Christian Soutou

3e édition

SQL pour Oracle

Applications avec Java, PHP et XML

Avec 50 exercices corrigés

EYROLLES

SQL pour Oracle

3e édition

CHEZ LE MÊME ÉDITEUR

Du même auteur

C. Soutou. – **De UML à SQL.**
N°11098, 2002, 500 pages.

C. Soutou. – **Apprendre SQL avec MySQL.**
N°11915, 2006, 398 pages.

Autour d'oracle et de sql

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

R. Bizoï – **Oracle 10*g* - Administration.**
N°11747, 2005, 744 pages.

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

R. Bizoï – **PL/SQL pour Oracle 10*g*.**
N°12056, 2006, 328 pages.

C. Pierre de Geyer et G. Ponçon – **Mémento PHP et SQL.**
N°11785, 2006, 14 pages.

G. Briard – **Oracle9*i* sous Windows.**
N°11220, 2003, 1040 pages.

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

B. Vidal – **Applications mobiles avec Oracle.**
N°9251, 2001, 686 pages.

R. Bizoï. – **Oracle9*i* : SQL et PL/SQL.**
N°11351, 2003, 468 pages.

M. Israel. – **SQL Server 2000.**
N°11027, 2001, 1078 pages.

M. Israel. – **SQL Server 2000.**
N°11027, 2001, 1078 pages.

Christian Soutou
Avec la participation d'Olivier Teste

SQL pour Oracle

3e édition

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

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2004, 2005, 2008, ISBN : 978-2-212-12299-2

Mise en page : TyPAO
Dépôt légal : février 2008
N° d'éditeur : 7760
Imprimé en France

Table des matières

Remerciements XVII

Avant-propos XIX

- **Guide de lecture** XIX
  - Première partie : SQL de base XX
  - Deuxième partie : PL/SQL XX
  - Troisième partie : SQL avancé XX
  - Annexe : bibliographie et webographie XX
- **Conventions d'écriture** XXI
- **Contact avec l'auteur et site Web** XXI

Introduction 1

- **SQL, une norme, un succès** 1
- **Modèle de données** 2
  - Tables et données 2
  - Les clés 3
- **Oracle** 3
  - Un peu d'histoire 4
  - Offre du moment 5
  - Notion de schéma 7
  - Accès à Oracle depuis Windows 8
  - Détail d'un numéro de version 9
- **Installation d'Oracle** 9
  - Mise en œuvre d'Oracle9*i* 9
  - Désinstallation de la 9*i* 12
  - Mise en œuvre d'Oracle 10*g* 12
  - Désinstallation de la 10*g* 15
  - Mise en œuvre d'Oracle 10*g* Express Edition 16
  - Mise en œuvre d'Oracle 11*g* 16
  - Désinstallation de la 11*g* 18
- **Les interfaces SQL*Plus** 19
  - Généralités 19
  - Premiers pas 24
  - Variables d'environnement 26
  - À propos des accents 27

© Éditions Eyrolles

Partie I SQL de base 29

1 Définition des données 31
Tables relationnelles 31
Création d'une table (CREATE TABLE) 31
Casse et commentaires. 32
Premier exemple 33
Contraintes 33
Conventions recommandées. 35
Types des colonnes 36
Structure d'une table (DESC) 39
Restrictions 39
Commentaires stockés (COMMENT) 40
Index 40
Classification 41
Index B-tree. 41
Index bitmap 42
Index basés sur des fonctions. 42
Création d'un index (CREATE INDEX) 43
Bilan 44
Tables organisées en index 44
Destruction d'un schéma 46

2 Manipulation des données 51
Insertions d'enregistrements (INSERT). 51
Syntaxe 51
Renseigner toutes les colonnes 52
Renseigner certaines colonnes. 52
Ne pas respecter des contraintes 53
Dates/heures. 53
Caractères Unicode. 56
Données LOB 56
Séquences 57
Création d'une séquence (CREATE SEQUENCE). 58
Manipulation d'une séquence 60
Modification d'une séquence (ALTER SEQUENCE). 61
Visualisation d'une séquence 62
Suppression d'une séquence (DROP SEQUENCE). 63
Modifications de colonnes 63
Syntaxe (UPDATE) 63
Modification d'une colonne 64
Modification de plusieurs colonnes 64
Ne pas respecter des contraintes 64
Dates et intervalles 65

© Éditions Eyrolles

Suppressions d'enregistrements . . . 70
Instruction DELETE . . . 70
Instruction TRUNCATE . . . 70
Intégrité référentielle . . . 71
Cohérences . . . 71
Contraintes côté « père » . . . 72
Contraintes côté « fils » . . . 72
Clés composites et nulles . . . 73
Cohérence du fils vers le père . . . 74
Cohérence du père vers le fils . . . 74
En résumé . . . 75
Flottants . . . 76
Valeurs spéciales . . . 77
Fonctions pour les flottants . . . 77

3 Évolution d'un schéma . . . 85
Renommer une table (RENAME) . . . 85
Modifications structurelles (ALTER TABLE) . . . 86
Ajout de colonnes . . . 86
Renommer des colonnes . . . 86
Modifier le type des colonnes . . . 87
Supprimer des colonnes . . . 88
Colonnes UNUSED . . . 88
Modifications comportementales . . . 89
Ajout de contraintes . . . 89
Suppression de contraintes . . . 90
Désactivation de contraintes . . . 92
Réactivation de contraintes . . . 94
Contraintes différées . . . 97
Directives DEFERRABLE et INITIALLY . . . 97
Instructions SET CONSTRAINT . . . 99
Instruction ALTER SESSION SET CONSTRAINTS . . . 99
Directives VALIDATE et NOVALIDATE . . . 99
Directive MODIFY CONSTRAINT . . . 101
Colonne virtuelle . . . 102
Table en lecture seule . . . 104

4 Interrogation des données . . . 109
Généralités . . . 109
Syntaxe (SELECT) . . . 110
Pseudo-table DUAL . . . 110
Projection (éléments du SELECT) . . . 111
Extraction de toutes les colonnes . . . 112
Extraction de certaines colonnes . . . 112

© Éditions Eyrolles

Alias. 113
Duplicatas 113
Expressions et valeurs nulles 114
Ordonnancement. 114
Concaténation 115
Pseudo-colonne ROWID 115
Pseudo-colonne ROWNUM 116
Insertion multiligne 116
Restriction (WHERE) 117
Opérateurs de comparaison 118
Opérateurs logiques 118
Opérateurs intégrés. 119
Fonctions 120
Caractères. 121
Numériques 124
Dates 125
Conversions 126
Autres fonctions. 128
Regroupements 128
Fonctions de groupe 129
Étude du GROUP BY et HAVING 130
Opérateurs ensemblistes 133
Restrictions 133
Exemple 134
Opérateur INTERSECT. 134
Opérateurs UNION et UNION ALL 135
Opérateur MINUS 135
Ordonner les résultats 136
Produit cartésien 137
Bilan 139
Jointures. 139
Classification 140
Jointure relationnelle 140
Jointures SQL2 140
Types de jointures 141
Équijointure 141
Autojointure 143
Inéquijointure. 144
Jointures externes 145
Jointures procédurales 150
Jointures mixtes 154
Sous-interrogations synchronisées 155
Autres directives SQL2 157

 © Éditions Eyrolles

Division . . . 159
Définition . . . 160
Classification . . . 160
Division inexacte en SQL . . . 161
Division exacte en SQL . . . 162
Requêtes hiérarchiques . . . 162
Point de départ du parcours (START WITH) . . . 163
Parcours de l'arbre (CONNECT BY PRIOR) . . . 163
Indentation . . . 164
Élagage de l'arbre (WHERE et PRIOR) . . . 165
Jointures . . . 167
Ordonnancement . . . 167
Nouveautés 10*g* . . . 168
Mises à jour conditionnées (fusions) . . . 172
Syntaxe (MERGE) . . . 172
Exemple . . . 173
Nouveautés 10*g* . . . 173
Exemple . . . 174
Expressions régulières . . . 175
Quelques exemples . . . 177
Fonction REGEXP_LIKE . . . 177
Fonction REGEXP_REPLACE . . . 180
Fonction REGEXP_INSTR . . . 181
Fonction REGEXP_SUBSTR . . . 183
Nouveautés 11*g* . . . 184
Extractions diverses . . . 185
Directive WITH . . . 185
Fonction WIDTH_BUCKET . . . 187

5 Contrôle des données . . . 193
Gestion des utilisateurs . . . 194
Classification . . . 194
Création d'un utilisateur (CREATE USER) . . . 194
Modification d'un utilisateur (ALTER USER) . . . 196
Suppression d'un utilisateur (DROP USER) . . . 197
Profils . . . 198
Console Enterprise Manager . . . 201
Privilèges . . . 206
Privilèges système . . . 206
Privilèges objets . . . 209
Privilèges prédéfinis . . . 213
Rôles . . . 214
Création d'un rôle (CREATE ROLE) . . . 215
Rôles prédéfinis . . . 216

© Éditions Eyrolles

Console Enterprise Manager. 217
Révocation d'un rôle 218
Activation d'un rôle (SET ROLE). 219
Modification d'un rôle (ALTER ROLE). 220
Suppression d'un rôle (DROP ROLE) 221
Vues 221
Création d'une vue (CREATE VIEW) 222
Classification 224
Vues monotables. 224
Vues complexes 229
Autres utilisations de vues. 232
Transmission de droits 236
Modification d'une vue (ALTER VIEW) 236
Suppression d'une vue (DROP VIEW) 236
Synonymes. 237
Création d'un synonyme (CREATE SYNONYM) 237
Transmission de droits 239
Suppression d'un synonyme (DROP SYNONYM) 239
Dictionnaire des données 239
Constitution 240
Classification des vues 240
Démarche à suivre 241
Principales vues 243
Objets d'un schéma. 245
Structure d'une table 245
Recherche des contraintes d'une table. 246
Composition des contraintes d'une table 246
Détails des contraintes référentielles 246
Recherche du code source d'un sous-programme 247
Recherche des utilisateurs d'une base de données 248
Rôles reçus 248

Partie II PL/SQL. 253

6 Bases du PL/SQL. 255
Généralités 255
Environnement client-serveur 255
Avantages 256
Structure d'un programme. 256
Portée des objets 257
Jeu de caractères 258
Identificateurs 258
Commentaires 259

© Éditions Eyrolles

Variables . . . 259
Variables scalaires . . . 260
Affectations . . . 260
Restrictions . . . 261
Variables %TYPE . . . 261
Variables %ROWTYPE . . . 262
Variables RECORD . . . 263
Variables tableaux (type TABLE) . . . 264
Résolution de noms . . . 266
Opérateurs . . . 266
Variables de substitution . . . 267
Variables de session . . . 268
Conventions recommandées . . . 268
Types de données PL/SQL . . . 269
Types prédéfinis . . . 269
Sous-types . . . 269
Conversions de types . . . 271
Nouveautés 11*g* . . . 271
Structures de contrôles . . . 272
Structures conditionnelles . . . 272
Structures répétitives . . . 275
Nouveauté 11*g* . . . 279
Interactions avec la base . . . 280
Extraire des données . . . 280
Manipuler des données . . . 282
Curseurs implicites . . . 284
Paquetage DBMS_OUTPUT . . . 285
Transactions . . . 288
Caractéristiques . . . 288
Début et fin d'une transaction . . . 289
Contrôle des transactions . . . 290
Transactions imbriquées . . . 291

7 Programmation avancée . . . 295
Sous-programmes . . . 295
Généralités . . . 295
Procédures cataloguées . . . 296
Fonctions cataloguées . . . 297
Codage d'un sous-programme PL/SQL . . . 298
Exemples . . . 298
Compilation . . . 301
Appels . . . 301
À propos des paramètres . . . 303
Récursivité . . . 304

Sous-programmes imbriqués . . . 304
Recompilation d'un sous-programme . . . 306
Destruction d'un sous-programme . . . 306
Paquetages (packages) . . . 306
Généralités . . . 306
Spécification . . . 307
Compilation . . . 308
Implémentation . . . 308
Appel . . . 309
Surcharge . . . 309
Recompilation . . . 309
Destruction d'un paquetage . . . 309
Curseurs . . . 310
Généralités . . . 310
Instructions . . . 310
Parcours d'un curseur . . . 311
Utilisation de structures (%ROWTYPE) . . . 312
Boucle FOR (gestion semi-automatique) . . . 313
Utilisation de tableaux (type TABLE). . . . 314
Paramètres d'un curseur . . . 315
Accès concurrents (FOR UPDATE et CURRENT OF) . . . 315
Variables curseurs (REF CURSOR) . . . 317
Exceptions . . . 319
Généralités . . . 319
Exception interne prédéfinie . . . 321
Exception utilisateur . . . 325
Utilisation du curseur implicite . . . 327
Exception interne non prédéfinie . . . 328
Propagation d'une exception . . . 329
Procédure RAISE_APPLICATION ERROR . . . 331
Déclencheurs . . . 332
À quoi sert un déclencheur ? . . . 332
Généralités . . . 333
Mécanisme général . . . 333
Syntaxe . . . 334
Déclencheurs LMD . . . 335
Transactions autonomes . . . 347
Déclencheurs LDD . . . 348
Déclencheurs d'instances . . . 348
Appels de sous-programmes . . . 349
Gestion des déclencheurs . . . 350
Ordre d'exécution . . . 351
Tables mutantes . . . 351
Nouveautés 11*g* . . . 352

 © Éditions Eyrolles

SQL dynamique 356
Classification 357
Utilisation de EXECUTE IMMEDIATE 357
Utilisation d'une variable curseur 359

Partie III SQL avancé 363

8 Le précompilateur Pro*C/C++ 365

Généralités 365
Ordres SQL intégrés 365
Variables 366
Variable indicatrice 367
Cas du VARCHAR 368
Zone de communication (SQLCA) 368
Connexion à une base 369
Gestion des exceptions 369
Transactions 370
Extraction d'un enregistrement 370
Mises à jour 372
Utilisation de curseurs 372
Variables scalaires 372
Variables tableaux 373
Utilisation de Microsoft Visual C++ 375

9 L'interface JDBC 377

Généralités 377
Classification des pilotes (drivers) 377
Les paquetages 378
Structure d'un programme 380
Variables d'environnement 380
Test de votre configuration 381
Connexion à une base 382
Base Access 382
Base Oracle 383
Déconnexion 385
Interface Connection 385
États d'une connexion 386
Interfaces disponibles 386
Méthodes génériques pour les paramètres 387
États simples (interface Statement) 387
Méthodes à utiliser 388
Correspondances de types 389

© Éditions Eyrolles

Interactions avec la base . . . 390
Suppression de données . . . 390
Ajout d'enregistrements . . . 391
Modification d'enregistrements . . . 391
Extraction de données . . . 391
Curseurs statiques . . . 392
Curseurs navigables . . . 393
Curseurs modifiables . . . 397
Suppressions . . . 399
Modifications . . . 400
Insertions . . . 400
Restrictions . . . 401
Interface ResultSetMetaData . . . 402
Interface DatabaseMetaData . . . 403
Instructions paramétrées (PreparedStatement) . . . 404
Extraction de données (executeQuery) . . . 405
Mises à jour (executeUpdate) . . . 406
Instruction LDD (execute) . . . 406
Appels de sous-programmes . . . 407
Appel d'une fonction . . . 408
Appel d'une procédure . . . 408
Transactions . . . 409
Points de validation . . . 410
Traitement des exceptions . . . 411
Affichage des erreurs . . . 412
Traitement des erreurs . . . 412

10 L'approche SQLJ . . . 415
Généralités . . . 415
Blocs SQLJ . . . 415
Précompilation . . . 416
Configurations . . . 416
Affectations (SET) . . . 418
Intégration de SQL . . . 418
Instructions du LDD . . . 418
Instructions du LMD . . . 419
Requêtes . . . 419
À propos des itérateurs . . . 423
Transactions . . . 426
Intégration de blocs PL/SQL . . . 426
Points de validation . . . 427
Appels de sous-programmes . . . 428
Résultats scalaires . . . 428
Résultats complexes . . . 430

© Éditions Eyrolles

Traitement des exceptions 431
Définition des données 431
Manipulation des données 432
Interrogation des données 432
Contextes de connexion 433
SQL dynamique 435
Expression 435
Restrictions 436

11 Procédures stockées et externes 439
Procédures stockées Java 439
Stockage d’une procédure 440
Interactions avec la base 445
Déclencheurs 449
Procédures externes Java 450
Compilation de la classe 451
Création d’une librairie 451
Publication d’une procédure externe 451
Appel d’une procédure externe 452

12 Oracle et le Web 453
Configuration minimale d’Apache 453
PL/SQL Web Toolkit 455
Détail d’une URL 455
Paquetages HTP et HTF 456
Pose d’hyperliens 460
Formulaires 462
Tables 463
Listes 463
PL/SQL Server Pages 464
Généralités 464
Balises 465
Chargement d’un programme PSP 466
Appel 466
Interaction avec la base 466
Intégration de PHP 468
Configuration adoptée 468
API de PHP pour Oracle 471
Interactions avec la base 473

13 Oracle XML DB 487
Généralités 487
Comment disposer de XML DB ? 487

© Éditions Eyrolles

Le type de données XMLType 488
Modes de stockage 489
Stockages XMLType 490
Création d'une table 491
Répertoire de travail 493
Grammaire XML Schema 493
Annotation de la grammaire 494
Enregistrement de la grammaire 496
Stockage structuré (object-relational) 498
Stockage non structuré (CLOB) 511
Stockage non structuré (binary XML) 512
Autres fonctionnalités 516
Génération de contenus 516
Vues XMLType 517
Génération de grammaires annotées 520
Dictionnaire des données 522
XML DB Repository 524
Interfaces. 524
Configuration 524
Paquetage XML_XDB 527
Accès par SQL 527

Annexe : Bibliographie et webographie 535

Index 537

© Éditions Eyrolles

Remerciements

Merci à Agnès Labat, Thierry Millan, Gratien Viel et Olivier Teste qui ont contribué à l'élaboration de la première édition de cet ouvrage en 2005.

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.

La documentation d'Oracle 11*g* représente plus de 1 Go de fichiers HTML et 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 et Master Pro).

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 intéressés par certaines nouvelles directives du langage.

Cette troisième édition ajoute à la précédente les nouvelles fonctionnalités de la version 11*g* en ce qui concerne SQL, PL/SQL ainsi que la présentation de XML DB, l'outil d'Oracle qui gère (stockage, mise à jour et extraction) du contenu XML.

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 décrit la technologie SQLJ (norme ISO) qui permet d’intégrer du code SQL dans un programme Java. Le chapitre 11 traite des procédures stockées et des procédures externes. Le chapitre 12 est consacré aux techniques qu’Oracle propose pour interfacer une base de données sur le Web (*PL/SQL Web Toolkit* et PSP *PL/SQL Server Pages*) ainsi que l’API PHP. Enfin, le chapitre 13 présente les fonctionnalités de XML DB et l’environnement *XML DB Repository.*

Annexe : bibliographie et webographie

Vous trouverez en annexe une bibliographie consacrée à Oracle ainsi que de nombreux sites Web que j’ai jugés intéressants de mentionner ici.

© Éditions Eyrolles

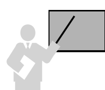
Conventions d'écriture

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 (exemple : `CREATE` { `TABLE` | `VIEW` }. Les symboles « [» et «] » précisent le caractère optionnel d'une directive au sein d'une commande (exemple : `CREATE TABLE Avion (…) [ORGANISATION INDEX];`).



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 que le code source est téléchargeable à partir du site des éditions Eyrolles (`www.editions-eyrolles.com`).

Ce pictogramme indique une astuce ou un conseil personnel.

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 (`soutou@iut-blagnac.fr`). Par ailleurs, il existe un site d'accompagnement qui contient les errata, compléments ainsi que le code des exemples et le corrigé de tous les exercices (`www.editions-eyrolles.com`).

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). Viennent ensuite les procédures d'installation des différentes éditions d'Oracle pour Windows actuellement sur le marché (9*i*, 10*g Express Edition*, 10*g* et 11*g*). Enfin, l'utilisation des interfaces de commandes est abordée pour que vous puissiez programmer avec SQL dès le chapitre 1.

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 (*Stuctured 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*:2003, *i* allant de 1 à 13). La plus récente partie de la norme de 2006 (ISO/IEC 9075-14:2006) est consacré à XML.

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`	Définition des données (LDD)
`INSERT – UPDATE – DELETE – MERGE – LOCK TABLE`	Manipulation des données (LMD)
`SELECT`	Interrogation des données (LID)
`GRANT – REVOKE – COMMIT – ROLLBACK – SAVEPOINT – SET TRANSACTION`	Contrôle des données (LCD)

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

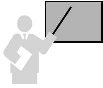
comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Pilote

brevet	nom	nbHVol	*compa*
PL-1	Amélie Sulpice	450	AF
PL-2	Thomas Sulpice	900	AF
PL-3	Paul Soutou	1000	SING

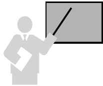
© Éditions Eyrolles

Les clés



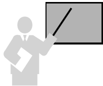
La clé primaire (*primary key*) d'une table est l'ensemble minimal de colonnes qui permet d'identifier de manière unique chaque enregistrement.

Dans la figure précédente, les colonnes « clés primaires » sont notées en gras. La colonne `comp` représente le code de la compagnie et la colonne `brevet` décrit le numéro du brevet.



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 notre exemple, les colonnes `nomComp` et `nom` peuvent être des clés candidates si on suppose qu'aucun homonyme n'est permis.



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.

© Éditions Eyrolles

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. La filiale française (Oracle France) a été créée en 1986, elle est composée de cinq départements (marketing, commercial, avant-vente, conseil et formation).

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 naisssance 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 Oracle 8*i*	*i* comme Internet, SQLJ – Linux – XML.
2001 Oracle9*i*	Services Web – serveur d'applications – architectures sans fil.
2004 Oracle10*g*	*g* comme *Grid computing* (ressources en *clusters*).
2007 Oracle11*g*	Auto-configuration.

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

© Éditions Eyrolles

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.

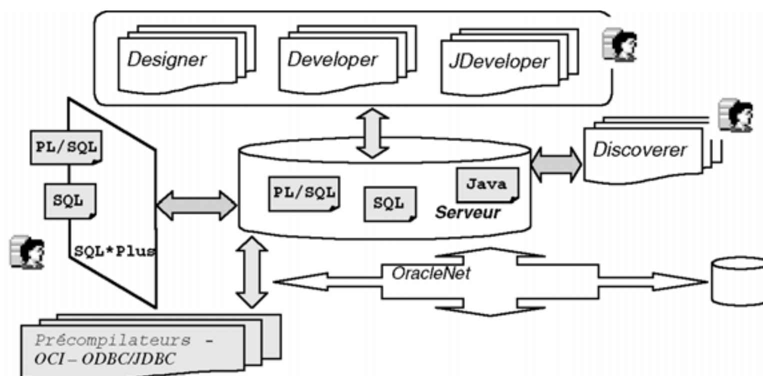
Offre du moment

Leader du marché des SGBD relationnels, Oracle disposait d'une part comprise entre 44 % (source IDC) et 47 % (source Gartner) en 2006, contre environ 21 % pour IBM et 18 % pour Microsoft. Oracle devrait être bien armé en 2008 (année probable de la *Release 2* de la version 11*g*) face à SQL Server 2008 de Microsoft, un des rares SGBD à lui grignoter des parts de marché.

Face à la montée en puissance des SGBD Open Source, fin 2005, Oracle (puis IBM, Sybase et Microsoft avec *SQL Server 2005 Express*) réagit en proposant le premier, une version gratuite du SGBD (*Oracle 10g Express Edition*) mais bridée en nombre de processeurs, d'enregistrements (4 Go) et d'espace mémoire (1 Go). Aucun éditeur ne veut être absent du marché très important des PME, sensible aux coûts et particulièrement réceptif aux promesses de l'Open Source.

La figure suivante schématise les produits d'Oracle qui se positionnent autour du serveur de données (SGBD) :

Figure I-2 *Offre Oracle*



Le langage SQL est utilisé explicitement par toute personne (ou par un outil) travaillant sur la base de données (administrateur, développeur, utilisateur). Le langage PL/SQL est le langage procédural d'Oracle. Il permet d'incorporer nativement tout ordre SQL dans un programme.

Les interfaces SQL*Plus sont le moyen minimal (plutôt spartiate mais efficace) pour se connecter et travailler avec la base. Le *middleware* OracleNet permet d'interconnecter des bases Oracle distantes entre elles ainsi que des bases non Oracle.

Les précompilateurs permettent d'intégrer SQL dans des programmes C, Fortran, COBOL, C++, etc. L'interface OCI (*Oracle Call Interface*) permet d'accéder à la base de données à partir de langages de troisième génération via des primitives de bas niveau.

Les produits *Designer*, *Forms* et *Reports* sont des outils d'aide à la conception et de création d'applications interactives ou d'états de tous types (écrans, rapports, *batchs* générés en PL/SQL, HTML ou VB).

Les produits *Developer* et *JDeveloper* sont des outils de développement d'applications client-serveur ou Internet-intranet.

Les produits *Discoverer* et *Express* sont spécialisés dans la gestion des info centres et des entrepôts de données (*datawarehouses*). Les utilisateurs ont la possibilité d'interroger la base en construisant graphiquement des requêtes.

L'intégration d'une machine virtuelle Java rebaptisée *Jserver* sous Oracle8*i* démontre la volonté d'Oracle d'inclure largement Java dans son offre. Oracle a ajouté par la suite un ORB (*Object Request Broker*) à son moteur pour s'ouvrir au protocole IIOP (*Inter Object Request Broker*) et supporter les composants EJB (*Enterprise Java Beans*). Le serveur Apache, inclus depuis peu au moteur d'Oracle, permet de programmer des *applets*, *servlets* ou des JSP (*Java Server Pages*). Enfin, le moteur accepte de conserver et de compiler des procédures stockées codées non plus en PL/SQL mais en Java.

Oracle offre également des outils d'administration en mode graphique. Les plus connus permettant d'exporter et d'importer des données, de charger une base à partir de fichiers externes (*SQL*Loader*), de configurer les couches réseaux (*Net Manager* et *Net Configuration Assistant*). La console principale se nomme *Oracle Enterprise Manager.* Elle permet d'administrer graphiquement une ou plusieurs bases de données locales ou distantes.

Oracle est composé de plusieurs domaines, cet ouvrage concerne simplement une infime partie du serveur de données.

Serveur de données (la base)

Oracle Database est disponible en plusieurs versions qualifiées de « Standard » ou « Enterprise ». Le nom du produit monoposte pour Windows ou Linux est *Personal Oracle*. Plusieurs options permettent de renforcer les performances, la sécurité, le traitement transactionnel et le *datawarehouse*. Citons *Oracle Data Guard*, *Oracle Real Application Clusters*, *Oracle Partitioning*, *Oracle Advanced Security, Oracle Label Security*, *Oracle Diagnostics Pack*, *Oracle Tuning Pack*, *Oracle OLAP*, *Oracle Data Mining, Oracle Spatial*.

© Éditions Eyrolles

Serveur d'applications

Oracle Application Server est un des serveurs d'applications compatible J2EE les plus complets et intégrés du marché. *OracleAS* regroupe des composants divers : serveur HTTP, portail (*Portal*), fonctions vocales et sans fil (*Wireless*), mise en cache des pages Web, etc.

Outils de développement

Les outils sont regroupés dans une offre appelée *Developer Suite* comprenant pour la partie *Application Developement* : *JDeveloper*, *Forms Developer* (outil RAD pour PL/SQL), *Designer* (conception et maintenance de schémas), *Software Configuration Manager*. La partie *Business Intelligence* inclut : *Warehouse Builder*, *Reports Developer*, *Discoverer* et *Business Intelligence Beans*.

*Suite Bureautique (*Collaboration Suite*)*

Cette offre est composée de *Oracle Email*, *Oracle Voicemail & Fax*, *Oracle Calendar*, *Oracle Files Management*, *Oracle Ultra Search*, *Oracle Wireless & Voice*. Cette offre permet d'utiliser une seule boîte de réception pour tous messages électroniques, vocaux et télécopies, et l'accès aux messages à partir de clients du marché, de navigateurs Web, de téléphones ou d'assistants numériques personnels.

Suites E-Business

L'offre *E-Business Suite* est un progiciel de gestion intégré (ensemble d'applications d'entreprise pour gérer les clients, produits, commandes, créances, etc.). La suite *E-Business Suite Special Edition* est dédiée aux PME et permet de préconfigurer *E-Business Suite* par une technologie Web.

*Externalisation (*Applications Outsourcing*)*

Ce produit est relatif au transfert à Oracle de la gestion des applications d'une entreprise (info gérance). Il est composé de *E-Business Suite*, *Oracle Small Business Suite*, *iLearning*, *Exchange*. Des services complets de gestion, de bases de données et de matériel sont proposés pour réduire les coûts de maintenance.

Divers

Oracle fournit enfin un service de support, de conseil (*Oracle Consulting*), de formation (*Oracle University*) et de financement (*Oracle Financing*).

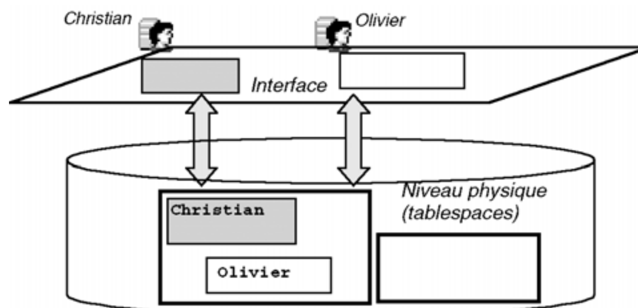
Notion de schéma

Un schéma est un ensemble comprenant des structures de données et des données. Il appartient à un utilisateur de la base et porte le nom de ce dernier. Chaque utilisateur possède ainsi

son propre schéma. Leurs éléments (Oracle les nomme d'une manière générique *objects*) sont créés et modifiés par des ordres SQL.

La figure suivante illustre deux utilisateurs travaillant sur leur schéma (stocké dans un espace de la base) par une interface qui peut être SQL*Plus (dans la majeure partie des enseignements), ou un langage de programmation – citons principalement C, COBOL, C++ ou Java.

Figure I-3 *Notion de schéma Oracle*



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

Tableau I-3 Éléments d'un schéma Oracle

Éléments étudiés – Chapitre	Aspects non étudiés
Déclencheurs (*triggers*) – 7	Clusters
Fonctions, procédures et paquetages – 7	Dimensions
Librairies de procédures externes – 11	Liens de bases de données (*database links*)
Index – 1	Opérateurs
Ressources Java – 9, 10, 11	Tables, types et vues objets
Séquences et synonymes – 2, 5	Vues matérialisées (anciennement « clichés »)
Tables et tables en index – 1	
Vues (*views*) – 5	
XML – 13	

Accès à Oracle depuis Windows

Une fois que vous aurez installé Oracle sur votre ordinateur, vous serez libre de choisir l'accès qui vous convient. Ce livre utilise essentiellement les interfaces SQL*Plus de l'éditeur, mais aussi Java via JDBC, le Web au travers des techniques décrites au chapitre 12 et XML avec XML DB.

 © Éditions Eyrolles

Il existe d'autres acccès possibles, citons Office (*Word*, *Excel*, *Powerpoint*, *Access*), Visual Studio, Active Server Pages (ASP) et les environnements .NET. Les passerelles employées s'appellent ODBC, Oracle OLE DB, Oracle Objects for OLE (OO4O), ActiveX Data Objects (ADO), Oracle Call Interface (OCI), ODBC .NET, Oracle OLE DB .NET, Oracle Data Provider for .NET.

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.

