces informations vont servir à Oracle pour décider des algorithmes à utiliser pour générer chaque plan d'exécution. L'optimiseur choisira, pour une requête donnée, le plan d'exécution le moins coûteux.

La commande `ANALYSE` est désormais obsolète et l'utilisation du paquetage `DBMS_STATS` est préconisé pour personnaliser vos collectes. Depuis la version 10*g*, la collecte des statistiques est automatique (par défaut un batch est exécuté entre 22 heures et 6 heures tous les jours).

L'optimiseur considère par défaut que les données de toute colonne sont réparties de façon uniforme (hypothèse de distribution uniforme des valeurs). Ce comportement peut entraîner la génération de plans d'exécution non optimaux en cas de répartition inégale des données. Les statistiques se doivent donc de refléter au mieux le contenu des tables. Si un histogramme est disponible sur une colonne, l'optimiseur l'utilise à la place du nombre de valeurs distinctes.

Les histogrammes

Le mécanisme des histogrammes permet de pallier au mieux le problème de répartition non homogène des données. Le fait de disposer d'histogrammes sur des colonnes contenant des données inégalement réparties (ou des valeurs présentant de grandes variations dans le nombre de doublons) aide l'optimiseur d'instructions à générer de bonnes estimations de sélectivité et à prendre de meilleures décisions concernant l'utilisation des index, les ordres de jointure, les méthodes de jointure, etc.

Les caractéristiques de tous les histogrammes sont stockées dans le dictionnaire des données (`DBA_TAB_HISTOGRAMS`, `DBA_PART_HISTOGRAMS` et `DBA_SUBPART_HISTOGRAMS`). La génération des histogrammes est l'opération la plus consommatrice de ressources lors de la collecte de statistiques.

Pour chaque colonne d'une table, en fonction du nombre de valeurs distinctes, Oracle peut créer deux types d'histogrammes.

- L'histogramme de fréquence où le nombre de valeurs distinctes d'une colonne est inférieur ou égal au nombre d'intervalles. Ce type d'histogramme sera créé si les données comportent moins de 254 valeurs distinctes et que le nombre d'intervalles n'est pas précisé.
- l'histogramme équilibré en hauteur où le nombre d'intervalles est inférieur au nombre de valeurs distinctes d'une colonne.

Les histogrammes ne sont pas utiles pour les colonnes qui n'apparaissent pas dans les clauses `WHERE` ou `JOIN` et celles déclarées avec une contrainte `UNIQUE`.

Les histogrammes sur les colonnes de type chaînes de caractères sont évalués sur les 32 premiers octets de chaque valeur.

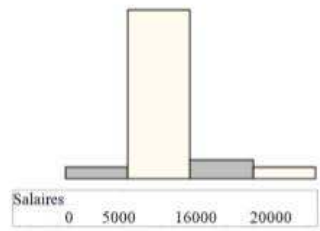
Considérons la table suivante contenant 1 000 lignes. Si Oracle collecte trois valeurs distinctes pour la colonne `compa` (on suppose qu'il n'existe que trois compagnies), alors les plans d'exécution générés considéreront cette répartition (pour la recherche des pilotes d'une compagnie donnée, Oracle s'attend à monter l'équivalent de 333 lignes en mémoire). Lorsqu'une colonne est distribuée de manière homogène, l'histogramme se présente sous la forme d'éléments de même hauteur. À l'inverse, lorsque la distribution est hétérogène, la colonne (ou l'index) est déséquilibrée en l'absence d'histogrammes statistiques.

En revanche, puisque le salaire se répartit inégalement (la plupart des pilotes gagne entre 5 000 et 16 000 €), Oracle générera un histogramme plus précis pour que l'optimiseur ne se base pas sur une répartition homogène et prévoit moins de pilotes à extraire pour des faibles ou des hauts salaires.

Figure 12-2 *Répartition des données dans une colonne*

Pilote

brevet	salaire	compa
PL-1	3400	AF
PL-2	4500	SING
PL-3	9000	AF
PL-4	10000	SING
PL-5	10050	SING
PL-6	16000	AF
PL-7	10060	CAST
PL-8	15000	
...	...	...
PL-1000		CAST



Collecte

La procédure `GATHER_TABLE_STATS` du paquetage `DBMS_STATS` collecte les statistiques sur une table. Les paramètres à renseigner obligatoirement sont le nom du schéma et celui de la table. Plusieurs autres paramètres sont intéressants à préciser :

- `METHOD_OPT` qui permet de prévoir les histogrammes soit par un nombre d'intervalles de l'histogramme (entre 1 et 254), soit par distribution en fonction des valeurs des données de la table.
- `CASCADE` collecte également des statistiques sur les index après avoir examiné la table.
- `ESTIMATE_PERCENT` indique le pourcentage estimé de lignes utilisées pour calculer les statistiques (`NULL` ou par défaut signifie toutes les lignes). Sa valeur peut être comprise entre 0,000001 et 100.
- `STALE_PERCENT` sert à déterminer le seuil à partir duquel les statistiques d'un objet sont considérées comme obsolètes.

Le tableau suivant détaille la collecte des statistiques sur la table `Adherent` du schéma `soutou`. En fonction de la version d'Oracle, le paramètre `METHOD_OPT` n'a pas la même valeur par défaut (en 8*i* et 9*i*, il vaut `FOR ALL COLUMNS SIZE 1` ; à partir de 10*g*, c'est `FOR ALL COLUMNS SIZE AUTO`).

Cette procédure est utile pour les traitements manipulant des tables temporaires que les statistiques automatiques ignorent et qui sont privées d'index.

Tableau 12-4 Collecte des statistiques sur une table

Calcul de statistiques	Commentaires

`BEGIN`
` DBMS_STATS.GATHER_TABLE_STATS(`
`  OWNNAME => 'SOUTOU',`
`  TABNAME => 'ADHERENT',`
`  METHOD_OPT => 'FOR ALL COLUMNS SIZE AUTO`
`  FOR COLUMNS tel SIZE 254',`
`  CASCADE => true);`
`END;`
	Toutes les colonnes de la table seront analysées pour produire des histogrammes qui conviennent le mieux. L'histogramme associé à la colonne `tel` est ici fixé arbitrairement à 254 intervalles.

Après avoir lancé cette procédure, il est possible d'examiner les résultats dans le dictionnaire des données.

Visualisation des statistiques

Les tableaux suivants présentent le détail des statistiques concernant la table des adhérents. La première requête concerne les valeurs des données de toutes les colonnes. On y trouve le nombre de valeurs distinctes et l'intervalle de ces valeurs (en hexadécimal).

Tableau 12-5 Informations sur les colonnes (valeurs)

Requête et résultats

```
SELECT COLUMN_NAME,NUM_DISTINCT,LOW_VALUE,HIGH_VALUE
       FROM USER_TAB_COLUMNS
       WHERE TABLE_NAME='ADHERENT';

COLUMN_NAM NUM_DISTINCT LOW_VALUE           HIGH_VALUE
---------- ------------ -------------------- --------------------
ADHID             24033 C102                 C3033E17
NOM               10738 41414241444C49       5A5953534D414E
PRENOM             3040 41204D41524945       C94C4F444945
CIVILITE              3 44722E               4D722E
DATE_NAIS         16570 77690414010101       786D071A010101
TEL               23286 30202020202020202020 30392038302030362030
```

La requête suivante illustre que la colonne `tel` dispose de valeurs `NULL`. On y trouve également la taille moyenne en octets pour chaque colonne.

Tableau 12-6 Informations sur les colonnes

Requête et résultats

```
SELECT COLUMN_NAME, NUM_NULLS, AVG_COL_LEN, SAMPLE_SIZE
       FROM USER_TAB_COLUMNS
       WHERE TABLE_NAME='ADHERENT';

COLUMN_NAM  NUM_NULLS AVG_COL_LEN SAMPLE_SIZE
---------- ---------- ----------- -----------
ADHID               0           5       24033
NOM                 0           9        5750
PRENOM              0           8        5750
CIVILITE            0           5        5750
DATE_NAIS           0           8       24033
TEL               765          15        5566
```

La requête suivante renseigne, pour chaque colonne, la densité, le type d'histogramme et la date de dernière analyse. Aucun histogramme n'est généré pour les colonnes `adhid` (clé primaire) et `date_nais` du fait de leur très forte sélectivité.

Tableau 12-7 Informations sur les histogrammes

Requête et résultats

```
SELECT COLUMN_NAME,DENSITY,NUM_BUCKETS,LAST_ANALYZED,HISTOGRAM
       FROM USER_TAB_COLUMNS
       WHERE TABLE_NAME='ADHERENT';

COLUMN_NAM    DENSITY NUM_BUCKETS LAST_ANA HISTOGRAM
---------- ---------- ----------- -------- ---------------
ADHID      ,000041609           1 06/05/10 NONE
NOM        ,000472367         254 06/05/10 HEIGHT BALANCED
PRENOM     ,002040816         254 06/05/10 HEIGHT BALANCED
CIVILITE   ,000020666           3 06/05/10 FREQUENCY
DATE_NAIS   ,00006035           1 06/05/10 NONE
TEL        ,000050813         254 06/05/10 HEIGHT BALANCED
```

La densité est calculée à partir de la formule suivante : *1/nombre de valeurs distinctes non nulles*. La colonne `DENSITY` exprime la sélectivité que l'optimiseur évalue pour toute équi-jointure (`a.adhid=p.adhid`) et prédicat d'égalité (`a.adhid=6`). Les valeurs possibles sont situées dans l'intervalle de 0 à 1 (0 si aucune ligne n'est sélectionnée, 1 si toutes le sont). Plus une colonne est sélective, moins l'optimiseur envisage de retourner des lignes à l'évaluation du prédicat. La sélectivité forme une partie importante de l'équation décidant du meilleur chemin.

La colonne NUM_BUCKETS exprime le nombre d'intervalles de valeurs (avec un maximum de 254). Par défaut Oracle a considéré 254 intervalles pour répartir les données des colonnes nom, prenom. Concernant la colonne civilite, 3 valeurs sont possibles ('Mr.', 'Mlle.' ou 'Mme.'), donc 3 intervalles suffisent à modéliser l'histogramme associé.

La colonne HISTOGRAM renseigne à propos du type d'histogramme :

- HEIGHT BALANCED, les valeurs des colonnes sont divisées dans des intervalles d'une manière homogène (concerne les colonnes nom, prenom et tel). On peut constater que le prénom compose la colonne la moins sélective, les plus sélectives étant le numéro d'adhérent et le téléphone.
- FREQUENCY, chaque intervalle contient le nombre d'occurences de cette valeur. Ce type d'histogramme est créé lorsque le nombre de valeurs distinctes est plus petit que le nombre d'intervalles de l'histogramme demandé (ici pour la colonne civilite).

Quand mettre à jour les statistiques ?

La meilleure fréquence d'actualisation des statistiques est *dès que nécessaire* ! Ce sont souvent des opérations très coûteuses, et il n'est pas envisageable d'actualiser les statistiques à chaque modification des tables. Par ailleurs, les statistiques deviennent obsolètes après de nombreuses mises à jour. Des statistiques perimées nuisent au plan d'exécution et peuvent provoquer de réels écarts de performances. Il est courant de les calculer toutes les semaines pour des bases en production voire toutes les nuits lorsque de forts mouvements peuvent avoir lieux au cours d'une journée.

Dans tous les cas, l'actualisation des statistiques s'impose après des modifications fréquentes et significatives au cours de la journée, une migration, une importation conséquente ou une modification du modèle physique (changement d'un paramètre de stockage, création d'index, partitionnement, réorganisation, etc.).

D'autres procédures jouent un rôle similaire ; citons GATHER_INDEX_STATS, GATHER_DATABASE_STATS et GATHER_SCHEMA_STATS du paquetage DBMS_STATS qui permettent de récolter les statistiques au niveau d'une base, d'un schéma et des index. Le module DBMS_SCHEDULER peut aussi être configuré pour exécuter l'action GATHER_STATS_JOB qui collectera des statistiques en mode *batch*.

Si les statistiques ne sont pas utilisées, Oracle collecte des statistiques partielles en fonction du paramètre OPTIMIZER_DYNAMIC_SAMPLING. Ce mécanisme est intéressant pour des tables à la forte volatilité.

Outils de mesure de performances

Cette section présente les principales méthodes basiques qui vous permettront de mesurer les performances de vos requêtes et évaluer différents scénarios.

- La visualisation des plans d'exécution dans une session SQL*Plus à l'aide de la commande `AUTOTRACE`, de l'instruction `EXPLAIN PLAN`, de l'événement 10046 ou par l'utilitaire `tkprof`.
- L'analyse de certaines vues du dictionnaire des données, principalement `V$SQLAREA` qui fournit des statistiques sur chaque instruction SQL en memoire, parsée et prête à l'exécution. Depuis la version 9*i*, les vues `V$SQLSTATS` et `V$SQL_PLAN` décrivent les plans d'exécution. Chaque *hard parse* met à jour ces vues et les mécanismes de *monitoring* sont majoritairement basés sur ces vues.
- L'utilisation du paquetage `DBMS_APPLICATION_INFO` qui peut être utile aux développeurs pour tracer et mieux contrôler leurs transactions.
- L'utilitaire `runstats` de Thomas Kyte (créateur et animateur du célèbre site *http://asktom.oracle.com*) permet de comparer deux implémentations.

Nous n'étudierons pas les paramètres d'initialisation de la base qui sont modifiés par `ALTER SESSION` (tâche du développeur) ou par `ALTER SYSTEM` (tâche du DBA). Ces paramètres conditionnent le plan d'exécution de toute requête. Vous devrez veiller à ce que vos environnements de test et de production soient comparables de ce point de vue.

Visualisation des plans d'exécution

Un plan d'exécution est le résultat de l'action de l'optimiseur qui présente au moteur d'exécution les opérations qu'il doit effectuer de la manière la plus efficace. Chaque plan est décrit sous la forme d'un arbre contenant les informations suivantes :

- l'ordre des tables auxquelles l'instruction fait référence ;
- une méthode d'accès pour chaque table mentionnée dans l'instruction ;
- une méthode de jointure pour chaque table affectée ;
- des opérations sur les données (filtrage, tri ou agrégation).

Les sources

Plusieurs sources peuvent être utilisées afin d'extraire un plan d'exécution :

- La table `PLAN_TABLE` (utilisée avec `AUTOTRACE` ou `EXPLAIN PLAN` sous SQL*Plus).
- Des vues du dictionnaire de données `V$SQL_PLAN`, `V$SQL_PLAN_MONITOR` (à partir de la version 11*g*), `DBA_HIST_SQL_PLAN` (référentiel AWR), `STATS$SQL_PLAN` (outil Statspack).

- La base de gestion SMB (*SQL Management Base*) qui stocke le journal des instructions, les historiques de plan, les *SQL Plan Baselines*, ainsi que les profils SQL.
- Des fichiers de trace de l'événement 10053 par un *dump* de l'état des processus et ceux générés par `DBMS_MONITOR`.

Bien que les commandes `SET AUTOTRACE` et `EXPLAIN PLAN` affichent un plan d'exécution que l'optimiseur est susceptible d'utiliser, la vue `V$SQL_PLAN` contient le plan réellement employé.

L'infrastructure du référentiel AWR et les rapports Statspack contiennent les plans des instructions les plus coûteuses en ressources.

L'affichage

Le package `DBMS_XPLAN` fournit cinq tables fonction.

- `DISPLAY` qui met en forme et affiche le contenu d'une table `PLAN_TABLE`.
- `DISPLAY_AWR` qui met en forme et affiche le contenu du plan d'exécution d'une instruction SQL stockée dans le référentiel AWR.
- `DISPLAY_CURSOR` qui met en forme et affiche le contenu du plan d'exécution d'un curseur chargé.
- `DISPLAY_SQL_PLAN_BASELINE` qui affiche un ou plusieurs plans d'exécution pour l'instruction SQL indiquée.
- `DISPLAY_SQLSET` qui met en forme et affiche le contenu du plan d'exécution des instructions stockées dans un ensemble *SQL Tuning Set* (STS).

L'arbre

Le plan d'exécution reflète une structure d'arbre dont chaque étape compose un nœud (ou une feuille) et dont la première étape à exécuter se trouve au niveau le plus bas et le plus à gauche (ou en haut suivant l'inspiration du dessinateur de l'arbre). À chaque étape est associé un coût qui contient le coût de toutes les étapes descendantes.

L'exemple suivant présente un plan d'exécution prévisionnel d'une jointure entre trois tables (extraction de l'identité des pratiquants de handball nés en 1995). Outre la visualisation de l'arbre, le plan peut éventuellement contenir des informations concernant le partitionnement et l'exécution en mode parallèle.

Tableau 12-8 Plan d'exécution d'une jointure

Requête et plan d'exécution

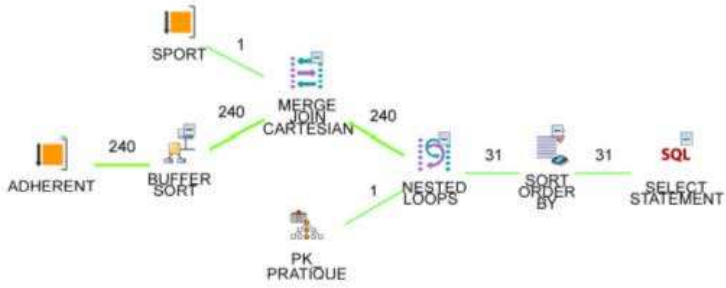
```
SELECT  a.adhid, a.prenom, a.nom
  FROM  Adherent a, Pratique p, Sport s
  WHERE TO_CHAR(DATE_NAIS,'YYYY')='1995' AND a.adhid = p.adhid
  AND   s.spid = p.spid AND s.splibelle = 'Hand-ball'
  ORDER BY nom;

-------------------------------------------------------------------------------
| Id  | Operation             | Name         | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |              |    31 |  2511 |    73   (3)|
|   1 |  SORT ORDER BY        |              |    31 |  2511 |    73   (3)|
|   2 |   NESTED LOOPS        |              |    31 |  2511 |    72   (2)|
|   3 |    MERGE JOIN CARTESIAN|             |   240 | 13200 |    72   (2)|
|*  4 |     TABLE ACCESS FULL | SPORT        |     1 |    25 |     3   (0)|
|   5 |     BUFFER SORT       |              |   240 |  7200 |    69   (2)|
|*  6 |      TABLE ACCESS FULL| ADHERENT     |   240 |  7200 |    69   (2)|
|*  7 |    INDEX UNIQUE SCAN  | PK_PRATIQUE  |     1 |    26 |     0   (0)|
-------------------------------------------------------------------------------
```

La mise en œuvre prévisionnelle de cette jointure fait intervenir deux algorithmes distincts : un tri fusion (*sort* puis *merge*) et une boucle imbriqué (*nested loops*). La figure 12-8 illustre l'arbre associé à ce plan d'exécution. Les nombres sur les liens correspondent aux lignes traitées. L'ordre des opérations est le suivant : 4, 6, 5, 3, 7, 2, 1 puis 0. Le coût principal (69) correspond au parcours entier de la table `Adherent` réalisé lors de la deuxième étape. À ce coût, on ajoute 3 pour l'extraction du code du sport correspondant au handball. Ainsi le coût de la troisième étape est égal à la somme des deux premières, etc.

La commande SET AUTOTRACE

La commande SQL*Plus `SET AUTOTRACE` existe depuis la version 7.3 ; elle permet d'obtenir après l'exécution d'une instruction, le plan d'exécution ainsi que des statistiques SQL.

Figure 12-3 Plan d'exécution



Les statistiques fournies par cette commande sont extraites de la vue `V$SESSTAT`.

La syntaxe de cette commande est la suivante. Si les deux dernières options sont omises, les statistiques et plans sont affichés par défaut.

```
SET AUTOT[RACE] {OFF | ON | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]
```

- `OFF` : désactive le suivi d'exécution automatique.
- `ON` : active le suivi d'exécution automatique.
- `TRACEONLY` : active le suivi d'exécution automatique des instructions SQL en occultant le résultat des instructions mais en exécutant toutefois la requête.
- `EXPLAIN` : affiche les plans d'exécution mais pas les statistiques.
- `STATISTICS` : affiche les statistiques mais pas les plans d'exécution.

Sous Oracle9*i*, vous devrez créer au préalable la table `PLAN_TABLE` via le script `utlxplan.sql` situé dans le répertoire *`ORACLE_HOME`*`/rdbms/admin`. Dans les autres cas, vous n'avez qu'à affecter le rôle `plustrace` à l'utilisateur qui désire bénéficier de l'autotraçage. Si ce rôle est absent, créez-le (sous `SYS AS SYSDBA`) à l'aide du script `plustrce.sql` situé dans le répertoire *`ORACLE_HOME`*`/sqlplus/admin`.

Pour vous connectez sous SQL*Plus : `connect SYS/mot_de_passe AS SYSDBA` puis `start ?\sqlplus\admin\plustrce.sql` (le symbole `?` sera automatiquement substitué par le chemin *ORACLE_HOME*).

Le tableau suivant présente le résultat et le plan d'exécution prévisionnel de la requête retournant le numéro, prénom et nom des femmes nées en mai 1995. Le numéro du plan d'exécution (*hash value*) est indiqué avant son détail.

Tableau 12-9 Plan d'exécution complet

Requête et plan d'exécution

```
SET AUTOTRACE ON
SELECT  adhid, prenom, nom FROM Adherent
  WHERE TO_CHAR(DATE_NAIS,'YYYY')='1995'
  AND   TO_CHAR(DATE_NAIS,'MM')='05'
  AND   civilite = 'Mme.'
  ORDER BY nom;

    ADHID PRENOM                         NOM
---------- ------------------------------ -------------------------
     21311 CHANTAL                        FOULON
     19600 DENISE                         LANIESSE
     …

Plan d'exécution
----------------------------------------------------------
Plan hash value: 1854364040
-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |     1 |    35 |    70   (3)| 00:00:0
|   1 |  SORT ORDER BY     |          |     1 |    35 |    70   (3)| 00:00:0
|*  2 |   TABLE ACCESS FULL| ADHERENT |     1 |    35 |    69   (2)| 00:00:0
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("CIVILITE"='Mme.' AND
              TO_CHAR(INTERNAL_FUNCTION("DATE_NAIS"),'YYYY')='1995' AND
              TO_CHAR(INTERNAL_FUNCTION("DATE_NAIS"),'MM')='05')
```

Le plan est présenté sous forme d'étapes (colonne `Operation`) imbriquées. Chaque étape retourne un ensemble de lignes (prévisionnel) qui sont utilisées à l'étape suivante jusqu'à l'extraction finale. Un ensemble de lignes peut provenir d'une table, d'une vue ou du résultat d'une jointure ou d'un regroupement. Le nom de la table (ou vues, colonne `Name`) est disposé en regard de la méthode d'accès (jointure, filtre, tri ou fonction d'agrégat). Les éventuels prédicats de chaque opération sont précisés par la suite. Le temps prévu (`Time`) est indiqué au format `HH:MI:SS`. Le coût prévu (`Cost`) est détaillé par étape.

Le plan d'exécution présenté peut ne pas être le plan utilisé. Les causes de cette différence sont principalement la présence de variables de lien (*bind variables*) dans l'instruction et l'obsolescence des statistiques. En revanche, les statistiques chiffrées de `AUTOTRACE` reflètent la réalité de l'instruction.

Le tableau suivant détaille les statistiques générées lors de l'exécution de la requête.

Tableau 12-10 Détails des statistiques issues de SET AUTOTRACE

Statistiques		Commentaires
Statistiques		

603	recursive calls	Nombre d'instructions internes.
0	db block gets	Nombre de blocs en mémoire pour une modification.
346	consistent gets	Nombre de blocs extraits de la mémoire.
196	physical reads	Nombre de blocs extraits du disque.
0	redo size	Taille (en octets) utilisée dans le buffer `redo log`.
785	bytes sent via SQL*Net to client	Nombre d'octets envoyés au serveur.
416	bytes received via SQL*Net from client	Nombre d'octets retournés au client.
2	SQL*Net roundtrips to/from client	Nombre d'allers-retours entre le client et le serveur.
31	sorts (memory)	Nombre d'opérations de tri en mémoire.
0	sorts (disk)	Nombre d'opérations de tri sur disque.
8	rows processed	Nombre de lignes extraites.

Quelques informations complémentaires à propos de ces indicateurs :

- Les appels internes, `recursive calls`, incluent la construction du plan lui-même, les exécutions des déclencheurs, des allocations mémoire pour les tris, des recherches et mises à jour du dictionnaire des données, etc. Ce nombre passe généralement à 0 à la prochaine exécution de la même requête (le plan d'exécution est utilisé et il n'a pas à être reconstruit).
- `redo size` est nul en principe pour un `SELECT` qui ne concerne pas les fichiers de journalisation, `redo log`, concernés par toute mise à jour, et utilisés au cours d'une restauration (*recovery*).
- `db block gets` concerne les blocs lus en mode `CURRENT` et non pas en mode lecture consistante. Les instructions `UPDATE` et `DELETE` ont besoin d'y accéder pour effectuer les mises à jour.
- `consistent gets` concerne les blocs en mode lecture consistante ; les instructions `SELECT` sont concernées de même que les blocs d'annulation (*rollback segments*).
- `physical reads` inclut aussi les lectures dans le cache du système de gestion de fichiers du système d'exploitation.

Il faut additionner `db block gets` et `consistent gets` pour obtenir le nombre de *buffers* lus logiquement.

Les tris doivent de préférence être effectués dans la mémoire plutôt que sur disque.

Instruction EXPLAIN PLAN

L'instruction EXPLAIN PLAN permet de générer un plan prévisionnel sans exécuter l'instruction. En fonction des divers paramétrages (de session, de l'instance) et du moment, l'exécution réelle de la requête pourra toutefois s'effectuer suivant un plan différent de celui hypothétiquement calculé.

Vous devez disposer d'une table qui contiendra le résultat des plans générés. Créez la table PLAN_TABLE en exécutant le script utlxplan.sql situé dans *ORACLE_HOME*/rdbms/admin. Vous pouvez utiliser à présent EXPLAIN PLAN avec la syntaxe suivante.

```
EXPLAIN PLAN
   [ SET STATEMENT_ID = chaine_caractere ]
   [ INTO [ schema. ] table [ @dblink ] ]
   FOR instruction_SQL ;
```

- SET STATEMENT_ID désigne l'identifiant du plan dans la table *plan table* (par défaut le plan de la dernière requête exécutée sera considéré).
- INTO désigne la table *plan table* (par défaut celle générée par le script d'Oracle : PLAN_TABLE).
- *instruction_SQL* désigne l'instruction à évaluer.

Afin d'obtenir le plan d'exécution de la jointure précédente, il convient de lancer le script suivant. Pour visualiser ce plan d'exécution, vous devez écrire une requête ou utiliser la procédure DISPLAY du paquetage DBMS_XPLAN (recommandé pour obtenir l'indentation du résultat qui permet d'interpréter le plan, car les étapes ne s'exécutent pas indépendamment les unes des autres).

Tableau 12-11 Obtention d'un plan d'exécution prévisionnel

Plan d'exécution d'une requête

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'exemple1' FOR
  SELECT a.adhid, a.prenom, a.nom
         FROM Adherent a, Pratique p, Sport s
         WHERE TO_CHAR(DATE_NAIS,'YYYY')='1995'
         AND  a.adhid = p.adhid AND  s.spid = p.spid
         AND  s.splibelle = 'Hand-ball' ORDER BY nom;

SELECT * FROM TABLE
              (DBMS_XPLAN.DISPLAY('PLAN_TABLE', 'exemple1', 'TYPICAL', NULL));
```

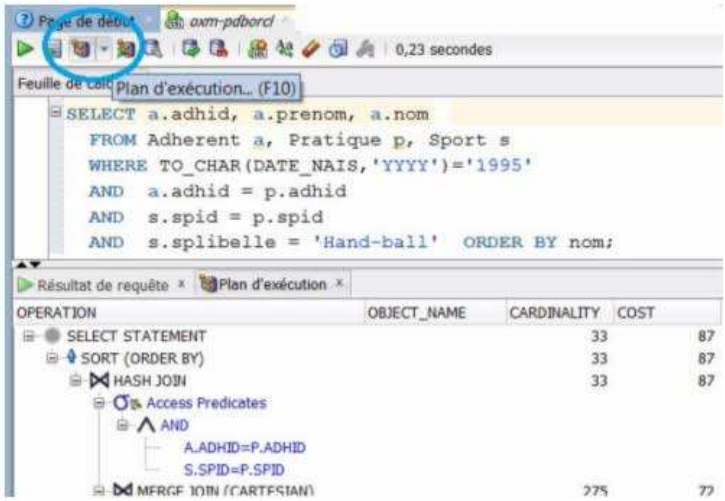
Vous devrez vérifier que le nombre de lignes (Rows) soit en adéquation avec le nombre réel de lignes ramenées car il s'agit d'une estimation basée sur les statistiques.

La commande EXPLAIN PLAN ne gère pas les conversions implicites de variables attachées de type DATE et peut afficher des plans différents que ceux réellement exécutés du fait de l'utilisation de variables de lien (*bind variables*).

Les consoles

Selon la version d'Oracle, les consoles d'administration peuvent disposer d'un onglet (dans la version 11*g*, il se nomme `Schema/Feuille de calcul SQL...`) pour visualiser les plans d'exécution. Pensez à donner à l'utilisateur les prérogatives nécessaires (par exemple, `SELECT ANY DICTIONARY`). L'outil SQL Developer fournit depuis longtemps un aperçu graphique d'un plan d'exécution :

Figure 12-4 *Plan d'exécution avec SQL Developer*



L'outil tkprof

L'outil tkprof n'est pas une commande SQL ou SQL*Plus mais un exécutable du système d'exploitation. Il utilise les fichiers de trace d'Oracle et fournit des informations détaillées (sous forme de fichiers texte) à propos des sessions. Les informations extraites concernent les temps des opérations *parse*, *execute* et *fetch*, le nombre de lignes traitées et les plans d'exécution réels.

La procédure à suivre pour l'utilisation de cet outil est la suivante :

- positionnement d'un certain nombre de paramètres affectant le traçage (statistiques et localisation du fichier de trace) ;
- activation du traçage de la session (`ALTER SESSION...` ou `DBMS_SESSION.SET_SQL_TRACE...`) ;
- exécution de la session dont les instructions sont désormais tracées ;

- désactivation du traçage (`ALTER SESSION...` ou `DBMS_SESSION.SESSION_TRACE_DISABLE...`);
- exécution de l'utilitaire tkprof afin de produire un rapport basé sur le fichier des traces.

Positionnement des paramètres

Le paramètre `TIMED_STATISTICS` doit être positionné à vrai au niveau de l'instance (fichiers de configuration ou par le biais de la commande `ALTER SYSTEM`). Au niveau d'une session seule l'instruction `ALTER SESSION SET TIMED_STATISTICS = TRUE;` peut suffire.

Assurez-vous de disposer d'une taille suffisante lors de la création du fichier de trace (exemple pour 1 Mo : `ALTER SESSION SET MAX_DUMP_FILE_SIZE = 1000000;`).

L'emplacement du répertoire `USER_DUMP_DEST` est déterminé au niveau de l'instance dans le fichier d'initialisation et il ne peut être modifié que par le DBA et ce, pour toute l'instance. La recherche du nom du fichier de trace (de la forme *nom_instance_ORA_numero*.trc) associé à la session en cours ne peut s'effectuer qu'en interrogeant des vues dynamiques du dictionnaire des données.

Tableau 12-12 Recherche du répertoire et du nom du fichier trace

Extraction du nom du ficher trace de la session courante

```
SQL> SELECT value AS "Fichier trace"
     FROM   v$diag_info
     WHERE  name = 'Default Trace File';

Fichier trace
-------------------------------------------------------------
C:\APP\CSOUTOU\diag\rdbms\orcl\orcl\trace\orcl_ora_4464.trc
```

Activation et désactivation du traçage

Vous devrez disposez de la prérogative `ALTER SESSION` pour activer le traçage dans une session. Le tableau suivant présente les instructions à exécuter dans une session SQL*Plus. La deuxième écriture présente l'avantage de pouvoir se trouver à l'intérieur d'une procédure cataloguée.

Tableau 12-13 Activation du traçage sous SQL*Plus

Avant 10*g*	Après 10*g*
`ALTER SESSION SET sql_trace=TRUE;`	`BEGIN DBMS_SESSION.SET_SQL_TRACE(sql_trace => true); END;`

En plaçant, de la même manière, à FALSE le paramètre sql_trace, la session n'est plus tracée. Vous devez ensuite choisir au niveau de la session d'inclure ou non les attentes et les variables de lien. L'instruction PL/SQL suivante positionne à vrai ces deux paramètres :

```
DBMS_SESSION.SESSION_TRACE_ENABLE(waits => true, binds => true);
```

Exécution de tkprof

Une fois vos requêtes ou vos procédures exécutées et la trace désactivée, vous pouvez faire appel à tkprof en précisant principalement le nom du fichier résultat et le chemin vers le fichier de trace d'Oracle. Beaucoup de paramètres sont disponibles ; nous ne les détaillerons pas. Dans l'exemple suivant, tkprof crée le fichier matrace.txt dans le répertoire C:\Temp. Les options choisies ici sont sys=no qui évite de tracer les instructions internes et waits=yes qui prend en compte les attentes.

```
tkprof C:\app\soutou\diag\rdbms\bdcs11gr2\bdcs11gr2\trace\BDCS11GR2_
ORA_5928.trc C:\Temp\ma_trace.txt sys=no waits=yes
```

Le tableau suivant présente la première partie du fichier de résultat (reformaté pour l'alléger) qui correspond à la trace de la requête qui extrait, par ordre alphabétique, l'identité des adeptes féminines du golf nées entre 1955 et 1994 et disposant d'un numéro de téléphone portable.

Tableau 12-14 Résultat (1/2) de tkprof

Première partie du fichier de sortie

```
TKPROF: Release 11.2.0.1.0 - Development on Lun. Mai 10 12:08:50 2010
Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.

Trace file: c:\app\soutou\diag\rdbms\bdcs11gr2\bdcs11gr2\trace\BDCS11GR2_ORA_5928.trc
Sort options: default
*** SESSION ID:(29.847) 2010-05-10 12:04:08.343

SELECT a.adhid, a.prenom, a.nom FROM Adherent a, Pratique p, Sport s
  WHERE TO_NUMBER(TO_CHAR(DATE_NAIS,'YYYY')) < 1995
  AND TO_NUMBER(TO_CHAR(DATE_NAIS,'YYYY')) > 1954
  AND NOT (CIVILITE = 'Mr.') AND SUBSTR (TEL,1,2) = '06'
  AND  a.adhid = p.adhid  AND  s.spid = p.spid  AND  s.splibelle = 'Golf'
  ORDER BY nom

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        8      0.06       0.08          0        250          0         104
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       10      0.07       0.09          0        250          0         104
...
```

Alors que EXPLAIN PLAN présentait des valeurs estimées (prédictives), cette trace indique par exemple que la requête a effectivement durée 9 centièmes de secondes. La signification de ces données est la suivante :

- `count` : nombre d'appels d'une phase (*parse*, *execute* et *fetch*) ; ici 8 opérations *fetch* ont été nécessaires pour extraire 104 lignes contenues dans 250 blocs.
- `cpu` : temps processeur (en secondes) ; s'il est à zéro, vous n'avez pas dû positionner à `TRUE` le paramètre `TIMED_SATISTICS`.
- `elapsed` : temps incluant les attentes qui ne sont ni CPU, ni entrée-sorties (verrouillages par exemple) ; s'il est à à zéro, voir l'item `cpu`…
- `disk` : nombre de lectures physiques (ici aucune car la requête avait déjà été exécutée, donc le résultat monté en SGA).
- `query` : nombre de blocs (lecture consistante, équivalent à `consistent gets` de `EXPLAIN PLAN`).
- `current` : nombre de blocs en mode `CURRENT` (équivalent à `db block gets` de `EXPLAIN PLAN`).
- `rows` : nombre de lignes traitées (n'inclut pas les sous-requêtes) ; pour les extractions, ce nombre se trouve dans la ligne `Fetch` ; pour les mises à jour, il se situe dans la ligne `Execute`.

Plutôt que de focaliser sur `cpu` et `elapsed`, surveillez `disk` et `query`. En effet, trouver systématiquement un grand nombre de lectures physiques pour la même extraction peut être inquiétant et peut mener à un travail sur la mémoire (*tuning*). De même, trouver un très grand nombre de blocs manipulés pour un faible nombre de lignes retournées peut imposer de réorganiser les données (index, *cluster*, etc.).

Plus le temps `cpu` d'une requête diffère du temps d'exécution avec attentes (`elapsed`), plus il est probable que certaines contentions du système d'exploitation existent (CPU trop sollicitée, beaucoup d'entrée-sorties, répartition physique mal adaptée, verouillages, etc.).

Le nombre de blocs lus (`query+current`) est le facteur influençant les autres, notamment `rows` ; il n'est pas anormal que ce dernier soit égal à 10^6 si 10^6 blocs sont également lus. En revanche, vous devrez peut être agir si l'extraction lit 10^6 blocs pour ne ramener que quelques lignes.

Tous ces facteurs sont à diviser par le nombre de fois où la requête a été exécutée (`count`). Traiter un million de blocs en un appel est bien plus performant que de les traiter en 10 000 appels.

La seconde partie du fichier de sortie concerne le plan d'exécution et les attentes.

Tableau 12-15 Résultat (2/2) de tkprof

Seconde partie du fichier de sortie

```
...
Rows     Row Source Operation
-------  ---------------------------------------------------
    104  SORT ORDER BY (cr=250 pr=0 pw=0 time=0 us cost=777 size=2892720 card=42540)
    104   HASH JOIN  (cr=250 pr=0 pw=0 time=717601 us cost=89 size=2892720 card=42540)
      1    TABLE ACCESS FULL SPORT (cr=7 pr=0 pw=0 time=0 us cost=3 size=1144 card=104)
   3281    HASH JOIN  (cr=243 pr=0 pw=0 time=123756 us cost=85 size=279756 card=4908)
   3195     TABLE ACCESS FULL ADHERENT (cr=190 pr=0 pw=0 time=35517 us cost=69 size=156555
             card=3195)
  27011     TABLE ACCESS FULL PRATIQUE (cr=53 pr=0 pw=0 time=74659 us cost=15 size=216088
             card=27011)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                        8        0.00          0.00
  Disk file operations I/O                         1        0.01          0.01
  asynch descriptor resize                         2        0.00          0.00
  SQL*Net message from client                      8        0.11          0.77
```

La signification de chaque ligne (*row source*) du plan d'exécution est la suivante. Afin d'obtenir ces détails, la trace de la session doit être désactivée avant d'exécuter le rapport (`cr` désigne les lectures consistantes, `r` les lectures physiques et `w` les écritures physiques). On retrouve à chaque étape, le temps d'exécution en microsecondes (`time`), le coût (`cost` qui engloble le coût des étapes précédentes), la taille en octets (`size`) et le nombre de lignes traitées (`card`). Concernant cette requête, on peut dire que le coût principal est le tri final de la jointure qui concerne 250 blocs.

Bilan

Ne tenez pas compte uniquement des plans d'exécution lors de l'analyse d'une trace *tkprof*.

L'étude du nombre de blocs est primordiale ; à ceux qui déclarent « *Le select met 50 secondes à répondre, pourtant il n'y a pas d'access full !* », on peut répondre qu'un access *full* ne rime pas avec requête mal écrite. Au contraire, quel que soit le volume d'une table, Oracle se dispense en fait souvent des index (à plus forte raison quand le volume de la table est faible).

Un index doit être examiné de près (à créer s'il n'existe pas) et modifié si le nombre de blocs traités est important par rapport aux lignes retournées. Plus le ratio (`query+current`)/`rows` est important, plus il faut surveiller l'instruction SQL.

Assurez-vous que vos configurations de tests et de production soient identiques sinon les plans d'exécution seront différents. De même, si vous tracez un programme, créez un index, puis analysez le rapport ; les *explains* peuvent utiliser cet index et vous ne serez pas dans les mêmes conditions.

Enfin, il est possible d'exécuter `tkprof` même si l'application n'est pas terminée. Bien que certaines données concernant les lignes, blocs et temps n'apparaissent pas, les plans d'exécution sont disponibles.

Paquetage DBMS_APPLICATION_INFO

Disponible depuis la version 9*i release* 2, le paquetage `DBMS_APPLICATION_INFO` sert à enregistrer des informations de surveillance utiles. Ces informations sont stockées dans les vues `V$SESSION` et `V$SQLAREA`. Ce paquetage permet de répondre aux problématiques ou scénarios suivants :

- Quelle est l'application qui exécute la session, quelle est la partie du code exécutée ou encore combien reste-t-il de temps avant de terminer l'opération en cours ?
- Une procédure s'exécute et il est nécessaire d'éviter des exécutions simultanées (une procédure doit être en mesure de savoir si une autre est déjà en cours d'exécution).
- Plusieurs procédures effectuent des mises à jour qui peuvent entraîner des erreurs dues aux verrous (`ORA-00054 - resource busy` ou `ORA-00060 - deadlock detected`).

Initialisation des informations

Afin d'informer d'autres applicatifs, votre application peut initialiser :

- le nom du module (en général le nom de l'application, de la procédure, du déclencheur, etc.) à l'aide de la procédure `SET_MODULE` ;
- le nom de l'action (en général le nom de la transaction, la valeur d'un compteur, etc.) à l'aide de la procédure `SET_ACTION` ;
- des informations à propos du client (de toute nature) à l'aide de la procédure `SET_CLIENT_INFO`.

Le tableau suivant décrit une procédure (qui ajoute une ligne à la table `Sport` et compte ensuite le nombre de sports) et informe de son exécution en utilisant le paquetage `DBMS_APPLICATION_INFO`. L'extraction des informations est réalisée en interrogeant la vue `V$SESSION` en précisant qu'il s'agit de la session en cours. Pour cela, il faut utiliser la vue `V$MYSTAT` qui concerne les statistiques de la session en cours et tester le contexte d'exécution à l'aide de l'identifiant de la session auditée (`audsid`).

Tableau 12-16 Utilisation de DBMS_APPLICATION_INFO

Procédure initialisant les informations

```
CREATE PROCEDURE ajoute_sport
 (v_spid NUMBER, v_splibelle VARCHAR) AS
  v_nbrsport NUMBER :=0;
BEGIN
  DBMS_APPLICATION_INFO.SET_CLIENT_INFO
   (client_info => 'client Web');
  DBMS_APPLICATION_INFO.SET_MODULE
   (module_name => 'ajoute_sport',
    action_name => NULL);
-- Transaction
  INSERT INTO Sport (spid, splibelle)
        VALUES (v_spid, v_splibelle);
  COMMIT;
  SELECT COUNT(*) INTO v_nbrsport
       FROM Sport;
  DBMS_APPLICATION_INFO.SET_ACTION
    (action_name => 'Nombre de sports : '
                  ||v_nbrsport);
END;
```

Interrogation du dictionnaire

```
SELECT sid, serial#, module,
       action, client_info
  FROM V$SESSION
  WHERE sid  =
     (SELECT DISTINCT sid
             FROM V$MYSTAT)
  AND audsid =
     SYS_CONTEXT('userenv',
                 'sessionid');
```

```
-- résultat de la requête
       SID    SERIAL# MODULE       ACTION                CLIENT_INFO
---------- ---------- ------------ --------------------- ---------------
        19        111 ajoute_sport Nombre de sports : 13 client Web
```

En principe, il faut réinitialiser le nom du module, de l'action et les informations du client à la fin de la transaction de sorte que ces données ne caractérisent pas toutes les transactions suivantes de la session. L'appel aux procédures suivantes : `SET_MODULE(NULL, NULL)` et `SET_CLIENT_INFO(NULL)` du paquetage `DBMS_APPLICATION_INFO` doivent donc précéder le `END` final.

Lecture des informations

Il est possible d'extraire le nom du module, de l'action et les éventuelles informations du client à l'aide de `READ_MODULE`, `READ_ACTION` et `READ_CLIENT_INFO` du paquetage `DBMS_APPLICATION_INFO`. Le tableau suivant décrit un bloc qui extrait les informations de la session en cours.

Tableau 12-17 Extraction d'informations avec DBMS_APPLICATION_INFO

Procédure qui extrait les informations	Résultat

DECLARE
v_client_info VARCHAR2(64);
v_module VARCHAR2(48);
v_action VARCHAR2(32);
BEGIN
DBMS_APPLICATION_INFO.READ_MODULE(v_module,v_action);
DBMS_APPLICATION_INFO.READ_CLIENT_INFO(v_client_info);
DBMS_OUTPUT.PUT_LINE
(v_module||'/'||v_action||'/'||v_client_info);
END;
	ajoute_sport/Nombre de sports : 13/client Web

Contrôle de la concurrence

Les informations du DBMS_APPLICATION_INFO permettent aussi de contrôler la concurrence des procédures cataloguées. En effet, au lieu de coder l'exclusivité des processus par des verrous ou par une valeur dans une table (ce qui posera toujours problème si une session est tuée ou finit anormalement), il suffit de consulter la vue V$SESSION pour constater si une procédure donnée est en cours ou terminée.

Le tableau suivant décrit la fonction qui s'assure de l'exécution exclusive des procédures p1 et p2. Le raisonnement vaut également si p1 et p2 ne font qu'une (on interdit alors à la procédure d'être multisession).

Tableau 12-18 Contrôle de la concurrence d'exécution

Fonction et les deux procédures