Pour contrôler la version du serveur sur lequel vous êtes connecté et celle des outils présents sur votre poste, exécutez le script suivant dans une interface SQL*Plus :

```
COL PRODUCT FORMAT A35
COL VERSION FORMAT A15
COL STATUS FORMAT A15
SELECT * FROM PRODUCT_COMPONENT_VERSION;
```

Installation d'Oracle

Rendez-vous à présent sur le site d'Oracle où vous devrez d'abord vous enregistrer (http://www.oracle.com/technology/software/products/database/index.html).

Cette section décrit les différentes procédures d'installation en fonction de la version que vous aurez choisi d'installer. Si vous avez un PC haut de gamme (au moins 2 Go de RAM et un processeur récent), vous pouvez opter pour la 11*g*, sinon préférez une version antérieure.

Vous pouvez aussi opter pour une version plus légère (*10*g *Express Edition*). Cependant, vous ne bénéficierez pas de tous les outils complémentaires, seule la programmation SQL sera possible en utilisant une simple interface en mode texte.

Mise en œuvre d'Oracle9i

Comptez une petite heure pour cette installation. Commencez par extraire chacun des fichiers téléchargés dans des sous-répertoires distincts d'un répertoire temporaire (exemple : `disk1`, `disk2` et `disk3` du répertoire `C:\Eyrolles`), puis exécutez `setup.exe`.

- Le choix est donné pour les répertoires source et cible.

Figure I-4 *Répertoires source et cible d'installation*

- Dans la fenêtre `Produits disponibles`, choisissez `Database/Suivant`.
- Dans la fenêtre `Types d'installations`, choisissez `Personal Edition` si vous envisagez de ne pas connecter par la suite d'autres clients sur votre machine. Dans le cas inverse, optez pour `Enterprise Edition`.
- Dans la fenêtre `Configuration de bases de données`, choisissez `Universel/Suivant`.
- Dans la fenêtre `Microsoft Transaction Server`, laissez le port par défaut (2030).
- Dans la fenêtre `Identification`, saisissez le nom de votre base sur 8 caractères maximum (exemple : `BDSoutou`). La possibilité est donnée d'appeler une base avec un domaine réseau (exemple : `BDSoutou.oracle.fr`) ; Oracle parle de *service name*.
- Dans la fenêtre `Emplacement des fichiers`, laissez le répertoire par défaut (le même que celui de l'installation).
- Dans la fenêtre `Jeu de caractères`, laissez le jeu par défaut.
- Lors du récapitulatif, Oracle vous prévient de la place qu'il va occuper. Assurez-vous que vous disposez de suffisamment d'espace disque.

© Éditions Eyrolles

Figure I-5 Nom de la base de données

De longues minutes vont s'écouler avant la saisie des mots de passe des comptes d'administration. Étirez bien la fenêtre de façon à voir le bouton `OK`. Saisissez deux mots de passe différents, et mémorisez-les car il n'existe aucun moyen simple de les retrouver (à part en sollicitant le support d'Oracle qui n'est pas gratuit).

Figure I-6 Gestion des mots de passe

Voilà, Oracle 9*i* est installé. Si ce n'est pas le cas, reportez-vous à la section suivante. Dernière chose : si vous n'utilisez pas souvent Oracle, pensez à arrêter et à positionner sur `Manuel` les quatre services d'Oracle (*Agent*, *Recovery*, *Listener*, *ServiceBase*). Il faudra redémarrer au moins les deux derniers pour la moindre instruction SQL.

Désinstallation de la 9i

Ne le cachons pas, la désinstallation d'Oracle peut devenir rapidement une véritable galère si vous supprimez des répertoires sans trop savoir ce que vous faites. L'ajout de produits supplémentaires n'est pas une mince affaire non plus. Il est étonnant qu'un éditeur comme Oracle ne fournisse toujours pas, sous Windows, une procédure propre qui arrête les services, supprime les fichiers et répertoires, les entrées dans les menus de démarrage, les variables d'environnement et nettoie la base de registres. Non, rien de tout ça, ou plutôt l'inverse : Oracle laisse tout après une désinstallation complète !

En bref, voici la procédure rapide et efficace à employer pour nettoyer sans rien oublier :

- Arrêtez tous les services d'Oracle.
- Supprimez le répertoire `Oracle` dans le répertoire `Program Files`. Redémarrez l'ordinateur.
- Supprimez le répertoire `Oracle` dans le répertoire d'installation. Si `oci.dll` dans le répertoire `bin` d'Oracle vous cause du tracas, arrêtez un des processus `SVCHOST.EXE` pour supprimer ce fichier.
- Entrez dans la base de registres (Menu `Démarrer/Exécuter…/regedit`) et supprimez la clé `ORACLE` dans `HKEY_LOCAL_MACHINE\SOFTWARE`.
- Pour les systèmes non XP, supprimez les clés correspondant à tous les services dans `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`.
- Pour les systèmes XP, recherchez les entrées « `Oracle` » dans la base de registres et supprimez les clés associées. Oui, c'est consternant, mais je n'ai pas trouvé mieux ! Supprimez également les clés correspondant à tous les services, il y en a neuf qui se trouvent dans : `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services` et `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet002\Services` ainsi que `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`.

Une fois que tout est fait, redémarrez votre machine et reprenez l'installation initiale. Il est plus prudent d'utiliser un nom de base différent à chaque nouvelle installation.

Mise en œuvre d'Oracle 10g

Comptez moins d'une heure pour installer Oracle10*g*. Il faut une assez grosse configuration (Pentium IV et 512 Mo de RAM). L'installation basique occupe 400 Mo en RAM et 2 Go sur le disque. Prudence toutefois si vous décidez d'installer ce produit tout en ayant déjà une

© Éditions Eyrolles

base version 9*i*. La procédure est plus simple que pour la version 9*i* qui nécessitait trois CD et davantage de menus. Le produit *Oracle Database Oracle 10g* est fournit avec deux CD. Le premier installe la base, la console d'administration et les interfaces de commande SQL*Plus. Le CD *Companion* contient les produits moins couramment installés (précompilateurs, pilotes et diverses extensions).

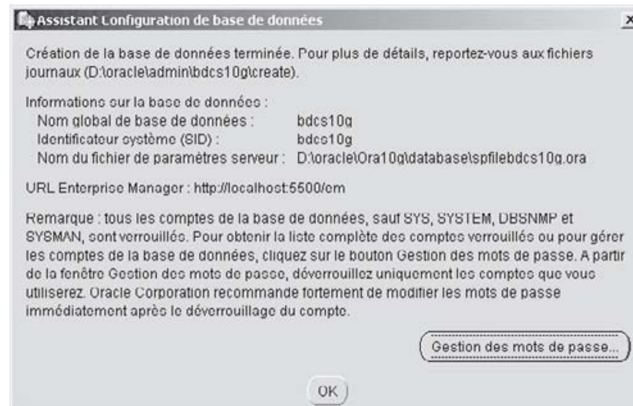
Extrayez le fichier téléchargé dans un répertoire temporaire puis exécutez `setup.exe`. Le choix est donné pour le répertoire cible (choisissez un répertoire vide si vous n'utilisez pas celui proposé par Oracle) et le type du produit (ici *Personal*). Donnez un nom à la base (ici `bdcs10g`), et un mot de passe aux comptes systèmes.

Figure I-7 *Répertoire cible d'installation*

Si vous disposez d'une base 9*i*, la fenêtre qui suivra vous offrira la possibilité de faire migrer cette base (*upgrade*) dans une nouvelle base 10*g*. Au récapitulatif, Oracle vous informe de la place qu'il va occuper. Si vous désirez installer d'autres produits ne figurant pas dans la liste, il faudra utiliser par la suite le CD *Companion*.

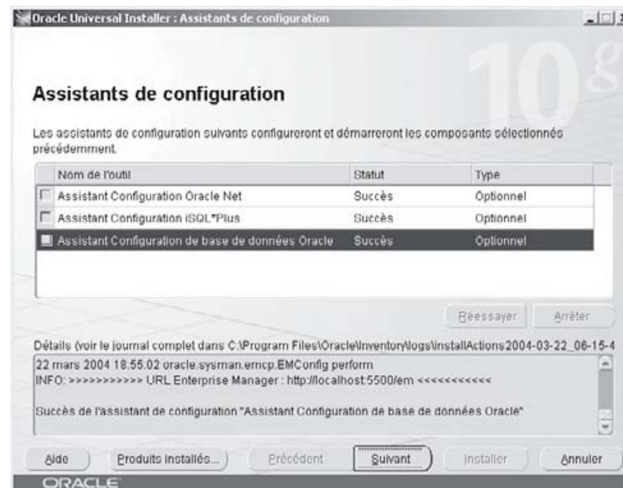
De longues minutes vont s'écouler avant de pouvoir modifier (éventuellement) les mots de passe des comptes d'administration. Personnellement, j'ai rencontré deux problèmes. Le premier à 43 % d'installation (échec de l'initialisation *Oracle Cluster Registry*), en choisissant `OK` le processus redémarre. À 63 %, l'installation bloque de nouveau quelques minutes (échec du démarrage du service *OracleCSService*), en choisissant `Réessayer` le processus redémarre.

Figure I-8 Modification éventuelle des mots de passe



Après ceci, l'installeur exécute différents assistants qui doivent tous se terminer avec succès.

Figure I-9 Assistants



Une fois Oracle installé, cliquez sur `Quitter`. Le navigateur démarre pour exécuter la nouvelle version de l'outil d'administration *Enterprise Manager*. Connectez-vous une première fois puis fermez le navigateur après avoir accepter les termes du contrat proposé. Oracle a désormais mis en place huit services (cinq sont automatiquement lancés).

© Éditions Eyrolles

Figure I-10 *Services initiaux d'Oracle 10g*

OracleCSService		Déma...	Automatique
OracleDBConsoleBD10G		Déma...	Automatique
OracleJobSchedulerBD10G			Désactivé
OracleOraDb10g_home1iSQL*Plus	iSQL*Plus ...	Déma...	Automatique
OracleOraDb10g_home1SNMPPeerEn...			Manuel
OracleOraDb10g_home1SNMPPeerM...			Manuel
OracleOraDb10g_home1TNSListener		Déma...	Automatique
OracleServiceBD10G		Déma...	Automatique

Si vous n'utilisez pas quotidiennement Oracle, pensez à arrêter ou positionner ces services sur `Manuel` pour économiser des ressources.

Désinstallation de la 10g

La désintallation d'Oracle 10*g* s'opère de la manière suivante :

- Arrêtez tous les services d'Oracle.
- Entrez dans la base de registres (Menu `Démarrer/Exécuter…regedit`) et supprimez les clés suivantes :
  - `ORACLE` dans `HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE` si vous n'avez pas d'autres base Oracle, sinon supprimez également les clés `ORA_CRS_HOME, KEY_OraDb10g_home1, Ocr, SCR` et `SYSMAN` ;
  - relatives à Oracle dans les clés `HKEY_CLASSES_ROOT` ;
  - pour les systèmes non XP, supprimez les clés correspondant aux huit services dans `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.` Pour les systèmes XP, ils se trouvent dans `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services et ControlSet002` ou `3`.
- Enlevez les chemins mis en place au niveau des variables d'environnement (Panneau de configuration, Système onglet `Avancé` sous XP).
- Supprimez, s'il existe, le répertoire `C:\Documents and Settings\utilisateur\Local Settings\Temp\OraInstallDate`.
- Enlevez les entrées du menu `Démarrer` (sous XP `C:\Documents and Settings\All Users\Menu Démarrer\Programmes`).
- Supprimez le répertoire `Oracle` sous `Program Files`. Rédémarrez votre ordinateur.
- Supprimez le répertoire où vous avez installé Oracle. Si `oci.dll` dans le répertoire `bin` vous cause tracas, arrêtez le processus `SVCHOST.EXE` (celui qui occupe le plus d'espace en mémoire) pour supprimer ce fichier. Videz la corbeille. Redémarrez votre ordinateur.

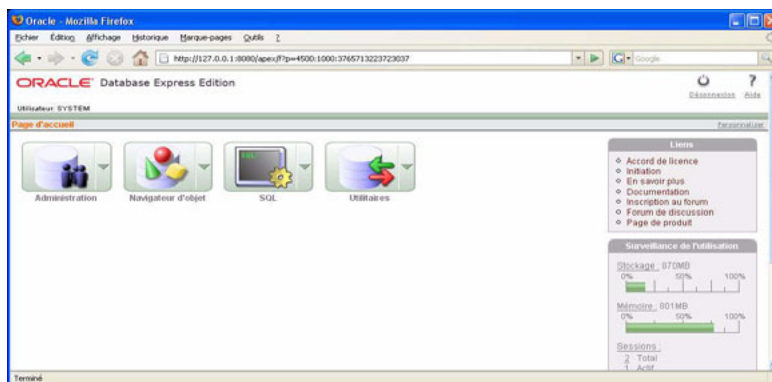
Défragmentez enfin l'unité de disque qui a contenu la base avant d'entreprendre une nouvelle installation. Il est aussi plus prudent d'utiliser un nom de base différent à chaque nouvelle installation.

© Éditions Eyrolles

Mise en œuvre d'Oracle 10*g* Express Edition

Vous devriez installer cette version en une vingtaine de minutes. Exécutez le fichier téléchargé, un assistant se lance, acceptez le contrat et validez l'emplacement du logiciel (par défaut `C:\oraclexe`). Il faudra ensuite saisir et confirmer le mot de passe des comptes administrateur d'Oracle. Il vous servira lors de la première connexion (lancement initial proposé à la fin de l'installation).

Figure I-11 *Console 10*g *Express Edition*



Trois services sont désormais installés (deux d'entre-eux seront automatiquement lancés), pensez à positionner ces services sur `Manuel` si vous n'utilisez pas souvent le SGBD.

Des entrées dans le menu `Démarrer` sont crées, elles permettent d'arrêter la base, de la relancer et de lancer une interface de commande (SQL*Plus en mode caractères). Vous pourrez travailler avec cette interface une fois que vous aurez créé un utilisateur à l'aide de la console (en vous connectant à l'aide de la commande `connect`).

Pour désinstaller cette version, suivez la procédure de désinstallation de la version 10*g*.

Mise en œuvre d'Oracle 11g

Si vous ne disposez pas d'au moins 2 Go de RAM, 5 Go de disque disponible et d'un processeur d'au moins 550 MHz (pour Vista, 800 MHz), il est préférable d'utiliser la version 10*g*. L'installation de la version 11*g* s'apparente à celle de la 10*g* comme nous allons le voir mais les outils SQL*Plus et *i*SQL*Plus ne sont plus présents.

Le produit *Oracle Database Oracle11g* est fournit en plusieurs parties. La première installe la base et certains outils comme XML DB et SQL Developer. La partie *Examples* contient les produits moins couramment installés (bases démos, précompilateurs, pilotes et diverses extensions). D'autres parties existent (*client*, *cluster* et *gateway*).

© Éditions Eyrolles

En fonction du nombre de processeurs, de la RAM et de la taille de la base souhaitée, vous devrez choisir parmi les différentes éditions du SGBD : *Personal*, *Standard* ou *Enterprise*. Il existe des options supplémentaires concernant la performance, la disponibilité, la sécurité et l'administration pour la version la plus complète.

Si vous envisagez d'utiliser la console d'administration (application Java gourmande en CPU), soignez au préalable votre configuration réseau. Vous êtes en DHCP : installez impérativement un adaptateur de boucle Microsoft (voir la documentation *Installation Guide for Microsoft Windows* chapitre 2 - *Oracle Database Preinstallation Requirements*). N'utilisez pas le navigateur de Microsoft avec un mode de sécurité avancé. Si vous changez d'adresse IP ou de nom de machine, vous devrez reconfigurer la console (par les commandes `emctl` et `emca`).

Extrayez le fichier téléchargé dans un répertoire temporaire puis exécutez `setup.exe` qui se trouve dans le répertoire `database`. Le choix est ensuite donné pour les répertoires cibles et l'édition du logiciel (ici *Personal*). Donnez un nom à la base (ici `bdcs11g`) et un mot de passe aux comptes système.

Figure I-12 *Installation d'Oracle 11*g

Après vérification de votre configuration, vous pouvez paramétrer votre support (si vous disposez d'un compte metalink). Un récapitulatif s'affiche ensuite. J'ai rencontré une attente à 89 % de l'assistant configuration de la base de données. Ensuite, vous devrez modifier les mots de passe des comptes d'administration, profitez-en pour déverrouiller certains comptes de démonstration (`SCOTT`, `XDB`, `HR` et `OE`). Placez dans vos marque-pages l'URL de la console d'administration (dans mon cas `https://camparols.iut-blagnac.fr:1158/em`).

Une fois Oracle installé, cliquez sur `Quitter`. En fonction de votre configuration, consultez les éventuelles tâches de post-installation (voir la documentation *Installation Guide for Microsoft Windows,* chapitre 4). Oracle a mis en place quatre services (trois seront automatiquement lancés).

Figure I-13 *Services initiaux d'Oracle 11g*

OracleDBConsolebdcs11g	Démarré	Automatique
OracleJobSchedulerBDCS11G		Désactivé
OracleOraDb11g_home1TNSListener	Démarré	Automatique
OracleServiceBDCS11G	Démarré	Automatique

Si vous ne travaillez pas quotidiennement avec Oracle, pensez à positionner ces services sur `Manuel` pour économiser des ressources (notamment la console si vous n'utilisez que SQL Developer). J'ai rencontré (sous XP) un message d'erreur en voulant redémarrer la console qui est toutefois opérationnelle par la suite.

La console Java *Enterprise Manager* n'est pas la même que celle de la version 10*g :* elle est désormais basée sur plusieurs onglets (Accueil, Performances, Disponibilité, Serveur, Schéma, Mouvement de données, Logiciel et fichiers associés). Attention, votre console ne fonctionnera pas si votre configuration réseau a changé depuis l'installation.

Figure I-14 *Console Oracle 11g*

Désinstallation de la 11g

Pour désintaller Oracle 11*g*, suivez la même procédure que pour la version 10*g*.

© Éditions Eyrolles

Les interfaces SQL*Plus

Les interfaces SQL*Plus permettent de dialoguer avec la base de différentes manières :

- exécution de commandes SQL 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.

Généralités

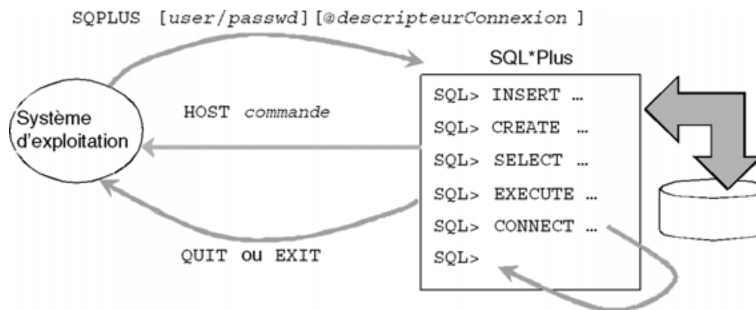
Plusieurs interfaces SQL*Plus sont disponibles sous Windows :

- en mode ligne de commande (qui ressemble à une fenêtre DOS ou `telnet`) ;
- avec l'interface graphique (qui est la plus connue dans le 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).

Les interfaces graphiques SQL*Plus et *i*SQL*Plus ne sont plus offertes depuis la version 11*g*. L'interface en mode ligne de commande reste toutefois disponible pour toutes les versions.

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-15 *Principe général des interfaces SQL*Plus*



Connexion à Oracle

Quel que soit le mode de connexion que vous allez choisir, il faudra saisir au moins deux paramètres (utilisateur et mot de passe). Un troisième paramètre est optionnel, il s'agit du descripteur de connexion qui indique la base à laquelle vous voulez vous connecter. Si vous n'avez qu'une base (Oracle parle d'« instance ») et que vous êtes sur la machine qui l'héberge, nul besoin de renseigner ce paramètre (ne rien mettre, ou inscrire le nom de la base). Dans le cas contraire, il faudra utiliser un descripteur de connexion, défini par l'outil *Net Configuration Assistant*, et qui se trouvera codé dans le fichier de configuration `tnsnames.ora` (situé dans un sous-répertoire d'Oracle `...\network\admin`) en fonction de la version dont vous disposerez.

Le code suivant décrit ma connexion locale : le descripteur de connexion est surligné (`CXBDSOUTOU`), le nom de la base en gras (`BDSOUTOU`). J'ai volontairement distingué ces deux identificateurs et je vous conseille d'en faire autant.

```
CXBDSOUTOU = (DESCRIPTION = (ADDRESS_LIST =
   (ADDRESS = (PROTOCOL = TCP)(HOST = Camparols)(PORT = 1521))  )
     (CONNECT_DATA = (SERVICE_NAME = BDSoutou)  )  )
```

Dans les exemples qui suivent, vous pouvez vous connecter sous l'utilisateur `SYSTEM` avec le mot de passe que vous avez indiqué lors de l'installation.

Mode ligne de commande

Dans une fenêtre de commandes, lancez `sqlplus`. Un nom d'utilisateur et un mot de passe sont demandés. Pour les connexions distantes, il faut relier le nom du descripteur de connexion à celui de l'utilisateur (exemple : `soutou@cxbdsoutou`).

Figure I-16 *Interface en mode ligne de commande*

```
Invite de commandes - sqlplus
C:\>sqlplus

SQL*Plus: Release 9.2.0.3.0 - Production on Sa Sep 20 14:10:06 2003

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.

Entrez le nom utilisateur : soutou@cxbdsoutou
Entrez le mot de passe :

Connectú ó :
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

SQL> _
```

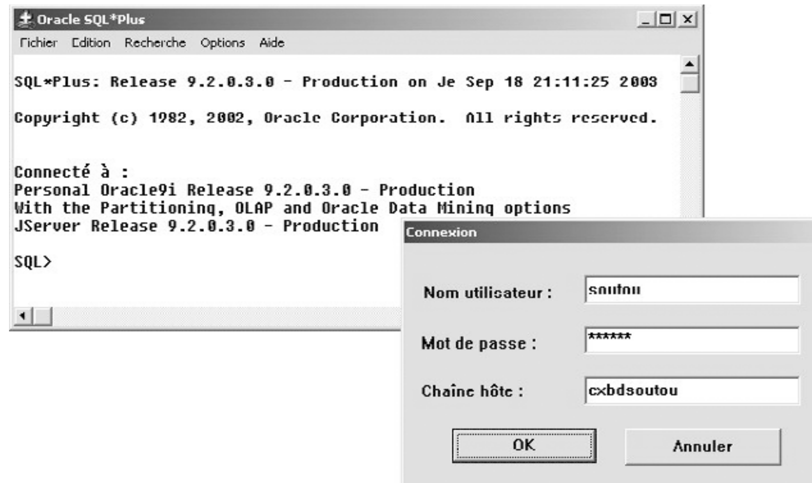
 © Éditions Eyrolles

*SQL*Plus graphique*

Cette interface est la plus pratique, car elle permet facilement de copier-coller des blocs instructions SQL et PL/SQL. Malheureusement la version 11*g* n'en dispose pas.

Pour lancer SQL*Plus graphique, menu `Démarrer Programmes/Oracle…/Application Development/SQL Plus`. Saisir les trois champs afin d'obtenir la fenêtre de commandes suivante :

Figure I-17 *Interface SQL*Plus graphique*



Vous pouvez placer sur le Bureau Windows l'icône de cette interface (exécutable `sqlplusw.exe` dans le répertoire `Oracle\ora92\bin` pour la version 9*i*).

*SQL*Plus Worksheet*

Présente en version 9*i*, cette interface dépend du produit *Enterprise Manager*. Elle possède de nouvelles fonctionnalités (historisation des commandes, colorisation des mots réservés SQL et PL/SQL, gestion des commandes déjà passées, etc.). Les instructions s'inscrivent dans la fenêtre du haut, celle du bas affiche les résultats. Il n'y a pas d'invite de commande (*prompt* `SQL>`). Pour lancer SQL*Plus Worksheet, menu `Démarrer Programmes/Oracle…/Application Development/SQL Worksheet`. Saisir les trois champs pour obtenir la fenêtre de commandes suivante :

Figure I-18 *Interface SQL*Plus Worksheet*

Vous pouvez placer l'icône sur votre bureau (exécutable `oemapp.bat worksheet` placé dans le répertoire `Oracle\ora92\bin` pour la version 9*i*).

*iSQL*Plus*

Cette interface n'existe que pour les versions 9*i* et 10*g*, elle est gérée par le serveur Web d'Oracle Apache (sous Oracle9*i*, le port est indiqué dans le fichier `httpd.conf` situé dans le répertoire `Oracle\ora92\Apache\Apache\conf` sous Windows). Le chapitre 12 indique la configuration que nous avons adoptée (port 77).

Pour lancer *i*SQL*Plus sous Oracle9*i*, entrez l'URL : `http://`*`nomMachine:port`*`/isqlplus` dans votre navigateur. La première fenêtre permet de se connecter.

Une fois connecté, la deuxième fenêtre (c'est-à-dire la fenêtre principale) présente de nombreuses possibilités que vous découvrirez facilement. Les résultats des commandes s'inscrivent à la suite de la fenêtre.

Pour lancer *i*SQL*Plus sous Oracle10*g*, inscrivez l'URL `http://`*`nomMachine`*`:5560/isqlplus` dans votre navigateur.

© Éditions Eyrolles

Figure I-19 *Connexion via iSQL*Plus*

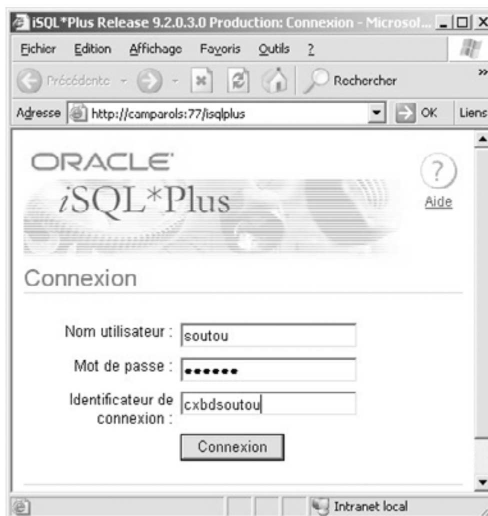
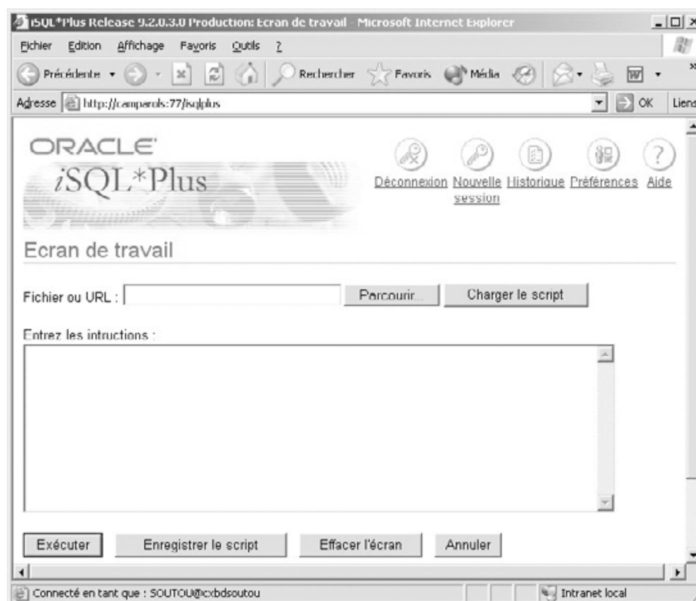


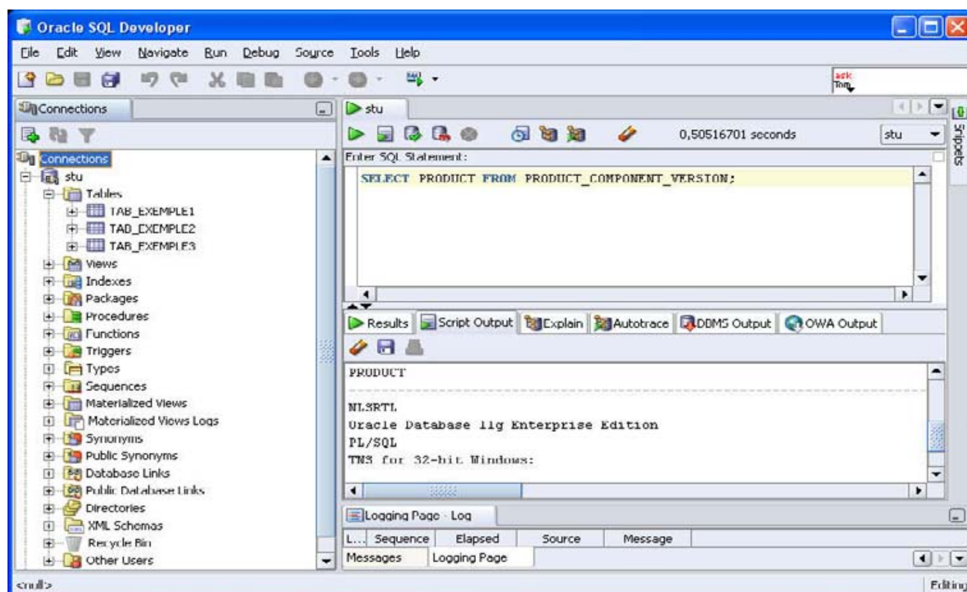
Figure I-20 *Interface iSQL*Plus*



SQL Developer

En l'absence d'interface graphique, la version 11*g* d'Oracle propose l'outil *SQL Developer* (menu `Démarrer Oracle.../Développement d'applications`). Au premier lancement, il vous sera demandé le chemin du répertoire contenant l'exécutable `java.exe`. *SQL Developer* permet de nombreuses fonctionnalités pour manipuler tous les objets d'un schéma (tables, procédures, déclencheurs, vues, etc.), cependant aucune commande SQL*Plus (`COL`, `ACCEPT`...) n'est prise en compte.

Figure I-21 *SQL Developer*



Premiers pas

La procédure suivante va guider vos premiers pas pour travailler sous les différentes interfaces d'une manière professionnelle. Il s'agit de stocker vos fichiers de commandes qui pourront servir à différentes actions (création de tables, de vues ou d'utilisateurs, insertions, modifications ou suppressions d'enregistrements, requêtes, blocs PL/SQL et sous-programmes PL/SQL, etc.).

Si vous travaillez avec la 11*g*, utilisez *SQL Developer* sinon, optez pour l'interface graphique SQL*Plus. Une fois familiarisé avec SQL*Plus, il vous sera aisé d'utiliser les autres interfaces graphiques (SQL*Plus Worksheet et *i*SQL*Plus) qui sont plus intuitives. Si vous envisagez

© Éditions Eyrolles

Unix, vous devrez également connaître les fonctionnalités de l'interface en mode ligne de commande.

Vos premières commandes : création d'un utilisateur

Si cela n'a pas été déjà fait, vous allez créer un utilisateur Oracle. Ouvrez le fichier `CreaUtilisateur.sql` qui se trouve dans le répertoire `Introduction`, à l'aide du Bloc-notes (ou d'un éditeur de texte de votre choix). Changez `<nom de l'utilisateur>` par le nom de l'utilisateur à créer – enlevez les symboles « < » et « > ». Inscrivez également un mot de passe. Enregistrez ce fichier dans l'un de vos répertoires (attention de ne pas utiliser le caractère espace dans le nom de vos répertoires).

Ouvrez ce fichier par le menu `Fichier/Ouvrir` (ou `GET` pour l'interface en ligne de commande, il faut exécuter `sqlplus` au niveau d'une fenêtre DOS dans le répertoire qui contient le fichier). Vous devez maintenant visualiser le fichier à l'écran. Sous l'interface graphique, il faut faire *`Enter`* et le prompt SQL s'affiche. Il reste à exécuter ce fichier avec :

- la commande `R` (*run*) pour l'interface en ligne de commande ou SQL*Plus graphique ;
- ou l'icône qui ressemble à un éclair sous SQL*Plus Worksheet ;
- ou le bouton *`Exécuter`* sous *i*SQL*Plus.