```
CREATE FUNCTION verifie_exec(v_module IN VARCHAR2) RETURN BOOLEAN IS
  v_nbre INTEGER;
BEGIN
  SELECT 1 INTO v_nbre FROM V$SESSION
         WHERE module = v_module AND ROWNUM = 1;
  RETURN(FALSE);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
       RETURN(TRUE);
END;
```

```
CREATE PROCEDURE p1 IS
BEGIN
 IF NOT verifie_exec('p2') THEN
    DBMS_OUTPUT.PUT_LINE
      ('p2 est en cours d''execution...');
    RETURN;
 END IF;
```

```
CREATE PROCEDURE p2 IS
BEGIN
 IF NOT verifie_exec('p1') THEN
    DBMS_OUTPUT.PUT_LINE
      ('p1 est en cours d''execution...');
    RETURN;
 END IF;
```

Tableau 12-18 Contrôle de la concurrence d'exécution *(suite)*

Fonction et les deux procédures	
DBMS_APPLICATION_INFO.SET_MODULE ('p1','ajout adherent'); -- transaction ajout adherent … DBMS_APPLICATION_INFO.SET_MODULE (NULL, NULL); END;	DBMS_APPLICATION_INFO.SET_MODULE ('p2','ajout adherent exterieur'); -- transaction ajout adherent exterieur DBMS_APPLICATION_INFO.SET_MODULE (NULL, NULL); END;

L'utilitaire runstats de Tom Kyte

Célèbre gourou Oracle et fondateur du site *http://asktom.oracle.com*, Thomas Kyte est aussi l'auteur du paquetage runstats (`runstats_pkg`) disponible sur *http://appsdba.com/techinfo/runstats.htm*. Il permet de comparer deux solutions d'implémentation différentes (requête, instruction, procédure, etc.) en se basant sur un certain type de verrous d'Oracle : les *latches*.

Ces verrous sont des mécanismes de sérialisation de bas niveau qui protègent les structures de mémoire partagée dans la SGA. Les *latches* préservent la mémoire accédée par plusieurs transactions concurrentes, en interdisant la modification de la zone mémoire en question par plusieurs process.

Le paquetage de Tom Kyte réalise un calcul différentiel des statistiques cumulatives contenues dans les vues `V$SYSSTAT` et `V$SESSTAT`. En comparaison avec les autres utilitaires, runstats permet de prévoir la solution qui conviendra le mieux en cas de montée en charge du volume des données.

Il est préférable d'utiliser ce paquetage en travaillant seul sur la base de sorte à ce que les mesures ne soient pas pertubées par d'autres transactions. Dans un mode mono-utilisateur, vous devrez privilégier la solution minimisant le temps d'exécution. En revanche, dans un mode multi-utilisateur, vous devrez préférer la solution minimisant les *latches*.

Préalables

Sous `SYS AS SYSDBA`, attribuez à l'utilisateur qui désire exécuter le paquetage runstats le droit `CREATE VIEW` et `SELECT` sur les vues `SYS.V_$TIMER`, `SYS.V_$MYSTAT`, `SYS.V_$LATCH` et `SYS.V_$STATNAME`. Exécutez ensuite le script `runstats.sql` qui définit et implémente le paquetage dans votre schéma.

Exemple

Le tableau suivant décrit deux implémentations de la mise à jour du solde de tous les adhérents. La première solution est la pire qui soit car elle vérouille la table, utilise un curseur parcourant toutes les lignes de la table puis accède individuellement à chaque enregistrement par son `rowid`. La seconde réalise la mise à jour globale en une seule instruction.

Tableau 12-19 Deux solutions d'implémentation

Mises à jour par itérations	Mise à jour globale
CREATE PROCEDURE test_plsql AS BEGIN LOCK TABLE Adherent IN EXCLUSIVE MODE; FOR rec IN (SELECT Adherent.*,ROWID AS rid FROM Adherent) LOOP UPDATE Adherent SET solde = solde * 1.1 WHERE rowid = rec.rid; END LOOP; COMMIT; END test_plsql;	CREATE PROCEDURE test_sql AS BEGIN UPDATE Adherent SET solde = solde * 1.1; COMMIT; END test_sql;

Principe d'utilisation

Le principe est d'appeler le paquetage au début de la comparaison, d'exécuter la première implémentation, d'appeler le paquetage, d'exécuter la seconde implémentation puis d'invoquer une dernière fois le paquetage afin d'obtenir les résultats.

Le seul paramètre concerne la procédure `rs_stop` qui attend une valeur seuil (`p_difference_threshold`) déterminant l'affichage des résultats (ici le seuil est fixé à 50). Les résultats basés sur les statistiques et les *latches* seront donc, pour chaque test, triés par ordre croissant si la différence des chiffres obtenus (en valeur absolue) excède la valeur 50.

Tableau 12-20 Utilisation de runstats

Exécution et résultats	
BEGIN runstats_pkg.rs_start; test_plsql; runstats_pkg.rs_middle; test_sql; runstats_pkg.rs_stop(50); END;	Appels au paquetage runstats entre l'exécution des deux solutions d'implémentation à comparer.

```
Run1 ran in 101 hsecs
Run2 ran in 25 hsecs
run 1 ran in 404% of the time
Name                                  Run1        Run2        Diff
STAT...recursive cpu usage              86          24         -62
STAT...Elapsed Time                    103          26         -77
STAT...CPU used by this sessio         105          23         -82
STAT...consistent gets from ca         474         558          84
STAT...table scan blocks gotte         419         507          88
LATCH.simulator hash latch           2,339       2,432          93
...
```

Tableau 12-20 Utilisation de runstats *(suite)*

Exécution et résultats

```
STAT...session uga memory max      123,452            0    -123,452
STAT...undo change vector size   2,117,576    3,525,000   1,407,424
STAT...redo size                 5,985,752   10,173,956   4,188,204

Run1 latches total versus runs -- difference and pct
Run1        Run2        Diff       Pct
178,215     198,794      20,579     89.65%
```

La seconde implémentation est plus rapide d'un facteur 4 par rapport à la première mais nécessite un peu plus de ressources concernant les *latches*. On notera aussi que la mise à jour globale implique deux fois plus de volume à la zone `redo`.

Bilan

L'article de Shirish Joshi (*http://www.devx.com/dbzone/Article/40778/1954*) compare ces mécanismes. Le premier tableau liste les caractéristiques des outils en fonction de leur capacité à répondre à des informations.

Tableau 12-21 Caractéristiques des utilitaires

	Explain plan	Autotrace	Runstats
Temps d'exécution		X	X
Détails du SQL	X		
Entrées-sorties logiques		X	
Montée en charge/latching			X
Événements d'attente			

Le second tableau compare les outils en fonction d'autres paramètres d'utilisation.

Tableau 12-22 Autres facteurs influant les utilitaires

	Explain plan	Autotrace	Runstats
Affecté par la mise en cache	Non	Oui	Oui
Estimé ou réalisé	Estimé	Estimé	Réalisé
Fourni par Oracle	Oui	Oui	Non
Post-processing	Non	Non	Non
Facilité de comparaison	Difficile	Oui	Oui

Explain Plan et Autotrace sont des solutions simples à mettre en œuvre, mais elles ne peuvent désigner que des plans prévisionnels. Enfin, les résultats de *runstats* peuvent fluctuer fortement en fonction du cache, de la version et du paramétrage du serveur.

Organisation des données

Cette section décrit les composants de la boîte à outils qui vous servira à optimiser vos applications. Plusieurs mécanismes peuvent être mis en œuvre de manière conjointe : les contraintes, les index, le cache, le partitionnement, les vues matérialisées et la dénormalisation.

Des contraintes au plus près des données

Vous devez définir, sur vos colonnes, le maximum de contraintes d'intégrité afin de renseigner au mieux l'optimiseur. Bien que la contrainte `CHECK` ne soit pas encore utilisée par l'optimiseur, il est possible que dans le temps cette fonctionnalité soit présente.

Les colonnes NOT NULL

Le fait de déclarer des contraintes `NOT NULL` ne vous empêche pas de réaliser aussi des tests du côté de l'application. En effet, il peut être utile de vérifier qu'une valeur est présente dans un champ de saisie d'un formulaire plutôt que d'attendre d'envoyer un grand nombre d'octets au serveur qui renverra une erreur du fait d'un `NOT NULL`.

En supposant que la table `Sport` dispose de la colonne `federation` (dont les valeurs actuelles sont non nulles), le tableau suivant présente deux déclarations de contrainte `NOT NULL`.

Tableau 12-23 Déclaration de NOT NULL

Déclaration avec `CHECK`	Déclaration en ligne (*in line*)

`ALTER TABLE Sport`
`  ADD CONSTRAINT ck_federation`
`  CHECK (federation IS NOT NULL);`
	`ALTER TABLE Sport`
	`  MODIFY federation NOT NULL;`

Définissez `NOT NULL` sur le plus de colonnes possibles pour renseigner l'optimiseur.

Préférez toujours la seconde écriture (*in line constraint*), pour que l'optimiseur puisse intégrer cette information, alors qu'il ignorera la contrainte déclarée avec `CHECK`.

Les colonnes UNIQUE

Pour toute contrainte `UNIQUE`, un index (unique) est créé. Une contrainte `UNIQUE` diffère d'une contrainte `PRIMARY KEY` par le fait que les valeurs `NULL` sont autorisées ; elle n'a donc pas vocation à identifier toute ligne.

Définissez `UNIQUE` sur les colonnes potentiellement uniques de sorte que l'optimiseur puisse bénéficier d'un index supplémentaire (la désactivation d'une contrainte `UNIQUE` provoque la suppression de l'index).

Le tableau suivant présente la déclaration d'une contrainte `UNIQUE` (création implicite d'un index de nom `un_nom_prenom_tel`) et sa désactivation (suppression implicite d'un index). Comme il existe des homonymes au sein des adhérents, la contrainte `UNIQUE` minimale à mettre en œuvre est composée du nom, prénom et numéro de téléphone.

Tableau 12-24 Déclaration de UNIQUE

Déclaration de la contrainte	Désactivation

`ALTER TABLE Adherent`
`  ADD CONSTRAINT un_nom_prenom_tel`
`  UNIQUE (nom,prenom,tel);`
	`ALTER TABLE Adherent`
	`  DISABLE CONSTRAINT un_nom_prenom_tel;`

L'index multicolonnes (`nom+prenom+tel`) sera bénéfique pour les extractions dont un prédicat est basé sur le nom, le prénom et le numéro, et sur un accès aux trois colonnes simultané.

Indexation

Les différents types d'index ont été brièvement présentés au chapitre 1. Sans index, toute recherche s'apparente à un parcours séquentiel de toute la table. Ainsi pour *n* lignes, le nombre moyen de lectures est égal *n*/2, ce qui est très pénalisant dès que le volume de données devient important. De plus, ce nombre d'accès croît proportionnellement avec le nombre de lignes (100 fois plus de lignes implique un temps d'accès 100 fois plus long).

Étudions les cas d'utilisation des index d'Oracle de sorte à rendre une requête plus optimale.

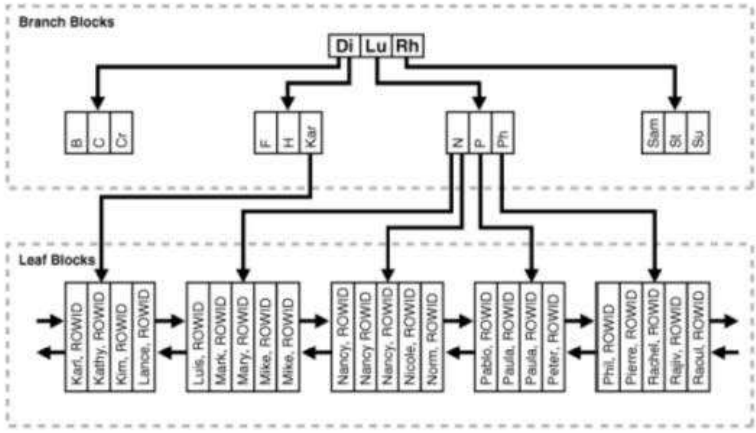
Index B-tree

Les index *B-tree* (*B* comme *Balanced*) sont constitués comme des arbres dont les noeuds aiguillent vers des sous-noeuds (suivant la valeur recherchée) jusqu'aux blocs feuille (*leaf blocks*) qui contiennent toutes les valeurs de l'index et les adresses de ligne (*rowid*) identifiant

le segment de données associé. Les blocs feuilles sont doublement chaînés de sorte que l'index puisse être parcouru dans les deux sens sans passer par la racine.

Ce mécanisme est bien plus performant qu'un accès séquentiel car pour n lignes, le nombre moyen de lectures n'est plus proportionnel à n mais à $log(n)$. La taille maximale d'une entrée d'index est environ égale à la moitié de la taille des blocs de données (soit de l'ordre de 4 000 pointeurs pour une taille de bloc de 8 Ko).

Figure 12-5 *Index B-tree (© doc. Oracle)*



Un index *B-tree* est conçu automatiquement lors de la création de la clé primaire d'une table et d'une contrainte `UNIQUE`. Les arbres *B-tree* présentent de nombreux avantages :

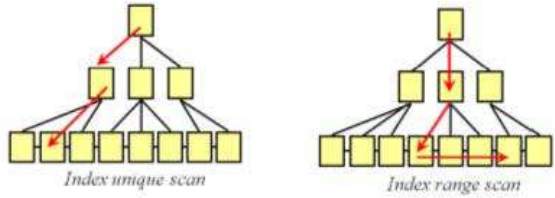
- Malgré les mises à jour de la table, ils restent équilibrés (les blocs feuilles sont au même niveau). En conséquence, quelle que soit la valeur cherchée, le temps de parcours est sensiblement identique. Les blocs intermédiaires sont remplis, en moyenne, aux trois-quarts de leur capacité.
- Les performances d'extraction, répondant à la majorité des prédicats des requêtes, sont excellentes, notamment les comparaisons d'égalité et d'intervalles.
- Les répercussions des mises à jour sont efficaces et ne se dégradent pas en fonction d'une forte augmentation de la taille des tables.

Nous ne traiterons pas ici des caractéristiques physiques des index (partitions, compression, pourcentages des tailles de blocs, etc.).

Les principales opérations que l'optimiseur réalise sur un index sont les suivantes :

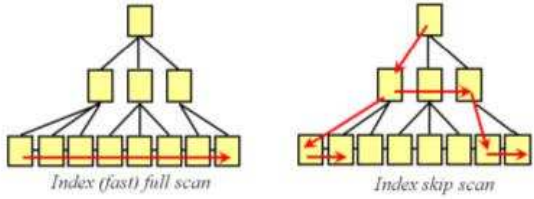
- *index unique scan* passe par la racine de l'arbre ; généralement toutes les colonnes de l'index sont concernées par une égalité dans le prédicat `WHERE`. Il s'agit en principe de la manière la plus optimale, mais qui n'est pas toujours utilisée par l'optimiseur au profit de *range scan*.
- *index range scan* passe par la racine de l'arbre et accède séquentiellement aux blocs feuille (doublement chaînées). Opération très utilisée par l'optimiseur, notamment lorsque une colonne de l'index est concernée par une inégalité dans le prédicat `WHERE`, et que l'index n'est pas unique. Dans tous ces cas, l'optimiseur juge qu'il est plus rapide de parcourir les feuilles de l'index plutôt que l'index lui-même.

Figure 12-6 *Accès direct et par parcours par intervalles d'un index B-tree*



- *index full scan* et *index fast full scan* sont une alternative au parcours *full table scan* quand l'index contient toutes les colonnes nécéssaires à la requête et qu'au moins une de ces colonnes est `NOT NULL`. Il ne peut pas être utilisé sur un index bitmap ; le parcours de l'index entier est plus rapide car il se réalise en mode lecture multibloc et peut être parallélisé.
- *index skip scan* (concerne les index multicolonnes) utilise l'index alors que la (ou les) première(s) colonne(s) de l'index n'est (ne sont) pas présente(s) dans le prédicat `WHERE`.

Figure 12-7 *Parcours séquentiel et par saut d'un index B-tree*



Généralement, afin d'isoler le stockage physique des index, on utilise un *tablespace* dédié (qui peut se trouver sur un autre disque que celui des données). Il est aussi d'usage de créer un index pour chaque clé étrangère afin de rendre plus efficace les jointures.

Le tableau suivant présente d'une part la création de l'espace de stockage pour héberger les index et, d'autre part, la création d'index affectés à cet espace (une clé primaire et une clé étrangère non unique).

Tableau 12-25 Création d'index en association avec un tablespace

Création de l'espace	Création d'index

`CREATE TABLESPACE tbs_index`
`  DATAFILE 'tbs_index.dat'`
`  SIZE 500M REUSE`
`  AUTOEXTEND ON`
`  NEXT 500K MAXSIZE 2000M;`
	`--colonne "clé primaire"`
	`ALTER TABLE Adherent`
	`  ADD CONSTRAINT pk_Adherent PRIMARY KEY`
	`(adhid)`
	`  USING INDEX TABLESPACE tbs_index;`
	`…`
	`--colonne "clé étrangère"`
	`CREATE INDEX idx_Pratique_adhid`
	`  ON Pratique (adhid) TABLESPACE tbs_index;`

Pour se convaincre de l'utilité des index, exécutez la requête avec et sans index (il s'agit d'une division) qui extrait les adhérents inscrits à tous les sports. L'adhérente la plus sportive est, sans conteste, Céline Larrazet et il faut 36 secondes sans indexage pour découvrir l'identité de la championne alors que la réponse est quasi instantanée en présence d'index sur les clés étrangères. Les chiffres sont éloquents même pour une volumétrie réduite (24 000 adhérents dans 1 800 blocs) : sans index, on recense 20 fois plus d'accès aux blocs et de nombreux tris.

Tableau 12-26 Performances d'une extraction avec et sans index

```
SELECT a.civilite, a.prenom, a.nom, a.tel FROM Adherent a
       WHERE NOT EXISTS
            (SELECT spid FROM Sport
             MINUS  SELECT spid FROM Pratique WHERE adhid = a.adhid)
       AND NOT EXISTS
           (SELECT spid FROM Pratique WHERE adhid = a.adhid
            MINUS SELECT spid FROM Sport);
```

CIVILITE	PRENOM	NOM	TEL
Mme.	CELINE	LARRAZET	05-62-18-04-76

Avec index	Sans index
3 centièmes de secondes	36 secondes
114 recursive calls	533 recursive calls
72734 consistent gets	1442719 consistent gets
0 sorts (memory)	48080 sorts (memory)

Bien que les index *B-tree* soient majoritairement employés, ils ne conviennent pas aux conditions suivantes :

- Données de faible cardinalité : on considère qu'une colonne disposant de moins de 200 valeurs disctinctes n'est pas une bonne candidate à un index *B-tree* (par exemple, la civilité qui ne comporte que 3 valeurs). Les index *bitmap* sont une alternative à cette limitation.
- Quand l'accès aux données s'effectue par une fonction SQL (*built-in function*), l'index *B-tree* n'est pas utilisé (par exemple `WHERE UPPER(prenom)='PAUL'` n'emploiera pas l'index sur `prenom`). Le fait de créér un index sur cette fonction est une alternative à cette limitation.

Index et expressions (built-in function)

Si vous utilisez des fonctions caractères (`UPPER`, `SUBSTR`, `RTRIM`, etc.) ou des fonctions numériques (`MOD`, `ROUND`, `TRUNC`, etc.) dans le prédicat de vos requêtes, n'espérez pas utiliser vos index.

Le tableau 12-27 présente les résultats de différentes requêtes selon deux stratégies d'indexage. La volumétrie de la table `Adherentbis` est de plus d'un million d'adhérents (88 Mo de donnnées occupant près de 90 000 blocs). Pour chaque requête, sont donnés : le type de parcours de l'index (*table access full* : l'index n'est pas utilisé), le nombre de blocs lus (*b*) et le coût (*c*).

Tableau 12-27 Utilisation d'index B-tree sur des expressions de colonnes

Index existants	Prédicats et résultats	

`CREATE INDEX idx_nom ON Adherentbis (nom) TABLESPACE tbs_index;`
 	
`CREATE INDEX idx_solde ON Adherentbis (solde) TABLESPACE tbs_index;`
	`WHERE nom='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'`
	Index range scan
	(578 *b* – 110 *c*)
		`WHERE UPPER(nom)='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'`
		Table access full
		(10236 *b* – 2797 *c*)

	`WHERE ROUND(solde,1)=9030.8` *Table access full* (10235 *b* – 2802 *c*)	`WHERE solde=9030.75` *Index range scan* (3 *b* – 3 *c*)

`CREATE INDEX idx_UPPERnom ON Adherentbis (UPPER(nom)) TABLESPACE tbs_index;`
 	
`CREATE INDEX idx_ROUNDsolde ON Adherentbis (ROUND(solde,1)) TABLESPACE tbs_index;`
	`WHERE nom='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'`
	Table access full
	(10229 *b* – 2795 *c*)
		`WHERE UPPER(nom)='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'`
		Index range scan
		(578 *b* – 107 *c*)

	`WHERE ROUND(solde,1)=9030.8` *Index range scan* (3 *b* – 3 *c*)	`WHERE solde=9030.75` *Table access full* (10229 *b* – 2795 *c*)

Les remarques que l'on peut déduire à propos de la première stratégie d'indexage sont les suivantes :

- Les fonctions `ROUND` et `UPPER` rendent inopérants les index définis pourtant sur les colonnes concernées.
- Les index sur les colonnes numériques sont plus performants que les index sur les colonnes chaînes de caractères.

Concernant la deuxième stratégie d'indexage, les fonctions `ROUND` et `UPPER` rendent opérationnels les index, mais les conditions simples sur les colonnes entraînent un parcours entier de la table.

Index et NULL

Le principe de fonctionnement des index *B-tree* ne permet pas une recherche directe (*unique scan*) sur une absence de valeur (`NULL`) ; en conséquence, si un index existe sur une colonne non nulle, il ne sera pas utilisé au mieux lors de la recherche des `NULL` (prédicat `IS NULL` ou `IS NOT NULL`).

Pour indexer efficacement une colonne qui peut contenir des `NULL`, plusieurs solutions s'offrent à vous :

- index basé sur une fonction déterministe qui retourne un entier quand la colonne vaut `NULL` ;
- index composé par une colonne qui n'est jamais `NULL` ;
- index basé sur une fonction adéquate comme `NVL2(chaine, valeur_si_NOT_NULL, valeur_si_NULL)`.

Appliquez ces différentes solutions à votre base de sorte à déterminer la plus performante. Le tableau suivant présente quelques résultats d'après la recherche du nombre d'adhérents en fonction de leur numéro de téléphone donné (`NULL`, valeur, `NOT NULL`). Concernant les données, 37 485 adhérents n'ont pas de numéro de téléphone (soit 3 % de la population). Pour chaque type d'indexage, sont donnés la taille de l'index en Mo, le type de parcours de l'index, le nombre de blocs lus (*b*) et le coût (*c*).

Tableau 12-28 Utilisation d'index B-tree sur une colonne ayant des valeurs NULL

`SELECT COUNT(nom) FROM Adherent WHERE…`	**Condition sur la nullité**	`tel='06-81-94-44-31'`	`tel IS NOT NULL`
Sans index	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Table access full* (10228 *b* – 2796 *c*)	*Table access full* (10228 *b* – 2793 *c*)
Index *B-tree* (taille : 38 Mo) **`CREATE INDEX idx_tel_btree ON Adherent (tel);`**	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Index range scan* (4 *b* – 5 *c*)	*Index fast full scan* (4756 *b* – 1293 *c*)

Tableau 12-28 Utilisation d'index B-tree sur une colonne ayant des valeurs NULL *(suite)*

SELECT COUNT(nom) FROM Adherent WHERE…	Condition sur la nullité	tel='06-81-94-44-31'	tel IS NOT NULL

Index fonction (taille : 0,68 Mo)
`CREATE FUNCTION f_tel_null (p_tel Adherentbis.tel%type) RETURN NUMBER DETERMINISTIC AS BEGIN IF p_tel IS NULL THEN RETURN 1; ELSE RETURN NULL; END IF; END f_tel_null;`
`CREATE INDEX idx_tel_btree ON Adherent (f_tel_null(tel));`
	`f_tel_null(tel)=1`
	Index fast full scan
	(28 *b* – 84 *c*)
		Table access full
		(10228 *b* – 2796 *c*)
			Sans objet

Index composé (taille : 41 Mo) `CREATE INDEX idx_tel_btree ON Adherent (tel,0);`	`tel IS NULL` *Index range scan* (166 *b* – 76 *c*)	*Index range scan* (4 *b* – 5 *c*)	*Index fast full scan* (5150 *b* – 1399 *c*)
Index fonction NVL2 (taille : 0,62 Mo) `CREATE INDEX idx_tel_btree ON Adherent (NVL2(tel,NULL,0));`	`NVL2(tel,NULL,0)=0` *Index fast full scan* (74 *b* – 21 *c*)	*Table access full* (10228 *b* – 2796 *c*)	Sans objet
Index unique (taille : 37 Mo) `CREATE UNIQUE INDEX idx_tel_btree ON Adherentbis (tel);`	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Index unique scan* (4 *b* – 3 *c*)	*Index fast full scan* (4599 *b* – 1250 *c*)

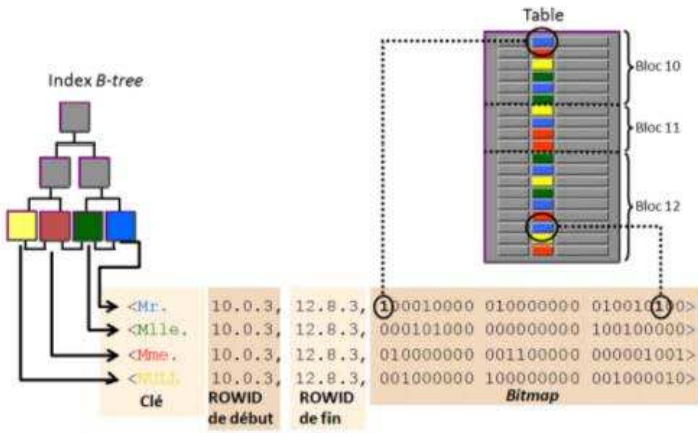
Les index les plus performants sont :

- Pour répondre au prédicat `IS NULL`, ceux qui utilisent une fonction (déterministe pour l'un et `NVL2` pour l'autre).
- Pour répondre au prédicat `col=valeur`, l'index classique, unique ou composé, qui offre les mêmes résultats.
- Pour répondre au prédicat `IS NOT NULL`, l'index unique.

Index bitmap

Un index *bitmap* est organisé comme un index *B-tree* dont chaque feuille permet de pointer vers plusieurs lignes. Chaque en-tête de *bitmap* contient un *rowid* de début et de fin. À partir de ces valeurs, un algorithme met les *bitmaps* en correspondance avec des *rowid*. Chaque position de *bitmap* correspond à une ligne potentielle de la table, même si cette ligne n'existe pas. Le contenu de cette position, pour une valeur particulière, indique si la ligne contient ou non (1 ou 0) cette valeur dans les colonnes du *bitmap*.

Figure 12-8 *Accès direct et par parcours par intervalle d'un index B-tree*



Les index *bitmap* sont très répandus dans les environnements OLAP (*OnLine Analytical Processing*) caractérisés par d'importants volumes de données et par l'absence de mises à jour. En effet, le verouillage d'un index *bitmap* entraîne aussi le verouillage de nombreuses lignes de la table concernée.

L'indexation *bitmap* est idéale pour des colonnes de faible cardinalité sur des tables volumineuses (quand le nombre de valeurs distinctes est très nettement inférieur au nombre de lignes de la table, de l'ordre de 1 %).

Les index *bitmap* sont performants pour les requêtes où chaque critère retourne beaucoup de lignes et la sélectivité de l'ensemble des critères est forte.

Il est déconseillé de créer des index *bitmap* sur des tables très fortement actualisées car il est très coûteux de reconstruire l'index à chaque mise à jour.

Les index *bitmap* sont de taille réduite et acceptent de gérer les `NULL`.

Le principe du *bitmap* consiste à créer pour chaque ligne de la table un mot binaire comportant autant de bits que de possibilités de valeurs de l'index. Lors d'une recherche, un simple *AND* binaire valide ou non les correspondances entre les colonnes testées.

Dans l'exemple, les valeurs possibles de la colonne `civilite` sont : `'Mr.'`, `'Mme.'` et `'Melle.'`. Chaque ligne de la table sera potentiellement associée à un des mots de 3 bits suivants : `000` pour `NULL`, `001` pour `'Melle.'`, `010` pour `'Mme.'` ou `100` pour `'Mr.'`. Le

tableau suivant présente quelques résultats de recherche du nombre d'adhérents en fonction de leur civilité. L'index le plus performant concernant la colonne `civilite` est sans conteste le *bitmap*.

Tableau 12-29 Utilisation d'index sur une colonne de faible cardinalité

`SELECT COUNT(nom) FROM Adherent WHERE…`	`civilite='Mr.'`	`civilite IN ('Mlle.','Mme.')`	`NOT (civilite='Mr.')`
Sans index	*Table access full* (10235 *b* – 2793 *c*)	*Table access full* (10235 *b* – 2795 *c*)	*Table access full* (10235 *b* – 2793 *c*)
Index bitmap (taille : 0,43 Mo) `CREATE BITMAP INDEX idx_civilite_bitmap ON Adherentbis (civilite) TABLESPACE tbs_index;`	*Bitmap index single value* (20 *b* – 14 *c*)	*Bitmap index single value* (38 *b* – 29 *c*)	*Bitmap index fast full scan* (53 *b* – 42 *c*)
Index *B-tree* (taille : 21 Mo) `CREATE INDEX idx_civilite_btree ON Adherentbis (civilite) TABLESPACE tbs_index;`	*Index fast full scan* (2602 *b* – 712 *c*)	*Index fast full scan* (2602 *b* – 741 *c*)	*Index fast full scan* (2602 *b* – 712 *c*)

L'utilisation optimale d'un index *bitmap* concerne les égalités `col=valeur`, les comparaisons à des ensembles `col IN (v1,v2…)` que l'optimiseur traduit en `col=v1 OR col=v2…`, et les inégalités `NOT(col=valeur)` que l'optimiseur traduit en `col<>valeur`.

Index multicolonnes

Un index peut être composé de plusieurs colonnes (on parle aussi d'index multicolonnes ou concaténé). Les colonnes de l'index ne doivent pas nécessairement être adjacentes.

Le nombre maximal de colonnes est fixé à 32 pour un index *B-tree* et 30 pour un index *bitmap*.

Les colonnes `LONG` et `LONG RAW` ne peuvent pas être indexées.

Les index multicolonnes offrent l'avantage de pouvoir combiner des colonnes (ou expressions) présentant une faible sélectivité pour former un index dont la sélectivité est plus élevée. Par ailleurs, si toutes les colonnes concernées par une interrogation se trouvent dans un index composé, l'accès à l'index suffira (pas de nécessité d'accéder à la table).

Un index composé est utile principalement lorsque vos clauses `WHERE` font souvent référence à l'ensemble, ou à la partie de tête, des colonnes de l'index. La version 10*g* a introduit la recherche *index skip scan* pour utiliser un index composé en s'affranchisant d'une partie de l'en-tête de cet index (ses premières colonnes).

Bien que l'ordre des colonnes d'un index composé ait moins d'importance qu'avant la version 9*i* (les statistiques de la version 11*g* permettent de représenter la distribution relative de données entre plusieurs colonnes), choisissez les colonnes les plus fréquemment utilisées pour constituer la tête de l'index.

L'optimiseur n'utilisera l'algorithme *index skip scan* que si la cardinalité de la (des) première(s) colonne(s) de l'index est relativement faible (selon ses statistiques).

Illustrons cet aspect des choses avec deux index créés sur la table `Adherentbis`. Le premier est composé du nom, prénom et civilité, le second de la civilité, du nom et du prénom. Soumettons à ces deux index plusieurs requêtes comportant dans la clause `WHERE` des prédicats basés sur des colonnes appartenant ou non aux index.

Tableau 12-30 Utilisation d'index multicolonnes

Type d'index	Requêtes et résultats		
Index sur nom+prénom+civilité Taille : 42 Mo `CREATE INDEX idx_nom_pre_civilite ON Adherentbis (nom,prenom,civilite) TABLESPACE tbs_index;`	`WHERE nom='…' AND prenom='…' AND civilite='…'` *Index range scan* (57 *b* – 4 *c*)	`WHERE prenom='…' AND civi-lite='…'` *Index fast full scan* (5244 *b* – 1388 *c*)	`WHERE prenom='…' AND civi-lite='…' AND tel='…'` ***Table access full*** (10229 *b* – 2795 *c*)
Index sur civilité+nom+prénom Taille : 42 Mo `CREATE INDEX idx_civilite_nom_pre ON Adherentbis (civilite,nom,prenom) TABLESPACE tbs_index;`	`WHERE nom='…' AND prenom='…' AND civilite='…'` *Index range scan* (56 *b* – 4 *c*)	`WHERE prenom='…' AND civi-lite='…'` *Index fast full scan* (5245 *b* – 1473 *c*)	`WHERE prenom='…' AND civi-lite='…' AND tel='…'` *Index range scan* (6830 *b* – 26 *c*)
	`WHERE nom='…' AND prenom='…'` ***Index skip scan*** (73 *b* – 6 *c*)		

Lorsque la colonne `nom` (de grande cardinalité) est en tête de l'index, aucun saut d'index n'est réalisé et toute requête n'utilisant pas cette colonne dans la clause `WHERE` est peu performante (accès *full*).

Lorsque la colonne `civilite` (de très faible cardinalité) est en tête de l'index, des sauts d'index peuvent se produire et améliorer très notablement les performances.

Si deux colonnes sont fréquemment utilisées simultanément, il vaut mieux employer un index composite que deux index monocolonnes.

Lors de la création d'index multicolonnes, ordonnez de préférence les colonnes de la plus faible à la plus forte sélectivité.

Index et expressions

Il est important de dissocier un index concernant une colonne et l'utilisation de cette colonne dans une requête *via* une expression.

Le fait d'indexer une colonne n'indique pas à l'optimiseur de mettre en œuvre l'index lorsque la comparaison concerne une expression sur la colonne et non la colonne elle-même. Vous devrez alors définir un index équivalent à l'expression.

Illustrons ce point avec deux index créés sur la table `Adherentbis`. Le premier est défini à partir de la date de naissance, le second sur une fonction qui formate cette date. Soumettons à ces deux index plusieurs requêtes comportant dans la clause `WHERE` différents prédicats. Le tableau suivant présente les résultats.

Tableau 12-31 Index basés sur une expression

Type d'index	Requêtes et résultats		
Index sur la date de naissance Taille : 25 Mo `CREATE INDEX idx_datenais ON Adherentbis (date_nais) TABLESPACE tbs_index;`	`SELECT COUNT(nom) FROM Adherentbis WHERE TO_CHAR (date_nais, 'DD/MM/YYYY') ='17/01/1980';`	`SELECT COUNT(DISTINCT nom), TO_CHAR(date_nais, 'DD/MM/YYYY') FROM Adherentbis GROUP BY date_nais;`	`SELECT a1.tel, a1.nom, a1.date_nais FROM Adherentbis a1, Adherentbis a2 WHERE NOT(a1.nom=a2.nom) AND a1.date_nais= a2.date_nais AND TO_CHAR (a1.date_nais, 'DD/MM/YYYY') ='17/01/1980';'`
	Index fast full scan (3135 *b* – 886 *c*)	*Table access full* (10243 *b* – 2845 *c*)	*Table access full* (20470 *b* – 5621 *c*)
Index sur une expression Taille : 25 Mo `CREATE INDEX idx_to_char_datenais ON Adherentbis (TO_CHAR(date_nais, 'DD/MM/YYYY')) TABLESPACE tbs_index;`	Même requête	Même requête	Même requête
	Index range scan (7 *b* – 3 *c*)	*Table access full* (10243 *b* – 2845 *c*)	***Index range scan*** (10291 *b* – 2872 *c*)

Le premier index (qui n'est pas basé sur l'expression) :

- Est utilisé dans la première requête bien qu'il ne soit pas déclaré sur la fonction `TO_CHAR`.
- N'est pas utilisé dans la seconde requête malgré le fait qu'il ne soit pas déclaré sur la colonne `date_nais` (un regroupement ne justifie pas le parcours de l'index).

- N'est pas utilisé dans la troisième requête malgré le fait que le prédicat porte sur une égalité des colonnes indexées (l'optimiseur juge qu'il est plus intéressant de parcourir les deux tables).

Le deuxième index (basé sur l'expression) :

- Est utilisé de façon optimale dans la première requête car il est déclaré sur la fonction `TO_CHAR`.
- N'est pas utilisé dans la seconde requête du fait qu'aucun prédicat ne le concerne.
- Est utilisé dans la troisième requête (et divise par 2 le coût) car un prédicat le concerne.

Choix d'indexage

Quelques règles relatives à la gestion des index

Créez les index après l'insertion des données dans la table. Les données sont souvent chargées *via* SQL*Loader ou un utilitaire d'importation et il est plus efficace d'insérer les données avant d'y associer des index.

Créez un index si vous désirez extraire souvent moins de 15 % des lignes d'une table volumineuse.

Il est important de connaître le nombre de blocs concernés par un balayage de l'index en comparaison d'un parcours entier de la table. Si une table contient un million de lignes dans 5 000 blocs, et que les valeurs d'une des colonnes soient réparties sur plus de 4 000 blocs, il ne sera pas optimal de créer un index.

Pour améliorer les jointures, indexez les clés étrangères.

Les tables de faible volumétrie ne nécessitent pas d'index.

Les colonnes potentiellement indexables présentent les caractéristiques suivantes :

- Il existe une grande plage de valeurs qui sont relativement uniques dans la colonne (index *B-tree*).
- Il existe une petite plage de valeurs (index *bitmap*).
- Elles contiennent de nombreux `NULL`, mais les extractions concernent très souvent les lignes dont la valeur est non nulle.

Taille d'un index

La procédure `CREATE_INDEX_COST` du paquetage `DBMS_SPACE` (dédié à l'évolution des segments et espaces pour les tables et index) permet de déterminer le coût de création d'un index. Avant d'utiliser cette procédure, vous devez vous assurer que la table existe et que les statistiques sont collectées.

Le script suivant décrit l'appel de cette procédure qui retourne la taille du futur index unique (42 Mo) sur les colonnes nom, prenom et tel du volume occupé (62 Mo) dans le segment du *tablespace* tbs_index.

Tableau 12-32 Estimation de la taille d'un index

Appel de la procédure et résultats

```
SQL> VARIABLE v_octets_index    NUMBER
SQL> VARIABLE v_octets_segment  NUMBER
SQL> VARIABLE v_sql             CHAR(100)
SQL> EXEC    :v_sql:='CREATE UNIQUE INDEX idx_nom_pre_tel
                      ON Adherentbis (nom, prenom, tel) TABLESPACE tbs_index'
Procédure PL/SQL terminée avec succès.

SQL> EXEC DBMS_SPACE.CREATE_INDEX_COST(:v_sql, :v_octets_index, :v_octets_segment)
Procédure PL/SQL terminée avec succès.

SQL> PRINT :v_octets_index
V_OCTETS_INDEX
--------------
      42394212
SQL> PRINT :v_octets_segment
V_OCTETS_SEGMENT
----------------
        62914560
```

Réglage des index

Les quatre principaux facteurs qui guident l'optimiseur à choisir ou non un index sont la sélectivité des données, la taille des blocs, la taille moyenne des lignes et la cardinalité.

En fonction de ces facteurs, l'optimiseur étudie le *clustering factor* qui indique la synchronisation entre l'ordre des *rowid* dans les feuilles de l'index et ceux de la table. Cette information se trouve dans la colonne CLUSTERING_FACTOR de la vue DBA_INDEXES pour chaque index. Plus ce nombre se rapproche du nombre de blocs du segment de la table, plus l'index est optimal. En revanche, plus il se rapproche du nombre de lignes de la table, moins l'index est optimal.

Un index sera utilisé de façon optimale pour les recherches de données sélectives s'il est caractérisé par un faible *clustering factor*.

Bien qu'indexant des données sélectives, un index de fort *clustering factor* manipulant des lignes de taille moyenne réduite ne sera généralement pas employé, au profit d'un parcours de la table.

La taille des blocs influence également la stratégie d'Oracle. Les index soumis à de nombreuses opérations *range scan* ou *fast full scan* (par lecture multibloc) seront d'autant plus performants en disposant d'une taille de blocs la plus grande (jusqu'à 32 Ko).

Reconstruction des index

Selon la fréquence de mise à jour et le volume de données modifiées d'une table, vous devrez reconstruire les index *B-tree* du fait de leur fragmentation. Plus la densité des blocs feuilles est élevée, meilleur est l'index. À l'inverse, il est souhaitable de reconstruire l'index lorsqu'il contient de nombreux blocs peu peuplés. Depuis la version 11*g*, l'outil *segment advisor* est capable de détecter les index défaillant et de les reconstruire automatiquement.

Vous devrez choisir entre deux stratégies : la reconstruction (clause `REBUILD` de `ALTER INDEX`) et la fusion (clause `COALESCE` de `ALTER INDEX`). Alors que `REBUILD` permet de déplacer un index vers un autre *tablespace* (ou de modifier des caractéristiques physiques), la fusion se focalise sur des branches de l'index. La reconstruction d'un index existant offre de meilleures performances que la destruction de l'index puis sa recréation.

Utilisation des index par l'optimiseur

Plusieurs autres facteurs influencent l'optimiseur sur la façon d'utiliser un index ou non :

- Le volume des tables où relativement peu de lignes vérifiant une condition ne nécessitent pas un parcours indexé, ou au contraire, un grand nombre de blocs où le parcours de la table sera choisi.
- Un conseil (*hint*) incompatible comme `/*+ NO_INDEX(...) */`.
- Le partitionnement de la table et l'absence d'index global (ou un non partitionnement et des index locaux).
- L'utilisation de variables de session (*bind variables*). L'optimiseur établit un plan d'exécution sans connaître a priori la valeur de ces variables. Une section est consacrée aux variables de session.

Jointures

Les jointures ont été étudiées au chapitre 4. Ainsi, plusieurs écritures sont possibles pour répondre à toute interrogation mettant en relation plusieurs tables (généralement basées sur l'égalité entre une colonne clé étrangère et clé primaire ou unique). Dans toute jointure entre deux tables, une ligne d'une table est appelée *inner*, l'autre *outer*. Pour choisir un plan d'exécution, l'optimiseur décide de la stratégie en fonction de plusieurs facteurs :

- Le chemin d'accès afin d'extraire les données de chaque table lors de la jointure.
- La méthode de jointure pour chaque paire de lignes jointes ; l'opération adoptée est soit un *nested loop*, soit un *sort merge join*, soit un *cartesian join*, ou *hash join*.
- L'ordre dans lequel les jointures doivent se réaliser lorsqu'il y a plus de 2 tables en relation. La deuxième jointure s'opère après la première, etc.

Ne vous souciez pas trop du style d'écriture de votre jointure (relationnel, SQL2, procédural ou mixte), ni de l'ordre des tables dans la clause `FROM`. L'optimiseur d'Oracle se chargera de réécrire, dans la majorité des cas, votre requête de la manière optimale et choisira le meilleur plan d'exécution (sous réserve d'une collecte des statistiques).

L'optimiseur détermine d'abord si la jointure retourne au final au moins une ligne. Cette réponse est basée sur les contraintes `UNIQUE` et `PRIMARY KEY` des tables. Si ces contraintes existent, l'optimiseur traite ces tables en premier puis rend optimale la suite des opérations en minimisant les coûts (via les statistiques) :

- des *nested loops* en se basant sur le coût des lectures en mémoire de chaque ligne de la table *outer* et chaque correspondance avec les lignes de la table *inner* ;
- des *hash join* en s'appuyant principalement sur le coût de construction d'une table de hachage ;
- des *sort merge join* en utilisant principalement le coût des lectures en mémoire de toutes les données et des tris.

Il existe aussi l'index *bitmap join* qui combine l'avantage du *bitmap* à celui d'un prédicat de jointure. Ce type d'indexage convient davantage aux très gros volumes de données (*datawarehouses*).

Nested loops

Une opération *nested loop* se déroule en trois temps. D'abord l'optimiseur choisit la table sur laquelle conduire l'itération (*outer table*) et désigne l'autre table en tant que *inner*. Ensuite, pour chaque ligne de la table *outer*, Oracle accède à toutes les lignes de la table *inner*.

Les jointures programmées avec l'opérateur *nested loop* sont très performantes lorsqu'un faible nombre de données de la première table (*outer*) est mis en jointure et que la condition de jointure accède efficacement à la deuxième table (*inner*). Si le chemin d'accès à la table *inner* est indépendant de la table *outer*, alors des mêmes lignes sont extraites à chaque itération de la table *outer* (cela dégrade les performances et l'optimiseur choisira une opération *hash join*).

Dans un plan d'exécution, l'itération se présente par le mot-clé `NESTED LOOPS`, la table *outer* apparaît avant la table *inner*. Dans l'exemple suivant, le *hint* `USE_NL` force l'opération sinon un *hash join* plus performant serait utilisé par l'optimiseur. Selon les versions et *releases* d'Oracle, vous trouverez différentes implémentations de l'opération *nested loop*. Pour cette requête, 2 904 blocs sont lus pour un coût de 142.

Tableau 12-33 Jointure mise en œuvre avec nested loop

Requête

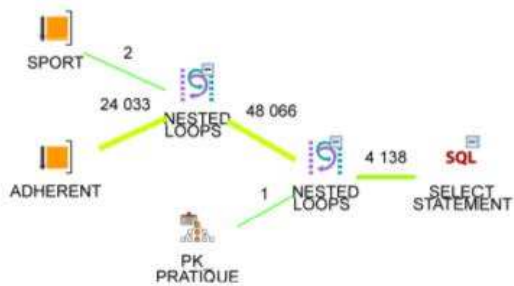
```
SELECT /*+ USE_NL(a p s) */
       a.adhid, a.nom, a.tel
 FROM  Adherent a, Sport s, Pratique p
  WHERE s.splibelle
       IN ('Escrime','Ping-pong')
  AND   a.adhid = p.adhid
  AND   s.spid  = p.spid;
```

Plan d'exécution