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

Il est inutile de toujours préciser le chemin du répertoire si vous exécutez à nouveau ce fichier ou d'autres se trouvant dans le même répertoire. Si vous désirez travailler à partir d'un autre répertoire, précisez le nouveau chemin (menu `Fichier/Ouvrir`).

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.

Commandes de base

Comme pour le langage SQL, les commandes de SQL*Plus sont insensibles à la casse. Nous les noterons toutes en majuscules pour respecter nos conventions. Une fois écrite, la commande (instruction SQL ou bloc PL/SQL) peut être manipulée, avant ou après son exécution. Le tableau suivant indique les commandes de base pour manipuler le *buffer* d'entrée de toutes les interfaces, sauf pour *i*SQL*Plus qui propose ces options d'une manière intuitive via des boutons.

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

Commande	Commentaires
R	Exécute (*run*).
L	Liste le contenu du buffer.
L*	Liste la ligne courante.
L*n*	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.

Notez qu'il est possible avec *i*SQL*Plus de charger un script distant via la commande START par les protocoles HTTP ou FTP. En ce cas, l'argument devient {http|ftp}:// *nomMachine*:*port*/*fichier.sql*.

Variables d'environnement

Les variables d'environnement permettent de paramétrer une session SQL*Plus. L'affectation d'une variable s'opère à l'aide de la commande SET ou aussi à l'aide d'un menu graphique (pour les deux interfaces graphiques et l'interface Web). Le tableau suivant résume les principales fonctions qui ne sont pas disponibles en ligne pour *i*SQL*Plus.

© Éditions Eyrolles

Tableau I-5 Variables d'environnement

Commande	Commentaires
AUTOCOMMIT {ON | OFF | IMMEDIATE | n}	Validation automatique après une ou *n* commandes.
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 {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.

L'état d'une variable d'environnement est donné par la commande SHOW (ou par un menu graphique). Le tableau suivant décrit quelques paramètres de cette commande.

Tableau I-6 Paramètres de la commande SHOW

Commande	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

Si vous envisagez d'utiliser des accents dans des sessions SQL*Plus pour vos tables, colonnes, etc., vous devez vérifier le paramétrage de la variable Oracle NLS_LANG sur le poste client (et non pas du côté du SGBD comme certains le pensent).

Mode ligne de commande

Dans le cas de SQL*Plus en ligne de commande, exécutez dans une fenêtre DOS la commande set NLS_LANG=FRENCH_FRANCE.WE8PC850. Lancez ensuite l'interface d'Oracle par la commande sqlplus (si vous avez différentes versions d'Oracle, positionnez-vous au préalable sur le répertoire contenant cet exécutable, cd C:\app\soutou\product\11.1.0\db_1\BIN dans mon cas). Pour tester votre configuration, exécuter ces instructions les unes après les autres.

```
CREATE TABLE tableAccentuée (colé VARCHAR2(50));
INSERT INTO tableAccentuée VALUES('Test éphémère sur SQL*Plus.');
SELECT * FROM tableAccentuée ;
DROP TABLE tableAccentuée;
```

Autres interfaces

Pour *SQL Developer* et les autres interfaces graphiques, c'est dans la base de registre que cela se joue. Assurez-vous d'avoir la valeur `NLS_LANG=FRENCH_FRANCE.WE8MSWIN1252` dans l'entrée `HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE`.

© Éditions Eyrolles

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)

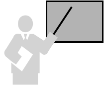
Pour pouvoir créer une table dans votre schéma, il faut que vous ayez reçu le privilège `CREATE TABLE`. Si vous avez le privilège `CREATE ANY TABLE`, vous pouvez créer des tables dans tout schéma. Le mécanisme des privilèges est décrit au chapitre « Contrôle des données ».

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 des lettres majuscules ou minuscules (accentuées ou pas), des chiffres et les symboles, par exemple : _, $ et #. Oracle est insensible à la casse et convertira au niveau du dictionnaire de données les noms de tables et de colonnes en majuscules.

- *colonnei typei* : nom d'une colonne (mêmes caractéristiques que pour les noms des tables) et son type (NUMBER, CHAR, DATE…). Nous verrons quels types Oracle propose. La directive DEFAULT fixe une valeur par défaut. La directive NOT NULL interdit que la valeur de la colonne soit nulle.



NULL représente une valeur qu'on peut considérer comme non disponible, non affectée, inconnue ou inapplicable. Elle est différente d'un espace pour un caractère ou d'un zéro pour un nombre.

- *nomContraintei typeContraintei* : 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

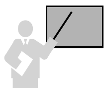
Dans toute instruction SQL (déclaration, manipulation, interrogation et contrôle des données), il est possible d'inclure des retours chariots, des tabulations, espaces et commentaires (sur une ligne précédée de deux tirets --, sur plusieurs lignes entre /* et */). De même, la casse n'a pas d'importance 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

<table>
<tr><th>Sans commentaire</th><th>Avec commentaires</th></tr>
<tr><td><pre>
CREATE TABLE MêmesévénementsàNoël
(colonne CHAR);

CREATE TABLE Test
 (colonne NUMBER(38,8));

CREATE table test (Colonne
NUMBER(38,8));
</pre></td><td><pre>
CREATE TABLE
     -- nom de la table
TEST(
     -- description
     COLONNE NUMBER(38,8)
)
-- fin, ne pas oublier le point-virgule.
;
CREATE TABLE Test (
    /* une plus grande description
       des colonnes */
     COLONNE NUMBER(38,8));
</pre></td></tr>
</table>



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

 © Éditions Eyrolles

« `nomComp='Air France'` » n'aura pas la même signification que l'expression « `nomComp ='AIR France'` ».

Comme nous le conseillons dans l'avant-propos, il est préférable d'utiliser les conventions suivantes :

- tous les mots-clés de SQL sont notés en `MAJUSCULES` ;
- les noms de tables sont notés en `Minuscules` (excepté la première lettre) ;
- les noms de colonnes et de contraintes en `minuscules`.

L'adoption de ces conventions rendra vos requêtes, scripts et programmes plus lisibles (un peu à la mode Java).

Premier exemple

Le tableau ci-dessous décrit l'instruction SQL qui permet de créer la table `Compagnie` illustrée par la figure suivante dans le schéma *soutou* (l'absence du préfixe « `soutou.` » aurait conduit au même résultat si *soutou* était l'utilisateur qui crée la table).

Figure 1-1 *Table à créer*

Compagnie

comp	nrue	rue	ville	nomComp

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

Instruction SQL	Commentaires

`CREATE TABLE soutou.Compagnie`
`(comp CHAR(4),`
`nrue NUMBER(3),`
`rue CHAR(20),`
`ville CHAR(15) DEFAULT 'Paris',`
`nomComp CHAR(15) NOT NULL);`
	La table contient cinq colonnes (quatre chaînes de caractères et une valeur numérique de trois chiffres).
	La table inclut en plus deux contraintes :
	• `DEFAULT` qui fixe *Paris* comme valeur par défaut de la colonne `ville` ;
	• `NOT NULL` qui impose une valeur non nulle dans la colonne `nomComp`.

Contraintes

Les contraintes 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 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*).

Oracle recommande de déclarer les contraintes `NOT NULL` en ligne, les autres peuvent être déclarées soit en ligne, soit nommées. Étudions à présent les types de contraintes nommées (*out-of-line*).

Quatre types de contraintes sont 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 au niveau de la table (les valeurs nulles font exception à moins que `NOT NULL` soit aussi appliquée sur les colonnes).
- La contrainte `PRIMARY KEY` déclare la clé primaire de la table. Un index est généré automatiquement sur la ou les colonnes concernées. Les colonnes clés primaires ne peuvent être ni nulles ni identiques (en totalité si elles sont composées de plusieurs colonnes).
- La contrainte `FOREIGN KEY` déclare une clé étrangère entre une table enfant (*child*) et une table père (*parent*). Ces contraintes définissent l'intégrité référentielle que nous aborderons plus tard. La directive `ON DELETE` dispose de deux options : `CASCADE` propagera la suppression de tous les enregistrements fils rattachés à l'enregistrement père supprimé, `SET NULL` positionnera seulement leur clé étrangère à `NULL` (voir la section « Intégrité référentielle » du chapitre 2).
- La contrainte `CHECK` impose un domaine de valeurs ou une condition simple ou complexe entre colonnes (exemple : `CHECK (note BETWEEN 0 AND 20)`, `CHECK (grade='Copilote' OR grade='Commandant')`).

Il n'est pas recommandé de définir des contraintes sans les nommer (bien que cela soit possible), car il sera difficile de faire évoluer les contraintes déclarées (désactivation, réactivation, suppression) et la lisibilité des programmes en sera affectée.

Si vous ne nommez pas une contrainte, un nom est automatiquement généré sous la forme suivante : `SYS_Cnnnnnn` (*n* entier).

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

 © Éditions Eyrolles

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`).

En respectant nos conventions, déclarons les tables de l'exemple suivant (`Compagnie` avec sa clé primaire et `Pilote` avec ses clés primaire et étrangère). Du fait de l'existence de la clé étrangère, la table `Compagnie` est dite « parent » (ou « père ») de la table `Pilote` « enfant » (ou « fils »). Cela résulte de l'implantation d'une association *un-à-plusieurs* entre les deux tables (bibliographie *UML 2 pour les bases de données*). Nous reviendrons sur ces principes à la section « Intégrité référentielle » du prochain chapitre.

Figure 1-2 *Deux tables à créer*

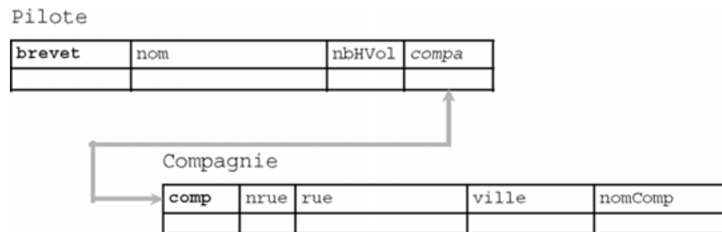


Tableau 1-3 Contraintes en ligne et nommées

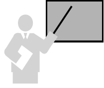
Tables	Contraintes

`CREATE TABLE Compagnie`
` (comp CHAR(4), nrue NUMBER(3),`
` rue CHAR(20), ville CHAR(15) DEFAULT 'Paris',`
` nomComp CHAR(15) NOT NULL,`
` CONSTRAINT pk_Compagnie PRIMARY KEY(comp));`
	Deux contraintes en ligne et une contrainte nommée de clé primaire.

`CREATE TABLE Pilote`
`(brevet CHAR(6), nom CHAR(15),`
` nbHVol NUMBER(7,2), compa CHAR(4),`
` CONSTRAINT pk_Pilote PRIMARY KEY(brevet),`
` CONSTRAINT nn_nom CHECK (nom IS NOT NULL),`
` CONSTRAINT ck_nbHVol CHECK`
`       (nbHVol BETWEEN 0 AND 20000),`
` CONSTRAINT un_nom UNIQUE (nom),`
` CONSTRAINT fk_Pil_compa_Comp FOREIGN KEY`
`       (compa) REFERENCES Compagnie(comp));`
	Aucune contrainte en ligne et cinq contraintes nommées :
	• Clé primaire
	• `NOT NULL`
	• `CHECK` (nombre d'heures de vol compris entre 0 et 20000)
	• `UNIQUE` (homonymes interdits)
	• Clé étrangère

© Éditions Eyrolles

Remarques



- L'ordre n'est pas important dans la déclaration des contraintes nommées.
- Une contrainte `NOT NULL` doit être déclarée dans un `CHECK` si elle est nommée.
- `PRIMARY KEY` équivaut à : `UNIQUE` + `NOT NULL` + index.
- L'ordre de création des tables est important quand on définit les contraintes en même temps que les tables (on peut différer la création ou l'activation des contraintes, voir le chapitre 3). Il faut créer d'abord les tables « pères » puis les tables « fils ». Le script de destruction des tables suit le raisonnement inverse.

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

Les types `CHAR` et `NCHAR` permettent de stocker des chaînes de caractères de taille fixe.

Les types `VARCHAR2` et `NVARCHAR2` permettent de stocker des chaînes de caractères de taille variable (`VARCHAR` est maintenant remplacé par `VARCHAR2`).

Les types `NCHAR` et `NVARCHAR2` permettent de stocker des chaînes de caractères Unicode (*multibyte*), méthode de codage universelle qui fournit une valeur de code unique pour chaque caractère quels que soient la plate-forme, le programme ou la langue. Unicode est utilisé par XML, Java, JavaScript, LDAP, et WML. Ces types Oracle sont proposés dans le cadre NLS (*National Language Support*).

Les types `CLOB` et `NCLOB` permettent de stocker des flots de caractères (exemple : du texte).

© Éditions Eyrolles

Tableau 1-4 Types de données caractères

Type	Description	Commentaire pour une colonne
CHAR(*n* [BYTE | CHAR])	Chaîne fixe de *n* caractères ou octets.	Taille fixe (complétée par des blancs si nécessaire). Maximum de 2000 octets ou caractères.
VARCHAR2(*n* [BYTE|CHAR])	Chaîne variable de *n* caractères ou octets.	Taille variable. Maximum de 4000 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 2000 caractères.
NVARCHAR2(*n*)	Chaîne variable de *n* caractères Unicode.	Taille variable. Mêmes caractéristiques que NCHAR sauf pour la taille maximale qui est ici de 4000 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. Plus utilisé mais fourni pour assurer la compatibilité avec les anciennes applications.

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 $\pm 9.99 \times 10^{125}$.

Tableau 1-5 Types de données numériques

Type	Description	Commentaires pour une colonne
NUMBER[(t,d)]	Valeur numérique de *t* chiffres dont *d* décimales.	Maximum pour *t* + *d* : 38. Espace maximum utilisé : 21 octets.

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.

© Éditions Eyrolles

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 1-6 Types de données date/heure

Type	Description	Commentaires pour une colonne
DATE	Date et heure du 1er janvier 4712 avant JC au 31 décembre 4712 après JC.	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.

Données binaires

Les types BLOB et BFILE permettent de stocker des données non structurées (structure opaque pour Oracle) comme le multimédia (images, sons, vidéo, etc.).

Tableau 1-7 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. Plus utilisé mais fourni pour assurer la compatibilité avec les anciennes applications.
LONG RAW	Données binaires.	Comme RAW, jusqu'à 2 gigaoctets.

© Éditions Eyrolles

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.

Retrouvons la structure des tables `Compagnie` et `Pilote` précédemment créées. Le type de chaque colonne apparaît :

Tableau 1-8 Structure brute des tables

Table Compagnie	**Table** Pilote

<pre>SQL> DESC Compagnie
 Nom   NULL ?   Type
  -------------------------------- 	
 COMP   NOT NULL CHAR(4)
 NRUE   NUMBER(3)
 RUE   CHAR(20)
 VILLE   CHAR(15)
 NOMCOMP   **NOT NULL** CHAR(15)</pre>
	<pre>SQL> DESC Pilote
	 Nom NULL ? Type
	  --------------------------------- 	
	 BREVET NOT NULL CHAR(6)
	 NOM CHAR(15)
	 NBHVOL NUMBER(7,2)
	 COMPA CHAR(4)</pre>

Les contraintes NOT NULL nommées (définies via les contraintes CHECK) n'apparaissent pas mais sont pourtant actives (c'est le cas de la contrainte nn_nom sur la colonne nom). Les colonnes clés primaires sont toujours définies en ligne NOT NULL.

Restrictions

La commande DESC n'affiche que les contraintes NOT NULL définies en ligne au niveau des colonnes (en gras dans le script).

Les noms des tables et contraintes ne doivent pas dépasser 30 caractères. Ces noms doivent être uniques dans le schéma (restriction valable pour les vues, index, séquences, synonymes, fonctions, etc.).

Les noms des colonnes doivent être uniques pour une table donnée (il est en revanche possible d'utiliser le même nom de colonne dans plusieurs tables).

Les noms des objets (tables, colonnes, contraintes, vues, etc.) ne doivent pas emprunter des mots-clés du SQL d'Oracle TABLE, SELECT, INSERT, IF… Si vous êtes francophone, cela ne vous gênera pas.

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 '';
```

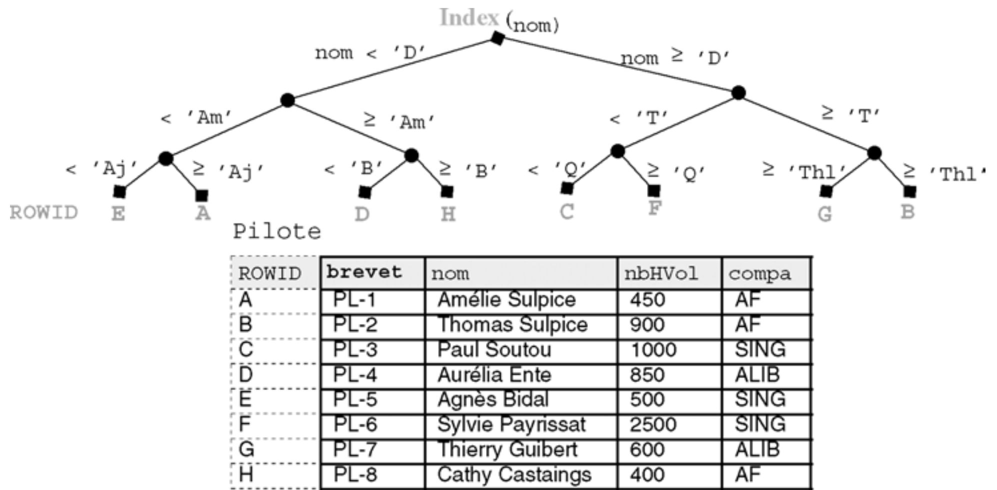
Index

Comme l'index de cet ouvrage vous aide à atteindre les pages concernées par un mot recherché, un index Oracle permet d'accélérer l'accès aux données d'une table. Le but principal d'un index est d'éviter de parcourir une table séquentiellement du premier enregistrement jusqu'à celui visé (problème rencontré si c'est le Français nommé « Zidane » qu'on recherche dans une table non indexée de plus de soixante millions d'enregistrements…). Le principe d'un index est l'association de l'adresse de chaque enregistrement (`ROWID`) avec la valeur des colonnes indexées.

Sans index et pour n enregistrements le nombre moyen d'accès nécessaire pour trouver un élément est égal à $n/2$. Avec un index, ce nombre tendra vers $\log(n)$ et augmentera donc bien plus faiblement en fonction de la montée en charge des enregistrements.

La figure suivante illustre un index sous la forme d'un arbre. Cet index est basé sur la colonne `nom` de la table `Pilote`. Cette figure est caricaturale, car un index n'est pas un arbre binaire (plus de deux liens peuvent partir d'un nœud). Dans cet exemple, trois accès à l'index seront nécessaires pour adresser directement un pilote via son nom au lieu d'en analyser huit au plus.

Figure 1-3 *Index sur la colonne nom*



ROWID	brevet	nom	nbHVol	compa
A	PL-1	Amélie Sulpice	450	AF
B	PL-2	Thomas Sulpice	900	AF
C	PL-3	Paul Soutou	1000	SING
D	PL-4	Aurélia Ente	850	ALIB
E	PL-5	Agnès Bidal	500	SING
F	PL-6	Sylvie Payrissat	2500	SING
G	PL-7	Thierry Guibert	600	ALIB
H	PL-8	Cathy Castaings	400	AF

Un index est associé à une table et peut être défini sur une ou plusieurs colonnes (dites « indexées »). Une table peut « héberger » plusieurs index. Ils sont mis à jour automatiquement après rafraîchissement de la table (ajouts et suppressions d'enregistrements ou modification des colonnes indexées). Un index peut être déclaré unique si on sait que les valeurs des colonnes indexées seront toujours uniques.

Classification

Plusieurs types d'index sont proposés par Oracle :

- l'arbre équilibré (*B-tree*), le plus connu, qui peut être défini sur trente-deux colonnes ;
- inverse (*reverse key*) qui concerne les tables « clusterisées » ;
- chaîne de bits (*bitmap*) qui regroupe chaque valeur de la (ou des) colonne(s) indexée(s) sous la forme d'une chaîne de bits. Ce type d'index peut être défini sur trente colonnes. Option disponible seulement avec la version *Enterprise Edition* ;
- basés sur des calculs entre colonnes (*function-based indexes*).

Index B-tree

La particularité de ce type d'index est qu'il conserve en permanence une arborescence symétrique (balancée). Toutes les feuilles sont à la même profondeur. Le temps de recherche est ainsi à peu près constant quel que soit l'enregistrement cherché. Le plus bas niveau de l'index (*leaf blocks*) contient les valeurs des colonnes indexées et le *rowid*. Toutes les feuilles de l'index sont chaînées entre elles. Pour les index non uniques (par exemple si on voulait définir

© Éditions Eyrolles

un index sur la colonne `compa` de la table `Pilote`) le *rowid* est inclus dans la valeur de la colonne indexée. Ces index, premiers apparus, sont désormais très fiables et performants, ils ne se dégradent pas lors de la montée en charge de la table.

Index bitmap

Alors qu'un index *B-tree*, permet de stocker une liste de *rowids* pour chaque valeur de la colonne indexée, un *bitmap* ne stocke qu'une chaîne de bits. Chacun d'eux correspond à une possible valeur de la colonne indexée. Si le bit est positionné à 1, pour une valeur donnée de l'index, cela signifie que la ligne courante contient la valeur. Une fonction de transformation convertit la position du bit en un *rowid*. Si le nombre de valeurs de la colonne indexée est faible, l'index *bitmap* sera très peu gourmand en occupation de l'espace disque.

Cette technique d'indexage est intéressante dans les applications décisionnelles (*On Line Analytical Processing*) qui manipulent de grandes quantités de données mais ne mettent pas en jeu un grand nombre de transactions. Pour les applications transactionnelles (*On Line Transaction Processing*), les index *B-tree* conviennent mieux.

La figure suivante présente un index *bitmap* basé sur la colonne `compa`. Chaque ligne est associée à une chaîne de bits de taille variable (égale au nombre de valeurs de la colonne indexée, ici trois compagnies sont recensées dans la table `Pilote`).

Figure 1-4 *Index bitmap sur la colonne compa*

Index *bitmap* sur `compa`

ROWID	AF	SING	ALIB
A	1	0	0
B	1	0	0
C	0	1	0
D	0	0	1
E	0	1	0
F	0	1	0
G	0	0	1
H	1	0	0

`Pilote`

ROWID	brevet	nom	nbHVol	compa
A	PL-1	Amélie Sulpice	450	AF
B	PL-2	Thomas Sulpice	900	AF
C	PL-3	Paul Soutou	1000	SING
D	PL-4	Aurélia Ente	850	ALIB
E	PL-5	Agnès Bidal	500	SING
F	PL-6	Sylvie Payrissat	2500	SING
G	PL-7	Thierry Guibert	600	ALIB
H	PL-8	Cathy Castaings	400	AF

Les index *bitmaps* sont très bien adaptés à la recherche d'informations basée sur des critères d'égalité (exemple : `compa = 'AF'`), mais ne conviennent pas du tout à des critères de comparaison (exemple : `nbHVol > 657`).

Index basés sur des fonctions

Une fonction de calcul (expressions arithmétiques ou fonctions SQL, PL/SQL ou C) peut définir un index. Celui-ci est dit « basé sur une fonction » (*function based index*).

Dans le cas des fonctions SQL (étudiées au chapitre 4), il ne doit pas s'agir de fonctions de regroupement (`SUM`, `COUNT`, `MAX`, etc.). Ces index servent à accélérer les requêtes contenant un calcul pénalisant s'il est effectué sur de gros volumes de données.

 © Éditions Eyrolles

Dans l'exemple suivant, on accède beaucoup aux comptes bancaires sur la base du calcul bien connu de ceux qui sont souvent en rouge : (`credit-debit`)*(1+(`txInt`/100))-`agios`.

Figure 1-5 *Index basé sur une fonction*

ROWID	Index fonction (credit-debit)*(1+(txInt/100))-agios)
C	-7,29
D	228,67
A	450,14
B	574,1

CompteEpargne

ROWID	ncompte	titulaire	debit	credit	txInt	agios
A	C1	Guibert	560	1000	3.6	5.7
B	C2	Soutou	250	850	4.1	50.5
C	C3	Teste	40	45	4.2	12.5
D	C4	Albaric	670	900	3.9	10.3

Un index basé sur une fonction peut être de type *B-tree* ou *bitmap*.

Il n'est pas possible de définir un tel index sur une colonne `LOB`, `REF`, ou collection (*nested table* et *varray*). Un index *bitmap* ne peut pas être unique.

Création d'un index (CREATE INDEX)

Pour pouvoir créer un index dans son schéma, la table à indexer doit appartenir au schéma. Si l'utilisateur a le privilège `INDEX` sur une table d'un autre schéma, il peut en créer un dans un autre schéma. Si l'utilisateur a le privilège `CREATE ANY INDEX`, il peut en constituer un dans tout schéma.

Un index est créé par l'instruction `CREATE INDEX`, modifié par la commande `ALTER INDEX` et supprimé par `DROP INDEX`.

En ce qui concerne les index basés sur des fonctions, l'utilisateur doit avoir le privilège `QUERY REWRITE`. La syntaxe de création d'un index est la suivante :

```
CREATE INDEX
   { UNIQUE | BITMAP } [schéma.]nomIndex
   ON [schéma.]nomTable ( {colonne1 | expressionColonne1 } [ASC |
   DESC ] …) ;
```

- `UNIQUE` permet de créer un index qui ne supporte pas les doublons.
- `BITMAP` fabrique un index « chaîne de bits ».
- `ASC` et `DESC` précisent l'ordre (croissant ou décroissant).

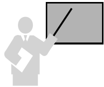
Créons plusieurs index sur la table des comptes bancaires. Le dernier (basé sur une fonction), doit faire apparaître les colonnes calculées dans ses paramètres après l'expression du calcul.

```
CREATE TABLE CompteEpargne
   (ncompte CHAR(4), titulaire VARCHAR(30), debit NUMBER(10,2),
    credit NUMBER(10,2), txInt NUMBER(2,1), agios NUMBER(5,2));
```

Tableau 1-9 Création d'index

Instruction SQL	Commentaires
`CREATE UNIQUE INDEX idx_titulaire_CompteEpargne ON CompteEpargne (titulaire DESC);`	Index *B-tree*, ordre inverse.
`CREATE INDEX idx_debitenFF_CompteEpargne ON CompteEpargne (debit*6.56);`	Index *B-tree*, expression d'une colonne.
`CREATE BITMAP INDEX idx_bitmap_txInt_CompteEpargne ON CompteEpargne (txInt);`	Index *bitmap*.
`CREATE INDEX idx_fct_Solde_CompteEpargne ON CompteEpargne ((credit-debit)*(1+(txInt/100))-agios, credit, debit, txInt, agios);`	Index basé sur une fonction.

Bilan



- Un index ralentit les rafraîchissements de la base (conséquence de la mise à jour de l'arbre ou des *bitmaps*). En revanche il accélère les accès ;
- Il est conseillé de créer des index sur des colonnes (majoritairement des clés étrangères) utilisées dans les clauses de jointures (voir chapitre 4) ;
- Les index *bitmaps* sont conseillés quand il y a peu de valeurs distinctes de la (ou des) colonne(s) à indexer. Dans le cas inverse, utilisez un index *B-tree*.
- Les index sont pénalisants lorsqu'ils sont définis sur des colonnes très souvent modifiées ou si la table contient peu de lignes.

Tables organisées en index

Une table organisée en index (*index-organized table*) peut être considérée comme la fusion d'une table et d'un index *B-tree*. Contrairement aux tables ordinaires (*heap-organized*) qui stockent des données sans ordre, toutes les valeurs d'une table organisée en index sont stockées au sein d'un index *B-tree*.

Apparu en version 8, ce type de tables est particulièrement utile pour les applications qui doivent extraire des informations basées essentiellement sur les clés primaires ou des éléments

 © Éditions Eyrolles

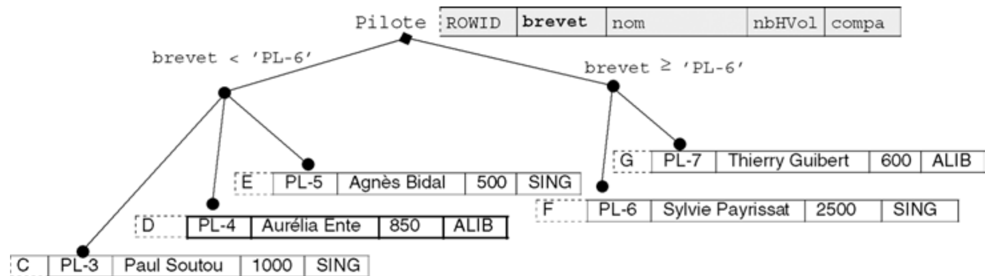
plus complexes (textes, images et sons). Le tableau suivant détaille les différences entre les deux types de tables.

Tableau 1-10 Caractéristiques des tables

Tables ordinaires	Tables organisées en index
La pseudo-colonne `ROWID` identifie chaque enregistrement. La clé primaire est optionnelle.	La clé primaire est obligatoire pour identifier chaque enregistrement.
Le `ROWID` physique permet de construire des index secondaires.	Le `ROWID` logique permet de construire des index secondaires.
Utilisation de clusters possible.	Utilisation interdite de clusters.
Peut contenir une colonne de type `LONG` et plusieurs colonnes de type `LOB`.	Peut contenir plusieurs colonnes `LOB` mais aucune de type `LONG`.

La figure suivante illustre la table `Pilote` organisée en index basé sur la clé primaire `brevet`.

Figure 1-6 *Table organisée en index*



La création d'une table organisée en index nécessite l'utilisation de la directive `ORGANIZATION INDEX` dans l'instruction `CREATE TABLE`. La clé primaire doit être obligatoirement déclarée. Des paramètres d'optimisation (`OVERFLOW` et `PCTTHRESHOLD`) peuvent également être mis en œuvre.

Dans notre exemple la syntaxe à utiliser est la suivante :

```
CREATE TABLE Pilote
   (brevet CHAR(6), nom CHAR(15), nbHVol NUMBER(7,2), compa CHAR(4),
   CONSTRAINT pk_Pilote PRIMARY KEY(brevet))
   ORGANIZATION INDEX ;
```

Les autres options de la directive `ORGANIZATION` sont :

- `HEAP` qui indique que les données ne sont pas stockées dans un ordre particulier (option par défaut) ;

- EXTERNAL qui précise que la table est en lecture seule et est située à l'extérieur de la base (sous la forme d'un fichier ASCII par exemple).

Destruction d'un schéma

- Il vous sera utile d'écrire un script de destruction d'un schéma (ici j'entends « schéma » comme ensemble de tables, contraintes et index composant une base de données et non pas en tant qu'ensemble de tous les objets d'un utilisateur) pour pouvoir recréer une base propre. Bien entendu si des données sont déjà présentes dans les tables, et que vous souhaitez les garder, il faudra utiliser une stratégie pour les réimporter dans les nouvelles tables. À ce niveau de l'ouvrage, vous n'en êtes pas là et le script de destruction va vous permettre de corriger vos erreurs de syntaxe du script de création des tables.
- Nous avons vu qu'il fallait créer d'abord les tables « pères » puis les tables « fils » (si des contraintes sont définies en même temps que les tables). L'ordre de destruction des tables, pour des raisons de cohérence, est inverse (il faut détruire les tables « fils » puis les tables « pères »). Dans notre exemple, il serait malvenu de supprimer la table Compagnie avant la table Pilote. En effet la clé étrangère compa n'aurait plus de sens.

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.

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

- 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 *Intégrité référentielle* du prochain chapitre).

Les éléments qui utilisaient la table (vues, synonymes, fonctions ou procédures) ne sont pas supprimés mais sont temporairement inopérants. Attention, une suppression ne peut pas être par la suite annulée.

Il suffit de relire à l'envers le script de création de vos tables pour en déduire l'ordre de suppression à écrire dans le script de destruction de votre schéma.

Attention à l'utilisation de CASCADE CONSTRAINTS (effets de bord).

Le tableau suivant présente deux écritures possibles pour détruire des schémas.

© Éditions Eyrolles

Tableau 1-11 Scripts équivalents de destruction

Avec `CASCADE CONSTRAINTS`	Les « fils » puis les « pères »

`--schéma Compagnie`
`DROP TABLE Compagnie CASCADE CONS-`
`TRAINTS;`
`DROP TABLE Pilote;`
	`--schéma Compagnie`
	`DROP TABLE Pilote;`
	`DROP TABLE Compagnie;`

`--schéma Banque`
`DROP INDEX idx_fct_Solde_CompteEpargne;`
`DROP INDEX idx_bitmap_txInt_CompteEpargne;`
`DROP INDEX idx_debitenFF_CompteEpargne;`
`DROP INDEX idx_titulaire_CompteEpargne;`
`DROP TABLE CompteEpargne;  /* Aurait aussi supprimé les index */`	

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*) qui s'appliquera à une base 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 1-12 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)

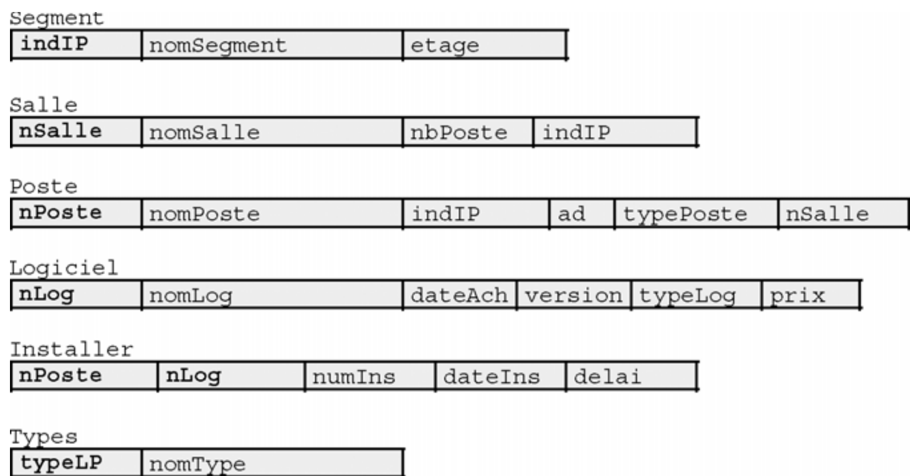
© Éditions Eyrolles

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 1-7 *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`). Comparer avec le schéma.

Exercice 1.4 Destruction des tables

Écrivez puis exécutez le script SQL de destruction des tables (que vous appellerez `dropParc.sql`). Lancer 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 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 1-13 Caractéristiques des colonnes à ajouter

Colonne	Commentaires	Types
kilometres	kilométrage d'un véhicule lors d'une sortie	NUMBER
n_conducteur	numéro de l'empoyé conducteur	VARCHAR2(4)
n_transporte	numéro de l'empoyé 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 ...);
```


© Éditions Eyrolles

Chapitre 2

Manipulation des données

Ce chapitre décrit l'aspect LMD (langage de manipulation des données) de SQL. Il existe une autre possibilité, que nous ne détaillerons pas, pour insérer des données dans les tables en utilisant des outils d'importation (*Oracle Enterprise Manager, Load Wizard* ou *SQL*Loader*). Nous verrons que SQL propose trois instructions 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 toutes les colonnes

Ajoutons trois lignes dans la table `Compagnie` en alimentant toutes les colonnes de la table par des valeurs. La deuxième insertion utilise le mot-clé `DEFAULT` pour affecter explicitement la valeur par défaut à la colonne `ville`. La troisième insertion attribue explicitement la valeur `NULL` à la colonne `nrue`.