```
-----------------------------------------------------
| Id  | Operation           | Name         | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT    |              |  4138 |
|   1 |  NESTED LOOPS       |              |  4138 |
|   2 |   NESTED LOOPS      |              | 48066 |
|*  3 |    TABLE ACCESS FULL| SPORT        |     2 |
|   4 |    TABLE ACCESS FULL| ADHERENT     | 24033 |
|*  5 |    INDEX UNIQUE SCAN| PK_PRATIQUE  |     1 |
-----------------------------------------------------
Predicate Information (identified by operation id):
-----------------------------------------------------
3 - filter("S"."SPLIBELLE"='Escrime' OR
           "S"."SPLIBELLE"='Ping-pong')
5 - access("A"."ADHID"="P"."ADHID" AND
           "S"."SPID"="P"."SPID")
Statistiques (partielles)
-----------------------------------------------------
       2904  consistent gets
        590  rows processed …
```

Comme l'illustre l'arbre suivant, la première itération met en jeu la table `Sport` (*outer* car elle ne filtre que 2 lignes) qui est combinée à la table `Adherent` (*inner* avec 24 033 lignes). Le chemin d'accès à la table *inner* est complètement indépendant de la table *outer*, car il n'existe pas de colonne commune (aucune ligne extraite ne sera donc identique pour chaque itération). Le résultat de cette jointure retourne 48 066 lignes. La deuxième itération combine ce résultat (table *outer*) avec la table d'association `Pratique` en utilisant son index. Des colonnes (`adhid` et `spid`) sont communes et la jointure de ces deux tables restitue finalement 4 138 lignes qui, après élimination de certains doublons, donnera 590 adhérents.

Figure 12-9 *Jointures par boucles imbriquées*



Le *hint* `NO_USE_NL(alias1 ...)` interdit à l'optimiseur l'utilisation de boucles imbriquées pour programmer les jointures.

Hash joins

Les jointures programmées avec l'opérateur *hash join* sont très performantes lorsqu'il s'agit de joindre d'importants volumes de données. L'optimiseur utilisera la table de plus faible volumétrie pour construire en mémoire une table de hachage sur la clé de jointure. Ensuite, un parcours de la table la plus volumineuse est réalisée, en vérifiant le prédicat par hachage pour extraire les lignes jointes.

L'opération de *hash join* est majoritairement utilisée pour les équijointures et quand un grand volume de données doit être joint ou une importante fraction d'une table de faible taille doit être jointe.

Dans un plan d'exécution, la jointure se présente par le mot-clé `HASH JOIN`, la table *outer* apparaît avant la table *inner*. L'exemple suivant illustre une jointure sur 3 tables (c'est la même requête que la précédente écrite sous la forme SQL2). Le plan d'exécution est plus performant que le plan *nested loop*, seulement 348 blocs sont lus (gain de 88 %) pour un coût de 88 (gain de 38 %). En écrivant ces jointures sous la forme relationnelle, le plan obtenu serait identique.

Tableau 12-34 Jointure mise en œuvre avec hash join

Requête

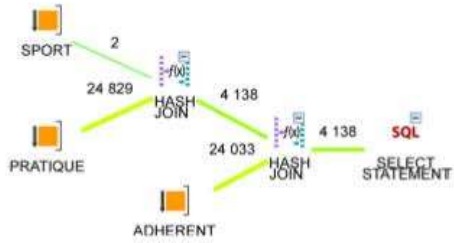
```
SELECT a.adhid, a.nom, a.tel
  FROM Adherent a
  INNER JOIN Pratique p
        ON p.adhid = a.adhid
    INNER JOIN  Sport s
          ON s.spid  = p.spid
    WHERE s.splibelle
          IN ('Escrime', 'Ping-pong');
```

Plan d'exécution

```
-----------------------------------------------
| Id  | Operation           | Name     | Rows  |
-----------------------------------------------
|   0 | SELECT STATEMENT    |          |  4138 |
|*  1 |  HASH JOIN          |          |  4138 |
|*  2 |   HASH JOIN         |          |  4138 |
|*  3 |    TABLE ACCESS FULL| SPORT    |     2 |
|   4 |    TABLE ACCESS FULL| PRATIQUE | 24829 |
|   5 |   TABLE ACCESS FULL | ADHERENT | 24033 |
-----------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("P"."ADHID"="A"."ADHID")
   2 - access("S"."SPID"="P"."SPID")
   3 - filter("S"."SPLIBELLE"='Escrime'
         OR "S"."SPLIBELLE"='Ping-pong')
Statistiques (partielles)
---------------------------------------------------
        348  consistent gets
        590  rows processed
```

Comme l'illustre l'arbre suivant, la première jointure met en jeu la table `Sport` (*outer* car elle ne filtre que 2 lignes) qui est combinée à la table d'association `Pratique` (*inner* avec 24 829 lignes). Le résultat de cette jointure par hachage retourne 4 138 lignes. La deuxième jointure combine ce résultat (table *outer*) avec la table `Adherent` sans utiliser son index. La colonne `adhid` est commune et le résultat final après élimination de certains doublons donnera 590 adhérents.

Figure 12-10 *Jointures par table de hachage*



Les *hints* `USE_HASH(alias1 ...)` et `NO_USE_HASH(alias1 ...)` permettent respectivement de forcer ou d'empêcher l'optimiseur à utiliser la jointure par hachage.

Sort merge joins

Bien que l'algorithme *hash join* soit souvent préféré aux autres (ce qui explique que la majorité des jointures soient mises en œuvre ainsi), l'opération *sort merge join* (tri-fusion) peut offrir de meilleures performances lorsque les lignes à joindre sont déjà triées ou qu'un tri doit être effectué.

Dans un *sort merge join*, il n'y a pas de table qui pilote une autre, mais deux étapes successives : le *sort join*, où les deux tables sont triées sur la clé de jointure, puis *merge join* qui fusionne les listes triées. Si les tables à joindre sont déjà triées, seule la deuxième étape est réalisée.

L'opération *sort merge join* peut être choisie pour les inéquijointures (<, <=, >, ou >=) où elle offre de meilleur résultats que les algorithmes *nested loop* et *hash join* pour d'importants volumes de données.

Dans un plan d'exécution, la jointure tri-fusion se présente par les mots-clés `SORT JOIN` puis `MERGE JOIN` ; la table *outer* apparaît avant la table *inner*.

L'exemple suivant illustre une inéquijointure sur 3 tables. On recherche les adhérents de code supérieur à ceux qui pratiquent un sport de code est inférieur au plus petit des codes de rugby et de moto. Dans le jeu de test, du fait qu'aucun adhérent ne soit adepte de moto et que le code du rugby est le plus grand (823), tous les sports sont sélectionnés. Cette inéquijointure effectue deux tris sur des volumes approchant un méga-octets, puis fusionne (25 Go). Les statistiques

indiquent que le nombre de blocs lus n'est pas important mais le transfert réseau, les tris et la fusion expliquent le temps de réponse (25 minutes d'attente dans mon environnement).

Tableau 12-35 Inéquijointure mise en œuvre avec sort merge join

Requête

```
SELECT  a.adhid, a.nom, a.tel
    FROM  Adherent a, Sport s, Pratique p
    WHERE s.splibelle  IN ('Rugby','Moto')
    AND   a.adhid > p.adhid
    AND   s.spid  > p.spid;
```

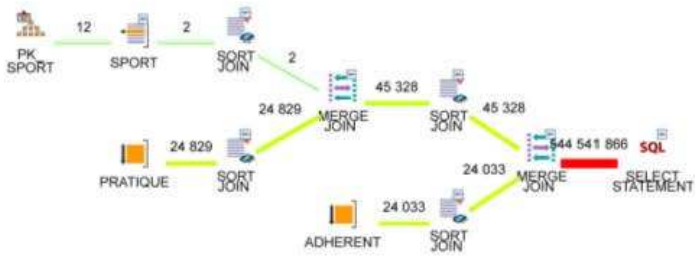
Résultats

```
Statistiques
----------------------------------------------------------
        303  consistent gets
21474946008  bytes sent via SQL*Net to client
  424319179  bytes received via SQL*Net from client
   38574435  SQL*Net roundtrips to/from client
          4  sorts (memory)
          0  sorts (disk)
  578616501  rows processed
```

```
| Id  | Operation                      | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |           |   544M|   25G|       |  3272  (83)| 00:00:40 |
|   1 |  MERGE JOIN                    |           |   544M|   25G|       |  3272  (83)| 00:00:40 |
|   2 |   SORT JOIN                    |           | 45328 |  929K|  2856K|   315   (2)| 00:00:04 |
|   3 |    MERGE JOIN                  |           | 45328 |  929K|       |    19  (11)| 00:00:01 |
|   4 |     SORT JOIN                  |           |     2 |   26 |       |     2   (0)| 00:00:01 |
|*  5 |      TABLE ACCESS BY INDEX ROWID| SPORT     |     2 |   26 |       |     2   (0)| 00:00:01 |
|   6 |       INDEX FULL SCAN          | PK_SPORT  |    12 |      |       |     1   (0)| 00:00:01 |
|*  7 |     SORT JOIN                  |           | 24829 |  193K|       |    17  (12)| 00:00:01 |
|   8 |      TABLE ACCESS FULL         | PRATIQUE  | 24829 |  193K|       |    15   (0)| 00:00:01 |
|*  9 |   SORT JOIN                    |           | 24033 |  680K|  1896K|   262   (1)| 00:00:04 |
|  10 |    TABLE ACCESS FULL           | ADHERENT  | 24033 |  680K|       |    68   (0)| 00:00:01 |
```

Comme l'illustre l'arbre suivant, le premier tri-fusion met en jeu la table `Sport` (*outer* car elle ne filtre que 2 lignes) et la table d'association `Pratique` (*inner* avec 24 829 lignes) du fait de leur colonne commune (`spid`). Le résultat de ce tri-fusion retourne 45 328 lignes. Le deuxième tri-fusion combine ce résultat (table *outer*) avec la table `Adherent` sans utiliser d'index pour restituer au final près de 590 millions de lignes.

Figure 12-11 *Jointures par tri-fusion*



Les *hints* `USE_MERGE(alias1 ...)` et `NO_USE_MERGE(alias1 ...)` permettent respectivement de forcer ou d'empêcher l'optimiseur d'utiliser la jointure par tri-fusion.

Algorithmes de jointure

D'autres algorithmes de jointure sont mis en œuvre par Oracle, citons :

- Les produits cartésiens (absence de clause de jointure dans la requête). Dans le plan d'exécution, vous trouverez l'opération `MERGE JOIN CARTESIAN`. L'optimiseur peut aussi décider d'utiliser cet algorithme pour joindre deux tables à faible volumétrie à une autre table de volume plus important.
- Les jointures externes unilatérales ou bilatérales (extractions de lignes ne correspondant pas aux critères de jointure). Dans le plan d'exécution, vous trouverez généralement les opérations `NESTED LOOPS OUTER`, `HASH JOIN OUTER` et `HASH JOIN FULL OUTER`.

Requêtes imbriquées

L'optimiseur traduit les requêtes imbriquées par des semi-jointures (*semi joins*) ou des anti-jointures (*anti joins*).

Les semi-jointures sont des jointures qui ont la particularité de ne pas parcourir la table *inner* complètement mais de stopper le parcours dès une certaine occurrence trouvée. Sont notamment concernées les requêtes utilisant les opérateurs `EXISTS` et `IN` (les jointures procédurales font donc partie de cette programmation).

Les antijointures sont des jointures qui ont la particularité d'extraire des lignes ne correspondant pas au prédicat de la requête imbriquée. Les requêtes qui sont concernées sont notamment celles qui utilisent l'opérateur `NOT EXISTS` et `NOT IN`.

Suivant la volumétrie des tables, la distribution des données et l'existence d'index, les plans d'exécution de vos semi-jointures utiliseront diverses opérations : `HASH JOIN RIGHT SEMI`, `HASH JOIN SEMI` ou `NESTED LOOPS SEMI` (l'opérateur historique par défaut). Concernant les antijointures, les opérateurs équivalents sont les suivants : `HASH JOIN ANTI`, `HASH JOIN RIGHT ANTI` et `NESTED LOOPS ANTI`.

Le tableau suivant décrit les opérateurs qui programment des antijointures par requête imbriquée. La volumétrie importante de la table `Adherenthis` explique que l'optimiseur choisisse l'algorithme *hash join* au détriment de *nested loop* (stratégie également choisie si on programme la requête avec `NOT IN`).

Tableau 12-36 Antijointures

Requête	Opérateur
`SELECT s.spid,s.splibelle FROM Sport s WHERE NOT EXISTS (SELECT p.adhid FROM Pratique p WHERE p.spid=s.spid);`	NESTED LOOPS ANTI
`SELECT s.spid,s.splibelle FROM Sport s WHERE s.spid NOT IN (SELECT p.adhid FROM Pratique p);`	NESTED LOOPS ANTI
`SELECT a.adhid, a.nom, a.tel FROM Adherentbis a WHERE NOT EXISTS (SELECT p.adhid FROM Pratique p WHERE p.adhid=a.adhid);`	HASH JOIN RIGHT ANTI

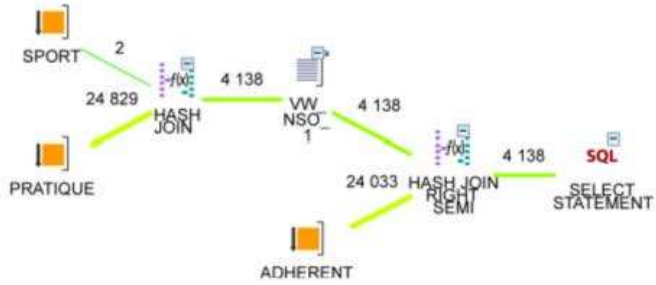
Le tableau suivant décrit les opérateurs qui programment des semi-jointures par requête imbriquée. Comme pour les antijointures, l'optimiseur choisit l'algorithme *hash join* au détriment du *nested loop* du fait de la volumétrie (stratégie également choisie si on programme la requête avec `IN`). La dernière requête écrite à la forme procédurale est celle étudiée aux sections *Hash joins* et *Nested loops*.

Tableau 12-37 Semi-jointures

Requête	Opérateur
`SELECT s.spid,s.splibelle FROM Sport s WHERE EXISTS (SELECT p.adhid FROM Pratique p WHERE p.spid=s.spid);`	NESTED LOOPS SEMI
`SELECT s.spid,s.splibelle FROM Sport s WHERE s.spid IN (SELECT p.adhid FROM Pratique p);`	NESTED LOOPS SEMI
`SELECT a.adhid, a.nom, a.tel FROM Adherentbis a WHERE EXISTS (SELECT p.adhid FROM Pratique p WHERE p.adhid=a.adhid);`	HASH JOIN RIGHT SEMI
`SELECT a.adhid, a.nom, a.tel FROM Adherent a WHERE a.adhid IN (SELECT p.adhid FROM Pratique p WHERE p.spid IN (SELECT s.spid FROM Sport s WHERE s.splibelle IN ('Escrime','Ping-pong')));`	HASH JOIN RIGHT SEMI

L'arbre suivant illustre la dernière jointure qui est programmée d'abord par une table de hachage puis par une semi-jointure. Elle offre des performances similaires à celle programmée exclusivement avec des algorithmes *hash joins*.

Figure 12-12 Semi-jointures



Le *hint* `NO_UNNEST` (à disposer dans la requête imbriquée) force l'optimiseur à n'utiliser ni semi-jointures ni antijointures. L'opérateur `FILTER` sera préféré (chaque ligne de la table *outer* examinera systématiquement plusieurs lignes de la table *inner*).

Autres algorithmes

D'autres algorithmes (cette liste n'est pas exhaustive) sont utilisés par l'optimiseur :

- `BUFFER SORT` qui utilise une table temporaire ou une zone de tri en mémoire pour stocker des données intermédiaires ; ces données ne sont pas nécessairement triées.
- `INLIST ITERATOR` qui est utilisé soit pour les clauses `IN` avec des valeurs, ou suite à des prédicats d'égalité liés par `OR`.
- `VIEW` qui gère un ensemble variable de données.
- `COUNT STOPKEY` qui programme une restriction définie via `ROWNUM`.
- `FIRST ROW` qui est déclenché par les opérateurs `MIN` et `MAX`.
- `FILTER` qui sert à éliminer une partie des lignes renvoyées par une autre étape (sous-interrogations et prédicats sur une seule table).
- `CONCATENATION` qui concatène plusieurs ensembles de lignes (suite à `UNION ALL` et des transformations de l'opérateur `OR`).

Variables de lien

L'utilisation des variables de lien est indispensable pour améliorer les performances d'une application OLTP (c'est moins vrai pour des applications OLAP et des traitements *batch*). En effet, chaque nouvelle instruction se traduit par *hard parse* souvent coûteux (surtout en mode multi-utilisateur) du fait de la pose de verrous internes.

Oracle assigne une valeur (*hash value*) à chaque nouvelle exécution d'une instruction SQL (cette valeur est visible au début du plan d'exécution). Toute modification du code de cette instruction (par exemple le nom d'un alias `a.nom` et `b.nom` ou d'une valeur `adhid=1` et `adhid=2`) donnera lieu à une nouvelle valeur *hash* générant à nouveau un *hard parse*.

L'optimiseur traite plusieurs écritures en tant que *bind variable* : les variables de session (substitution), les variables PL/SQL, les variables SQL dynamique (avec la directive `USING`). Les variables des instructions de toute API du marché (ODBC, ADO, JDBC, etc.) sont aussi considérées *bind variables*. Dans le cas de Java, il s'agit par exemple de la classe `PreparedStatement`. Le tableau suivant illustre quelques instructions contenant des *bind variables*.

Tableau 12-38 Différentes implémentations de bind variables

Contexte	Codage

SQL*Plus	`VARIABLE g_nom VARCHAR2(25)`
	`BEGIN`
	`  :g_nom := 'LARRAZET';`
	`END;`
	`SELECT prenom, tel FROM Adherentbis WHERE nom = :g_nom`
	`  AND civilite = 'Mme.' AND tel LIKE '05%';`

Code PL/SQL	`DECLARE`
	`  v_nom Adherentbis.nom%TYPE;`
	`  v_p  Adherentbis.prenom%TYPE;`
	`  v_t  Adherentbis.tel%TYPE;`
	`BEGIN`
	`  v_nom := 'LARRAZET';`
	`  SELECT prenom, tel INTO v_p, v_t FROM Adherentbis`
	`  WHERE nom = v_nom AND civilite = 'Mme.' AND tel LIKE '05%';`
	`END;`

SQL dynamique	`…`
	`  EXECUTE IMMEDIATE`
	`  'SELECT prenom,tel FROM Adherentbis  WHERE nom = :v_nom`
	`  AND civilite = ''Mme.'' AND tel LIKE ''05%'' '`
	`  INTO v_p, v_t USING v_nom;`

Attention ! Ne confondez pas une instruction construite dynamiquement dans le code et une instruction intégrant des *bind variables*. En conséquence, n'utilisez pas de signe concaténation (`||`) dans vos instructions au profit du SQL dynamique.

Pour vous en convaincre, exécutez le script en téléchargement qui compare l'extraction des 250 000 premiers adhérents. Le premier bloc utilise le curseur suivant (qui ne fait pas usage de variables de lien) `OPEN curseur FOR 'SELECT nom FROM Adherentbis WHERE adhid='||i`. La seconde solution utilise une variable de lien : `OPEN curseur FOR 'SELECT`

nom FROM Adherentbis WHERE adhid=:v_ad' USING i. Vous constaterez que la seconde solution s'exécute bien plus rapidement (de l'ordre de 85 % de gain).

Lorsque l'optimiseur prend en compte le premier plan d'exécution généré avec des *bind variables*, on parle de *bind peeking*. Depuis la version 11*g*, Oracle peut construire plusieurs plans d'exécution pour une même requête (avec des valeurs différentes pour les *bind variables* associées) : il s'agit de *l'adaptive cursor sharing*.

Comment réaliser des fetchs multilignes ?

Il est toujours préférable de retourner plusieurs lignes par *fetch*, on parle de *array fetch*. Plusieurs solutions existent ; citons la commande SET ARRAYSIZE sous SQL*Plus et l'option BULK COLLECT de l'instruction FETCH dans un bloc PL/SQL (consulter la section *Utilisation de LIMIT et BULK COLLECT* du chapitre 7).

Le tableau suivant présente une comparaison sous SQL*Plus de ces deux modes d'exécution. La requête pratiquant le *fetch* monoligne s'exécute en 30 secondes pour extraire plus d'un million d'adhérents. Le *fetch* multiligne réduit ce temps à 3 secondes. Le nombre de blocs lus sur disque est identique ; en revanche, le nombre de blocs manipulés en mémoire et transférés sur le réseau est minimisé par la lecture multiligne.

Tableau 12-39 Fetch monoligne vs multiligne

1 ligne par *fetch* (par défaut)	100 lignes par *fetch*
SQL> SET ARRAYSIZE 1 SQL> SET AUTOTRACE TRACEONLY SQL> SELECT a.adhid,a.nom,a.tel,a.date_nais FROM Adherentbis a; 1177618 ligne(s) sélectionnée(s). Ecoulé : 00 :00 :**30.46** Statistiques ---------- **594028** consistent gets **10224** physical reads **124308135** bytes sent via SQL*Net to client **6477304** bytes received via SQL*Net from client **588810** SQL*Net roundtrips to/from client	SQL> SET ARRAYSIZE **100** SQL> SET AUTOTRACE TRACEONLY SQL> SELECT a.adhid,a.nom,a.tel,a.date_nais FROM Adherentbis a; 1177618 ligne(s) sélectionnée(s). Ecoulé : 00 :00 :03.21 Statistiques ---------- **21911** consistent gets **10224** physical reads **49294367** bytes sent via SQL*Net to client **129952** bytes received via SQL*Net from client **11778** SQL*Net roundtrips to/from client

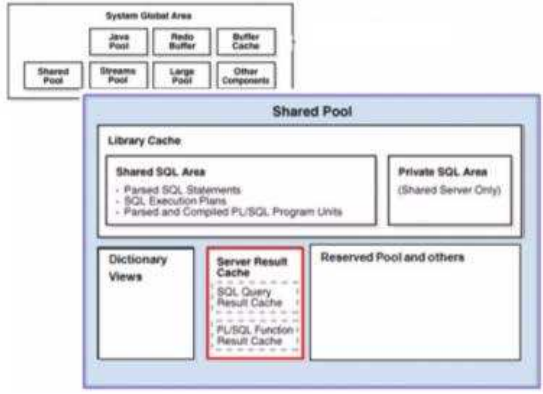
Sous SQL*Plus, la valeur de `ARRAYSIZE` peut être comprise entre 1 et 5 000. L'efficacité dépend aussi de la configuration *middleware* et des capacités du réseau. Une valeur excessive de ce paramètre peut nuire aux performances.

Le nombre de lignes à extraire par lecture doit être positionné entre 100 et 200 (cette valeur devrait convenir à la majorité des applications).

Gestion du cache

Depuis la version 11*g*, Oracle dispose d'un cache de résultat, faisant partie de l'espace mémoire qu'il gère (la SGA), et dont les requêtes SQL et les fonctions PL/SQL peuvent bénéficier. Il est très intéressant pour des requêtes coûteuses et répétitives qui nécessitent d'être exécutées à chaque appel, même si le plan d'exécution est préétabli et les blocs de données sont tous en mémoire. Il va sans dire qu'après toute mise à jour d'une donnée dans le cache, celui-ci est invalidé avant de se reconstruire à l'interrogation suivante.

Figure 12-13 *Cache SQL et PL/SQL (© doc. Oracle)*



Les paramètres de votre cache disponible sont visibles à l'aide de la requête suivante :

```
SQL> SELECT name, value FROM v$parameter
                        WHERE name LIKE 'result_cache%';
NAME                           VALUE
------------------------------ ----------
result_cache_mode              MANUAL
result_cache_max_size          4292608
result_cache_max_result        5
result_cache_remote_expiration 0
```

La taille ici est de 4 Mo et le pourcentage du cache utilisable pour un résultat est de 5 % (par défaut). Le cache est modifiable par `ALTER SESSION SET RESULT_CACHE_MODE = {FORCE|MANUAL}` ou `ALTER SYSTEM`. La valeur `MANUAL` est celle adoptée par défaut, la valeur `FORCE` l'active pour la session ou pour les sessions à venir. Cette dernière n'est pas recommandée puisqu'elle aurait pour conséquence la mise en cache de toutes les requêtes, même celles qui sont non déterministes (dont le résultat peut changer avec les mêmes paramètres en entrée). Les autres paramètres sont également modifiables au niveau du système seulement (pas de la session).

Pour vos tests, sachez que `ALTER SYSTEM FLUSH SHARED_POOL` vide la SGA mais pas le cache, tandis que `ALTER SYSTEM FLUSH BUFFER_CACHE` vide seulement le cache.

Cache pour les requêtes

L'activation et la désactivation du cache au niveau d'une requête peut se programmer à l'aide de hints : `/*+RESULT_CACHE*/` fait passer le paramètre `RESULT_CACHE_MODE` à `FORCE` et `/*+NO_RESULT_CACHE*/` fait passer le même paramètre à `MANUAL`.

Si vous avez déjà mis en place le cache avec `ALTER SESSION` ou `ALTER SYSTEM` et qu'une requête utilise un des hints précités, le hint l'emporte.

Comme les résultats du cache doivent être non aléatoires, vous ne devez pas utiliser des requêtes avec des tables temporaires, séquences, pseudo-colonnes ou expressions avec une date variable (`SYSDATE` par exemple).

Le tableau 12-40 présente les deux écritures de la même requête qui dispose le résultat en cache. Vous devrez peut-être exécuter plusieurs fois de suite la requête pour bénéficier du cache. Dans le plan d'exécution, vous retrouverez la preuve de l'utilisation du cache et la requête associée à l'aide de la vue `v$result_cache_objects`.

Tableau 12-40 Utilisation du cache pour les requêtes

Avec un hint

```
SQL> SELECT a.adhid,a.civilite…
     FROM adherent a …

     ADHID CIVIL PRENOM      NOM
---------- ----- ----------- --------
     27000 Mlle. JENIFER     THIRIET
Ecoulé : 00 :00 :37.71

SQL> SELECT /*+ RESULT_CACHE */
     a.adhid,a.civilite…
     FROM adherent a …

SQL> SELECT /*+ RESULT_CACHE */
…

     ADHID CIVIL PRENOM     NOM
---------- ----- ---------- ---------
     27000 Mlle. JENIFER    THIRIET
Ecoulé : 00 :00 :00.00
```

Avec le paramètre d'initialisation

```
ALTER SESSION SET RESULT_CACHE_MODE = FORCE;
SELECT…

Plan d'exécution
-----------------------------------------------
| Id  | Operation             | Name         | R
-----------------------------------------------
|   0 | SELECT STATEMENT      |              | 2
|   1 |  RESULT CACHE         | a8c0hndk7y0wb56uuzgm4ctayu |

SQL> SELECT id, type, creation_timestamp, status, name
     FROM v$result_cache_objects
     WHERE cache_id = 'a8c0hndk7y0wb56uuzgm4ctayu';
        ID TYPE       CREATION STATUS    NAME
---------- ---------- -------- --------- -----
         4 Result     12/07/14 Published
         SELECT a.adhid,a.civilite,a.prenom,a.nom
         FROM adherent a WHERE NOT EXISTS…
```

Cache pour les fonctions PL/SQL

Avant la version 11*g*, la zone de cache (en PGA) qui était utilisée par les tableaux associatifs dans les paquetages (*package level collections*, voir l'exemple de la section « Comment retourner une table » du chapitre 6) était spécifique à chaque session. Il pouvait se produire des problèmes en cas de montée en charge. Depuis la version 11*g*, une fonction peut retourner des résultats mis en cache disponibles pour d'autres sessions ; il s'agit de la fonction *result cache*.

Disposée au niveau de la création d'une fonction PL/SQL, l'option `RESULT_CACHE` permet de bénéficier d'une zone de cache pour stocker les résultats à chaque appel avec de nouveaux paramètres. L'option `RELIES_ON` (implicite depuis la version 11*g*R2) assure que toute mise à jour de données qui pourrait invalider le cache de résultat, entre deux appels identiques, déclenche une invalidation dudit résultat et une réévaluation.

Il est recommandé que votre fonction ait le moins possible d'effets de bord (modification d'une table, écriture d'une trace `DBMS_OUTPUT` ou envoi d'un e-mail, par exemple). De plus, méfiez-vous des fonctions utilisant des données dont le format peut dépendre des paramétrages d'une session (`NLS_DATE_FORMAT` ou `TIME ZONE`, par exemple).

Le tableau 12-41 présente une fonction qui retourne la concaténation du nom et du téléphone d'un adhérent dont le numéro passe en paramètre. Cette fonction est appelée 101 fois dans le bloc (100 fois avec le même numéro) et une fois avec un autre numéro. Le résultat est que seuls deux appels à la fonction sont opérés et 99 accès aux caches sont effectués…

Tableau 12-41 Utilisation du cache de résultat pour une fonction PL/SQL

Fonction result cache	101 appels de la fonction

`CREATE FUNCTION f_adhlu_cache`
`(p1 IN adherent.adhid%TYPE)`
`RETURN VARCHAR2`
`RESULT_CACHE`
`RELIES_ON(adherent)`
`IS`
`result VARCHAR2(100);`
`BEGIN`
`SELECT nom||'-'||tel INTO result`
`FROM adherent WHERE adhid = p1;`
`compteur.incremente();`
`RETURN result;`
`END f_adhlu_cache;`
`/`
	`SQL> DECLARE`
	`adhid_in adherent.adhid%TYPE := 27342;`
	`3 v_res VARCHAR2(100);`
	`4 BEGIN`
	`5 FOR i IN 1 .. 100`
	`6 LOOP`
	`7 v_res := f_adhlu_cache(adhid_in);`
	`8 END LOOP;`
	`9`
	`10 v_res := f_adhlu_cache(27504);`
	`11 DBMS_OUTPUT.PUT_LINE('Compteur : '||compteur.affiche());`
	`12 END;`
	`13 /`
	`Compteur : 2`

Plusieurs restrictions concernent les fonctions *result cache* :

- elle ne doit pas être *pipelined* ;
- elle ne peut disposer de paramètres de retour (`OUT` ou `IN OUT`) ;
- les paramètres d'entrée sont simples (pas de `BLOB`, `CLOB`, `NCLOB`, `REF CURSOR`, collections, objets ou *record*) ;
- le type de retour est simple (pas de `BLOB`, `CLOB`, `NCLOB`, `REF CURSOR`, objets). Les collections ou *record* ne doivent pas contenir un type précité.

Cache pour les tables

Depuis la version 11*g*R2, le cache de résultat peut être aussi utilisé au niveau d'une table (instructions concernées : `CREATE TABLE` et `ALTER TABLE`). L'option `RESULT_CACHE (MODE {FORCE | DEFAULT})` sert à activer ou désactiver le cache pour tout accès à la table. Une autre possibilité lors de la création d'une table consiste à employer les directives `CACHE` ou `NOCACHE`.

Dans la première écriture, `DEFAULT` caractérise le comportement par défaut d'une table (à savoir pas de mise en cache si ce n'est les blocs de données qui montent en mémoire naturellement lors de toute extraction). Cette option diffère toute décision de l'utilisation du cache par un paramètre système, de session, ou finalement un hint sur une requête particulière. Dans le second type d'écriture, `NOCACHE` exprime le même état de fait.

Pour mettre en cache une jointure, vous devrez effectuer ce procédé pour toutes les tables concernées. Il indique que les blocs extraits de ces tables sont placés en bonne place au niveau de la liste LRU (*Least Recently Used*) dans le *buffer cache*. Pour les tables de taille réduite qui ne sont pas mises à jour très fréquemment, c'est très efficace.

Le tableau 12-42 présente la mise en cache de deux tables existantes et la création d'une nouvelle table mise en cache dès le début. La colonne `result_cache` du dictionnaire de données renseigne l'état de chaque table. En traçant vos requêtes, vous constaterez que l'utilisation du cache, à partir du deuxième appel, fait passer le nombre de *gets* à zéro.

Tableau 12-42 Utilisation du cache de table

En modification	En création

```
SQL> ALTER TABLE sport RESULT_CACHE (MODE FORCE);
SQL> ALTER TABLE pratique RESULT_CACHE (MODE FORCE);

SQL> SELECT table_name, result_cache
  2  FROM   user_tables
  3  WHERE  table_name IN
           ('PRATIQUE','SPORT', 'ADHERENT');
TABLE_NAME RESULT_CACHE
---------- ---------------
PRATIQUE   FORCE
ADHERENT   DEFAULT
SPORT      FORCE
```

```
SQL> CREATE TABLE adherentes
  2  RESULT_CACHE (MODE FORCE)
  3  AS SELECT *
  4    FROM   adherent
  5    WHERE  civilite IN ('Mme.','Mlle.');

Table créée.
```

Figure 12-14 *Trace d'une jointure*

```
SQL> SET AUTOTRACE TRACEONLY
SQL> SELECT  s.splibelle, p.adhid
  2  FROM    pratique p, sport s
  3  WHERE   p.adhid = 777
  4  AND     p.spid=p.spid;
Plan d'exécution
----------------------------------------------------------
| Id  | Operation             | Name
----------------------------------------------------------
|   0 | SELECT STATEMENT      |
|   1 |  RESULT CACHE         | 55npxxz5z20fd511b6f0szukrd
|   2 |   MERGE JOIN CARTESIAN|
|*  3 |    INDEX RANGE SCAN   | IDX_PRATIQUE_ADHID
|   4 |    BUFFER SORT        |
|   5 |     TABLE ACCESS FULL | SPORT
----------------------------------------------------------
```

Figure 12-15 *Trace d'un accès full*

```
SQL> SET AUTOTRACE TRACEONLY
SQL> SELECT  nom, prenom
  2  FROM    adherentes
  3  WHERE   adhid < 777;
Plan d'exécution
----------------------------------------------------------
| Id  | Operation           | Name                      |
----------------------------------------------------------
|   0 | SELECT STATEMENT    |                           |
|   1 |  RESULT CACHE       | 3yr3qfadv7cyd91uyyjj3vff2j |
|*  2 |   TABLE ACCESS FULL| ADHERENTES                 |
----------------------------------------------------------
```

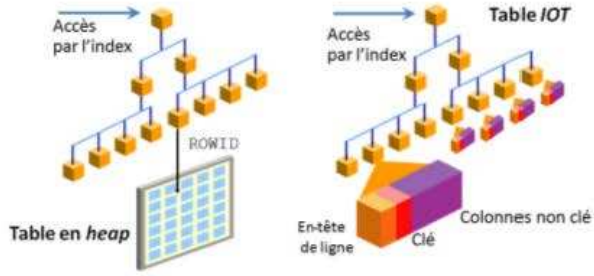
Le cache n'est pas possible pour les tables organisées en index (IOT).

Tables organisées en index

Évoquée au chapitre 1, une table organisée en index (IOT, *index-organized table*) est stockée physiquement dans une structure d'index *B-tree*. Chaque entrée d'index (blocs feuilles) contient alors les lignes entières de la table, au lieu de comporter `ROWID` dans le cas d'un index classique accédant à une table classique (en *heap*). La clé primaire d'une table IOT peut être composée de plusieurs colonnes.

Les tables IOT sont particulièrement intéressantes à utiliser si on désire rapprocher physiquement des données ou les disposer dans un ordre particulier (base de données spatiales et multidimensionnelles). Ce mécanisme existe aussi avec IBM DB2 et se nomme *clustered index* avec Microsoft SQL-Server. Depuis la version 9*i* d'Oracle, il est également possible de créer des index additionnels (sur des colonnes qui ne sont pas des clés). Le besoin de stockage est en théorie réduit, car les colonnes clés ne sont pas dupliquées dans une table et dans un index.

Figure 12-16 *Tables organisées en index*



Il est impératif de spécifier une clé primaire à la création d'une table organisée en index. La contrainte ne doit pas être déclarée `DEFERRABLE`.

Par construction, les tables IOT offrent un accès rapide pour les extractions basées sur une égalité de la clé primaire. Les recherches par inégalité ou intervalle sur la clé ne sont pas pénalisantes pour autant.

Comparatif

Le tableau suivant compare les caractéristiques des tables classiques (*heap-organized*) et des tables IOT.

Tableau 12-43 Comparatifs des tables heap-organized /index-organized

Caractéristique	Tables en *heap*	Tables IOT
Identification	Le *rowid* identifie toute ligne. Une clé primaire peut être déclarée.	La clé primaire doit être déclarée pour identifier toute ligne.
Rowid et indexage secondaire	La pseudo-colonne `ROWID` (adresse physique) permet un indexage secondaire.	La pseudo-colonne `ROWID` (adresse logique) permet un indexage secondaire.
Accès direct	Toute ligne peut être accessible directement par son *rowid*.	Toute ligne est accessible indirectement par la clé primaire.
Parcours entier	`full scan table` retourne toutes les lignes dans un certain ordre.	`index full scan` ou `fast full index scan` retournent toutes les lignes dans un ordre similaire.
Clusters	Peut être stockée dans un *cluster* avec d'autres tables.	Pas de *cluster* possible.
Colonnes multimédia	Peut contenir des colonnes `LONG` et *LOB*.	Peut contenir des colonnes *LOB* mais pas des `LONG`.
Colonnes virtuelles	Autorisé.	Interdit.

Une table organisée en index peut être utilisée pour des tables de référence, des tables d'association ou des tables auxquelles on accède toujours via leur clé primaire.

Les tables ayant peu de colonnes qui ne sont pas des clés et de taille relativement restreinte peuvent également être de bonnes candidates.

Les débordements

Si les lignes de la table IOT sont volumineuses, vous devrez gérer les segments de débordement (*overflow segments*). Ce problème ne se pose pas pour les tables associées à des index *B-tree* car la taille de chaque entrée d'index est relativement réduite (valeur de la clé et *rowid* associés).

Lorsqu'une zone de débordement est définie (directive `OVERFLOW`), chaque ligne de la table IOT est divisée en deux parties :

- L'index qui contient les colonnes clé primaire, un *rowid* qui référence le reste de la ligne dans la zone de débordement et, éventuellement, quelques colonnes qui ne sont pas des clés. Tout ceci est stocké dans le segment d'index.
- Le reste des colonnes de la ligne est stocké dans le segment de débordement.

Ce mécanisme présente un inconvénient majeur lorsque la majorité des accès aux données s'effectue dans les segments de débordement ; l'efficacité de la table organisée en index est alors considérablement réduite.

Création d'une IOT

La directive `ORGANIZATION INDEX` de l'instruction `CREATE TABLE` permet de définir une table (une clé primaire doit aussi être déclarée). Il existe plusieurs options :

- `OVERFLOW` préserve la densité de l'index permettant le stockage de colonnes non-clés dans un segment séparé (de débordement).
- `PCTTHRESHOLD` (valeur comprise entre 1 et 50, 50 par défaut) précise le pourcentage d'espace libre réservé dans un bloc d'index pour chaque ligne. La colonne non-clé (et les suivantes) qui dépasseront cette taille seront stockées dans le segment de débordement.
- `INCLUDING` indique le nom d'une colonne non-clé à partir de laquelle sera effectuée la séparation entre le segment d'index (privilégiant ainsi les accès à cette colonne et aux précédentes) et le segment de débordement.
- `COMPRESS` ne concerne que les tables IOT dont la clé primaire est composée de plusieurs colonnes ; l'index est alors compressé en fonction de valeurs communes de certains préfixes.

Comparaison avec une table en heap

Afin de comparer avec la table en *heap* `Opérateurs` et son index *B-tree* sur la clé primaire (colonne `opeid`), définissons la table organisée en index `Operateurs_iot` de la manière suivante.

Tableau 12-44 Création d'une table IOT

Code SQL	Commentaires

`CREATE TABLE Operateurs_iot`
`(opeid CHAR(4) NOT NULL,`
`nomope VARCHAR(25) NOT NULL,`
`…`
	Colonnes de la table.

`CONSTRAINT pk_Operateurs_iot` `PRIMARY KEY (opeid))`	Définition de la clé primaire.
`ORGANIZATION INDEX` `TABLESPACE tbs_index`	Le segment d'index se trouvera dans le *tablespace* `tbs_index`.
`INCLUDING` `creaope`	Les 3 premières colonnes seront dans le segment d'index.
`PCTTHRESHOLD` `20`	Si la taille d'une ligne dépasse 20 % de la taille d'un bloc, alors la colonnes et les suivantes seront stockées dans le *tablespace* `users`.
`OVERFLOW` `TABLESPACE users;`	

En analysant la requête qui extrait le code et le nom d'une dizaine d'opérateurs (prédicat `WHERE opeid IN ('SFR','Orange','M6T',…)`), il apparaît que 3 fois moins de blocs sont extraits da la table IOT que de la table en *heap*.

Limites

Une table IOT ne peut ni contenir plus de 1 000 colonnes, ni définir une clé primaire de plus de 32 colonnes.

Sans clause de débordement, le nombre de colonnes contenues dans le segment d'index est limité à 255.

Partitionnement

Le partitionnement permet de décomposer une table volumineuse (et ses index) en parties de taille plus réduite : les partitions. Chaque partition est un objet (au sens Oracle, table ou index) nommé, composé de la même structure (colonnes et contraintes) et disposant de ses propres caractéristiques de stockage (*tablespace*, options de blocs, etc.).

L'accès aux partitions est transparent ; si vous adoptez le partitionnement en évolution d'une implémentation classique, vous ne devrez pas réécrire le code. La stratégie de partitionnement pour une table comme pour un index présente de nombreux autres avantages :

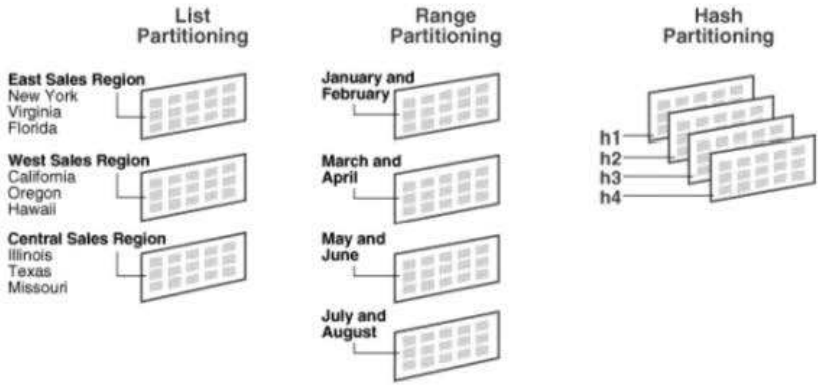
- maintenance et administration plus précise (ajout, suppression, fusion, division, modification d'une partition) ;
- disponibilité accrue : le fait qu'une partition soit indisponible n'implique pas l'indisponibilité de la partition entière. L'optimiseur supprime automatiquement du plan d'accès les partitions indisponibles ;
- réduction des contentions sur des ressources partagées (bases OLTP) et amélioration des requêtes sur des *datawarehouses* (bases OLAP).

La clé de partition

La clé de partition est composée d'une ou de plusieurs colonnes qui déterminent la partition d'accueil de la ligne en question. Chaque ligne est affectée à une seule partition.

Les stratégies basiques de partitionnement (*single level partitioning*) sont apparues progressivement : par intervalle (*range partitioning* en version 8.0), par hachage (*hash partitioning* en version 8.1) et par liste de valeurs (*list partitioning* en version 9.0). La figure suivante illustre les types de partitions possibles : la liste conduit à répartir les données selon les noms des régions, l'intervalle à un classement de manière temporelle (ici tous les deux mois) et le hachage à une présentation homogène en utilisant un algorithme interne.

Figure 12-17 *Stratégies basiques de partitionnement (© doc. Oracle)*



Les tables qui sont de très bonnes candidates au partitionnement :

- Sont de taille supérieure à 2 Go.
- Contiennent des données historiques pour lesquelles des informations récentes s'ajoutent à une nouvelle partition. Par exemple, une table contenant des données modifiables sur le mois en cours et où les information concernant les mois précédents sont en lecture seule.
- Sont celles qui nécessitent différentes unités de stockage.

Le partitionnement ne peut pas s'appliquer à une table en *cluster* ou à une table contenant une colonne de type `LONG` ou `LONG RAW`.

Intéressons-nous à présent à l'application de ces différentes techniques de partitionnement dans le but d'améliorer les performances des requêtes. Nous étudierons ensuite l'utilisation des index partitionnés locaux et globaux.

Partitions par intervalle

La directive `PARTITION BY RANGE` de l'instruction `CREATE TABLE` précise les colonnes de partition et définit les intervalles.

Avec le partionnement par intervalle, les lignes de la table sont réparties en fonction des valeurs de la clé de partitionnement. La directive `VALUES LESS THAN` indique la limite supérieure (sans l'inclure) de chaque partition. Toute ligne dont la colonne clé de la partition est égale ou supérieure à cette limite sera disposée automatiquement dans une des partitions sui-

vantes. Toutes les partitions, à l'exception de la première, disposent implicitement d'une limite basse (limite haute de la partition qui précède).

Le mot-clé `MAXVALUE` est utilisé dans la dernière partition.

En se basant sur la table `Adherentbis` qui contient des adhérents nées entre le 1/1/1920 et le 31/12/2005, comparons les deux implémentations (avec et sans partition). La table partitionnée en intervalles `Adherent_partition_range` est créée de la manière suivante : trois partitions sont définies, chacune dans un *tablespace* distinct. La première partition (`retraites`) contiendra les adhérents nés avant le 1er janvier 1945, la deuxième (`actifs`) contiendra les adhérents nés entre le 1er janvier 1945 et le 1er janvier 1993 et la dernière (`mineurs`) sera peuplée des adhérents nés après le 1er janvier 1993.

Tableau 12-45 Création d'une table partitionnée en intervalles

Code SQL	Commentaires

`CREATE TABLE Adherent_partition_range`
`(adhid NUMBER(10) NOT NULL,`
`nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL,`
`civilite VARCHAR(12) NOT NULL,`
`date_nais DATE NOT NULL,`
`tel VARCHAR2(20), solde NUMBER(8,2))`
	Colonnes de la table.

`PARTITION BY RANGE (date_nais)`
`(PARTITION retraites VALUES LESS THAN`
`(TO_DATE('01/01/1945','DD/MM/YYYY'))`
`TABLESPACE tbs_part1,`
`PARTITION actifs VALUES LESS THAN`
`(TO_DATE('01/01/1993','DD/MM/YYYY'))`
`TABLESPACE tbs_part2,`
`PARTITION mineurs VALUES LESS THAN (MAXVALUE)`
`TABLESPACE tbs_part3);`
	Définition de la clé de partition.
	La première partition se trouvera dans le *tablespace* `tbs_part1`, etc.

Afin de comparer les deux implémentations, considérons une requête qui extrait l'identité des adhérents selon la date de naissance (prédicat du type `WHERE TO_CHAR(a.date_nais,'DD/MM/YYYY') >= '01/01/1920' AND TO_CHAR(a.date_nais,'DD/MM/YYYY') < '01/01/1945'`). Sans aborder l'indexation, un gain de l'ordre de 40 % apparaît déjà au niveau des lectures physiques.

Intervalles automatiques

Que faire lorsque vous désirez créer des partitions d'intervalles portant sur des étendues de valeurs identiques sans connaître le nombre de partitions dont vous disposerez, la clé de partition pouvant évoluer dans le temps ? Dans ce cas, utilisez les partitions à intervalles automatiques.

Le mot-clé INTERVAL permet de gérer automatiquement les partitions *range* (en fixant une étendue de valeurs) sans devoir lister exhaustivement les intervalles. L'inconvénient est qu'il est impossible de disposer ces partitions dans des espaces prédéfinis car leur nombre est, a priori, inconnu.

Une partition initiale doit être définie, fixant le premier intervalle.

« Partionnons » la table des adhérents selon le solde (valeur comprise entre 0 et 30 000) en considérant des tranches de salaires de 8 000 euros. La première partition (p1) contiendra les adhérents dont le solde est inférieur à 8 001 euros. Les autres partitions seront créées automatiquement lors de l'insertion de lignes et selon la valeur de la colonne solde. Le nom des partitions peut varier d'une session à l'autre.

Tableau 12-46 Création d'une table partitionnée en intervalles automatiques

Code SQL	Visualisation des partitions

`CREATE TABLE Adherent_part_range_interval`
`(adhid NUMBER(10) NOT NULL, …`
`tel VARCHAR2(20),`
`solde NUMBER(8,2))`
`PARTITION BY RANGE (solde)`
`INTERVAL (8000)`
`(PARTITION p1 VALUES LESS THAN (8001));`
	`SELECT partition_name, high_value`
	`FROM USER_TAB_PARTITIONS`
	`WHERE table_name = 'ADHERENT_PART_RANGE_INTERVAL'`
	`ORDER BY partition_name;`
	
	`PARTITION_NAME HIGH_VALUE`
	`--------------- ---------`
	`P1 8001`
	`SYS_P35 16001`
	`SYS_P36 32001`
	`SYS_P37 24001`

Sans considérer l'indexation, l'optimiseur n'accédera qu'à une seule partition (économie sur les blocs lus) pour les prédicats filtrant un solde particulier (ou un intervalle contenu dans une partition). L'accès à une seule partition est décelé dans un plan d'exécution par la présence de l'opérateur PARTITION RANGE SINGLE, l'accès à plusieurs par PARTITION RANGE INLIST et l'accès à toutes par PARTITION RANGE ALL.

La clé d'un partitionnement par intervalle ne peut pas être du type ROWID, LONG, LOB, XMLType, ou TIMESTAMP WITH TIME ZONE.

Partitions par hachage

La directive PARTITION BY HASH de l'instruction CREATE TABLE précise les colonnes de partition, dénombre les partitions et indique les *tablespaces*.

Avec un partitionnement par hachage, les lignes de la table sont réparties selon un algorihme interne en fonction des valeurs de la clé de partitionnement. La répartition est homogène entre les partitions qui sont de tailles à peu près identiques. Ce mécanisme convient parfaitement lorsque aucune sémantique n'intervient dans la clé de partionnement.

Divisons la table des adhérents en quatre partitions par hachage sur le prénom. La table partitionnée `Adherent_partition_hash` est créée de la manière suivante. Si le nombre d'espaces défini est inférieur au chiffre indiqué dans `PARTITIONS`, Oracle gère les partitions d'une manière cyclique (pour 5 partitions et 3 *tablespaces*, la quatrième partition ira dans le premier *tablespace* et la dernière partition se trouvera dans le deuxième *tablespace*).

Tableau 12-47 Création d'une table partitionnée par hachage

Code SQL	Commentaires

`CREATE TABLE Adherent_partition_hash`
`(adhid NUMBER(10) NOT NULL,`
`nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL,`
`civilite VARCHAR(12) NOT NULL,`
`date_nais DATE NOT NULL,`
`tel VARCHAR2(20), solde NUMBER(8,2))`
`PARTITION BY HASH(prenom)`
`PARTITIONS 4`
`STORE IN (tbs_part1,tbs_part2,tbs_part3,tbs_part4);`
	Colonnes de la table.
	
	Définition de la clé de partition et du nombre de partitions. La première partition se trouvera dans le *tablespace* `tbs_part1`, etc.

Dans ce type de partitionnement, n'espérez pas des gains de performances pour toutes vos requêtes. Néanmoins, sans index, l'optimiseur n'accédera qu'à une seule partition (économie de 75 % sur les blocs lus) pour les prédicats filtrant un prénom particulier, deux partitions au plus pour les prédicats filtrant deux prénoms (gain de 50 %), etc. Les opérateurs que vous verrez apparaître dans vos plans d'exécution sont `PARTITION HASH SINGLE` (pour une égalité sur un prénom), `PARTITION HASH INLIST` (pour une comparaison avec un ensemble de prénoms) ou `PARTITION HASH ALL` (parcours de toutes les partitions).

Partitions par liste

La directive `PARTITION BY LIST` de l'instruction `CREATE TABLE` précise les colonnes de partition et liste les valeurs de la clé.

Le partitionnement par liste, répartit les lignes de la table selon les valeurs de la clé de partitionnement. L'avantage de ce mécanisme est qu'il permet de faire intervenir une sémantique dans la clé de partionnement (pas de notion d'ordre comme pour les intervalles).

La partition DEFAULT permet d'éviter de lister exhaustivement toutes les valeurs possibles de la clé de partitionnement en ainsi d'inclure toutes les lignes de la table d'une manière exclusive par rapport à leur partition d'accueil.

Fragmentons les adhédents en trois partitions par liste suivant leur sexe. La table partitionnée Adherent_partition_list est créée de la manière suivante.

Tableau 12-48 Création d'une table partitionnée par liste

Code SQL	Commentaires
`CREATE TABLE Adherent_partition_list (adhid NUMBER(10) NOT NULL, nom VARCHAR(25) NOT NULL, prenom VARCHAR(30) NOT NULL, civilite VARCHAR(12) NOT NULL, date_nais DATE NOT NULL, tel VARCHAR2(20), solde NUMBER(8,2))`	Colonnes de la table.
`PARTITION BY LIST (civilite) (PARTITION femmes VALUES ('Melle.', 'Mme.') TABLESPACE tbs_part1, PARTITION hommes VALUES ('Mr.') TABLESPACE tbs_part2, PARTITION autres VALUES (DEFAULT) TABLESPACE tbs_part3);`	Définition de la clé de partition. La première partition se trouvera dans le *tablespace* tbs_part1, etc.

L'optimiseur n'accédera qu'à une seule partition (économie de 50 % sur les blocs lus) pour les prédicats filtrant un sexe particulier. Les opérateurs que vous verrez apparaître dans vos plans d'exécution sont PARTITION LIST SINGLE (pour une comparaison d'égalité), PARTITION LIST INLIST (pour une comparaison avec un ensemble de sexes) ou PARTITION LIST ALL (parcours de toutes les partitions).

Partitions par référence

Si la table à partitionner est une table de référence (des clés étrangères d'autres tables pointent vers elle), il est possible de partitionner les tables enfants (*reference-partitioned table*) de la même manière que la table parent. Ce mécanisme n'est pas limité à une table enfant avec sa table parent mais peut être utilisé en cascade.

Ce dispositif offre plusieurs avantages car il rapproche physiquement des lignes qui étaient déjà reliées logiquement. Des jointures peuvent être plus performantes si des index locaux sont aussi créés. De plus, toute gestion d'une partition affectera des lignes de différentes tables mais d'une manière cohérente et homogène.

La directive PARTITION BY REFERENCE de l'instruction CREATE TABLE spécifie le nom de la contrainte de clé étrangère entre la table enfant et la table de référence. Cette contrainte doit être vérifiée et active.

Fragmentons la table des opérateurs selon l'année de création en trois partitions disposées dans des *tablespaces* distincts. En décidant de partitionner les adhérents (qui sont rattachés à un opérateur) de la même manière, vous devez créér les tables de la manière suivante.

Tableau 12-49 Création de tables partitionnées par référence

Table parent