Tableau 2-1 Insertions de toutes les colonnes

Instruction SQL	Commentaires
`INSERT INTO Compagnie VALUES ('SING', 7, 'Camparols', 'Singapour', 'Singapore AL');`	Toutes les valeurs sont renseignées dans l'ordre de la structure de la table.
`INSERT INTO Compagnie VALUES ('AC', 124, 'Port Royal', DEFAULT, 'Air France');`	`DEFAULT` explicite.
`INSERT INTO Compagnie VALUES ('AN1', NULL, 'Hoche', 'Blagnac', 'Air Nul1');`	`NULL` explicite.

Renseigner certaines colonnes

Insérons deux lignes dans la table `Compagnie` en ne précisant pas toutes les colonnes. La première insertion affecte implicitement la valeur par défaut à la colonne `ville`. La deuxième donne implicitement la valeur `NULL` à la colonne `nrue`.

Tableau 2-2 Insertions de certaines colonnes

Instruction SQL	Commentaires
`INSERT INTO Compagnie(comp, nrue, rue, nomComp) VALUES ('AF', 8, 'Champs Elysées', 'Castanet Air');`	`DEFAULT` implicite.
`INSERT INTO Compagnie(comp, rue, ville, nomComp) VALUES ('AN2', 'Foch', 'Blagnac', 'Air Nul2');`	`NULL` sur `nrue` implicite.

La table `Compagnie` contient à présent les lignes suivantes :

Figure 2-1 *Table après les insertions*

Compagnie — Valeur NULL — Valeur par défaut

comp	nrue	rue	ville	nomComp
SING	7	Camparols	Singapour	Singapore AL
AF	124	Port Royal	**Paris**	Air France
AN1		Hoche	Blagnac	Air Nul1
AC	8	Champs Elysées	**Paris**	Castanet Air
AN2		Foch	Blagnac	Air Nul2

© Éditions Eyrolles

Ne pas respecter des contraintes

Insérons des enregistrements dans la table `Pilote` qui ne respectent pas des contraintes. Le tableau suivant décrit les messages renvoyés pour chaque erreur (le nom de la contrainte apparaît dans chaque message, les valeurs erronées sont notées en gras). La dernière erreur signifie que la clé étrangère référence une clé primaire absente (nous reviendrons sur ces problèmes dans la section « Intégrité référentielle »).

Tableau 2-3 Insertions et contraintes

Insertions vérifiant les contraintes	Insertions ne vérifiant pas les contraintes
INSERT INTO Pilote VALUES ('PL-1', 'Amélie Sulpice', 450, 'AF');	INSERT INTO Pilote VALUES (**'PL-1'**, 'Amélie Sulpice', 450, 'AF'); ORA-00001: violation de contrainte unique (SOUTOU.**PK_PILOTE**)
	INSERT INTO Pilote VALUES ('NomNul', **NULL**, 450, 'AF'); ORA-02290: violation de contraintes (SOUTOU.**NN_NOM**) de vérification
INSERT INTO Pilote VALUES ('PL-2', 'Thomas Sulpice', 900, 'AF');	INSERT INTO Pilote VALUES ('PbHvol','Trop volé!',20000.**01**,'AF'); ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOUTOU.**CK_NBHVOL**) de vérification
INSERT INTO Pilote VALUES ('PL-3', 'Paul Soutou', 1000, 'SING');	INSERT INTO Pilote VALUES ('Unique',**'Amélie Sulpice'**,450,'AF'); ORA-00001: violation de contrainte unique (SOUTOU.**UN_NOM**)
	INSERT INTO Pilote VALUES ('PL-2','Thomas Sulpice',900, **'TOTO'**); ORA-02291: violation de contrainte (SOUTOU.**FK_PIL_COMPA_COMP**) d'intégrité touche 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

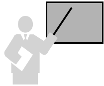
Déclarons la table `Pilote` qui contient deux colonnes de type `DATE`.

```
CREATE TABLE Pilote
   (brevet VARCHAR(6), nom VARCHAR(20), dateNaiss DATE,
   nbHVol NUMBER(7,2),
```

```
    dateEmbauche DATE, compa VARCHAR(4),
  CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

L'insertion du pilote initialise la date de naissance au 5 février 1965 (à zéro heure, zéro minute et zéro seconde), ainsi que la date d'embauche à la date (heures, minutes, secondes) du moment par la fonction SYSDATE.

```
INSERT INTO Pilote
  VALUES ('PL-10', 'Christian Soutou', '05-02-1965', 900, SYSDATE,
  'AF');
```



La fonction TO_DATE peut être aussi utilisée pour appliquer un format à la date. Cette même fonction permet aussi d'estimer un instant plus précis, exemple le 5 février 1965 à 6h30, 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.

Les écritures suivantes sont équivalentes (si le français est la langue choisie à l'installation) :

- '05-02-1965','05-02-65', '05/02/65'
- TO_DATE('Février 5, 1965','MONTH DD, YYYY')
- TO_DATE('5 Février 1965','DD MONTH YYYY')
- TO_DATE('5 02 1965','DD MM YYYY')

Types TIMESTAMP

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

```
CREATE TABLE Evénements
  (arrivé TIMESTAMP, arrivéLocalement TIMESTAMP WITH TIME ZONE);
```

L'insertion suivante initialise :

- la colonne arrivé 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 arrivéLocalement au 16 janvier 1965 à 12 heures, 30 minutes, 5 secondes et 98 centièmes dans le fuseau décalé vers l'est de 4h30 par rapport au méridien de Greenwich.

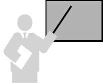
```
INSERT INTO Evénements
```

 © Éditions Eyrolles

```
  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 `Durées` contient la colonne `duréeAnnéesMois` (`INTERVAL YEAR TO MONTH`) pour stocker des intervalles en années et en jours, et la colonne `duréeJourSecondes` (`INTERVAL DAY TO SECOND`) pour stocker des intervalles en jours, heures, minutes, secondes et fractions de secondes.

```
CREATE TABLE Durées
  (duréeAnnéesMois   INTERVAL YEAR TO MONTH,
   duréeJourSecondes INTERVAL DAY TO SECOND);
```

L'insertion suivante initialise :

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

```
INSERT INTO Durées 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.

© Éditions Eyrolles

L'exemple suivant montre que le script a été exécuté le 23 avril 2003 à 19h33, 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
DBTIME                                     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,* % ` ' ( ) * + – , . / \ : ; < > = ! _ & ~ { } | ^ ? $ # @ " [ ], il faut travailler avec le jeu de caractères Unicode `NCHAR` et `NCHAR2` (la version à taille variable du précédent).

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.

La table `CaractèresUnicode` contient la colonne `unNCHARde1` de type `NCHAR(1)` pour stocker un caractère du jeu Unicode de l'alphabet courant.

```
CREATE TABLE CaractèresUnicode (unCHARde15 CHAR(15), unNCHARde1
NCHAR(1));
```

La première insertion initialise la colonne `nchar` à la valeur retournée par la fonction `UNISTR` qui convertit une chaîne en Unicode (ici : ¿). La deuxième insertion utilise le préfixe `N` car aucune transformation n'est nécessaire.

```
INSERT INTO CaractèresUnicode VALUES('Quid Espagnol', UNISTR('\
0345'));
INSERT INTO CaractèresUnicode VALUES('Quid Toulousain', N'?');
```

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).

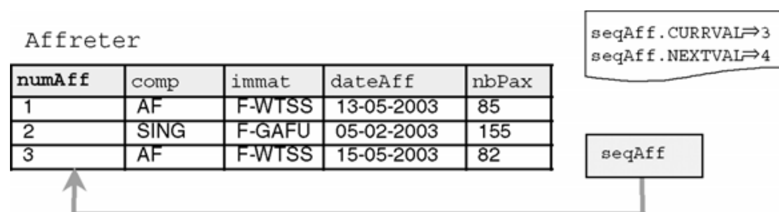
Séquences

Une séquence est un objet virtuel qui ne contient aucune donnée mais qui s'utilise pour générer automatiquement des valeurs (NUMBER). Elles sont utiles pour composer des clés primaires de tables quand vous ne disposez pas de colonnes adéquates à cet effet. Vous pouvez posséder plusieurs séquences dans votre schéma.

Les séquences sont gérées indépendamment des tables. Une séquence est en général affectée à une table mais vous pouvez l'utiliser pour plusieurs tables ou variables PL/SQL. Une séquence peut être partagée par plusieurs utilisateurs.

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 2-2 *Séquence appliquée à une clé primaire*



numAff	comp	immat	dateAff	nbPax
1	AF	F-WTSS	13-05-2003	85
2	SING	F-GAFU	05-02-2003	155
3	AF	F-WTSS	15-05-2003	82

© Éditions Eyrolles

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, 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.
- MAXVALUE: donne la valeur maximale de la séquence (ne pas dépasser 10^{29}-1). 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^{29}-1 pour une séquence ascendante et à -1 pour une séquence descendante.
- MINVALUE précise la valeur minimale de la séquence (ne pas dépasser la valeur -10^{27}-1). 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.

 © Éditions Eyrolles

- `NOCYCLE` (par défaut) indique que la séquence ne doit plus générer de valeurs une fois la limite atteinte.
- `CACHE` spécifie le nombre de valeurs de la séquence que le cache doit contenir (préallocation de mémoire). Le minimum est 2 et le maximum théorique est donné par la formule :

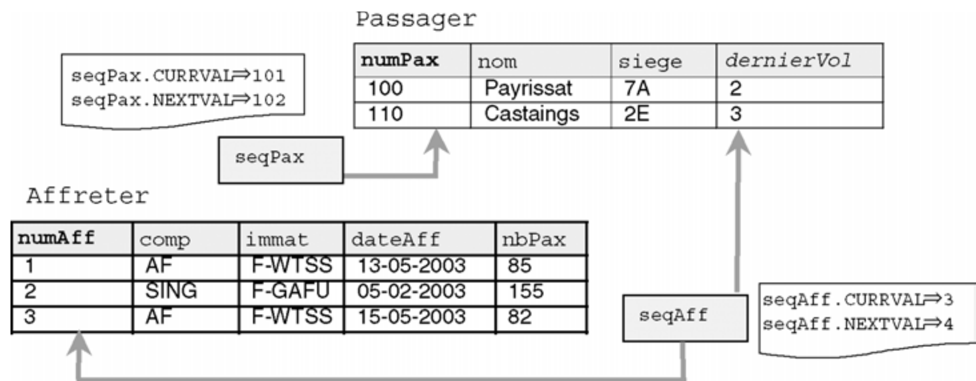
```
(CEIL (MAXVALUE - MINVALUE)) / ABS(INCREMENT)
```

- `NOCACHE` indique qu'aucune valeur de la séquence n'est dans le cache. Si les options `CACHE` et `NOCACHE` sont absentes de l'instruction, vingt valeurs de la séquence seront mises en cache.
- `ORDER` garantit que les valeurs de la séquence sont générées dans l'ordre des requêtes. Il faut utiliser cette option si la séquence est employée comme un point dans le temps (*timestamp*) dans des applications concurrentes (*Real Application Clusters*) ou dans celles qui fonctionnent en mode de verrous exclusifs. Cette option n'est pas importante pour les séquences clés primaires.
- `NOORDER` (défaut) ne prend pas en compte l'option précédente.

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 2-3 *Séquences*



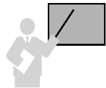
Le script SQL de définition des données est indiqué ci-après. Notez que les déclarations sont indépendantes, ce n'est qu'au moment des insertions qu'on affectera aux colonnes concernées les valeurs des séquences.

Tableau 2-4 Tables et séquences

Tables	Séquences
`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;`
`CREATE TABLE Passager (numPax NUMBER(6), nom CHAR(15), siege CHAR(4), dernierVol NUMBER(5), CONSTRAINT pk_Passager PRIMARY KEY(numPax), CONSTRAINT fk_Pax_vol_Affreter FOREIGN KEY(dernierVol) REFERENCES Affreter(numAff));`	**`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 2-5 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

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;
```

Modifications de colonnes

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 :

© Éditions Eyrolles

```
UPDATE [schéma.] nomTable
SET     colonne1 = expression | (requête_SELECT) | DEFAULT
        (colonne1, colonne2…) = (requête_SELECT)
[WHERE (coition)] ;
```

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 colonne

Modifions la compagnie de code 'AN1' en affectant la valeur 50 à la colonne `nrue`.

```
UPDATE Compagnie SET nrue = 50 WHERE comp = 'AN1';
```

Modification de plusieurs colonnes

Modifions la compagnie de code 'AN2' en affectant simultanément la valeur 14 à la colonne `nrue` et la valeur par défaut (`'Paris'`) à la colonne `ville`.

```
UPDATE Compagnie SET nrue = 14, ville = DEFAULT WHERE comp = 'AN2';
```

La table `Compagnie` contient à présent les lignes suivantes.

Figure 2-4 *Table après les modifications*

Compagnie — Modification 1

comp	nrue	rue	ville	nomComp
SING	7	Camparols	Singapour	Singapore AL
AF	124	Port Royal	Paris	Air France
AN1	**50**	Hoche	Blagnac	Air Nul1
AC	8	Champs Elysées	Paris	Castanet Air
AN2	**14**	Foch	**Paris**	Air Nul2

Modifications 2

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.

© Éditions Eyrolles

Tableau 2-7 Tables, données et contraintes

<table>
<tr><th>Table et données</th><th>Contraintes</th></tr>
<tr><td>Pilote

brevet	nom	nbHVol	compa
PL-1	Amélie Sulpice	450	AF
PL-2	Thomas Sulpice	900	AF
PL-3	Paul Soutou	1000	SING

Figure 2-5 Données</td><td>

```
CONSTRAINT pk_Pilote
 PRIMARY KEY(brevet),
CONSTRAINT nn_nom CHECK
 (nom IS NOT NULL),
CONSTRAINT ck_nbHVol CHECK
 (nbHVol BETWEEN 0 AND 20000),
CONSTRAINT un_nom UNIQUE (nom),
CONSTRAINT fk_Pil_compa_Comp
 FOREIGN KEY(compa)
 REFERENCES Compagnie(comp)
```

</td></tr>
</table>

À partir de la table `Pilote`, le tableau suivant décrit des modifications (certaines ne vérifient pas de contraintes). La mise à jour d'une clé étrangère est possible si elle n'est pas référencée par une clé primaire (voir la section « Intégrité référentielle »).

Tableau 2-8 Modifications

<table>
<tr><th>Vérifiant les contraintes</th><th>Ne vérifiant pas les contraintes</th></tr>
<tr><td>

```
--Modif d'une clé étrangère
UPDATE  Pilote
 SET   compa  = 'SING'
 WHERE brevet = 'PL-2';

--Modif d'une clé primaire
UPDATE  Pilote
 SET   brevet = 'PL3bis'
 WHERE brevet = 'PL-3';
```

Pilote

brevet	nom	nbHVol	compa
PL-1	Amélie Sulpice	450	AF
PL-2	Thomas Sulpice	900	**SING**
PL3bis	Paul Soutou	1000	SING

Figure 2-6 Après modifications</td><td>

```
UPDATE  Pilote SET brevet='PL-2'
 WHERE brevet='PL-1';
ORA-00001: violation de contrainte
unique (SOUTOU.PK_PILOTE)
UPDATE  Pilote SET nom = NULL
 WHERE brevet = 'PL-1';
ORA-02290: violation de contraintes
(SOUTOU.NN_NOM) de vérification
UPDATE  Pilote SET nbHVol = 20000.01
 WHERE brevet='PL-1';
ORA-02290: violation de contraintes
(SOUTOU.CK_NBHVOL) de vérification
UPDATE  Pilote SET nom = 'Paul Soutou'
 WHERE brevet = 'PL-1';
ORA-00001: violation de contrainte
unique (SOUTOU.UN_NOM)
UPDATE  Pilote SET compa = 'TOTO'
 WHERE brevet = 'PL-1';
ORA-02291: violation de contrainte
(SOUTOU.FK_PIL_COMPA_COMP) d'intégrité
-touche parent introuvable
```

</td></tr>
</table>

Dates et intervalles

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

© Éditions Eyrolles

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 :

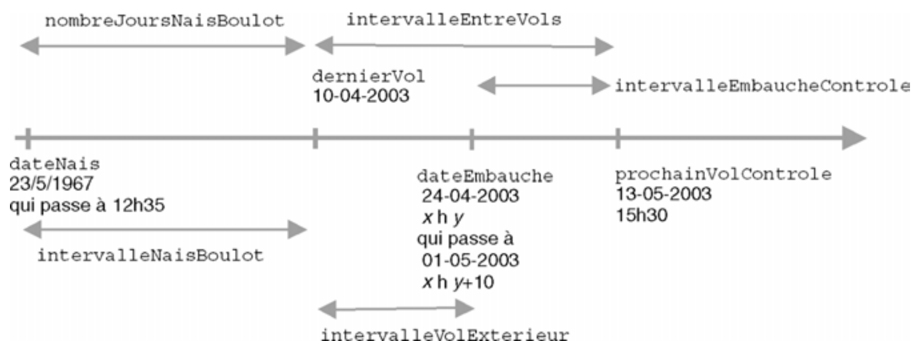
```
CREATE TABLE Pilote
  (brevet VARCHAR(6), nom VARCHAR(20), dateNaiss DATE, dernierVol DATE,
   dateEmbauche DATE, prochainVolControle DATE,
    nombreJoursNaisBoulot      NUMBER,
   intervalleNaisBoulot        INTERVAL DAY(7) TO SECOND(3),
   intervalleVolExterieur      INTERVAL DAY(2) TO SECOND(0),
   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, 15h30 (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 2-7 Intervalles à calculer



 © Éditions Eyrolles

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';
```

© Éditions Eyrolles

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 2-8 *Ligne modifiée par des calculs de dates*

Pilote

brevet	nom	dateNaiss	dernierVol	dateEmbauche	prochainVolControle
PL-1	Thierry Albaric	*25-03-1967* 25-03-1967 **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	Commentaire
`TO_CHAR(dateNaiss ,'J')`	`2439575`	Le calendrier julien est utilisé ici (comptage du nombre de jours depuis le 1er janvier, 4712 av. JC 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	Commentaire
`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	Commentaire
`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	Commentaire
`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 dans la table. Si aucune condition n'est précisée, toutes les lignes seront supprimées. Si la condition ne sélectionne aucune ligne, aucun enregistrement ne sera supprimé.

Détaillons les possibilités de cette instruction en considérant les différentes tables précédemment définies. La première commande détruit tous les pilotes de la compagnie de code 'AF', la seconde, avec une autre écriture, détruit la compagnie de code 'AF'.

```
DELETE FROM Pilote WHERE compa = 'AF';
DELETE Compagnie WHERE comp = 'AF';
```

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 (SOUTOU.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.

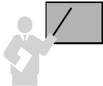
 © Éditions Eyrolles

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).

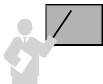


La contrainte référentielle concerne toujours deux tables – une table « père » aussi dite « maître » (*parent/referenced*) et une table « fils » (*child/dependent*) – possédant une ou plusieurs colonnes en commun. Pour la table « père », ces colonnes composent la clé primaire (ou candidate avec un index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

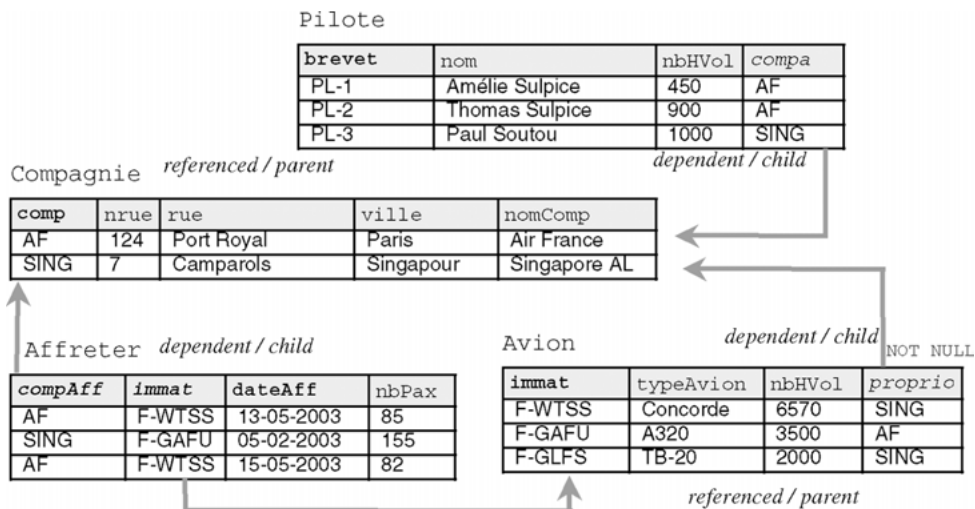


Deux types de problèmes sont automatiquement résolus par Oracle pour assurer l'intégrité référentielle :

- 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.

© Éditions Eyrolles

Figure 2-9 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 CHAR(4), nrue NUMBER(3),`
`rue CHAR(20), ville CHAR(15),`
`nomComp CHAR(15),`
`CONSTRAINT pk_Compagnie`
`PRIMARY KEY(comp));`
	`CREATE TABLE Compagnie`
	`(comp CHAR(4), nrue NUMBER(3),`
	`rue CHAR(20), ville CHAR(15),`
	`nomComp CHAR(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

© Éditions Eyrolles

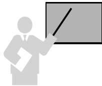
contrainte après la définition de la colonne. Cette écriture est préférable car elle convient aussi aux clés composées de plusieurs colonnes de par sa lisibilité.

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

Colonne et contrainte	Contrainte et colonne

CREATE TABLE Pilote
 (brevet CHAR(6) **CONSTRAINT**
  pk_Pilote **PRIMARY KEY**,
  nom CHAR(15), nbHVol NUMBER(7,2),
  compa CHAR(4) **CONSTRAINT** fk_Pil_compa_Comp
  REFERENCES Compagnie(comp));
	CREATE TABLE Pilote
	 (brevet CHAR(6), nom CHAR(15),
	  nbHVol NUMBER(7,2), compa CHAR(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 convient à notre exemple (la syntaxe de création des deux premières tables a été discutée plus haut) et étudions ensuite les mécanismes programmés par ces contraintes.

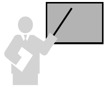
```
CREATE TABLE Compagnie …

CREATE TABLE Pilote …

CREATE TABLE Avion
   (immat CHAR(6), typeAvion CHAR(15), nbhVol NUMBER(10,2),
   proprio CHAR(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 CHAR(4), immat CHAR(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.

© Éditions Eyrolles

- 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.

© Éditions Eyrolles

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.

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.

Tableau 2-19 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 1.2×10^{-38}, plus petite valeur négative -1.2×10^{-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 2.3×10^{-308}, plus petite valeur négative -2.3×10^{-308}.

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.

© Éditions Eyrolles

Valeurs spéciales

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 8 octets 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
```


© Éditions Eyrolles

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.

© Éditions Eyrolles

- 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
```


© Éditions Eyrolles

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 2-20 Données des tables

Table	Données
Segment	``` INDIP NOMSEGMENT ETAGE ----------- -------------------- ---------- 130.120.80 Brin RDC 130.120.81 Brin 1er étage 130.120.82 Brin 2ème étage ```

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

```
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
```

```
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 2-20 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 2-21 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```

© Éditions Eyrolles

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 Naviguant;`	`ALTER TABLE Pilote RENAME TO Naviguant;`

Modifications structurelles (ALTER TABLE)

Considérons la table suivante que nous allons faire évoluer :

Figure 3-1 *Table à modifier*

```
CREATE TABLE Pilote
 (brevet VARCHAR2(4), nom VARCHAR2(20));

INSERT INTO Pilote
 VALUES ('PL-1', 'Agnès Labat');
```

Pilote

brevet	nom
PL-1	Agnès Labat

Ajout de 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	nom	nbHVol	compa	ville
PL-1	Agnès Labat		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.

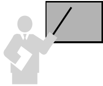
© Éditions Eyrolles

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, nom) VALUES ('PL-2', 'Laurent Boutrand');`	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`.

Si vous désirez diminuer la taille de colonnes non vides `CHAR`, il faut positionner le paramètre `BLANK_TRIMMING` à `TRUE`.

La table est désormais la suivante :

Figure 3-3 Après modification des colonnes

Pilote

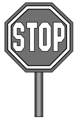
brevet	nom	nbHVol	compa	adresse
PL-1	Agnès Labat		AF	Paris
PL-2	Laurent Boutrand		SING	Paris

© Éditions Eyrolles

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;
```


© Éditions Eyrolles

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 3-4 *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)`

© Éditions Eyrolles

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));
```

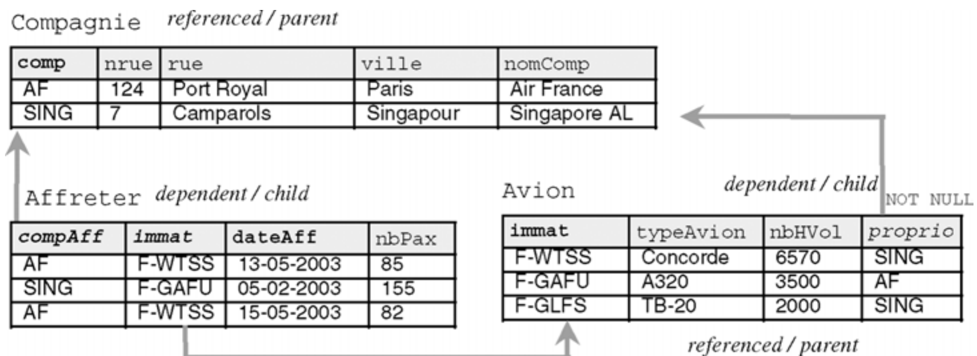
Clé primaire

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 3-5 *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];
```


© Éditions Eyrolles

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)

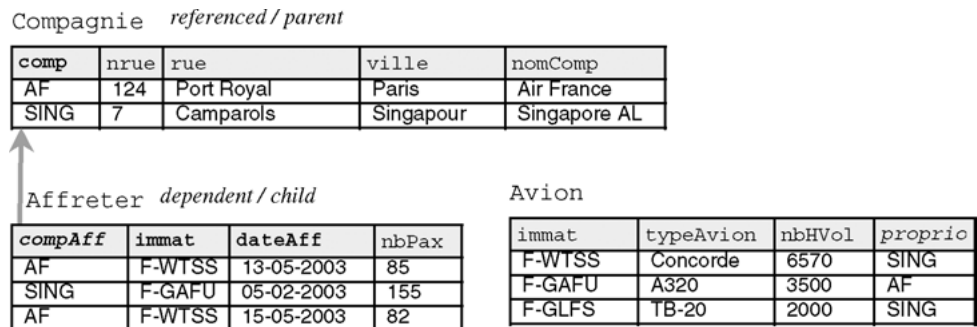
Supprimons la clé primaire de la table `Avion`. Il faut préciser `CASCADE`, car cette table est référencée par une clé étrangère dans la table `Affreter`. Cette commande supprime à la fois la clé primaire de la table `Avion` mais aussi les contraintes clés étrangères des tables dépendantes (ici seule 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 3-6 *Après suppression de contraintes*



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

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

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

© Éditions Eyrolles

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

Tableau 3-3 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

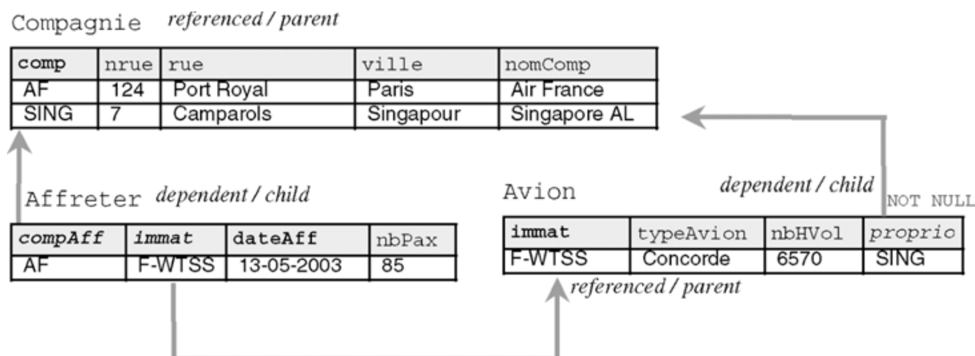
La syntaxe générale est la suivante :

```
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 3-7 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 :

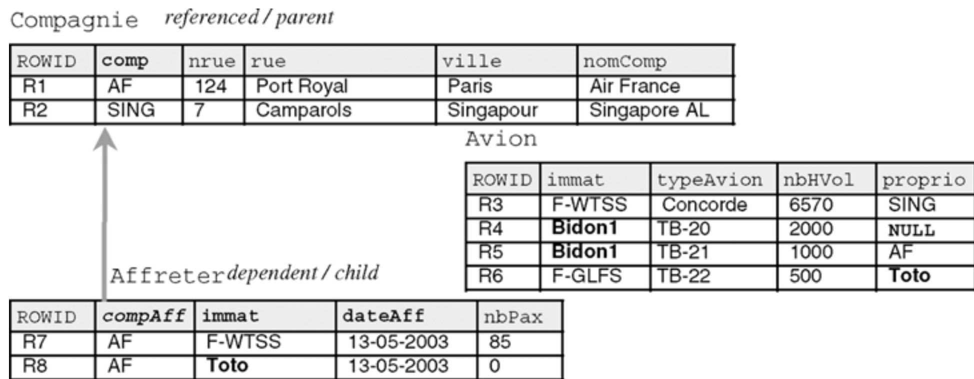
```
INSERT INTO Affreter VALUES ('AF', 'Toto', '13-05-2003', 0);
```

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.