```
CREATE TABLE Operateurs_partition_ref
 (opeid       CHAR(4) NOT NULL,
  nomope      VARCHAR(25) NOT NULL,
  creaope     DATE NOT NULL,
  siegesocial VARCHAR(15) NOT NULL,
  nbclients   NUMBER(7) NOT NULL,
  CONSTRAINT  pk_Ope_partition_ref
              PRIMARY KEY(opeid))
 PARTITION BY RANGE(creaope)
   ( PARTITION P_1999_2003  VALUES LESS THAN
        (TO_DATE('01-01-2004','DD-MM-YYYY'))
         TABLESPACE tbs_part1,
     PARTITION P_2004_2007 VALUES LESS THAN
        (TO_DATE('01-01-2008','DD-MM-YYYY'))
         TABLESPACE tbs_part2,
     PARTITION P_apres_2008 VALUES LESS THAN
         (MAXVALUE) TABLESPACE tbs_part3);
```

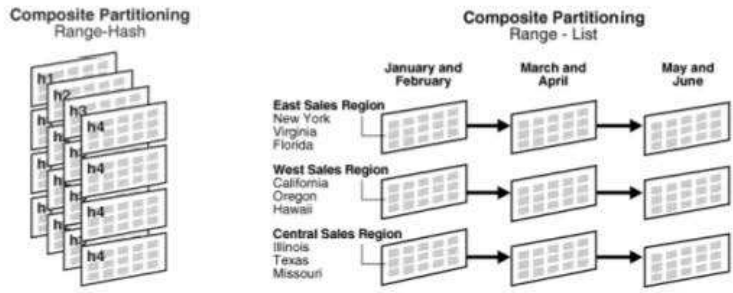
Table enfant