© Éditions Eyrolles

Figure 3-8 *Après désactivation de contraintes*



Compagnie *referenced / parent*

ROWID	comp	nrue	rue	ville	nomComp
R1	AF	124	Port Royal	Paris	Air France
R2	SING	7	Camparols	Singapour	Singapore AL

Avion

ROWID	immat	typeAvion	nbHVol	proprio
R3	F-WTSS	Concorde	6570	SING
R4	**Bidon1**	TB-20	2000	**NULL**
R5	**Bidon1**	TB-21	1000	AF
R6	F-GLFS	TB-22	500	**Toto**

Affreter *dependent / child*

ROWID	compAff	immat	dateAff	nbPax
R7	AF	F-WTSS	13-05-2003	85
R8	AF	**Toto**	13-05-2003	0

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;
```


© Éditions Eyrolles

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 :

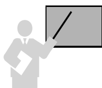
- La première, de type `ROWID`, contiendra les adresses des enregistrements ne respectant pas la contrainte ;
- la deuxième colonne de type VARCHAR2(30) contiendra le nom du propriétaire de la table ;
- la troisième colonne de type VARCHAR2(30) contiendra le nom de la table ;
- la quatrième, de type 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 3-4 Tables de 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 3-9 *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* R5 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';
```

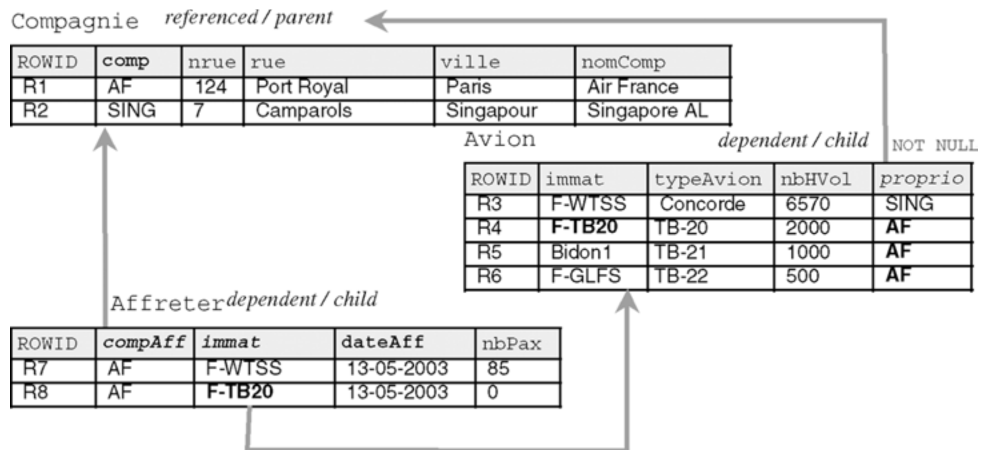
Avant de réactiver à nouveau les contraintes, il convient de supprimer les lignes de la table d'exceptions (ici `Problemes`). La réactivation de toutes les contraintes avec l'option `EXCEPTIONS INTO` ne génère plus aucune erreur et la table d'exceptions est encore vide.

 © Éditions Eyrolles

```
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 3-10 *Tables après modification et réactivation des contraintes*



Compagnie

ROWID	comp	nrue	rue	ville	nomComp
R1	AF	124	Port Royal	Paris	Air France
R2	SING	7	Camparols	Singapour	Singapore AL

Avion

ROWID	immat	typeAvion	nbHVol	proprio
R3	F-WTSS	Concorde	6570	SING
R4	**F-TB20**	TB-20	2000	**AF**
R5	Bidon1	TB-21	1000	**AF**
R6	F-GLFS	TB-22	500	**AF**

Affreter

ROWID	compAff	immat	dateAff	nbPax
R7	AF	F-WTSS	13-05-2003	85
R8	AF	**F-TB20**	13-05-2003	0

Contraintes différées

Une contrainte est dite « différée » (*deferred*) si elle déclenche sa vérification dès la première instruction `commit` rencontrée. 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`.

© Éditions Eyrolles

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 3-5 Contrainte DEFFERABLE

<table>
<tr><th>Mode différé</th><th>Mode immédiat</th></tr>
<tr><td colspan="2"><pre>CREATE TABLE Compagnie
 (comp CHAR(4), nrue NUMBER(3), rue CHAR(20), ville CHAR(15), nomComp CHAR(15),
 CONSTRAINT pk_Compagnie PRIMARY KEY(comp) NOT DEFERRABLE INITIALLY IMMEDIATE);</pre></td></tr>
<tr><td><pre>CREATE TABLE Avion
 (immat CHAR(6), typeAvion CHAR(15),
  nbhVol NUMBER(10,2), proprio CHAR(4),
  CONSTRAINT fk_Avion_comp_Compag
      FOREIGN KEY(proprio)
      REFERENCES Compagnie(comp)
      DEFERRABLE INITIALLY DEFERRED,
  CONSTRAINT pk_Avion
      PRIMARY KEY(immat));</pre></td><td><pre>CREATE TABLE Avion
 (immat CHAR(6), typeAvion CHAR(15),
  nbhVol NUMBER(10,2), proprio CHAR(4),
  CONSTRAINT fk_Avion_comp_Compag
      FOREIGN KEY(proprio)
      REFERENCES Compagnie(comp)
      DEFERRABLE INITIALLY IMMEDIATE,
  CONSTRAINT pk_Avion
      PRIMARY KEY(immat));</pre></td></tr>
<tr><td><pre>-- 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 (SOU-
TOU.FK_AVION_COMP_COMPAG) d'intégrité -
touche parent introuvable</pre></td><td><pre>-- fils sans père
INSERT INTO Avion VALUES
('F-WTSS', 'Concorde', 6570, 'SING');

ORA-02091: transaction annulée
ORA-02291: violation de contrainte (SOU-
TOU.FK_AVION_COMP_COMPAG) d'intégrité -
touche parent introuvable</pre></td></tr>
<tr><td></td><td><pre>-- 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</pre></td></tr>
</table>

 © Éditions Eyrolles

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 :

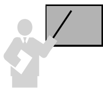
```
ALTER SESSION SET CONSTRAINTS = { IMMEDIATE | DEFERRED | DEFAULT }
```

- 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 ;

© Éditions Eyrolles

- 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 3-6 VALIDATE et NOVALIDATE

<table>
<tr><th>ENABLE NOVALIDATE</th><th>DISABLE VALIDATE</th></tr>
<tr><td colspan="2"><pre>CREATE TABLE Compagnie
 (comp CHAR(4), nrue NUMBER(3), rue CHAR(20), ville CHAR(15), nomComp CHAR(15),
 CONSTRAINT pk_Compagnie PRIMARY KEY(comp));</pre></td></tr>
<tr><td><pre>CREATE TABLE Avion
 (immat CHAR(6), typeAvion CHAR(15),
  proprio CHAR(4),
  CONSTRAINT fk_Avion_comp_Compag
    FOREIGN KEY(proprio)
    REFERENCES Compagnie(comp)DISABLE,
  CONSTRAINT pk_Avion
     PRIMARY KEY(immat));</pre></td><td><pre>CREATE TABLE Avion
 (immat CHAR(6), typeAvion CHAR(15),
  proprio CHAR(4),
  CONSTRAINT fk_Avion_comp_Compag
    FOREIGN KEY(proprio)
    REFERENCES Compagnie(comp),
  CONSTRAINT pk_Avion
     PRIMARY KEY(immat));</pre></td></tr>
<tr><td colspan="2"><pre>INSERT INTO Compagnie
 VALUES ('SING', 7, 'Camparols', 'Singapour', 'Singapore AL');</pre></td></tr>
<tr><td><pre>INSERT INTO Avion VALUES
 ('F-WTSS', 'Concorde', 'Toto');

ALTER TABLE Avion ENABLE NOVALIDATE
     CONSTRAINT fk_Avion_comp_Compag;</pre></td><td><pre>INSERT INTO Avion
   VALUES ('F-WTSS', 'Concorde',
'SING');

ALTER TABLE Avion DISABLE
    CONSTRAINT fk_Avion_comp_Compag;</pre></td></tr>
</table>

© Éditions Eyrolles

Tableau 3-6 VALIDATE et NOVALIDATE *(suite)*

<table>
<tr><th>ENABLE NOVALIDATE</th><th>DISABLE VALIDATE</th></tr>
<tr><td><pre>--interdit
INSERT INTO Avion VALUES
 ('F-TB20', 'Concorde', 'Toto');
ORA-02291: violation de contrainte (SOU-
TOU.FK_AVION_COMP_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</pre></td><td><pre>--permis
INSERT INTO Avion
   VALUES ('F-TB20', 'Concorde', 'Toto');

--interdit (dernier avion ne vérifie pas)
ALTER TABLE Avion DISABLE VALIDATE
     CONSTRAINT fk_Avion_comp_Compag;
ORA-02298: impossible de valider (SOU-
TOU.FK_AVION_COMP_COMPAG) - clés parents
introuvables

--Table Avion
Immat TypeAvion   Proprio
------ --------------- -------
F-WTSS Concorde   SING
F-TB20 Concorde   Toto</pre></td></tr>
</table>

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;
```

© Éditions Eyrolles

Nouveautés 11g

Deux nouvelles fonctionnalités sont intéressantes, il s'agit du concept de colonne virtuelle et de table en lecture seule.

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 CHAR(6), typeAvion CHAR(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
```


© Éditions Eyrolles

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 3-11 *Colonne virtuelles*

Avion

immat	typeAvion	nbHVol	age	nbPax	freqVolMois	HeurePax
F-WTSS	Concorde	20 000	18	90	92,5925...	1111,11
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`.

Table en lecture seule

Une table peut être déclarée en lecture seule temporairement ou définitivement. L'instruction qui permet cette fonctionnalité est la suivante :

```
ALTER TABLE nomTable { READ ONLY |  READ WRITE } ;
```

Déclarons la table `Avion` en lecture seule puis tentons d'insérer un enregistrement, il vient l'erreur « `ORA-12081 : opération de mise à jour interdite sur la table…` ». Après avoir annulé cette restriction, les insertions ou mises à jour sont permises.

```
ALTER TABLE Avion READ ONLY;
Table modifiée.

INSERT INTO Avion (immat,typeAvion,nbhVol,age,nbPax)
       VALUES ('F-NEW', 'A318', 90, 1, 140);
ERREUR à la ligne 1 : ORA-12081: opération de mise à jour interdite
sur la table "SOUTOU"."AVION"

ALTER TABLE Avion READ WRITE;
Table modifiée.

INSERT INTO Avion (immat,typeAvion,nbhVol,age,nbPax)
       VALUES ('F-NEW', 'A318', 90, 1, 140);
1 ligne créée.

SELECT immat,heurePax  FROM Avion;
IMMAT    HEUREPAX
------ ----------
F-WTSS    1111,11
F-GHTY        900
F-NEW          90
```


© Éditions Eyrolles

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 3-7 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 :

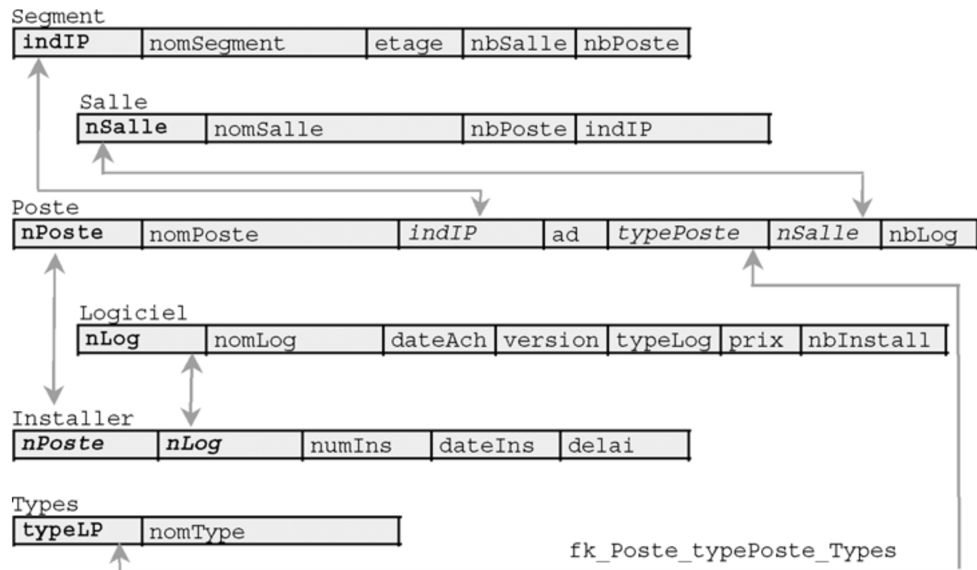
```
SELECT * FROM Salle;
SELECT * FROM Segment;
```

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`).

© Éditions Eyrolles

Figure 3-12 Contraintes référentielles à créer



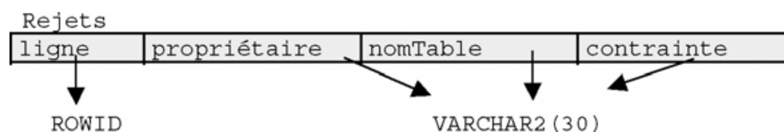
Si l'ajout d'une contrainte référentielle renvoie une erreur, vérifier les enregistrements des tables « pères » et « fils » (notamment au niveau de la casse des chaînes de caractères, 'Tx' est différent de 'TX' par exemple).

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)

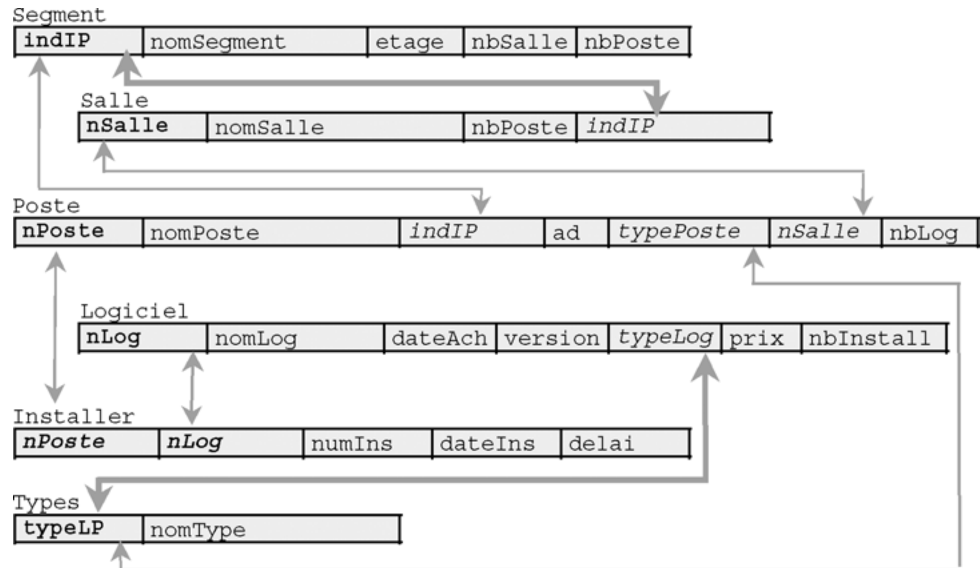


Cette table permettra de retrouver les enregistrements qui ne vérifient pas de contraintes lors de la réactivation.

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.

© Éditions Eyrolles

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`.

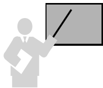
© Éditions Eyrolles

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.

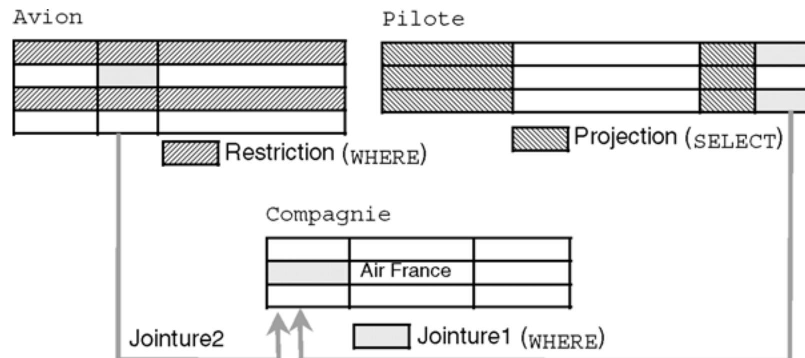
Généralités



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 :

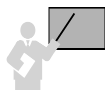
```
SELECT [ { DISTINCT | UNIQUE } | ALL ] listeColonnes
   FROM nomTable1 [,nomTable2]…
   [ WHERE condition ]
   [ clauseHiérarchique ]
   [ clauseRegroupement ]
   [ HAVING condition ]
   [ { UNION | UNION ALL | INTERSECT | MINUS } ( sousRequête )]
   [ clauseOrdonnancement ] ;
```

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).

© Éditions Eyrolles



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).

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 chaque option

Figure 4-2 *Table Pilote*

Pilote

brevet	nom	nbHVol	compa
PL-1	Gratien Viel	450	AF
PL-2	Didier Donsez	0	AF
PL-3	Richard Grin	1000	SING
PL-4	Placide Fresnais	2450	CAST
PL-5	Daniel Vielle		AF

Extraction de toutes les colonnes

L'extraction de toutes les colonnes d'une table requiert l'utilisation du symbole « * ».

Tableau 4-1 Utilisation de « * »

Requête SQL	Résultat sous SQL*Plus

`SELECT * FROM Pilote;`	<pre>BREVET NOM NBHVOL COMPA
	------ ---------------- ---------- -----
	PL-1 Gratien Viel 450 AF
	PL-2 Didier Donsez 0 AF
	PL-3 Richard Grin 1000 SING
	PL-4 Placide Fresnais 2450 CAST
	PL-5 Daniel Vielle AF</pre>

Extraction de certaines colonnes

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

Tableau 4-2 Liste de colonnes

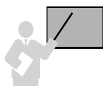
Requête SQL	Résultat sous SQL*Plus

`SELECT` **`compa`**`,` **`brevet`** `FROM Pilote;`	<pre>COMPA BREVET
	----- ------
	AF PL-1
	AF PL-2
	SING PL-3
	CAST PL-4
	AF PL-5</pre>

© Éditions Eyrolles

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 4-3 Alias (colonnes et tables)

Alias de colonnes	Alias de table

`SELECT compa AS c1,`
`  nom AS NometPrénom, brevet c3`
`  FROM Pilote;`
	`SELECT aliasPilotes.compa AS c1,`
	`       aliasPilotes.nom`
	`  FROM Pilote aliasPilotes;`

`C1   NOMETPRÉNOM      C3`
`---- ---------------- ------`
`AF   Gratien Viel     PL-1`
`AF   Didier Donsez    PL-2`
`SING Richard Grin     PL-3`
`CAST Placide Fresnais PL-4`
`AF   Daniel Vielle    PL-5`
	`C1   NOM`
	`---- ----------------`
	`AF   Gratien Viel`
	`AF   Didier Donsez`
	`SING Richard Grin`
	`CAST Placide Fresnais`
	`AF   Daniel Vielle`

Duplicatas

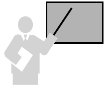
Les directives DISTINCT ou UNIQUE éliminent les éventuels duplicatas. Pour la deuxième requête, les écritures DISTINCT compa, UNIQUE(compa) et UNIQUE compa sont équivalentes. La notation entre parenthèses est nécessaire lorsque l'on désire éliminer des duplicatas par paires, triplets, etc.

Tableau 4-4 Gestion des duplicatas

Avec duplicata	Sans duplicata
`SELECT compa FROM Pilote;`	`SELECT DISTINCT(compa) FROM Pilote;`

`COMPA`
`-----`
`AF`
`AF`
`SING`
`CAST`
`AF`
	`COMPA`
	`-----`
	`AF`
	`CAST`
	`SING`

Expressions et valeurs nulles



Il est possible d'évaluer et d'afficher simultanément des expressions dans la clause SELECT (types NUMBER, DATE et INTERVAL).

Les opérateurs arithmétiques sont évalués par ordre de priorité (*, /, + et –).

Le résultat d'une expression comportant une valeur NULL est évalué à NULL.

Nous avons déjà étudié les opérations sur les dates et intervalles (chapitre 2). Dans l'exemple suivant l'expression 10*nbHVol+5/2 est calculée en multipliant par 10 le nombre d'heures de vol puis en ajoutant le résulat de 5 divisé par 2.

Tableau 4-5 Expressions numériques

Requête	Résultat sous SQL*Plus

`SELECT brevet, nbHVol,`
`  **nbHVol*nbHVol** AS auCarré,`
`  **10*nbHVol+5/2**`
`  FROM Pilote;`
	```BREVET NBHVOL **AUCARRÉ** **10*NBHVOL+5/2**```
	`------ ---------- ---------- -------------`
	`PL-1 450 202500 4502,5`
	`PL-2 0 0 2,5`
	`PL-3 1000 1000000 10002,5`
	`PL-4 2450 6002500 24502,5`
	`PL-5`

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.

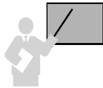
© Éditions Eyrolles

Tableau 4-6 Ordonnancement

Options par défaut	Option sur les valeurs NULL

`SELECT brevet, nom FROM Pilote`
`ORDER BY nom;`
 	
`BREVET NOM`
`------ ----------------`
`PL-5   Daniel Vielle`
`PL-2   Didier Donsez`
`PL-1   Gratien Viel`
`PL-4   Placide Fresnais`
`PL-3   Richard Grin`
	`SELECT brevet,nbHVol FROM Pilote`
	`ORDER BY nbHvol ASC NULLS FIRST;`
	
	`BREVET     NBHVOL`
	`------ ----------`
	`PL-5`
	`PL-2            0`
	`PL-1          450`
	`PL-3         1000`
	`PL-4         2450`

Concaténation



L'opérateur de concaténation s'écrit avec deux barres verticales (||). Il permet de concaténer des expressions (colonnes, calculs, fonctions ou constantes). La colonne résultante est considérée comme une chaîne de caractères.

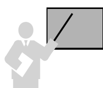
L'exemple suivant présente un alias dans l'en-tête de colonne ("Embauche") qui met en forme les résultats. La concaténation concerne deux colonnes et la constante « vole pour ».

Tableau 4-7 Concaténation

Requête	Résultat sous SQL*Plus

`SELECT brevet,`
`nom||' vole pour '||compa AS "Embauche"`
`FROM Pilote;`
	`BREVET Embauche`
	`------ ------------------------------`
	`PL-1   Gratien Viel vole pour AF`
	`PL-2   Didier Donsez vole pour AF`
	`PL-3   Richard Grin vole pour SING`
	`PL-4   Placide Fresnais vole pour CAST`
	`PL-5   Daniel Vielle vole pour AF`

Pseudo-colonne ROWID

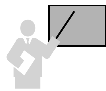


Le format du *rowid* de chaque enregistrement inclut le numéro de l'objet, le numéro relatif du fichier, le numéro du bloc dans le fichier et le déplacement dans le bloc. Le mot-clé qui désigne cette pseudo-colonne non modifiable (mais accessible) est ROWID.

Tableau 4-8 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 4-9 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

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;
```


© Éditions Eyrolles

Tableau 4-10 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;`

```
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
```

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
 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 :

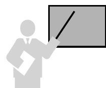
Figure 4-3 *Table Pilote*

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.



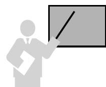
Les écritures « `prime=500` » et « `(prime=500)` » sont équivalentes. Les écritures « `prime<>500` » et « `NOT (prime=500)` » sont équivalentes. Les parenthèses sont utiles pour composer des conditions.

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

Tableau 4-11 Égalité, inégalité et comparaison

Égalité	Comparaison et inégalité
`SELECT brevet, nom AS "Prime 500"`	`SELECT brevet, nom, prime`
`       FROM Pilote WHERE prime = 500;`	`       FROM Pilote WHERE prime <= 400;`
`BREVET Prime 500`	
`------ ----------------`	`BREVET NOM                   PRIME`
`PL-1   Gratien Viel`	`------ ---------------- ----------`
`PL-4   Placide Fresnais`	`PL-3   Richard Grin             90`
	`PL-6   Francoise Tort            0`
`SELECT brevet, nom "de Air-France"`	
`       FROM Pilote WHERE compa = 'AF';`	`SELECT brevet, nom, prime FROM Pilote`
	`       WHERE prime <> 500;`
`BREVET de Air-France`	`BREVET NOM                   PRIME`
`------ ----------------`	`------ ---------------- ----------`
`PL-1   Gratien Viel`	`PL-3   Richard Grin             90`
`PL-2   Didier Donsez`	`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 (>, =, <, >=, <=, <>) 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.

© Éditions Eyrolles

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 4-12 Opérateurs logiques

Requête	Résultat sous SQL*Plus

`SELECT brevet, nom, compa`
` FROM Pilote WHERE`
`(compa = 'SING' OR`
` compa = 'AF' AND nbHVol < 500 );`
	`BREVET NOM              COMP`
	`------ ---------------- ----`
	`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              COMP`
	`------ ---------------- ----`
	`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 4-13 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              COMP`
	`------ ---------------- ----`
	`PL-3   Richard Grin     SING`
	`PL-4   Placide Fresnais CAST`
	`PL-5   Daniel Vielle    SING`
	`PL-6   Francoise Tort   CAST`

© Éditions Eyrolles

Tableau 4-13 Opérateurs intégrés *(suite)*

Opérateur	Exemple

`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              COMP`
	`------ ---------------- ----`
	`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              COMP`
	`------ ---------------- ----`
	`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 COMP`
	`-------------- --------- -------- ----`
	`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 »).

© Éditions Eyrolles

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 4-14 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

Tableau 4-14 Fonctions pour les caractères *(suite)*

<table>
<tr><th>Fonction</th><th>Objectif</th><th>Exemple</th></tr>
<tr><td>INSTR(c1, c2 [,p [,o]])</td><td>Premier indice d'une sous-chaîne c2 dans une chaîne c1.
Exemple : indice du 2e 'Air' après le 9e caractère.</td><td>SELECT INSTR('Infos-Air : AirBus pour Air-France','Air', 9, 2) "Indice"
FROM DUAL;

Indice

25</td></tr>
<tr><td>LOWER(c)</td><td>Tout en minuscules.</td><td>SELECT LOWER(prenom) || ' ' ||
LOWER(nom) "Etat civil"
FROM Pilote WHERE compa = 'SING';

Etat civil
----
richard grin
daniel vielle</td></tr>
<tr><td>LENGTH(c)</td><td>Longueur de la chaîne.</td><td>SELECT LENGTH('Infos-Air : AirBus pour Air-France') "Taille" FROM DUAL;

Taille

34</td></tr>
<tr><td>LPAD(c1,n,c2)</td><td>Insertion à gauche de c2 dans c1 sur n caractères.</td><td>SELECT LPAD('Rien',20,'-.-') "sur 20"
FROM DUAL;

sur 20

-.--.--.--.--.--Rien</td></tr>
<tr><td>LTRIM(c1,c2)</td><td>Enlève c2 à c1 en examinant la gauche de c1.</td><td>SELECT
LTRIM('B747B747A380 à Blagnac','B747')
"Bye les Jumbo" FROM DUAL;

Bye les Jumbo

A380 à Blagnac</td></tr>
<tr><td>REPLACE(c1,c2, c3)</td><td>Recherche les c2 présentes dans c1 et les remplace par c3.</td><td>SELECT
REPLACE('Matra et Aerospatiale',
'Matra','EADS') "Changement"
FROM DUAL;

Changement

EADS et Aerospatiale</td></tr>
<tr><td>RPAD(c1,n,c2)</td><td>Insertion à droite de c2 dans c1 sur n caractères.</td><td>SELECT RPAD('Rien',19,'-.-') "sur 19"
FROM DUAL;

sur 19

Rien-.--.--.--.--.-</td></tr>
</table>

© Éditions Eyrolles

Tableau 4-14 Fonctions pour les caractères *(suite)*

Fonction	Objectif	Exemple
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 COMP ---------- ---------- ---- donsez **smith** AF vielle **jones** SING
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'**, '012345678**9A**B**C**D**E**FGHIJK**L**MN**O**PQ**R**', 'ChaineVer**s**-C**o**dage-012**3**45**6**78**9**') "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

© Éditions Eyrolles

Numériques

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

Tableau 4-15 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*.	

© Éditions Eyrolles

Tableau 4-15 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.	

Dates

Le tableau suivant décrit les principales fonctions pour les dates.

Tableau 4-16 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 4-17 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`

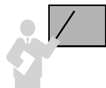
Arrondi du 28 octobre 2005 au niveau du mois.
`SELECT ROUND(TO_DATE('28-OCT-2005'),'MONTH') "Arrondi" FROM DUAL;`
	`Arrondi`
	`--------`
	`01/11/05`

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

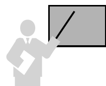


Il est possible d'affecter dans une expression ou dans une instruction SQL (`INSERT`, `UPDATE`…), une donnée de type `NUMBER` (ou `DATE`) à une donnée de type `VARCHAR2` (ou `CHAR`). Il en va de même pour l'affectation d'une colonne `VARCHAR2` par une donnée de type `DATE` (ou `NUMBER`). On parle ainsi 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`.

© Éditions Eyrolles

Les fonctions de conversion sont décrites dans le tableau suivant :

Tableau 4-18 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'.
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 Í 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.

© Éditions Eyrolles

Autres fonctions

D'autres fonctions n'appartenant pas à la classification précédente sont présentées dans le tableau suivant :

Tableau 4-19 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'.
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).

© Éditions Eyrolles

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 4-20 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 4-21 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`

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… ] ;
```


© Éditions Eyrolles

- 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 4-22 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`

© Éditions Eyrolles

Tableau 4-22 Exemple de fonctions de groupe avec GROUP BY *(suite)*

<table>
<tr><th>Fonction</th><th>Exemples</th></tr>
<tr><td>COUNT</td><td>Nombre de pilotes (et ceux qui ont de l'expérience du vol) par compagnie.
<pre>
SELECT compa, COUNT(*), COUNT(nbHVol) FROM Pilote
       GROUP BY(compa);

COMP   COUNT(*) COUNT(NBHVOL)
---- ---------- -------------
AF   3   3
CAST   1   0
SING   2   2
</pre></td></tr>
<tr><td>MAX</td><td>Nombre d'heures de vol le plus élevé, date d'embauche la plus récente pour chaque compagnie.
<pre>
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
</pre></td></tr>
<tr><td>STDEV – SUM (avec WHERE)</td><td>Écarts type des primes et sommes des heures de vol des pilotes volant sur 'A320' de chaque compagnie.
<pre>
SELECT compa, STDDEV(prime), SUM(nbHVol) FROM Pilote
      WHERE typeAvion = 'A320' GROUP BY(compa);

COMP STDDEV(PRIME) SUM(NBHVOL)
---- ------------- -----------
AF   0   450
SING   1000
</pre></td></tr>
<tr><td>Plusieurs colonnes dans le GROUP BY</td><td>Nombre de pilotes qualifiés par type d'appareil et par compagnie.
<pre>
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
</pre></td></tr>
<tr><td>GROUP BY et HAVING</td><td>Compagnies (et nombre de leurs pilotes) ayant plus d'un pilote.
<pre>
SELECT compa, COUNT(brevet) FROM Pilote
       GROUP BY(compa)
       HAVING COUNT(brevet)>=2;

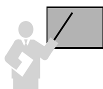
COMP COUNT(BREVET)
---- -------------
AF   3
SING   2
</pre></td></tr>
</table>

© Éditions Eyrolles

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 permis 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*

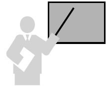
AviondeAF

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 4-23 Exemples avec INTERSECT

Besoin	Requête

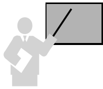
Quels sont les types d'avions que les deux compagnies exploitent en commun ?	`SELECT typeAvion FROM AvionsdeAF`
	`INTERSECT`
	`SELECT typeAv FROM AvionsdeSING;`
	
	`TYPEAVION`
	`----------`
	`A320`
	`A340`

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`

 © Éditions Eyrolles

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



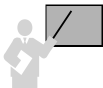
Les opérateurs UNION et UNION ALL sont commutatifs. L'opérateur UNION permet d'éviter les duplicatas (comme DISTINCT ou UNIQUE dans un SELECT). UNION ALL ne les élimine pas.