```
CREATE TABLE Adherent_partition_ref
 (adhid      NUMBER(10) NOT NULL,
  nom        VARCHAR(25) NOT NULL,
  prenom     VARCHAR(30) NOT NULL,
  civilite   VARCHAR(12) NOT NULL,
  date_nais  DATE NOT NULL,
  tel        VARCHAR2(20),
  solde      NUMBER(8,2),
  opeid      CHAR(4) NOT NULL,
  CONSTRAINT fk_adh_part_ref_ope
    FOREIGN KEY (opeid)
    REFERENCES Operateurs_partition_ref(opeid)
  ) PARTITION BY REFERENCE(fk_adh_part_ref_ope);
```

Sous-partitions

Si ces mécanismes de partitionnement ne sont pas assez précis, rien ne vous empêche d'utiliser le sous-partitionnement (*composite partitioning*) en divisant chaque partition. Dans la figure suivante, la première table, déjà partitionnée par intervalle de dates, est sous-partitionnée par hachage. La deuxième table, déjà partitionnée par intervalle de dates, est sous-partitionnée par liste de régions.

Figure 12-18 *Sous-partitionnements (© doc. Oracle)*



En combinant un partitionnement sur l'âge des adhérents (*range*) avec un sous-partitionnement sur le sexe (*list*), il vient la table `Adherent_range_list`. La directive `SUBPARTITION` définit chaque sous-partition ; l'option `TEMPLATE` permet d'adopter le même sous-partitionnement pour chaque partition.

Tableau 12-50 Création d'une table partitionnée par hachage

Code SQL	Commentaires

`CREATE TABLE Adherent_range_list`
`(adhid NUMBER(10) NOT NULL, nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL, civilite VARCHAR(12) NOT NULL,`
`date_nais DATE NOT NULL, tel VARCHAR2(20), solde NUMBER(8,2))`
	Colonnes de la table.

`PARTITION BY RANGE` `(date_nais)`	Définition de la clé de partition.
`SUBPARTITION BY LIST (civilite)`	Définition de la clé de sous-partitionnement.
`SUBPARTITION TEMPLATE`	Définition des sous-partitions.
`(SUBPARTITION femmes VALUES ('Melle.','Mme.'),`	
`SUBPARTITION hommes VALUES ('Mr.'),`	
`SUBPARTITION autres VALUES (DEFAULT))`	La première partition se trouvera dans le *tablespace* `tbs_part1`, etc.
`(PARTITION` `retraites`	
`VALUES LESS THAN (TO_DATE('01/01/1945','DD/MM/YYYY'))`	
`TABLESPACE tbs_part1,`	
`PARTITION` `actifs`	
`VALUES LESS THAN (TO_DATE('01/01/1993','DD/MM/YYYY'))`	
`TABLESPACE tbs_part2,`	
`PARTITION` `mineurs`	
`VALUES LESS THAN (MAXVALUE) TABLESPACE tbs_part3);`	

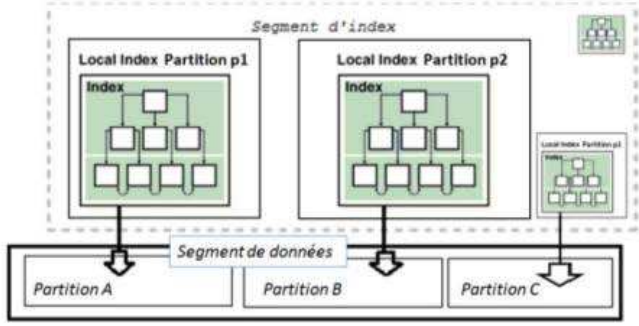
Index partitionné

Un index peut être partitionné de sorte à rendre le meilleur accès à une table malgré les différentes stratégies de partitionnement. Comme pour les tables, le partitionnement d'index facilite la gestion et l'évolution, tout en améliorant la disponibilité et les performances.

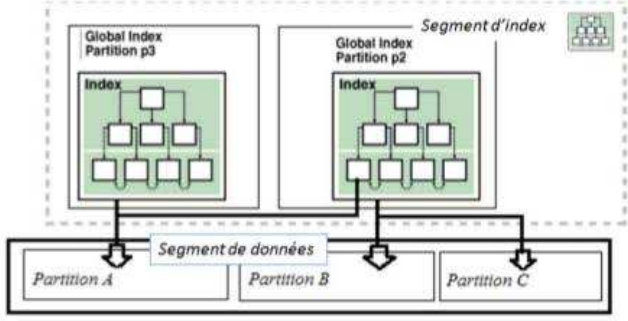
Il existe deux types d'index :

- les index partitionnés locaux (directive `LOCAL` de `CREATE INDEX`) sont partitionnés sur les mêmes colonnes que la table. Chaque partition d'index adresse une seule partition de la table. Les avantages de rapprocher données et index sont nombreux. En particulier, il s'agit d'éviter de reconstruire tout l'index si une des partitions des données venait à être modifiée, supprimée ou rendue indisponible suite à une action de maintenance ;
- les index partitionnés globaux (directive `GLOBAL` de `CREATE INDEX`) sont partitionnés indépendamment de la méthode de partitionnement de la table. L'avantage est d'optimiser un accès à toutes les partitions en les considérant comme une table unique.

Figure 12-19 Index partitionnés locaux et globaux



Index local partitionné



Index global partitionné

Index partitionné local

Un index partitionné local est un index *B-tree* dont les colonnes de l'index sont les colonnes clés de partitionnement de la table. Un index partitionné local peut adopter tout type de partitionnement (intervalle, hachage, liste et composite). De même, l'index et la table disposent du même nombre de partitions. Il est possible de créer un index local *bitmap* sur des tables partitionnées.

Tout index partitionné local concerné est automatiquement reconstruit suite aux modifications de partitions de la table. L'ajout d'une partition à un index local ne peut s'effectuer que suite à l'ajout d'une partition à la table associée. De même, la suppression d'une partition d'un index local s'opère seulement si la partition associée de la table est supprimée.

Index partitionné global

Un index partitionné global est un index *B-tree* qui est partitionné indépendamment du partitionnement de la table associée. Des feuilles d'un tel index peuvent adresser toute partition de la table. Un index global ne peut être partitionné qu'en intervalle ou par hachage.

Les index partitionnés globaux sont limités à 32 colonnes qui doivent correspondre à un préfixe des colonnes de l'index (*left prefix*). Par exemple, si l'index est défini sur les colonnes (`nom, prenom, tel`), alors les colonnes de partitionnement peuvent être (`nom, prenom, tel`), (`nom, prenom`) ou (`nom, tel`). Les combinaisons (`prenom, tel`), (`tel`) et (`prenom, nom`) seront invalides.

Il est impossible d'utiliser la technique du *bitmap* sur un index partitionné global.

Une colonne clé de partitionnement d'un index partitionné global ne peut pas être du type `ROWID`.

Le tableau suivant présente quelques index qu'il serait intéressant d'appliquer aux tables partitionnées des exemples précédents.

Tableau 12-51 Création d'index partitionnés

Index	Commentaires

`CREATE INDEX Adh_part_date_nais_loc_idx`
`  ON Adherent_partition_range`
`  (TO_CHAR(date_nais,'DD/MM/YYYY'))`
`  LOCAL;`
	L'index local accédera optimalement à chaque partition d'adhérents en fonction de la date de naissance.

`CREATE INDEX Adh_part_nom_loc_idx`
`  ON Adherent_partition_hash(nom)`
`  LOCAL`
`  STORE IN (tbs_part1, tbs_part3);`
	L'index local accédera optimalement à chaque partition d'adhérent en fonction de son nom. Les deux partitions de l'index se trouvent dans deux *tablespaces* distincts.

`CREATE INDEX Adh_part_nom_pre_glob_idx`
`  ON Adherent_partition_range(nom,prenom)`
`  GLOBAL`
`  TABLESPACE tbs_index;`
	L'index global accédera optimalement, indépendamment des partitions d'adhérents (partitionnés sur la date de naissance), en fonction des noms et prénoms. La partition de l'index se trouve dans le *tablespace* dédié aux index.

`CREATE INDEX Adh_part_solde_glob_idx`
`  ON Adherent_partition_range (solde)`
`  GLOBAL PARTITION BY RANGE (solde)`
`  (PARTITION p1 VALUES LESS THAN (10000),`
`  PARTITION p2 VALUES LESS THAN (25000),`
`  PARTITION p3 VALUES LESS THAN (MAXVALUE))`
`  TABLESPACE tbs_index;`
	L'index global accédera optimalement indépendamment des partitions d'adhérents (partitionnés sur la date de naissance) en fonction du montant du solde. Les 3 partitions de l'index se trouvent dans le *tablespace* dédié aux index.

1. Si les colonnes de partitionnement sont incluses dans les colonnes de l'index à mettre en œuvre, optez pour un index local. Sinon, passez à l'étape 2.

2. Si l'index est unique et n'inclut pas les colonnes de partitionnement, optez pour un index global. Sinon, passez à l'étape 3.

3. Si vous donnez priorité à la facilité de gestion, optez pour un index local. Sinon, passez à l'étape 4.

4. Si vous privilégiez les temps de réponse (applications OLTP), optez pour un index global. Dans un contexte OLAP, optez pour un index local.

Opérations sur les partitions et index

L'évolution des caractéristiques des tables et index partitionnés est possible. Ainsi, plusieurs opérations sont disponibles ; concernant les tables, il faudra agir à l'aide de l'instruction `ALTER TABLE` :

- ajout/suppression d'une partition via la clause `ADD/DROP/TRUNCATE PARTITION` ;
- union de partitions via la clause `COALESCE PARTITION` ;
- transformation/fusion de partitions via la clause `EXCHANGE/MERGE PARTITION` ;
- modification de partitions via la clause `MODIFY DEFAULT ATTRIBUTES FOR PARTITION` ;
- déplacement/éclatement de partitions via la clause `MOVE/SPLIT PARTITION` ;
- changement du nom/d'une partition via la clause `RENAME PARTITION`.

Dans la majorité des cas et à moins que vous n'utilisiez conjointement la clause `UPDATE INDEXES`, vous devrez reconstruire vos index partitionnés qui seront notifiés `UNUSABLE`.

Partitionnement des tables IOT

En version 8*i*, les tables organisées en index pouvaient aussi être partitionnées mais seulement en intervalle. La version 9*i* permettait le partitionnement par hachage. Depuis la version 10*g*, tout type de partitionnement est autorisé (*range*, *list* ou *hash*) avec les caractéristiques suivantes :

- les colonnes de partition sont incluses (tout ou partie) dans la clé primaire ;
- l'indexation en *bitmap* est possible et les index secondaires peuvent être locaux ou globaux ;
- la directive `PARTITION` peut inclure l'option `OVERFLOW` pour préciser le segment de débordement au niveau de chaque partition.

La table suivante organisée en index est partionnée selon la date des contrats (avant 2000, entre 2000 et 2010, et après).

Tableau 12-52 Création d'une table IOT partitionnée

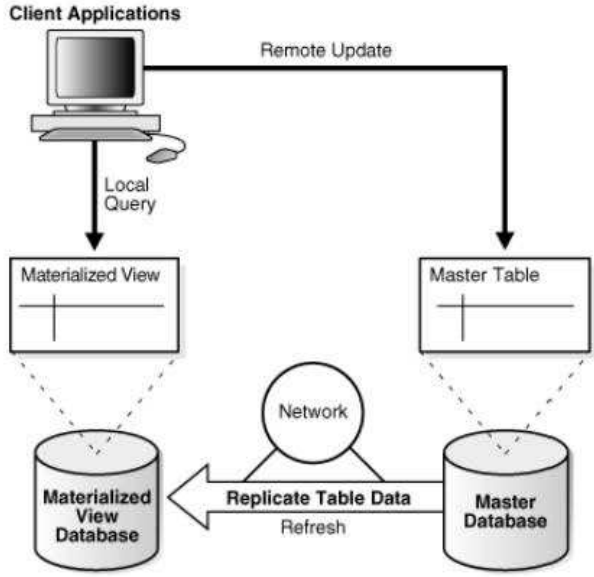
Code SQL	Commentaires
`CREATE TABLE Historique_partition_iot`	Colonnes de la table.
`(opeid CHAR(4) NOT NULL,`	
`adhid NUMBER(10) NOT NULL,`	
`date_contrat DATE NOT NULL,`	
`categorie VARCHAR(15) NOT NULL,`	
`reliquat NUMBER(7) NOT NULL,`	
`CONSTRAINT pk_Historique_partition_iot`	
`PRIMARY KEY (opeid,adhid,date_contrat))`	Définition de la clé primaire.
`ORGANIZATION INDEX`	Le segment d'index se trouvera dans le *tablespace* `tbs_index`.
`TABLESPACE tbs_index`	
`INCLUDING date_contrat`	Les 3 premières colonnes seront dans le segment d'index, le débordement dans le *tablespace* `users`.
`OVERFLOW TABLESPACE users`	
`PARTITION BY RANGE (date_contrat)`	Le partitionnement est défini sur la date du contrat et chaque partition se trouve dans un *tablespace* distinct.
`(PARTITION avant_2000 VALUES LESS THAN`	
`(TO_DATE('01/01/2000','DD/MM/YYYY'))`	
`TABLESPACE tbs_part1,`	
`PARTITION de_2000_a_2010 VALUES LESS THAN`	
`(TO_DATE('01/01/2011','DD/MM/YYYY'))`	
`TABLESPACE tbs_part2,`	
`PARTITION apres VALUES LESS THAN (MAXVALUE)`	
`TABLESPACE tbs_part3);`	

Vues matérialisées

Les vues (dématérialisées) étudiées au chapitre 5 permettent de simplifier l'écriture de certaines requêtes particulièrement complexes mais ne garantissent rien en regard des performances. Dans le pire des cas, une vue peut être consommatrice de ressources si d'autres vues sont impliquées en cascade dans la requête.

Les vues matérialisées (*materialized views*, anciennement *snapshots*) sont formées à partir de requêtes dont le résultat est stocké (comme les lignes d'une table). Une requête composant une vue matérialisée peut concerner des tables, vues et vues matérialisées. Dans un contexte de réplication, l'utilisation première de ces vues, une vue matérialisée s'appelle *master table*. Dans un contexte de *datawarehouse*, une vue matérialisée est nommée *detail table*.

Figure 12-20 *Vues matérialisées (© doc. Oracle)*

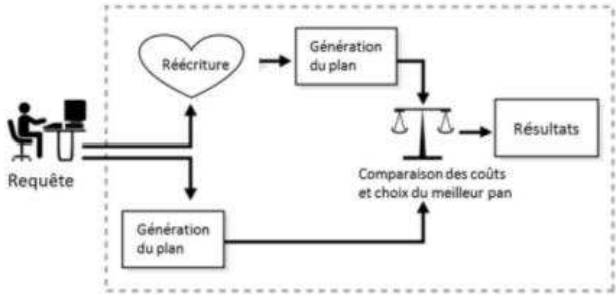


Sans aborder les avantages de ces vues dans une architecture répartie ou d'entrepôts de données, à propos des performances, les vues matérialisées répondent parfaitement à l'amélioration des jointures du fait du stockage de lignes précalculées et de la possibilité de réécriture de requêtes (*query rewrite*). De plus, le partitionnement et l'indexation sont possibles.

Réécriture de requêtes

La réécriture de requêtes est une technique d'optimisation qui transforme une requête complexe émise sur une table volumineuse en une requête sémantiquement équivalente interrogeant la vue matérialisée. Dès qu'il est plus intéressant d'utiliser la vue matérialisée parce qu'elle contient des résultats déjà calculés (agrégats et jointures), toute requête est réécrite (d'une manière transparente pour l'utilisateur) et utilise la vue à la place de la table. Aucun code n'est à ajouter dans l'instruction SQL qui ne référence que la (ou les) tables interrogées.

Figure 12-21 *Réécriture de requêtes*



Le rafraîchissement

Du fait du stockage redondant des données (dans les tables et dans la vue matérialisée), des méthodes de synchronisation (*refresh*) sont disponibles. La méthode de rafraîchissement peut être incrémentale (*fast refresh*) ou complète (*complete refresh*).

Le rafraîchissement incrémental évite de reconstruire entièrement la vue matérialisée. Cependant, ce mécanisme doit s'opérer relativement rapidement (à la demande ou périodiquement) pour garantir l'intégrité des données. Chaque table est associée à un journal d'opérations (*materialized view log*) qui recense toutes les modifications effectuées sur la table.

Le rafraîchissement complet se produit à la création de la vue matérialisée (définie avec `BUILD IMMEDIATE`). Bien que ce procédé puisse être coûteux si les volumes de données manipulés sont importants, les requêtes interrogeant ces tables seront bien plus performantes.

Exemples

Dans un contexte de réplication, les vues matérialisées permettent de maintenir sur une base locale des copies de données distantes. Ces copies peuvent être modifiables sous réserve d'utiliser l'option *Advanced Replication*. En général, ces vues sont basées sur la clé primaire des tables (ou les *rowid*). Dans un contexte d'entrepôts de données, les vues matérialisées composent généralement des regroupements (agrégations) et des jointures.

Utilisons une vue matérialisée pour préparer les extractions d'adeptes de l'escrime et du tennis de table et comparons quelques extractions avec une solution classique.

Tableau 12-53 Création d'une vue matérialisée

Code SQL	Commentaires

`CREATE MATERIALIZED VIEW adh_escrime_pingpong`
`TABLESPACE tbs_cluster`
`BUILD IMMEDIATE`
`REFRESH COMPLETE`
`ENABLE QUERY REWRITE`
`AS SELECT a.adhid, s.spid, s.splibelle,`
`a.nom, a.prenom, a.tel, a.date_nais, a.solde`
`FROM Adherentbis a, Sport s, Pratiquebis p`
`WHERE s.splibelle IN ('Escrime','Ping-pong')`
`AND a.adhid = p.adhid`
`AND s.spid = p.spid`
`ORDER BY a.adhid,a.nom;`
	Création de la vue matérialisée située dans le tablespace `tbs_cluster`.
	La construction est immédiate et la vue est éligible à la réécriture de requête (`ENABLE QUERY REWRITE`).

Les requêtes de jointure entre adhérents et l'escrime ou le tennis de table utiliseront la vue à la place des tables d'une manière bien plus efficace. L'opérateur que vous verrez apparaître dans vos plans d'exécution est `MAT_VIEW REWRITE ACCESS FULL`.

Le rafraîchissement automatique nécessite de créer un journal d'opérations par table interrogée. Les exemples suivants décrivent des vues matérialisées qui seront mises à jour automatiquement. La première sera actualisée dès la modification de la table `Sport`. La deuxième le sera tous les lundis à 15 h 00.

Tableau 12-54 Rafraichissement automatique

Code SQL	Commentaires

`CREATE MATERIALIZED VIEW LOG`
`ON Sport WITH PRIMARY KEY, ROWID;`
`CREATE MATERIALIZED VIEW catalogue_sports`
`REFRESH FAST`
`ON COMMIT`
`WITH PRIMARY KEY`
`AS SELECT spid, splibelle FROM Sport;`
	Création du journal des opérations.
	Création de la vue matérialisée avec une contrainte *primary key* (colonne clé primaire de la table).
	Rafraîchissement incrémental (*fast*) après modifications sur la table.

`CREATE MATERIALIZED VIEW LOG`
`ON Adherentbis`
`WITH PRIMARY KEY, ROWID`
`PURGE REPEAT INTERVAL '5' DAY;`
`CREATE MATERIALIZED VIEW Adherent_hommes`
`REFRESH FAST`
`START WITH ROUND(SYSDATE + 1) + 11/24`
`NEXT`
`NEXT_DAY(TRUNC(SYSDATE)/ LUNDI')+15/24`
`WITH PRIMARY KEY`
`AS SELECT a.adhid, a.nom, a.prenom,`
`a.tel, a.date_nais, a.solde`
`FROM Adherentbis a`
`WHERE a.civilite = 'Mr.';`
	Création du journal des opérations qui sera vidé tous les 5 jours.
	
	Création de la vue matérialisée avec une contrainte *primary key* (colonne clé primaire de la table).
	Rafraîchissement incrémental, première actualisation : le lendemain à 11 h puis tous les lundis à 15 h.

Dénormalisation

La dénormalisation suppose que vos tables soient d'abord en forme normale (au moins la troisième forme).

La majorité des problèmes de performances des applications en production survient à la montée en charge au niveau du volume des données. En d'autres termes, manipuler des tables mal conçues s'avère pénalisant seulement quand elles deviennent volumineuses et ne tiennent plus en RAM. Des experts étudient alors le code pour se rendre compte que les tables ne sont pas normalisées et il est souvent un peu trop tard. Partez sur de bonnes bases (c'est le cas de le dire) : normalisez au maximum en amont !

Si vos tables sont normalisées (au minimum en troisième forme normale), que toutes vos colonnes clés étrangères disposent d'un index associé, que ni les *clusters*, le partitionnement ou les vues matérialisées ne peuvent répondre à vos problématiques de temps de réponse, vous pouvez tenter de dénormaliser quelques-unes de vos tables.

Vous pouvez dénormaliser une table en y ajoutant des nouvelles colonnes qui permettront de stocker soit des colonnes calculées (qui éviteront des calculs), soit des données redondantes (mais plus accessibles) ou des clés primaires (de taille plus réduite, comme une séquence) ou étrangères (qui diminueront les jointures). Dans bien des cas, vous devrez programmer des déclencheurs afin de maintenir l'intégrité dans le temps de votre nouvelle base.

La dénormalisation sera profitable si votre application effectue de nombreuses lectures et peu de mises à jour. Si beaucoup de mises à jour sont réalisées, la dénormalisation dégradera sensiblement certaines performances. En revanche, si votre application effectue peu de lectures, dénormaliser ne sert à rien (a fortiori lorsqu'il existe beaucoup de mises à jour).

Colonnes calculées

Dans l'exemple suivant, l'ajout des colonnes `totaljp` (nombre de jours à facturer) et `totalj` (nombre de jours de formation dans l'année) évitera tout calcul impliquant des jointures avec la table `Inscriptions`.

Figure 12-22 *Dénormalisation par colonnes calculées*

Cours

ncours	titre	nbj	TP	totaljp
UMD	UML 2.0, conception de bases de données	2	1	4
PAT	Conception et design patterns	2	0.5	2
UJE	Modéliser une application JEE avec UML 2.0	3	1.5	3
UTR	UML 2.0, pour le temps réel	5	2.5	5

Inscriptions

nco	nci	mois	lieu
UMD	2	11	Pa
UMD	3	11	Pa
UJE	1	12	Na
UTR	3	02	Pa
PAT	3	04	Ly

Client

numCi	identite	societe	totalj
1	Laurent Robert	TOTAL	3
2	Estelle Allemand	Airbus	2
3	Michel Belli	Airbus	9

Le revers de la médaille consiste en la nécessité de programmer plusieurs déclencheurs sur la table `Inscriptions` qui mettront à jour ces deux colonnes à chaque inscription et à toute désinscription. Une ou plusieurs vues matérialisées peuvent être également utilisées.

Duplication de colonnes

Dans l'exemple suivant, l'ajout des colonnes `identite2` (nom du client) et `ste2` (société du client) évitera toute jointure avec la table `Client` pour lister les détails des inscriptions. Ici on régresse de la troisième à la première forme normale.

Figure 12-23 *Dénormalisation par ajout de colonnes*

Cours

ncours	titre	nbj	TP
UMD	UML 2.0, conception de bases de données	2	1
PAT	Conception et design patterns	2	0.5
UJE	Modéliser une application JEE avec UML 2.0	3	1.5
UTR	UML 2.0, pour le temps réel	5	2.5

Inscriptions

nco	nci	mois	lieu	identite2	ste2
UMD	2	11	Pa	Estelle Allemand	Airbus
UMD	3	11	Pa	Michel Belli	Airbus
UJE	1	12	Na	Lauren Robert	TOTAL
UTR	3	2	Pa	Michel Belli	Airbus
PAT	3	4	Ly	Michel Belli	Airbus

Client

numCi	identite	societe
1	Lauren Robert	TOTAL
2	Estelle Allemand	Airbus
3	Michel Belli	Airbus

Les inconvénients de ce mécanisme sont d'une part l'espace utilisé et d'autre part le risque d'incohérences si le nom d'un client (ou le nom de sa société) vient à changer. La solution consiste à utiliser un déclencheur sur la table `Client` ou une vue matérialisée (contenant entre autres ces colonnes additionnelles).

Ajout de clés étrangères

Dans l'exemple suivant, l'ajout de la colonne `nseg` (numéro de segment) évitera toute jointure avec la table `Salles` pour relier des postes à leur segment. Ici on régresse de la troisième à la deuxième forme normale.

Figure 12-24 *Dénormalisation par ajout de clé*

Postes

idP	typOS	RAM	IP	*ns*	*nseg*
P1-S1	XP	2	01	S1	192.167.10
P2-S1	Vista	1	02	S1	192.167.10
P3-S1	XP	2.5	03	S1	192.167.10
P1-S2	Linux	1.5	01	S2	192.167.20

Salles

nsalle	etage	nbp	*nseg*
S1	12	12	192.167.10
S2	12	10	192.167.20
S3	16	14	192.167.25
S4	10	8	192.167.10

Segments

nseg	debit	type
192.167.10	10	paire blindée
192.167.20	1000	fibre V2
192.167.25	100	fibre

On retrouve les mêmes inconvénients que précédemment (espace perdu et redondances). Pour y remédier, il faudra programmer un déclencheur ou passer par une vue matérialisée.

Exemple de stratégie

Dénormaliser constituera une action forte sur la base car la structure des tables sera altérée, des redondances apparaîtront et des déclencheurs devront être mis en œuvre. En conséquence, cette technique doit être la dernière solution à un problème de temps de réponse en exploitation et en aucun cas justifiée par l'intérêt du programmeur.

Normalisez au maximum avant de mettre en exploitation puis auditer régulièrement quelques semaines, mois ou années suivants la montée en charge.

Appliquez les requêtes les plus pénalisantes à une base dénormalisée mais équivalente en termes de données. Comparez les temps de réponse entre les deux bases. Si ce dernier est inférieur de plus de 30 %, il devient vraiment intéressant de dénormaliser.

Derniers conseils

Pour en terminer avec tous ces mécanismes d'optimisation, vous trouverez quelques conseils d'ordre général à propos de vos requêtes et de l'utilisation d'un SGBD.

Les bases de données sont comme bien des situations dans la vie. Les avantages s'accompagnent inéluctablement d'inconvénients, et chaque problème a sa solution. En d'autres termes, une solution d'optimisation peut vous faire gagner du temps et de l'argent à un endroit et vous pénaliser ailleurs. Vous devrez toujours peser le pour et le contre de toute optimisation et décider finalement en connaissance de causes. Bonne chance.

Requêtes inefficaces

Le tableau suivant présente quelques erreurs classiques et les moyens d'y remédier.

Tableau 12-55 Quelques requêtes inefficaces

Requête inefficace	Commentaires

`SELECT COUNT(*)`
`  FROM produits p`
`  WHERE prod_list_price <`
`  1.15 * (SELECT avg(unit_cost)`
`  FROM couts c`
`  WHERE c.prod_id = p.prod_id)`
	On cherche le nombre de produits pour lesquels l'écart entre le prix catalogue et le coût moyen est inférieur à 15 %. Le calcul de la moyenne s'exécute pour chaque produit. Il est préférable d'écrire :
	`SELECT COUNT(*) FROM produits p,`
	`  (SELECT prod_id, AVG(unit_cost) ac`
	`  FROM couts GROUP BY prod_id) c`
	`  WHERE p.prod_id = c.prod_id`
	`  AND p.prod_list_price < 1.15 * c.ac`

`SELECT *`
`  FROM job_history jh, employes e`
`  WHERE SUBSTR(TO_CHAR(e.employe_id),2)=`
`  SUBSTR(TO_CHAR(jh.employe_id),2)`
	N'utilisez plus jamais * !
	La requête applique des expressions aux colonnes de jointure. Afin de bénéficier d'index, il faudra définir deux index basés sur l'expression en question.

`SELECT cde_date`
`  FROM commandes`
`  WHERE cde_id_char = 1205`
	Le prédicat effectue une conversion implicite de type. Alors que la colonne est de type caractère, la constante, est numérique. Si l'index est déclaré, comparez la colonne à `TO_CHAR(1205)` ou à `'1205'`.

`SELECT emp_id, emp_name`
`  FROM employes`
`  WHERE TO_CHAR(salaire) = :sal`
	La conversion de type `(TO_CHAR)` est appliquée à la valeur de colonne plutôt qu'à la constante et est appelée pour chaque ligne de la table. Modifiez le prédicat : `salaire=TO_NUMBER(:sal)` pour bénéficier d'un index et éviter de multiples calculs.

`SELECT … FROM avions_AF`
`UNION`
`SELECT … FROM avions_BRITAIR`
	Contrairement à `UNION ALL`, l'opérateur `UNION` évite les doublons mais nécessite de réaliser un tri unique. Si vous savez qu'a priori, il n'existe pas de lignes communes aux deux extractions, préférez `UNION ALL`.

Pensez à la nécessité de modifier ultérieurement vos requêtes. Toute modification devrait minimiser le besoin de redéployer des modules. L'utilisation de procédures cataloguées peut répondre à ce besoin de maintenance.

Les 10 commandements de F. Brouard

Paru en 2008 sur le blog de F. Brouard (*http://blog.developpez.com/sqlpro*), l'article *Les 10 meilleures pratiques pour développer avec un SGBDR* cite un certain nombre de postulats qui sont ici résumés.

1. Une base de données relationnelle doit gérer des relations et non des fichiers.

   Assurez-vous que vos tables ne contiennent pas un nombre trop important de colonnes. Respectez la normalisation introduite par le modèle relationnel. Contrairement à une idée reçue, dans votre base, plus le nombre de tables ayant peu de colonnes est élevé, meilleures sont les performances de vos requêtes.

2. Une clé primaire artificielle est préférable à une clé métier (sémantique).

   En choisissant le numéro d'immatriculation d'une voiture, qui vous dit qu'à la création de l'enregistrement l'information sera connue ou que cette valeur n'évoluera pas dans le temps (entraînant des effets de bord très coûteux) ? La clé métier est en général plus volumineuse qu'une simple colonne `NUMBER`. L'idéal est de définir une clé primaire artificielle et de disposer aussi d'une clé métier (contrainte `UNIQUE`).

3. Ne codez jamais (à part dans vos tests ou démonstrations), une requête du type `SELECT * ...`, ceci pour ces 3 principales raisons :

   – Moins de données circulent sur le réseau, plus les temps de réponse sont courts. Il est donc préférable d'indiquer dans la liste des colonnes uniquement celles qui sont nécessaires.

   – Allégez la charge du transformateur de requêtes en lui évitant de rechercher les informations dans les tables système pour déduire la liste de toutes les colonnes et les privilèges associés.

   – Allez-vous interdire implicitement que vos tables évoluent en terme de structure ? Ajouter ou supprimer une colonne risque de rendre le code inopérant à tout endroit où cette instruction se trouvera.

4. Évitez si possible d'utiliser des curseurs dans vos transactions.

   Les curseurs imposent une programmation itérative (où les données sont traitées ligne par ligne comme avec un simple fichier) et non ensembliste. Un SGBD est optimisé pour traiter de manière ensembliste les données avec SQL. Depuis la version SQL 1999, la récursivité est supportée et SQL devient un langage complet (au sens de la machine de Turing) où tout traitement peut être théoriquement programmé à l'aide de requêtes.

5. Écourtez la durée de vos transactions et programmez côté serveur (procédures cataloguées).

   Une transaction nécessite d'accéder souvent exclusivement aux données et des verrous sont automatiquement mis en œuvre. Ces derniers induisent des temps d'attente pour les utilisateurs concurrents. Si votre code n'est pas optimisé ou s'il s'exécute du côté du client, la contention devient inévitable.

6. Utilisez le SQL dynamique pour écrire des requêtes simples.

   Le fait d'écrire une instruction avec du SQL dynamique évite au transformateur de requêtes un certain nombre de tâches et permet la réutilisation du plan d'exécution.

7. Paramétrez la bonne collation.

   Une collation sert à gérer la manière dont les chaînes de caractères, constituant les données de la base, vont se comporter face aux opérateurs de comparaison et à l'ordonnancement des données (tri). La gestion des majuscules/minuscules, accents, ligatures (comme dans cœur), etc., doit être prévue. Consultez la section *À propos des accents et jeux de caractères* de l'introduction.

8. N'utilisez jamais une requête du type `SELECT MAX(...)+1` ... pour générer une clé primaire.

   Ce mode de calcul est à proscrire car il peut conduire, un jour ou l'autre, au mieux à un télescopage de clé, au pire à un blocage total. En effet tant que la nouvelle ligne pourvue de cette nouvelle clé n'est pas encore insérée dans la base, toute autre transaction peut effectuer le calcul générant la même valeur. Afin de rendre cohérent ce mécanisme, il faudrait programmer une transaction intégrant le calcul de *max+1* et l'ordre d'insertion avec la nouvelle clé (ce qui n'est pas si simple qu'il y paraît). Préférez le mécanisme d'auto-incrément de votre SGBD qui pour Oracle est la séquence.

9. Utilisez avec parcimonie les tables temporaires.

   Chaque objet temporaire créé au sein d'un SGBDR déclenche une écriture coûteuse au journal de transactions et dans le dictionnaire des données. Si de nombreuses transactions sont effectuées en parallèle et qu'elles génèrent de nombreux objets temporaires, des points de contention peuvent apparaître. Préférez l'utilisation de requêtes contenant des fonctions table ou des CTE.

10. Utilisez des index mais à bon escient.

    Jusqu'à 5 index, une table d'une base OLTP est dans une moyenne habituelle. Au-delà, il faut analyser les raisons qui ont motivé une telle indexation (qui peuvent être justifiables). Indexez vos clés étrangères et ne vous trompez pas dans le mode d'indexage (*B-tree*, *bitmap* ou IOT). Si une table IOT convient à une séquence, elle peut mener à des performances désastreuses pour une clé sémantique alphanumérique de taille variable.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X