Tableau 4-24 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 4-25 Exemples avec l'opérateur MINUS

Web

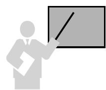
Besoin	Requête

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



Le résultat d'une requête contenant des opérateurs ensemblistes est trié par défaut par ordre croissant sauf avec l'opérateur `UNION ALL`.

La clause `ORDER BY` n'est utilisable qu'une fois en fin d'une requête incluant des opérateurs ensemblistes. Cette clause accepte le nom des colonnes de la première requête ou la position de ces colonnes.

Le tableau suivant présente trois écritures différentes de la même requête ensembliste contenant une clause `ORDER BY`. Le besoin est de connaître tous les types d'avions que les deux compagnies exploitent (classement par ordre décroissant).

Notez que la troisième requête produit le même résultat en faisant intervenir un `SELECT` dans le `FROM`. Ce mécanisme est autorisé par SQL2, il permet de construire dynamiquement la table à interroger.

Il faut affecter des alias aux expressions de la première requête pour pouvoir les utiliser dans le `ORDER BY` final.

© Éditions Eyrolles

Tableau 4-26 Exemples avec la clause ORDER BY

Technique	Requête

Nom de la colonne.	`SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING`
	`ORDER BY` `typeAvion DESC ;`

Position de la colonne.	… **`ORDER BY`** `1 DESC;`

`SELECT` dans le `FROM`.	`SELECT typeAvion` **`FROM`**
	`(SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING)`
	`ORDER BY` `typeAvion DESC ;`

```
TYPEAVION
----------
Concorde
A340
A330
```

Pour illustrer cette restriction, supposons que nous désirions faire la liste des avions avec 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. Ajoutons donc au `SELECT` de cette table, dans le tableau suivant, la valeur `0` pour rendre possible l'opérateur `UNION`.

Tableau 4-27 Alias pour ORDER BY

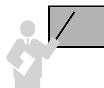
Requête	Résultat

`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 $y \in T2$.

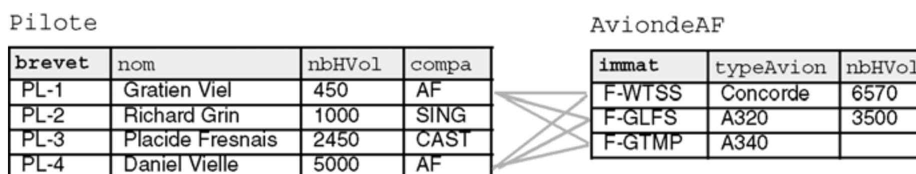
© Éditions Eyrolles



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* ∈ *T1* et *c2* ∈ *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 4-28 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`

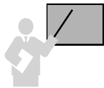
 © Éditions Eyrolles

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).

Jointures

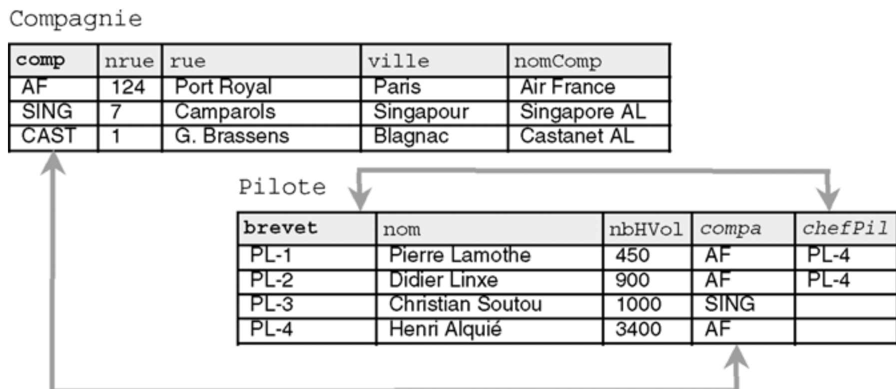
Les jointures permettent d'extraire des données issues de plusieurs tables. Le processus de normalisation du modèle relationnel est basé sur la décomposition et a pour conséquence d'augmenter le nombre de tables d'un schéma. Ainsi, la majorité des requêtes utilisent des jointures nécessaires pour pouvoir 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 4-9 *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	

© Éditions Eyrolles

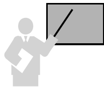
Classification

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 ];
```


© Éditions Eyrolles

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

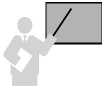
Bien que dans le vocabulaire courant, on ne parle que de « jointures » en fonction de la nature de l'opérateur utilisé dans la requête, de la clause de jointure et des tables concernées, on distingue :

- 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 4-29 Exemples de conditions

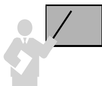
Type de jointure	Syntaxe de la condition
Équijointure	`WHERE comp = compa;`
Autojointure	`WHERE alias1.chefPil = alias2.brevet;`
Jointure externe	`WHERE comp= compa (+) ;`



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



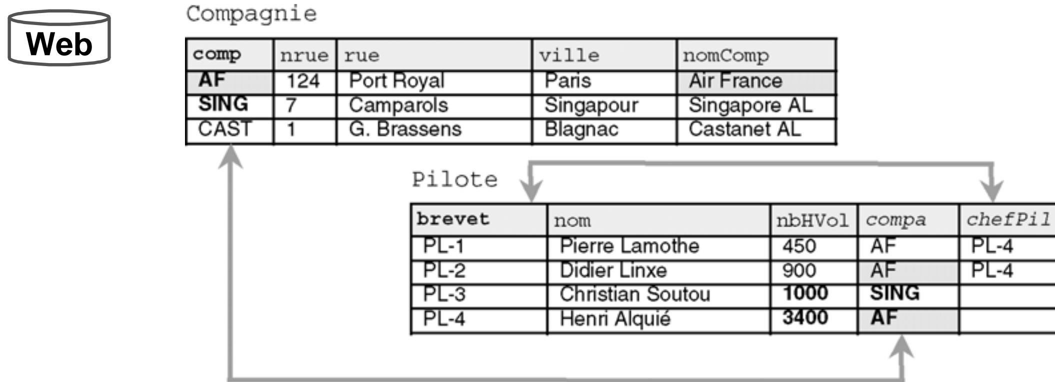
Une équijointure utilise l'opérateur d'égalité dans la clause de jointure et compare généralement des clés primaires avec des clés étrangères.

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.

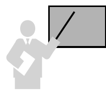
Figure 4-10 *Équijointures*



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

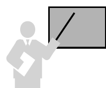
brevet	nom	nbHVol	*compa*	*chefPil*
PL-1	Pierre Lamothe	450	AF	PL-4
PL-2	Didier Linxe	900	AF	PL-4
PL-3	Christian Soutou	**1000**	**SING**	
PL-4	Henri Alquié	**3400**	**AF**	

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.

 © Éditions Eyrolles

Tableau 4-30 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 4-11 *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 4-31 Exemples d'autojointures

Requête	Jointure relationnelle	Jointure SQL2

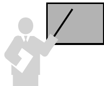
R3	`SELECT p1.brevet, p1.nom`
	`FROM Pilote p1, Pilote p2`
	`WHERE p1.chefPil = p2.brevet`
	`AND p2.nom LIKE '%Alquié%';`
		`SELECT p1.brevet, p1.nom`
		`FROM Pilote p1`
		`JOIN Pilote p2`
		`ON p1.chefPil = p2.brevet`
		`WHERE p2.nom LIKE '%Alquié%';`

	`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.

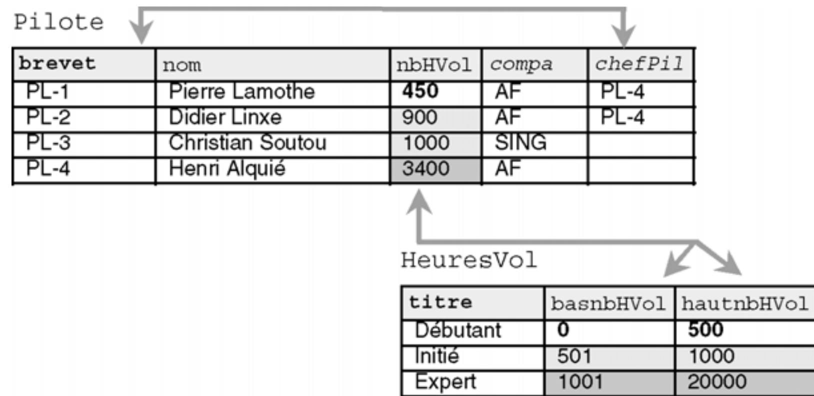
En considérant les tables suivantes, extrayons par exemple :

- 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.

 © Éditions Eyrolles

Figure 4-12 *Inéquijointures*



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	

HeuresVol

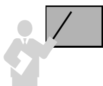
titre	basnbHVol	hautnbHVol
Débutant	0	500
Initié	501	1000
Expert	1001	20000

Le tableau suivant détaille ces requêtes, les clauses d'inéquijointures sont surlignées :

Tableau 4-32 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 s'ils ne satisfont pas aux conditions de jointure).

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).

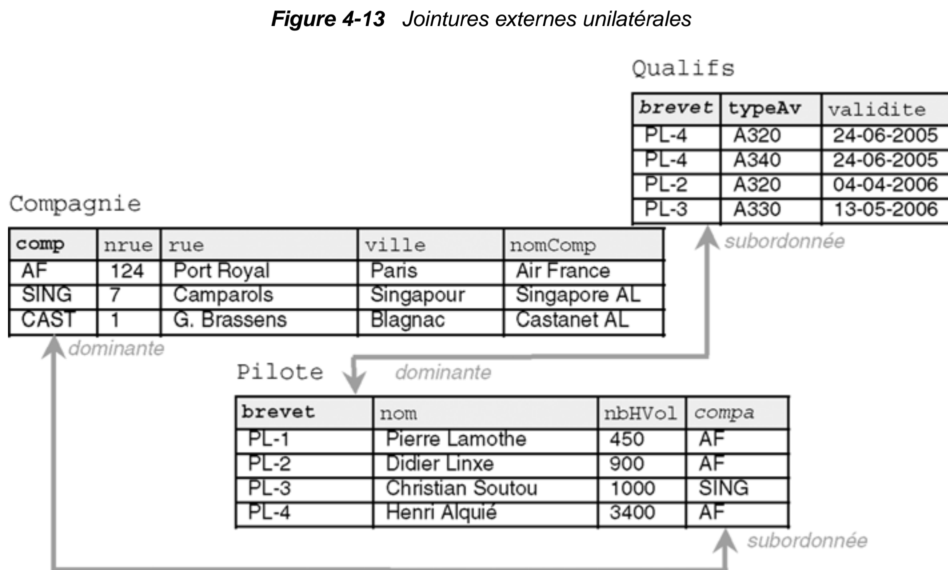
Jointures unilatérales

En considérant les tables suivantes, une jointure externe unilatérale permet d'extraire :

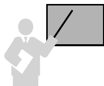
- la liste des compagnies et leurs pilotes, même les compagnies n'ayant pas de pilote (requête *R7*). Sans une jointure externe, la compagnie 'CAST' ne peut être extraite ;
- la liste des pilotes et leurs qualifications, même les pilotes n'ayant pas encore de qualification (requête *R8*).

La figure illustre les tables dominantes et subordonnées :

Figure 4-13 *Jointures externes unilatérales*



Écriture « relationnelle »

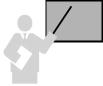


- La directive de jointure externe « `(+)` » se place du côté de la table subordonnée.
- Cette directive peut se placer à gauche ou à droite d'une clause de jointure, pas des deux côtés.

 © Éditions Eyrolles

- 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 4-33 Écritures équivalentes de jointures externes unilatérales

<table>
<tr><th>Requête</th><th>Jointures relationnelles</th><th>Jointures SQL2</th></tr>
<tr><td>R7</td><td><pre>SELECT cpg.nomComp, pil.brevet,
       pil.nom
 FROM Pilote pil, Compagnie cpg
 WHERE cpg.comp = pil.compa(+);
--équivalent à
 WHERE pil.compa(+) = cpg.comp;</pre></td><td><pre>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;</pre></td></tr>
<tr><td></td><td colspan="2"><pre>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</pre></td></tr>
<tr><td>R8</td><td><pre>SELECT qua.typeAv, pil.brevet,
       pil.nom
 FROM Pilote pil, Qualifs qua
 WHERE qua.brevet(+)=pil.brevet;

--équivalent à

 WHERE pil.brevet=qua.brevet(+);</pre></td><td><pre>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;</pre></td></tr>
<tr><td></td><td colspan="2"><pre>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</pre></td></tr>
</table>

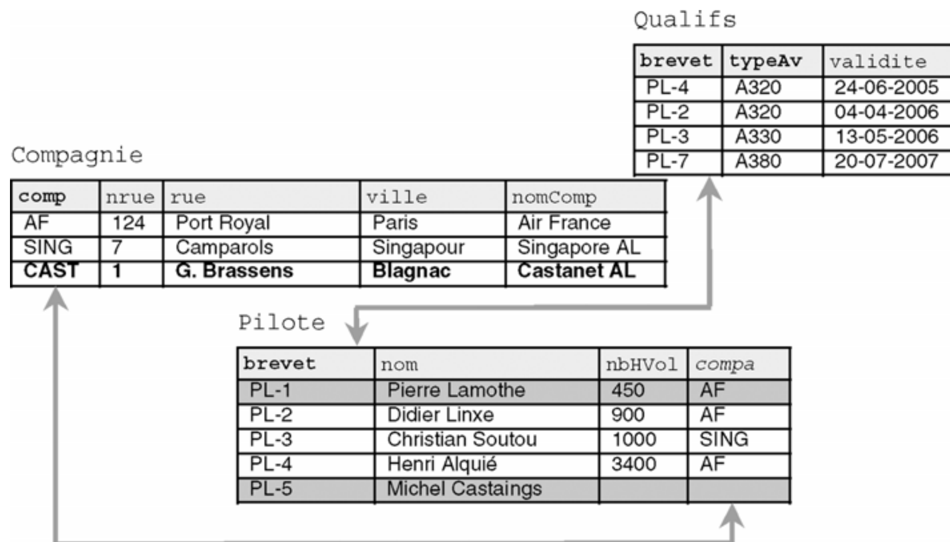
Jointures bilatérales

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 4-14 *Jointures externes bilatérales*



Qualifs

brevet	typeAv	validite
PL-4	A320	24-06-2005
PL-2	A320	04-04-2006
PL-3	A330	13-05-2006
PL-7	A380	20-07-2007

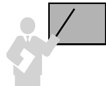
Compagnie

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

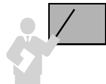
brevet	nom	nbHVol	*compa*
PL-1	Pierre Lamothe	450	AF
PL-2	Didier Linxe	900	AF
PL-3	Christian Soutou	1000	SING
PL-4	Henri Alquié	3400	AF
PL-5	Michel Castaings		

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.

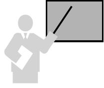
© Éditions Eyrolles

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 4-34 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).

© Éditions Eyrolles

Tableau 4-35 Sous-interrogations monolignes

Opérateur	Besoin	Requête

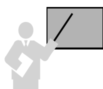
= pour les équijointures ou autojointures (= teste une ligne)	*R1* (Pilotes de la compagnie de nom 'Air France' ayant plus de 500 heures de vol.)	`SELECT brevet, nom`
		`FROM Pilote`
		`WHERE compa =`
		`(SELECT comp`
		`FROM Compagnie`
		`WHERE nomComp = 'Air France')`
		`AND nbHVol>500;`

	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 4-36 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 4-15 *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` :

© Éditions Eyrolles

Tableau 4-37 Opérateurs ALL et ANY

<table>
<tr><th>Opérateur</th><th>Besoin</th><th>Requête</th></tr>
<tr><td rowspan="2"><code>ANY</code></td><td>R11. Avions dont le nombre d'heures de vol est inférieur à celui de n'importe quel A320.</td><td><pre>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</pre></td></tr>
<tr><td>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'.</td><td><pre>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</pre></td></tr>
<tr><td rowspan="2"><code>ALL</code></td><td>R13. Avions dont le nombre d'heures de vol est inférieur à tous les A320.</td><td><pre>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</pre></td></tr>
<tr><td>R14. Compagnies et leurs avions dont le nombre d'heures de vol est supérieur à tous les avions de la compagnie de code 'AF'.</td><td><pre>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</pre></td></tr>
</table>

Jointures mixtes

Une jointure mixte combine des clauses de jointures relationnelles, procédurales (avec des sous-interrogations) ou des clauses de jointures SQL2.

Jointure relationnelle procédurale

La jointure mixte suivante combine une clause de jointure relationnelle (**en gras**) avec une jointure procédurale (en surligné) pour programmer la 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-interrogation dans la clause FROM

Introduite dans SQL2, la possibilité de construire dynamiquement une table dans la clause `FROM` d'une requête est désormais opérationnelle sous Oracle.

```
SELECT listeColonnes
     FROM table1 aliasTable1, (SELECT… FROM table2 WHERE…) aliasTable2
   [ WHERE (conditionsTable1etTable2) ];
```

Considérons la table suivante. Le but est d'extraire le pourcentage partiel de pilotes par compagnie. Dans notre exemple, il y a 5 pilotes dont 3 dépendent de 'AF'. Pour cette compagnie le pourcentage partiel de pilotes est de 3/5 soit 60%.

Figure 4-16 *Table Pilote*

Pilote

brevet	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		

 © Éditions Eyrolles

La requête suivante construit dynamiquement deux tables (alias a et b) dans la clause FROM pour répondre à cette question :

Tableau 4-38 SELECT dans un FROM

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

Sous-interrogations synchronisées

Une sous-interrogation est synchronisée si elle manipule des colonnes d'une table du niveau supérieur.

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.z…
                              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

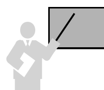
Le tableau suivant détaille un exemple d'opérateur mathématique appliqué à une sous-interrogation synchronisée :

Tableau 4-39 Sous-interrogation synchronisée

<table>
<tr><th>Besoin</th><th>Requête</th></tr>
<tr><td>R15. Avions dont le nombre d’heures de vol est supérieur au nombre d’heures de vol moyen des avions de leur compagnie (ici 700h pour 'AF' et 1115h pour 'SING').</td><td><pre>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</pre></td></tr>
</table>

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 Lamothe	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 :

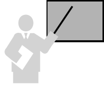
Tableau 4-40 Opérateur EXISTS

<table>
<tr><th>Besoin</th><th>Requête</th></tr>
<tr><td>R15. Pilotes ayant au moins un pilote sous leur responsabilité.</td><td><pre>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</pre></td></tr>
</table>

 © Éditions Eyrolles

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 4-41 Opérateur NOT EXISTS

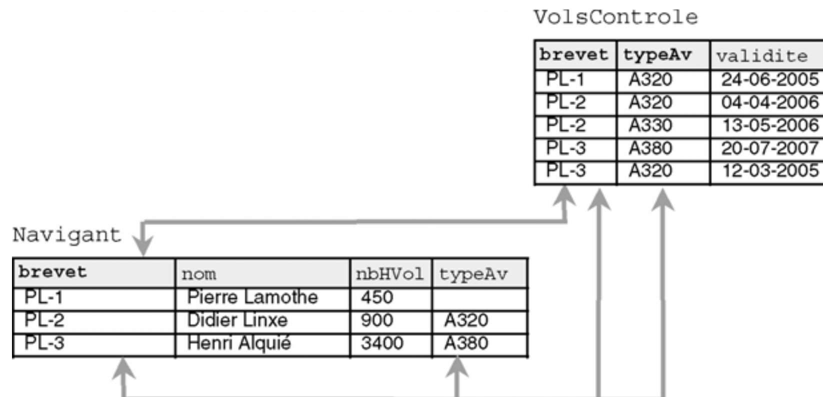
Besoin	Requête

Liste des compagnies n'ayant pas de pilote.	`SELECT cpg.* FROM Compagnie cpg WHERE NOT EXISTS (SELECT compa FROM PiloteWHERE compa = cpg.comp);`
	
	`COMP NRUE RUE VILLE NOMCOMP`
	`---- ------- ---------------- ------------ ----------`
	`CAST 1 G. Brassens Blagnac Castanet AL`

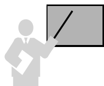
Autres directives SQL2

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*



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.

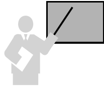
Tableau 4-42 Jointures naturelles

Besoin	Jointures SQL2

Navigants qualifiés sur un type d'appareil et instructeurs sur ce même type	`SELECT brevet, nom, typeAv, validite FROM Naviguant NATURAL JOIN VolsControle;`
	`--équivalent à`
	`SELECT brevet, nom, typeAv, validite FROM VolsControle NATURAL JOIN Naviguant;`

```
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 4-43 Jointures naturelles restreintes

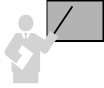
Besoin	Jointures SQL2

Nom des navigants avec leurs qualifications et dates de validité.	`SELECT nom, v.typeAv, v.validite FROM Naviguant JOIN VolsControle v USING(brevet);`
	
	`SELECT nom, v.typeAv, v.validite FROM VolsControle v JOIN Naviguant 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
```


© Éditions Eyrolles

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 4-44 Produit cartésien

Besoin	Jointures SQL2

Combinaison de toutes les lignes des deux tables.	`SELECT *`
	`FROM Naviguant CROSS JOIN VolsControle;`
	`--équivalent à`
	`SELECT *`
	`FROM VolsControle CROSS JOIN Naviguant;`

```
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 !

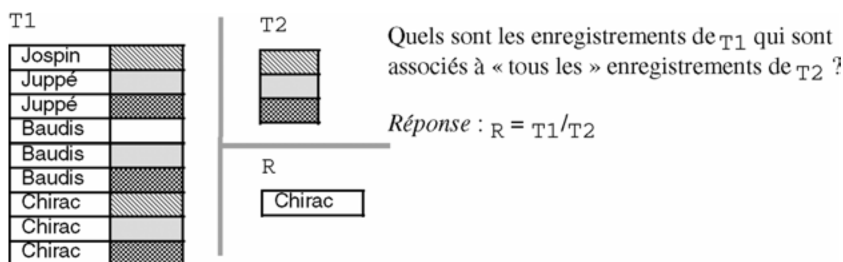
Cet opérateur permet de traduire le terme « pour tous les » des requêtes qu'on désire programmer en SQL.

On peut aussi dire que lorsque vous voulez comparer un ensemble avec un groupe de référence, il faut programmer une division.

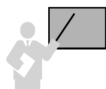
La division se traduit sous SQL par l'opérateur ensembliste `MINUS` et la fonction `NOT EXISTS`.

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 $ti \in$ T3 (de structure *[a1,…,an]*), $tj \in T2$ (*tj* de structure *[b1,…,bn]*) et $ti,tj \in T1$ (*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

 © Éditions Eyrolles

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 :

Parcours de tous les avions

```
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);
```

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
    SELECT comp FROM Compagnie WHERE pays = 'F');                  B – A
```

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.

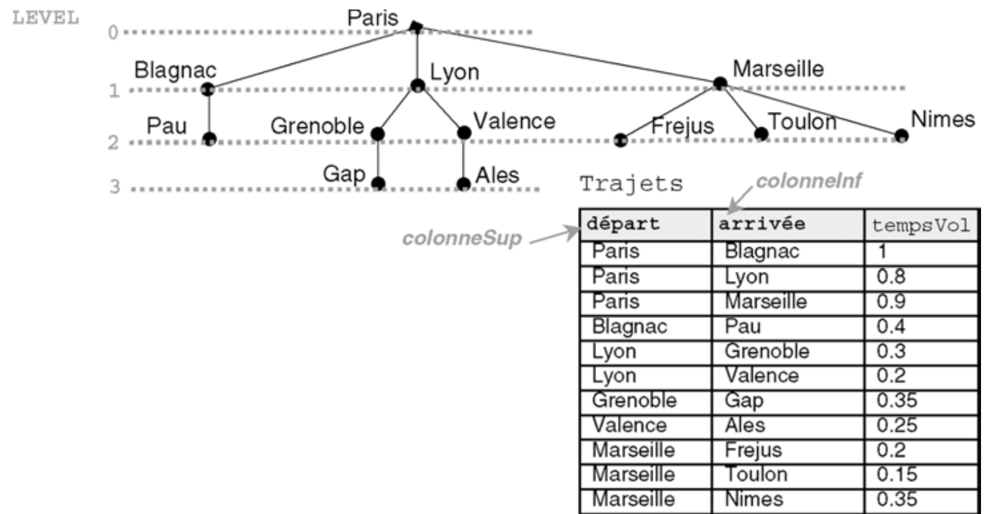
L'exemple décrit un arbre qui comprend trois niveaux. La table `Trajets` décrit cet arbre. Des deux colonnes qui assurent l'association, il est facile de distinguer celle qui désigne l'élément supérieur (*colonneSup* ici `départ`) de celle qui désigne un élément inférieur (*colonneInf* ici `arrivée`).

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;
```

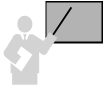

© Éditions Eyrolles

Figure 4-21 *Arbre représenté par une table*



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)

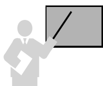


Le point de départ est spécifié par la directive `START WITH`. Ce n'est pas forcément la racine la plus haute de la hiérarchie.

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 4-45 Requêtes hiérarchiques

<table>
<tr><th>Besoin</th><th>Requête et résultat</th></tr>
<tr><td>Parcours de l'arbre de bas en haut en partant de la ville de Paris.</td><td><pre>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</pre></td></tr>
<tr><td>Parcours de l'arbre de haut en bas en partant de la ville de Lyon.</td><td><pre>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</pre></td></tr>
</table>

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 (que nous étudierons dans l'annexe consacrée à l'interface SQL*Plus) permet de substituer un libellé à une colonne, à l'affichage.

© Éditions Eyrolles

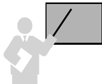
La requête suivante décale à gauche de quatre espaces les affichages pour chaque niveau (le premier niveau n'est pas décalé, le deuxième l'est de quatre espaces, etc.). La concaténation de ce décalage avec la colonne `arrivée` est renommée dans une variable (`DepartParis`), déclarée ici, de quinze caractères.

Tableau 4-46 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 4-47 Élagage d'arbres

<table>
<tr><th>Besoin</th><th>Requête et résultat sous SQL*Plus</th></tr>
<tr><td>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.</td><td><pre>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;

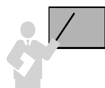
DEPARTPARIS   TEMPSVOL
--------------- ----------
Blagnac   1
    Pau   ,4
        Gap   ,35
        Ales   ,25
Marseille   ,9
    Frejus   ,2
    Toulon   ,15
    Nimes   ,35</pre></td></tr>
<tr><td>Parcours de l'arbre en entier de haut en bas en partant de la ville de Paris sans prendre en compte les trajets depuis Lyon.</td><td><pre>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</pre></td></tr>
<tr><td>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.</td><td><pre>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</pre></td></tr>
</table>

© Éditions Eyrolles

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.

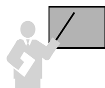
Supposons que nous disposions de la table `Aéroports` ci-dessous. L'équijointure relationnelle permet d'afficher les fréquences des aéroports sur les parcours.

Tableau 4-48 Requête hiérarchique (jointure relationnelle)

Table `Aéroport`	Requête et résultat sous SQL*Plus
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	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 4-49 Ordonner une requête hiérarchique

Web

Requête	Résultat sous SQL*Plus

<pre>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 ;</pre>
	<pre>DEPARTPARIS TEMPSVOL
	--------------- ----------
	Marseille ,9
	    Toulon ,15
	    Nimes ,35
	    Frejus ,2
	Lyon ,8
	    Valence ,2
	        Ales ,25
	    Grenoble ,3
	        Gap ,35
	Blagnac 1
	    Pau ,4</pre>

Nouveautés 10g

Depuis la version 10*g*, de nouvelles fonctions sont disponibles pour interroger des hiérarchies. Les exemples qui suivent les présentent.

Sys_Connect_By_Path

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.

Web

```
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;
```


© Éditions Eyrolles

```
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
```

Connect_By_Root

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,
       SYS_CONNECT_BY_PATH(départ,'/') chemin
     FROM Trajets WHERE LEVEL > 2
     CONNECT BY PRIOR arrivée = départ;

De Paris à CONNECT_BY Chemin...
---------- ---------- ------------------------------
Gap        Lyon       /Paris/Lyon/Grenoble
Ales       Lyon       /Paris/Lyon/Valence
```

Connect_By_Isleaf

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.

© Éditions Eyrolles

Web

```
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.

Web

```
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
```

Connect_By_IsCycle

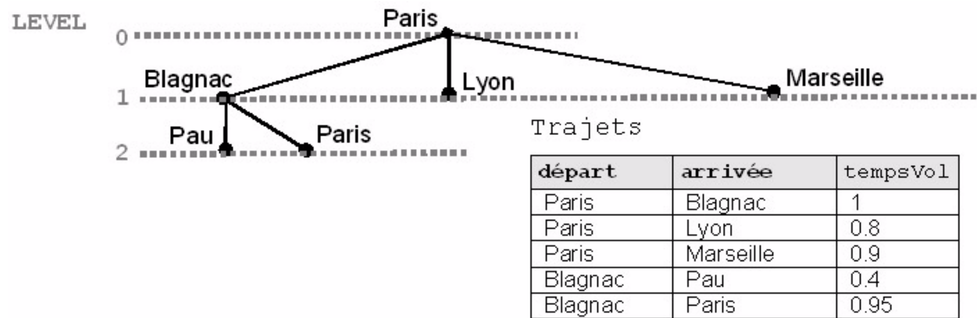
La pseudo colonne `CONNECT_BY_ISCYCLE` retourne la valeur 1 si l'enregistrement courant est associé à un enregistrement enfant qui est également son ancêtre dans la hiérarchie désignée par la condition dans la clause `CONNECT BY`. Dans le cas inverse, cette pseudo colonne vaut 0. Elle n'a de sens que 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
```


© Éditions Eyrolles

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*



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, '/')
chemin
       FROM Trajets
       WHERE CONNECT_BY_ISCYCLE = 0 AND LEVEL < 3
       START WITH départ = 'Paris'
       CONNECT BY NOCYCLE PRIOR arrivée = départ;
```

© Éditions Eyrolles

```
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.

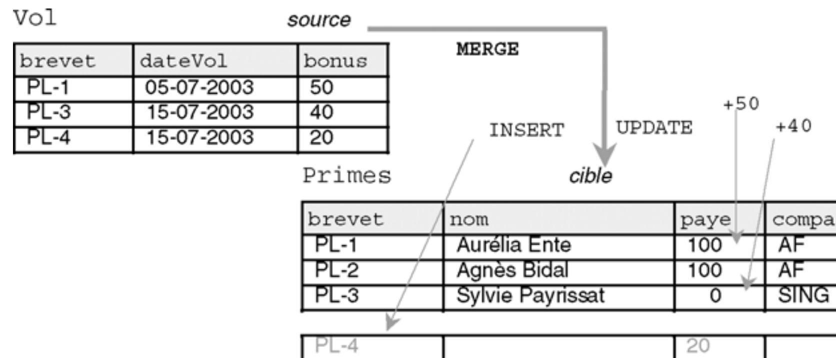
L'instruction MERGE est déterministe : il n'est pas possible de mettre à jour plusieurs fois le même enregistrement de la table cible en une seule instruction.

© Éditions Eyrolles

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 sa paye au bonus reçu. 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.

Tableau 4-50 Fusion par MERGE

Requête	Résultat sous SQL*Plus

<pre>**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);</pre>
	<pre>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</pre>

Nouveautés 10g

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 :

© Éditions Eyrolles

```
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.

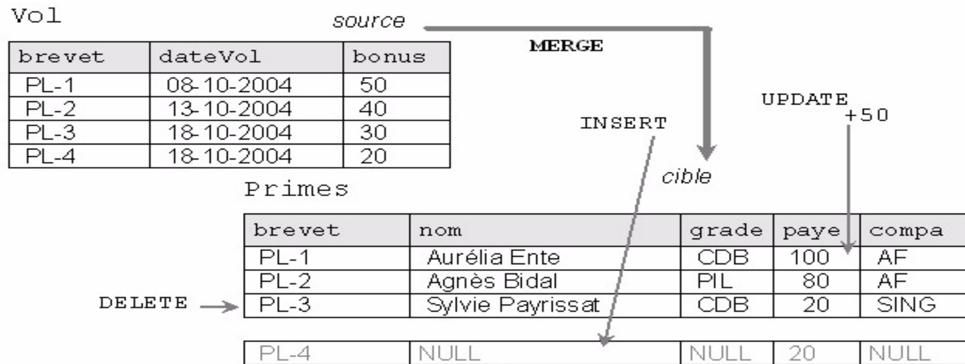
Il n'est pas possible de modifier une colonne référencée dans la clause de jointure `ON`.

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`).

 © Éditions Eyrolles

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 4-51 Fusion par MERGE

Requête	Résultat sous SQL*Plus

<pre>**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);</pre>
	<pre>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</pre>

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 4-52 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 ä, â, à…).

Le tableau suivant recence les classes d'équivalence disponibles.

Tableau 4-53 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.

© Éditions Eyrolles

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.

© Éditions Eyrolles

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 4-54 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
mòbile
pájaro
zurück

Test2

col
resume
résumé
résume
resumé
rasumé
ràsume

© Éditions Eyrolles

Tableau 4-55 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.

© Éditions Eyrolles

Tableau 4-56 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 4-57 Utilisations de la fonction REGEXP_INSTR

<table>
<tr><th>Requête</th><th>Résultat SQL*Plus</th></tr>
<tr><td><pre>SELECT REGEXP_INSTR('IUT Dept GTR, 31703 Blagnac',
                     '[^]+', 1, 4) FROM DUAL;</pre></td><td><pre>Exemple 1

        15</pre></td></tr>
<tr><td><pre>SELECT REGEXP INSTR('500 Oracle Parkway,Redwood Shores,
          CA','[s|r|p][[:alpha:]]{6}', 3, 2, 1,'i')
        "Exemple 2" FROM DUAL;</pre></td><td><pre>Exemple 2

        28</pre></td></tr>
<tr><td><pre>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');</pre></td><td><pre>ENDROIT SURFACE
-------- ------------
-
P1   1800 acres
P4   217-acre
P5   60,000 acres
P6   40,000+ acres</pre></td></tr>
</table>

© Éditions Eyrolles

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 4-58 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

© Éditions Eyrolles

Nouveautés 11g

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 de 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.

© Éditions Eyrolles

- 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 (REGEXP_COUNT)

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 enfin deux autres fonctions d'extraction disponibles depuis la 10*g*.

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.

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.

Web

```
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

Compagnie

comp	nomComp
AF	Air France
SING	Singapore AL
CAST	Castanet AL

Pilote

brevet	nbHVol	compa
PL-1	3400	AF
PL-2	4500	AF
PL-3	9000	AF
PL-4	10000	SING
PL-5	10050	SING
PL-6	16000	SING
PL-7	10000	CAST
PL-8	15000	CAST

comp_charges

nbHVol	nomComp
16900	Air France
36050	Singapore AL
25000	Castanet AL

moy_charges

nbHVol
25983.34

© Éditions Eyrolles

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 chapitre 4 (tableau 4-15).

```
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 4000 (soit 10 plages de 340 unités). La première ira de 600 à 940 (non inclus), la seconde de 940 à 1280 (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
```

Exercices

Les objectifs de ces exercices sont :

- de créer dynamiquement 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 4-28 *Structures des nouvelles tables*

Softs

nomSoft	version	prix

PCSeuls

nP	nomP	seg	ad	typeP	salle

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.

© Éditions Eyrolles

Exercice 4.5 Modifications synchronisées

Écrivez le script `modifSynchronisées.sql` pour ajouter les lignes suivantes dans la table `Installer` :

Figure 4-29 *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'.

© Éditions Eyrolles

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 les entre le 2 et le 3 du mois d'une année et d'un mois donné avec le nombre de visites pour chacun d'eux.

39 En déduire les chantiers les plus visités.

40 Nombre de visites des employés (transportés comme 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.

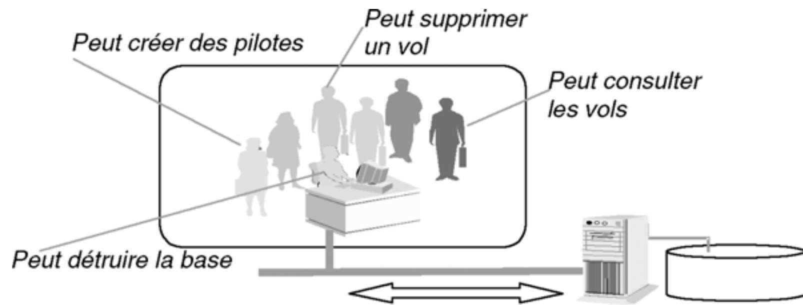
© Éditions Eyrolles

Chapitre 5

Contrôle des données

Comme dans tout système multi-utilisateur, l'usager d'un SGBD doit être identifié avant de pouvoir utiliser des ressources. L'accès aux informations et à la base de données doit être contrôlé à des fins de sécurité et de cohérence. La figure suivante illustre un groupe d'utilisateurs dans lequel existe une classification entre ceux qui peuvent consulter, mettre à jour, supprimer des enregistrements, voire les tables.

Figure 5-1 *Conséquences de l'aspect multi-utilisateurs*



Nous verrons dans cette section les aspects du langage SQL qui concernent le contrôle des données et des accès. Nous étudierons :

- la gestion des utilisateurs à qui on associe des espaces de stockage *(tablespaces)* dans lesquels se trouveront leurs objets (tables, index, séquences, etc.) ;
- la gestion des privilèges qui permettent de donner des droits sur la base de données (privilèges système) et sur les données de la base (privilèges objets) ;
- la gestion des rôles qui regroupent des privilèges système ou objets affectés par la suite à un ou plusieurs utilisateurs ;
- la gestion des vues ;
- la gestion des synonymes ;
- l'utilisation du dictionnaire des données.

Nous verrons dans chaque section comment utiliser la console d'administration pour s'affranchir d'écrire des instructions SQL.

© Éditions Eyrolles

Gestion des utilisateurs

Un utilisateur (*user* ou *username*) est identifié au niveau de la base par son nom et peut se connecter puis accéder aux objets de la base sous réserve d'avoir reçu un certain nombre de privilèges.

Un schéma est une collection nommée (du nom de l'utilisateur qui en est propriétaire) d'objets (tables, vues, séquences, index, procédures, etc.).

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 DBA (*DataBase Administrator*). Il en existe au moins un. Une petite base peut n'avoir qu'un seul administrateur. Une base importante peut en regrouper plusieurs qui se partagent les tâches suivantes :
  - installation et mises à jour de la base et des outils éventuels ;
  - gestion de l'espace disque et des espaces pour les données (*tablespaces*) ;
  - gestion des utilisateurs et de leurs objets (s'ils ne les gèrent pas eux-mêmes) ;
  - optimisation des performances ;
  - sauvegardes, restaurations et archivages ;
  - contact avec le support technique d'Oracle.
- L'administrateur réseaux (qui peut être le DBA) se charge de la configuration de l'intergiciel (*middleware*) Oracle Net au niveau des postes clients.
- Les développeurs qui conçoivent et mettent à jour la base. Ils peuvent aussi agir sur leurs objets (création et modification des tables, index, séquences, etc.). Ils transmettent au DBA leurs demandes spécifiques (stockage, optimisation, sécurité).
- Les administrateurs d'applications qui gèrent les données manipulées par l'application ou les applications. Pour les petites et les moyennes bases, le DBA joue ce rôle.
- Les utilisateurs qui se connectent et interagissent avec la base à travers les applications ou à l'aide d'outils (interrogations pour la génération de rapports, ajouts, modifications ou suppressions d'enregistrements).

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`.

© Éditions Eyrolles

Syntaxe

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`).

En l'absence de clause sur les espaces disque, le *tablespace* `SYSTEM` est associé à l'utilisateur en tant qu'espace de travail et d'espace temporaire. Il existe d'autres *tablespaces* créés par Oracle, citons `USERS` (celui que vous devriez utiliser pour votre espace par défaut) et `TEMP` (celui que vous devriez employer pour votre espace temporaire). Vous pouvez aussi créer vos espaces via la console d'administration La clause `ALTER USER` permet d'affecter un espace de travail ou temporaire différent de celui du départ.

En l'absence de profil, le profil `DEFAULT` est affecté à l'utilisateur.

Exemples

Le tableau suivant décrit la création de deux utilisateurs :

Tableau 5-1 Création d'utilisateurs

Instruction SQL	Résultat
CREATE USER Paul IDENTIFIED BY Pokémon DEFAULT TABLESPACE USERS QUOTA 10M ON USERS TEMPORARY TABLESPACE TEMP QUOTA 5M ON TEMP PASSWORD EXPIRE;	*Paul* est déclaré « utilisateur », ses objets (pas plus de 10 mégaoctets) seront stockés dans USERS, certaines de ses opérations nécessiteront de ranger des données dans TEMP (pas plus de 5 mégaoctets). Il devra changer son mot de passe à la première connexion.
CREATE USER Paul2 IDENTIFIED BY Pokémon DEFAULT TABLESPACE USERS ACCOUNT LOCK;	*Paul2* est déclaré « utilisateur », ses objets seront stockés dans USERS, son espace temporaire est SYSTEM. Le 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.

Quelques utilisateurs connus

Lors de l'installation vous avez dû noter la présence des utilisateurs SYS (mot de passe par défaut : CHANGE_ON_INSTALL) et SYSTEM (mot de passe par défaut : MANAGER). Le premier est le propriétaire des tables du dictionnaire de données. Il est préférable de ne jamais se connecter sous SYS en ligne. L'utilisateur SYSTEM est le DBA qu'Oracle vous offre. Il vous permettra d'effectuer vos tâches administratives en ligne ou par la console Enterprise Manager (créer des utilisateurs par exemple).

Si vous voulez qu'un autre utilisateur prenne cette fonction, il faudra que SYSTEM lui affecte les privilèges à retransmettre (ou le rôle DBA qui contient tous les privilèges système, voir la section « Privilèges »). L'affectation du rôle DBA est à manier avec précaution pour éviter qu'on ne détruise tout et n'importe quoi… Le rôle DBA n'inclut pas les privilèges système SYSDBA et SYSOPER qui permettent des tâches élémentaires comme la création d'une base, son démarrage ou son arrêt.

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.

Syntaxe

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
```

 © Éditions Eyrolles

```
[ 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).

Exemples

Le tableau suivant décrit des modifications des utilisateurs créés auparavant :

Tableau 5-2 Modification d'utilisateurs

Instruction SQL	Résultat
ALTER USER Paul IDENTIFIED BY X_Men TEMPORARY TABLESPACE TEMP QUOTA UNLIMITED ON TEMP;	*Paul* a changé de mot de passe, son espace temporaire est illimité dans `TEMP`. Il ne devra plus changer son mot de passe à la première connexion.
ALTER USER Paul2 DEFAULT TABLESPACE USERS QUOTA 10M ON USERS ACCOUNT UNLOCK;	L'espace de travail de *Paul2* est limité à 10 mégaoctets dans `USERS`. 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`.

Syntaxe

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.

Conséquences

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 :
```


© Éditions Eyrolles

```
{ 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 :

© Éditions Eyrolles

Tableau 5-3 Modification d'utilisateurs

Instructions SQL	Explications
CREATE PROFILE profil_Etudiants **LIMIT** SESSIONS_PER_USER 3 CPU_PER_CALL 3000 CONNECT_TIME 45 LOGICAL_READS_PER_CALL 1000 PRIVATE_SGA 15K IDLE_TIME 40 FAILED_LOGIN_ATTEMPTS 5 PASSWORD_LIFE_TIME 70 PASSWORD_REUSE_TIME 60 PASSWORD_REUSE_MAX UNLIMITED PASSWORD_LOCK_TIME 1/24 PASSWORD_GRACE_TIME 10;	• 3 sessions simultanées autorisées. • Un appel système ne peut pas consommer plus de 30 secondes de CPU. • Chaque session ne peut excéder 45 minutes. • Un appel système ne peut lire plus de 1 000 blocs de données en mémoire et sur le disque. • Chaque session ne peut allouer plus de 15 ko de mémoire en SGA. • Pour chaque session, 40 minutes d'inactivité maximum sont autorisées. • 5 tentatives de connexion avant blocage du compte. • Le mot de passe est valable pendant 70 jours et il faudra attendre 60 jours avant qu'il puisse être utilisé à nouveau. • 1 seul jour d'interdiction d'accès après que les 5 tentatives de connexion ont été atteintes. • La période de grâce qui prolonge l'utilisation du mot de passe avant son changement est de 10 jours.

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.

© Éditions Eyrolles

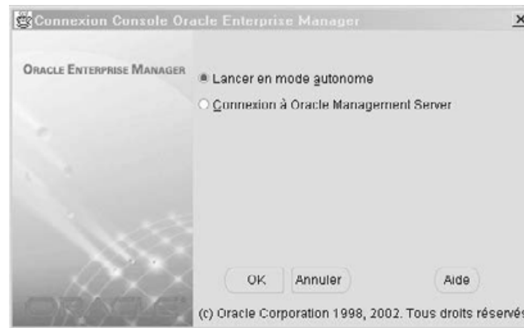
Console Enterprise Manager

Enterprise Manager est un outil graphique d'administration écrit en Java. Depuis la version 10*g*, la console s'exécute dans un navigateur. Cet outil centralise principalement la création, le diagnostic, le paramétrage (*tuning*), la sauvegarde et la restauration de bases locales ou distantes. En outre, il permet de gérer les couches réseaux, de programmer des tâches journalières, etc. Nous décrirons brièvement les aspects relatifs aux utilisateurs et aux profils.

*Console sous Oracle9*i

La console se lance sous Windows à partir de `Démarrer/Programmes/Oracle…/Enterprise Manager Console`. Apparaît ensuite l'écran ci-dessous pour lequel, dans notre cas, il faut choisir le mode autonome :

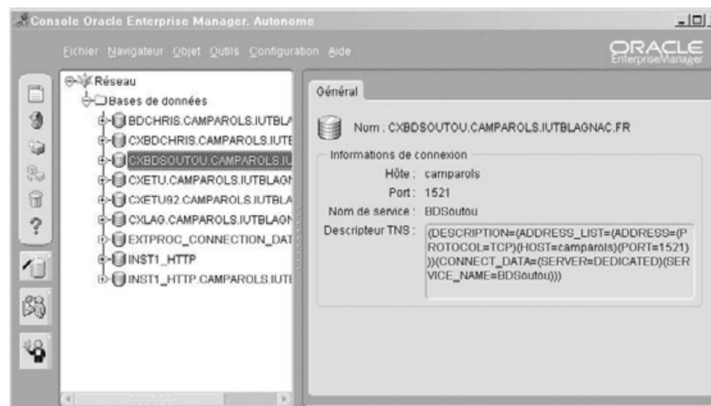
Figure 5-2 *Mode d'exécution de la console*



Choix de la connexion

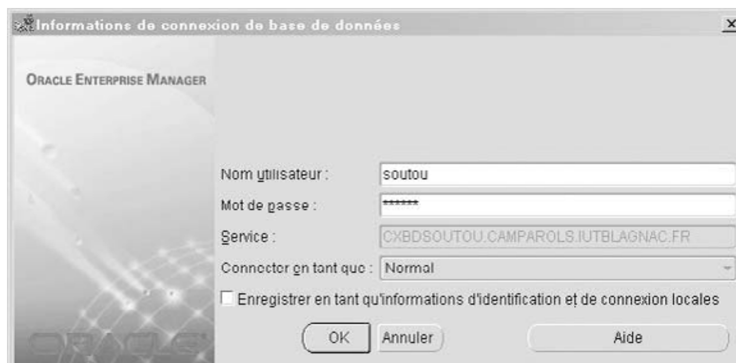
L'écran suivant décrit l'architecture générale accessible par la console. Il faut choisir une connexion à une base de données (ici le choix se porte sur la base de nom de service `BDSoutou`) identifiée par la chaîne de connexion `CXBDSOUTOU`.

Figure 5-3 *Connexions disponibles*



Une fois la connexion choisie, il faut s'identifier au niveau de la base de données cible. Si aucun utilisateur n'existe, identifiez-vous sous `SYSTEM`, mot de passe `MANAGER` (à moins que vous n'ayez modifié ce mot de passe lors de l'installation).

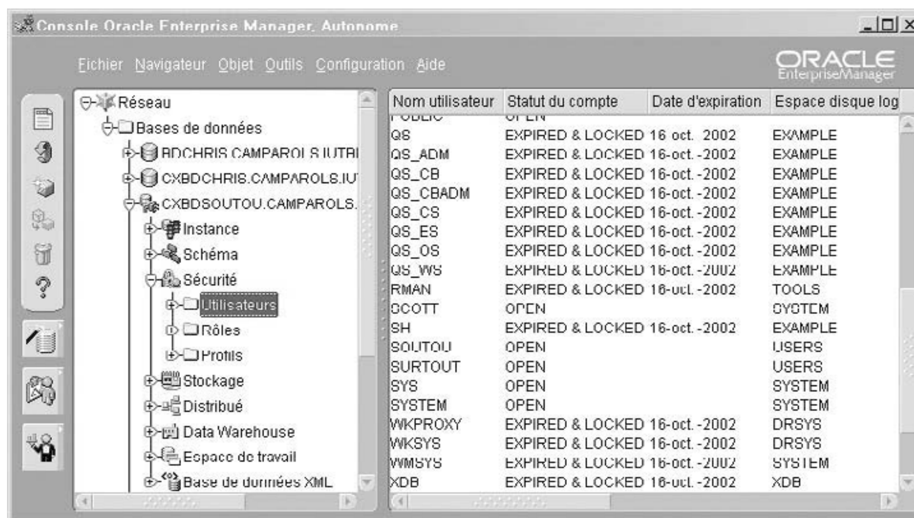
Figure 5-4 *Connexion à une base*



Choix de la fonction

Lorsque la connexion est établie pour l'utilisateur choisi, il est possible de gérer la base tout en respectant ses propres prérogatives. Ici, nous choisissons de lister les utilisateurs de la base `BDSoutou`.

Figure 5-5 *Liste des utilisateurs*

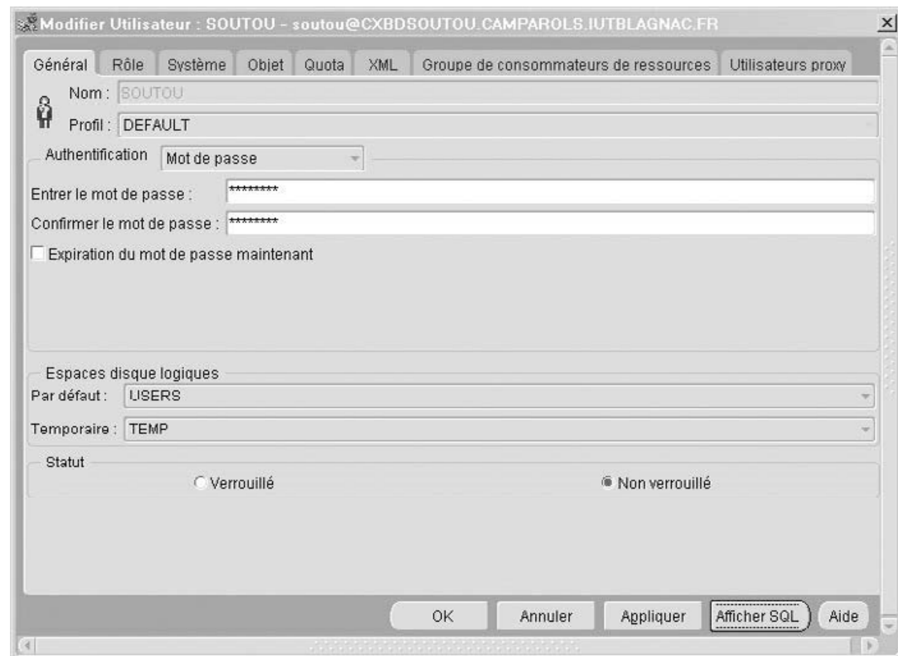


© Éditions Eyrolles

Caractéristiques d'un utilisateur

En sélectionnant un utilisateur, on retrouve graphiquement ses options initialisées par des commandes SQL en ligne. Pour toute modification dans quelque fenêtre que ce soit, il est possible d'extraire la commande SQL générée automatiquement (bouton `Afficher SQL`). Cette option est très précieuse pour les administrateurs qui pourront stocker les sources de toutes leurs opérations pour les réutiliser à la demande, si nécessaire.

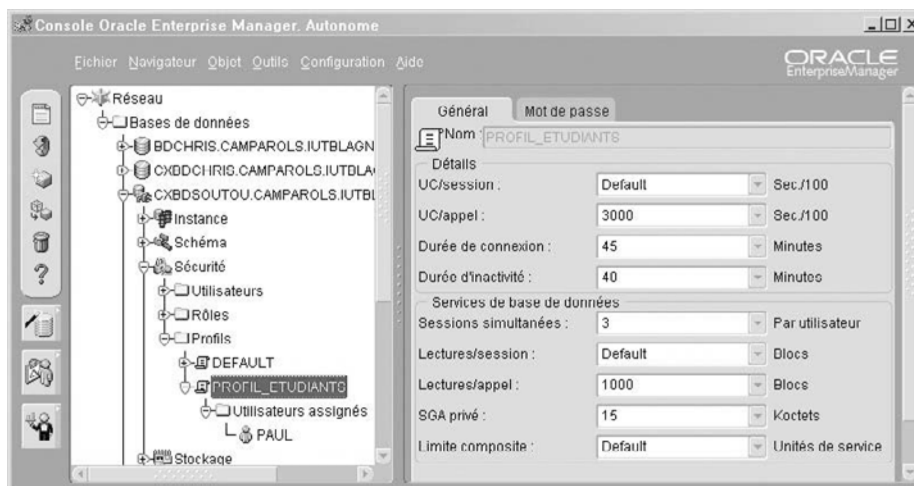
Figure 5-6 *Caractéristiques d'un utilisateur*



Caractéristiques d'un profil

L'écran suivant (figure 5-7) détaille une partie du profil que nous avons créé en ligne précédemment. L'arborescence permet de connaître les utilisateurs rattachés à ce profil. Par un clic droit sur le nom du profil, il sera possible de l'enlever, de regénérer le script de création (choix `afficher le DDL de l'objet`). Une autre fonction intéressante est de pouvoir trouver les autres objets qui font référence à ce profil (choix `rechercher des objets de bases de données…`).

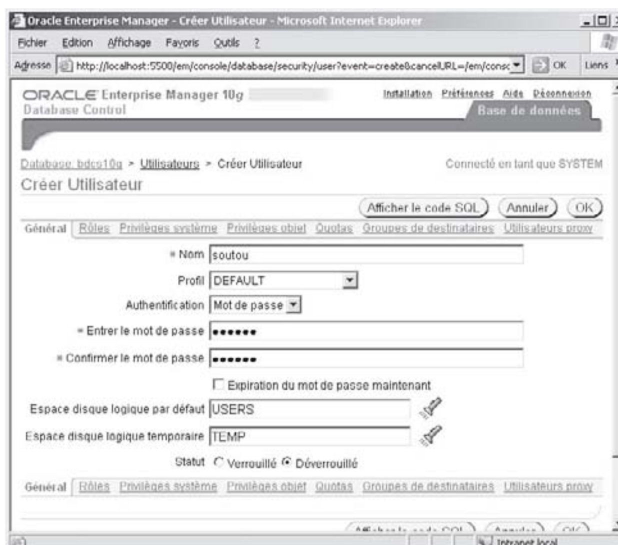
Figure 5-7 *Caractéristiques d'un profil*



*Console sous Oracle 10*g

Pour créer un utilisateur, connecter l'administrateur sur `http://localhost:5500/em` en choisissant l'onglet `Administration` et le lien `Utilisateurs`. Penser à affecter manuellement (lien `Quotas`) une valeur en Mo pour les espaces `USERS` et `TEMP`. Il semble que cette interface Web ne permette pas d'affecter un quota illimité.

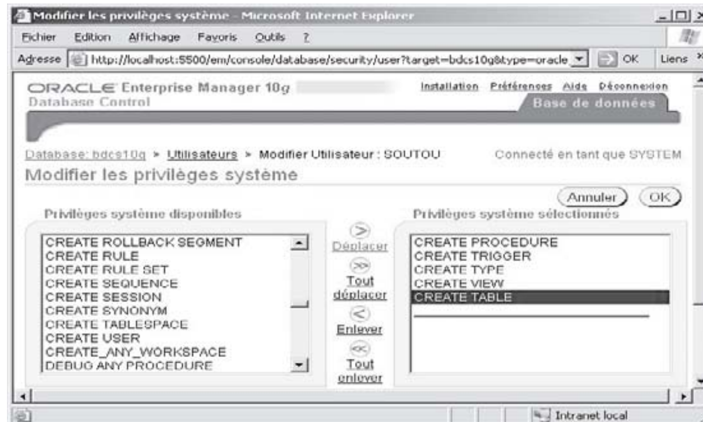
Figure 5-8 *Création d'un utilisateur (10*g*)*



© Éditions Eyrolles

Le rôle `CONNECT` est affecté par défaut à un nouvel utilisateur (il permet entre autre de pouvoir se connecter, attention il n'est pas suffisant pour créer des tables, types, etc.). Le rôle `RESOURCE` n'apparaît plus (toutefois il existe toujours), il faudra donc alimenter les privilèges d'un utilisateur explicitement (en relançant si nécessaire la console d'administration et en modifiant l'utilisateur au niveau des privilèges systèmes). L'écran suivant illustre ce propos en autorisant l'utilisateur *Soutou* à créer ses procédures, déclencheurs, types, vues et tables.

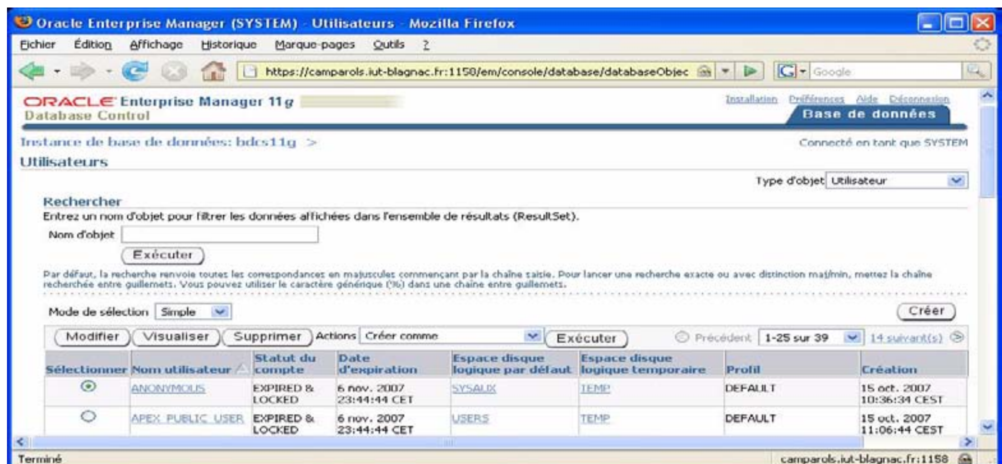
Figure 5-9 *Affectation de privilèges à un utilisateur (10*g*)*



*Console sous Oracle 11*g

Pour créer un utilisateur, sélectionner l'onglet `Serveur` de la console (dans mon cas `https://camparols.iut-blagnac.fr:1158/em`) puis choisir le lien `Utilisateurs`. Une fois la liste des schémas existants obtenue, cliquer sur `Créer`.

Figure 5-10 *Création d'un utilisateur (11*g*)*

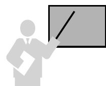


Les écrans suivants sont similaires à ceux de la version 10*g*, le rôle CONNECT est affecté par défaut à un nouvel utilisateur. Comme depuis la 10*g*, il faut alimenter les privilèges de tout utilisateur (au niveau des privilèges systèmes). La procédure à suivre pour affecter des privilèges à un utilisateur Oracle 11*g* est la même que pour la version précédente.

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.



Pour autoriser l'accès au schéma SYS par des privilèges assortis de la mention ANY, il faut passer le paramètre d'initialisation O7_DICTIONARY_ACCESSIBILITY à TRUE avec la commande ALTER SYSTEM ou par la console *Enterprise Manager* via l'arborescence Instance/Configuration.

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*.

© Éditions Eyrolles

Tableau 5-4 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 :

© Éditions Eyrolles

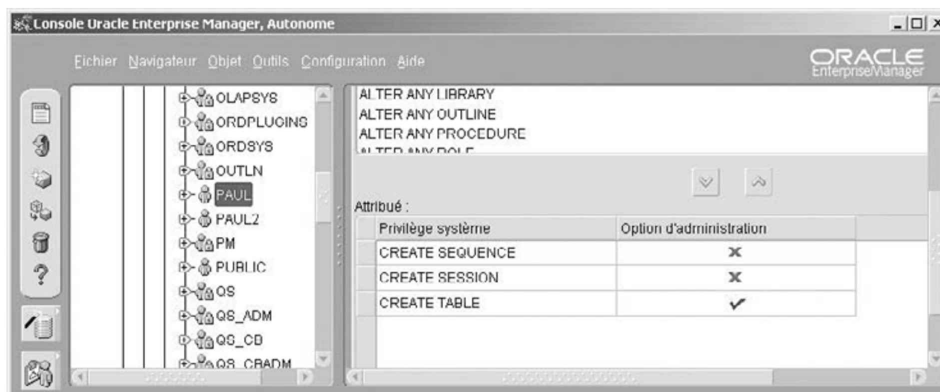
Tableau 5-5 Affectation de privilèges système

Administrateur	Explications
`GRANT CREATE SESSION, CREATE SEQUENCE TO Paul;`	*Paul* peut se connecter à la base sous SQL*Plus par un outil (la console par exemple), par programme. Il peut créer des séquences.
`GRANT CREATE TABLE TO Paul WITH ADMIN OPTION;`	*Paul* peut créer des tables dans son schéma et peut retransmettre ce privilège à un tiers.
`GRANT CREATE SESSION, CREATE ANY TABLE, DROP ANY TABLE TO Paul2;`	*Paul2* peut se connecter à la base, créer et détruire des tables dans tout schéma.

Console sous Oracle9i

L'outil graphique d'administration *Enterprise Manager 9i* permet de visualiser, d'affecter, de révoquer des privilèges système. L'écran suivant illustre l'arborescence qu'il faut utiliser (`Sécurité/Utilisateurs`) en choisissant un utilisateur particulier (ici *Paul*) et en sélectionnant l'onglet `Système`. On retrouve les trois privilèges donnés auparavant par la commande `GRANT`.

Figure 5-11 *Privilèges système de Paul*



Pour les versions 10*g* et 11*g*, il suffit d'opérer la même manipulation en choisissant les onglets appropriés.

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`.

© Éditions Eyrolles

```
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 5-6 Révocation de privilèges système

Administrateur	Explications
REVOKE CREATE SESSION **FROM** Paul, Paul2;	*Paul* et *Paul2* ne peuvent plus se connecter à la base. Ils conservent néanmoins leurs autres privilèges. Un tiers peut ainsi créer par exemple des tables dans leur schéma (ils ont tous deux le privilège CREATE TABLE).
REVOKE ALL PRIVILEGES **FROM** Paul2;	Commande incorrecte car *Paul2* n'a pas reçu tous les privilèges système.

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 5-7 Options possibles de quelques privilèges objets

Privilège	Table	Vue	Séquence	Programme PL/SQL
ALTER			×	
DELETE	×	×		
EXECUTE				×
INDEX	×			
INSERT	×	×		
REFERENCES	×			
SELECT	×	×	×	
UPDATE	×	×		

© Éditions Eyrolles

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.

© Éditions Eyrolles

Tableau 5-8 Affectations de privilèges objets

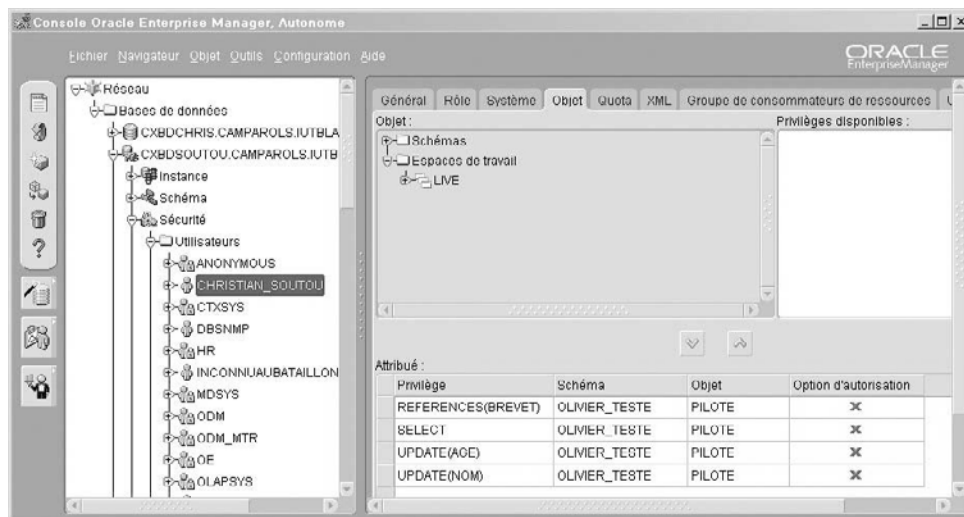
<table>
<tr><th>olivier_teste</th><th>christian_soutou</th></tr>
<tr><td><pre>--Table Pilote
BREVET NOM   AGE ADRESSE
------ -------- ----- ----------------
P1   Laroche   39 Montauban
P2   Lamothe   34 Ramonville
P3   Albaric   34 Vieille-Toulouse
P4   Labat   33 Pau</pre></td><td><pre>--Table Qualif
TYPEQUALIF PIL
---------- ------
PPL   P1
FI/A   P1
PPL   P4
CPL   P4
IFR   P3</pre></td></tr>
<tr><td>Affectation des privilèges de lecture de la table <code>Pilote</code>, de modification des colonnes <code>nom</code> et <code>age</code> et de référence à la clé primaire <code>brevet</code> à l'utilisateur christian_soutou.<pre>GRANT REFERENCES(brevet),
      UPDATE(nom, age), SELECT
 ON Pilote
 TO christian_soutou;</pre></td><td>Modification des colonnes <code>nom</code> et <code>age</code> de la table <code>Pilote</code> de olivier_teste.<pre>UPDATE olivier_teste.Pilote
 SET   nom = 'Boutrand', age = age+1
 WHERE nom = 'Labat';</pre>Lecture de la table <code>Pilote</code> de olivier_teste.<pre>SELECT * FROM olivier_teste.Pilote
       WHERE nom = 'Boutrand';
BREVET NOM   AGE ADRESSE
------ -------- ------- ----------------
P4   Boutrand   34 Pau</pre>Déclaration d'une clé étrangère vers la table <code>Pilote</code> de olivier_teste.<pre>ALTER TABLE Qualifications
 ADD CONSTRAINT dans_Pilote_olivier_teste
 FOREIGN KEY(pil)
 REFERENCES olivier_teste.Pilote(brevet);</pre></td></tr>
</table>

L'option `REFERENCES` permet d'implanter une contrainte d'intégrité entre deux tables de schémas distincts. Ici, l'ajout d'une qualification n'est permise que si le pilote est référencé dans la table `Pilote` du schéma *olivier_teste*.

Console d'administration

L'écran suivant illustre l'arborescence `Sécurité/Utilisateurs` de l'outil *Enterprise Manager9i* qu'il faut utiliser pour travailler avec les privilèges objets. En choisissant un utilisateur particulier et en sélectionnant l'onglet `Objet`, on retrouve les privilèges donnés auparavant par la commande `GRANT`. Les flèches permettent d'affecter ou de révoquer graphiquement des privilèges objets.

Figure 5-12 *Privilèges objets de christian_soutou*



Pour les versions 10*g* et 11*g*, il suffit d'opérer la même manipulation en choisissant les onglets appropriés.

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.

 © Éditions Eyrolles

- `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 5-9 Révocation de privilèges objets

olivier_teste	Explications
REVOKE UPDATE, SELECT **ON** Pilote **FROM** christian_soutou;	*christian_soutou* ne peut plus modifier ni lire la table `Pilote` de *olivier_teste*.
REVOKE REFERENCES ON Pilote FROM christian_soutou;	Commande incorrecte car l'option `CASCADE CONSTRAINT` doit être utilisée.
REVOKE REFERENCES **ON** Pilote **FROM** christian_soutou CASCADE CONSTRAINTS ;	*christian_soutou* ne peut plus bénéficier de la table `Pilote` pour programmer une contrainte référentielle via une clé étrangère.

Privilèges prédéfinis

Oracle propose des privilèges prédéfinis pour faciliter la gestion des droits. Le tableau suivant en décrit quelques-uns :

Tableau 5-10 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;
```

© Éditions Eyrolles

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 :

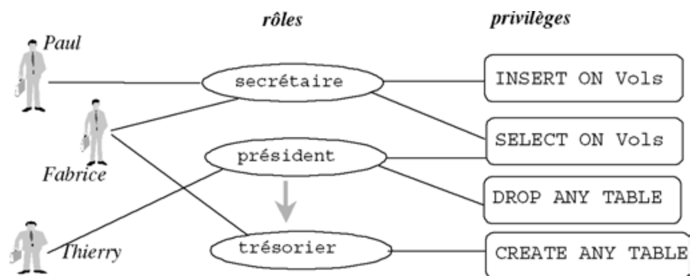
```
GRANT SELECT ANY DICTIONARY TO utilisateur;
```

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 président 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 5-13 *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.

© Éditions Eyrolles

Création d'un rôle (CREATE ROLE)

Pour pouvoir créer un rôle vous devez posséder le privilège `CREATE ROLE`. La syntaxe SQL est la suivante :

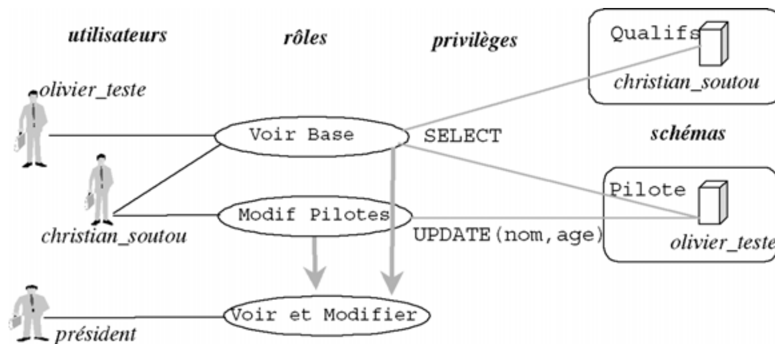
```
CREATE ROLE nomRôle
[ NOT IDENTIFIED | IDENTIFIED
   { BY motdePasse | USING [schéma.]paquetage | EXTERNALLY |
   GLOBALLY } ] ;
```

- `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 `olivier_teste` au niveau des colonnes `nom` et `age`. `Voir_et_Modifier` hérite des deux rôles précédents et est affecté à l'utilisateur `président`.

Figure 5-14 *Rôles à définir*



Le tableau suivant décrit la chronologie à respecter pour créer, alimenter et affecter ces rôles :

Tableau 5-11 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 olivier_teste.Pilote TO Voir_Base ;`
`GRANT SELECT ON christian_soutou.Qualifications TO Voir_Base ;`
`GRANT UPDATE (nom,age) ON olivier_teste.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, olivier_teste;`
`GRANT Voir_et_Modifier TO président;`
	Affectation des trois rôles à des utilisateurs.

Rôles prédéfinis

Oracle propose des rôles prédéfinis attribués aux utilisateurs SYSTEM et SYS. Ils sont générés lors de la création de la base par des scripts accessibles (sous Oracle9*i* dans le sous-répertoire rdbms\admin). Il est possible d'utiliser ces rôles en les affectant à des utilisateurs ou en enrichissant d'autres rôles.

Les trois premiers rôles du tableau suivant sont fournis pour maintenir la compatibilité avec les versions précédentes d'Oracle et ne seront plus créés automatiquement dans des versions futures. Il sera donc préférable de créer des rôles administratifs personnalisés sans utiliser ces « bons vieux » CONNECT et RESOURCE.

 © Éditions Eyrolles

Dans la version actuelle d'Oracle, si vous voulez aller vite en besogne, affectez le rôle CONNECT à un utilisateur qui manipule des tables. Ajoutez RESOURCE s'il doit programmer sous PL/SQL.

Tableau 5-12 Quelques rôles prédéfinis

Rôle	Privilèges
CONNECT	ALTER SESSION, CREATE CLUSTER, CREATE DATABASE LINK, **CREATE SEQUENCE**, **CREATE SESSION**, CREATE SYNONYM, **CREATE TABLE**, CREATE VIEW
RESOURCE	CREATE CLUSTER, CREATE INDEXTYPE, CREATE OPERATOR, **CREATE PROCEDURE**, CREATE SEQUENCE, CREATE TABLE, **CREATE TRIGGER**, CREATE TYPE
DBA	Tous les privilèges système avec WITH ADMIN OPTION
EXP_FULL_DATABASE	Privilèges requis pour des exportations : SELECT ANY TABLE, BACKUP ANY TABLE, EXECUTE ANY PROCEDURE, EXECUTE ANY TYPE… avec les rôles EXECUTE_CATALOG_ROLE et SELECT_CATALOG_ROLE
IMP_FULL_DATABASE	Privilèges requis pour des importations
EXECUTE_CATALOG_ROLE	EXECUTE sur les objets du dictionnaire des données
SELECT_CATALOG_ROLE	SELECT sur les objets du dictionnaire des données

Console Enterprise Manager

L'écran suivant illustre l'arborescence (Sécurité/Rôles) de l'outil *Enterprise Manager 9i* qu'il faut utiliser pour travailler avec les rôles. Nous retrouvons les rôles définis plus haut par des commandes SQL.

En choisissant un rôle et en sélectionnant l'onglet Objet, on retrouve ses privilèges associés. Les flèches permettent de le modifier en affectant ou en révoquant graphiquement d'autres privilèges. Le rôle Voir_Base est décrit dans l'écran suivant ainsi que la possibilité de le modifier en travaillant sur la table Qualifications de l'utilisateur *christian_soutou*.

Pour les versions 10*g* et 11*g*, il suffit d'opérer la même manipulation en choisissant les onglets appropriés.

© Éditions Eyrolles

Figure 5-15 Rôles sous Enterprise Manager

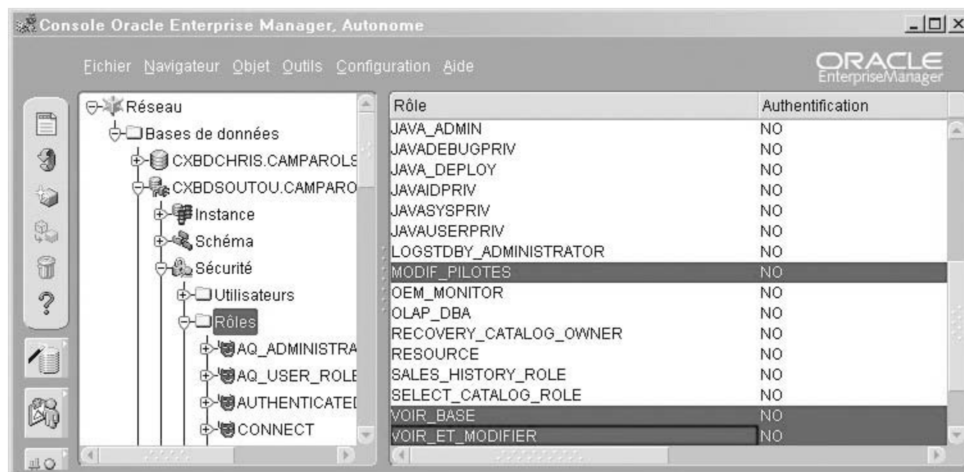
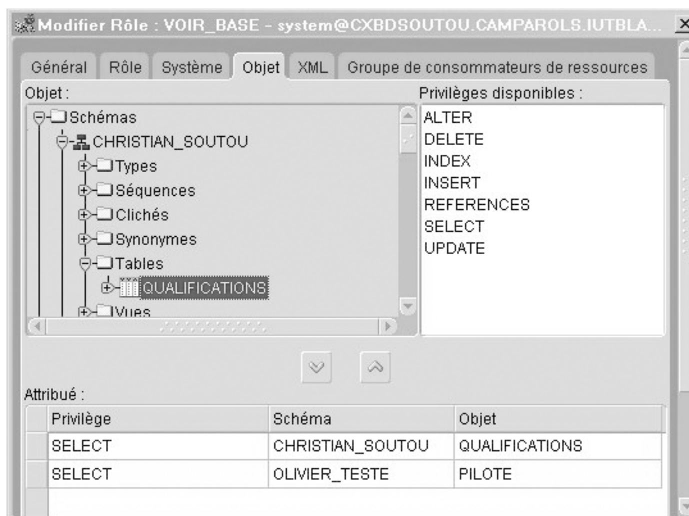


Figure 5-16 Détail d'un rôle sous Enterprise Manager



Révocation d'un rôle

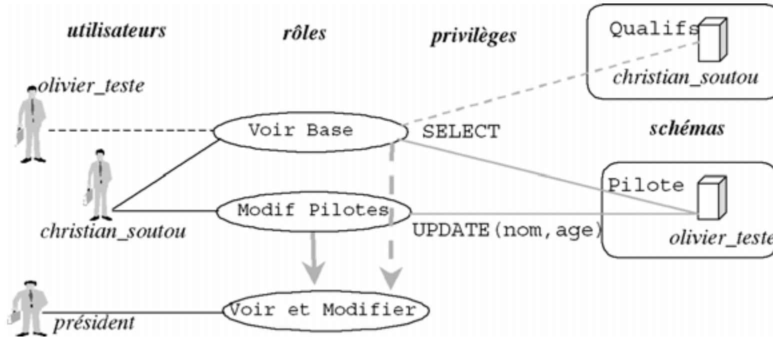
La révocation de privilèges d'un rôle existant se réalise à l'aide de la commande `REVOKE` précédemment étudiée dans les sections « Privilèges ». Pour pouvoir annuler un rôle, vous devez détenir au préalable ce rôle avec l'option `ADMIN OPTION` ou avoir reçu le privilège système `GRANT ANY ROLE`.

 © Éditions Eyrolles

```
REVOKE nomRôle [, nomRôle…]
   FROM {utilisateur | nomRôle | PUBLIC} [,{utilisateur | nomRôle |
   PUBLIC}]… ;
```

La figure suivante décrit, sous la forme de pointillés, les révocations à programmer. Il s'agit de révoquer le rôle `Voir_Base` de l'utilisateur *oliver_teste*, le rôle `Voir_Base` de rôle `Voir_et_Modifier` et le privilège `SELECT` (de la table `Qualifs`) du rôle `Voir_Base`.

Figure 5-17 *Rôles à révoquer*



Le tableau suivant décrit les instructions SQL à employer à cet effet :

Tableau 5-13 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 olivier_teste;`	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
motdePasse]]…
```

© Éditions Eyrolles

```
  | 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 5-14 Révocations de rôles et de privilèges

Administrateur	Explications
CREATE ROLE Supprime_Pilotes IDENTIFIED BY suppil;	Création d'un rôle identifié.
GRANT DELETE ON olivier_teste.Pilote TO Supprime_Pilotes;	Alimentation du rôle.
GRANT Supprime_Pilotes TO christian_soutou;	Attribution du rôle à un utilisateur.
Connexions de *christian_soutou*	
--Possible car rôle actif par défaut DELETE FROM olivier_teste.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 olivier_teste.Pilote; **ERREUR à la ligne 1 :** **ORA-00942:** Table ou vue inexistante	--Possible car rôle actif de nouveau DELETE FROM olivier_teste.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).

© Éditions Eyrolles

```
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 5-15 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

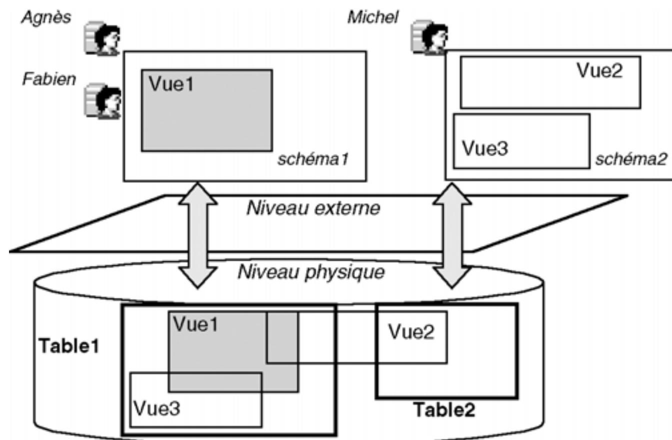
Outre le contrôle de l'accès aux données (privilèges), la confidentialité des informations est un aspect important qu'un SGBD relationnel doit prendre en compte. La confidentialité est assurée par l'utilisation de vues *(views)*, qui agissent comme des fenêtres sur la base de données. Ce chapitre décrit les différents types de vues qu'on peut rencontrer.

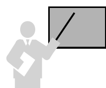
Les vues correspondent à ce qu'on appelle *le niveau externe* qui reflète la partie visible de la base de données pour chaque utilisateur. Seules les tables contiennent des données et pourtant, pour l'utilisateur, une vue apparaît comme une table. En théorie, les utilisateurs ne devraient accéder aux 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 5-18 *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 :

© Éditions Eyrolles

```
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 de 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.

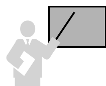
- WITH READ ONLY déclare la vue non modifiable par INSERT, UPDATE, ou DELETE.
- WITH CHECK OPTION garantit que toute mise à jour de la vue par INSERT ou UPDATE s'effectuera conformément au prédicat contenu dans la requête de définition. Il existe toutefois des situations particulières et marginales qui 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 5-16 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).

 © Éditions Eyrolles

Figure 5-19 Deux vues d'une table

Pilote

brevet	nom	nbHVol	adresse	compa
PL-1	Soutou	890	Castanet	CAST
PL-2	Laroche	500	Montauban	CAST
PL-3	Lamothe	1200	Ramonville	AF
PL-4	Albaric	500	Vieille-Toulouse	AF
PL-5	Bidal	120	Paris	ASO
PL-6	Labat	120	Pau	ASO
PL-7	Tauzin	100	Bas-Mauco	ASO

```
CREATE VIEW PilotesAF
  AS SELECT *
  FROM Pilote
  WHERE compa = 'AF';
```

```
CREATE VIEW Etat civil
  AS SELECT nom, nbHVol, adresse,
  compa FROM Pilote;
```

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 5-17 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 5-18 Vue avec alias

Écriture 1	Écriture 2
CREATE OR REPLACE VIEW PilotesPasAF (codepil,nomPil,heuresPil, adressePil, société) AS SELECT * FROM Pilote WHERE NOT (compa = 'AF');	**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 5-19 Vue d'une vue

Création	Contenu de la vue
CREATE OR REPLACE VIEW EtatCivilPilotesPasAF AS SELECT nomPil,heuresPil,adressePil FROM PilotesPasAF;	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`.

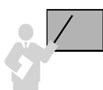
Redéfinissons la vue `PilotesPasAF` à l'aide de cette option. Les messages d'erreur induits par la clause de lecture seule, générés par Oracle ne sont pas très parlants.

 © Éditions Eyrolles

Tableau 5-20 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.

© Éditions Eyrolles

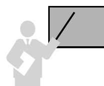
Tableau 5-21 Mises à jour de vues

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 5-22 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`

© Éditions Eyrolles

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 5-23 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.

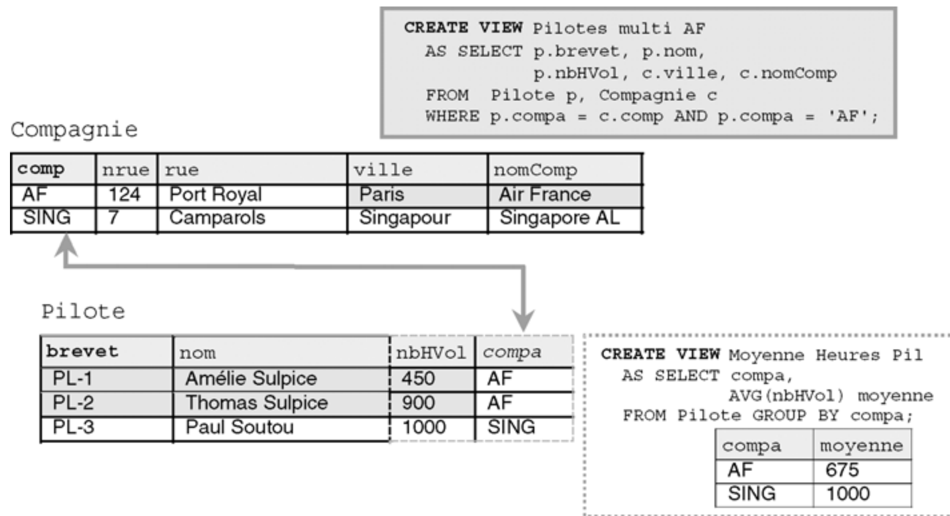
Les restrictions de création sont les suivantes :

- 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`.

© Éditions Eyrolles

Figure 5-20 *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 5-24 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

 © Éditions Eyrolles

l'être). Les suppressions se répercutent aussi sur les enregistrements de la table protégée par clé (Pilote).

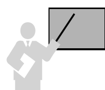
Tableau 5-25 Mises à jour d'une vue multitable

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).

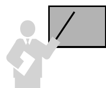
En considérant les données initiales, pour la vue multitable Vue_Multi_Comp_Pil, la table préservée est la table Pilote, car la colonne brevet identifie chaque enregistrement extrait de la vue alors que la colonne comp ne le fait pas.

Tableau 5-26 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 5-27 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.

© Éditions Eyrolles

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 5-28 Fonctions et variables d'environnement

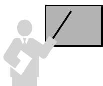
<table>
<tr><th>Variable / Fonction</th><th colspan="2">Signification</th></tr>
<tr><td>USER</td><td colspan="2">Nom de l'utilisateur connecté.</td></tr>
<tr><td>UID</td><td colspan="2">Numéro d'identification de l'utilisateur connecté.</td></tr>
<tr><td rowspan="4">USERENV('paramètre')</td><td rowspan="4">Fonction utilisant un des paramètres ci-contre.</td><td>SESSIONID : numéro de la session.</td></tr>
<tr><td>TERMINAL : nom du terminal dans le système d'exploitation hôte.</td></tr>
<tr><td>ENTRYID : numéro chronologique de la commande SQL dans la session.</td></tr>
<tr><td>LANGUAGE : langage utilisé.</td></tr>
</table>

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).

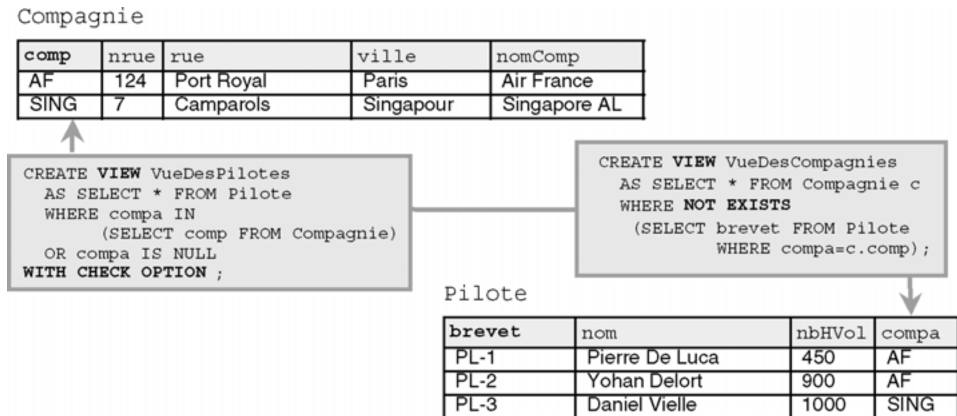


La cohérence référentielle entre deux tables *t1* (table « père ») et *t2* (table « fils ») se programme :

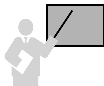
- du « père » vers le « fils » par l'utilisation d'une vue *v1* de la table *t1* définie avec la clause `NOT EXISTS` ;
- du « fils » vers le « père » par l'utilisation d'une vue *v2* de la table *t2* définie avec la clause `WITH CHECK OPTION`.

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 5-21 *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 :

© Éditions Eyrolles

Tableau 5-29 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 5-30 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 8h30 à 17h30.

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 5-31 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];
```


© Éditions Eyrolles

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];
```

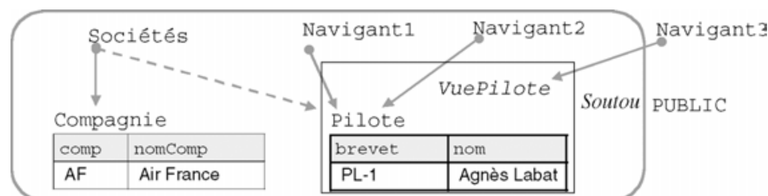
- `OR REPLACE` recrée le synonyme même s'il en existe déjà un de ce nom (cela vous évite de le détruire puis de le créer). Il existe une restriction pour les synonymes de types dont dépend une table (extension objet non étudiée dans ce livre).
- `PUBLIC` crée un synonyme public, accessible par tous, 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 (`Sociétés`, `Navigant1` et `Navigant2`) et un synonyme public (`Navigant3`).

Figure 5-22 *Synonymes de l'utilisateur Soutou*



Les instructions SQL sont les suivantes :

Tableau 5-32 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 Sociétés FOR Pilote;`
 	
`CREATE` **`OR REPLACE`** `SYNONYM Sociétés`
`FOR Compagnie;`
	Remplacement du synonyme privé `Sociétés` de la table `Compagnie` à la place de la table `Pilote`.

 © Éditions Eyrolles

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 5-33 Privilèges sur les synonymes

Utilisateur *Soutou*	Utilisateur *Paul*
`GRANT INSERT,SELECT ON Navigant2 TO Paul;`	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 Paul;`	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

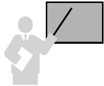
Le dictionnaire des données (*data dictionary*) est une partie majeure d'une base de données Oracle qu'on peut assimiler à une structure centralisée. Le dictionnaire est constitué d'un ensemble de tables système à partir desquelles sont définies environ six cents vues distinctes. Celles-ci stockent toutes 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.

© Éditions Eyrolles

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 5-34 Recherche du nom des vues du dictionnaire des données (à partir de TABLE_NAME)

Commande SQL	Résultat

`DESC DICTIONARY`	<pre>Nom NULL ? Type
	----------------- ------- ------------
	TABLE_NAME VARCHAR2(30)
	COMMENTS VARCHAR2(4000)</pre>

<pre>SELECT * FROM DICTIONARY
  WHERE table_name
  LIKE '%SEQUENCE%';</pre>
	<pre>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</pre>

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 5-35 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 5-36 Choix des colonnes d'une vue du dictionnaire des données

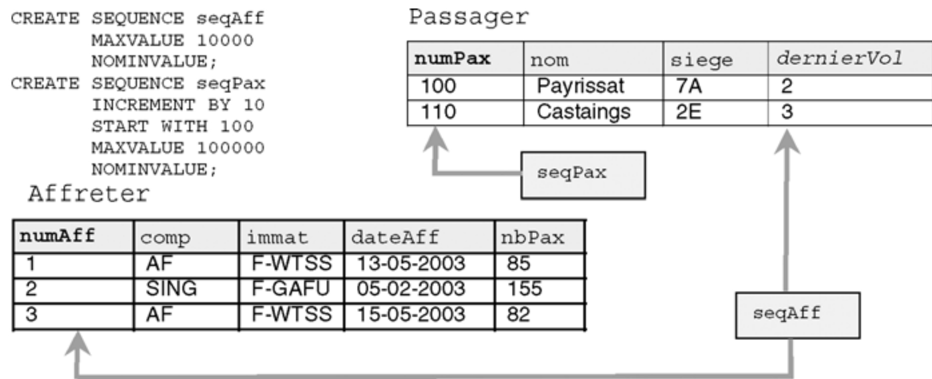
Commande SQL	Résultat
DESC USER_SEQUENCES	Nom NULL ? Type --------------------------- -------- ------------ SEQUENCE_NAME NOT NULL VARCHAR2(30) MIN_VALUE NUMBER MAX_VALUE NUMBER INCREMENT_BY NOT NULL NUMBER CYCLE_FLAG VARCHAR2(1) ORDER_FLAG VARCHAR2(1) CACHE_SIZE NOT NULL NUMBER LAST_NUMBER NOT NULL NUMBER

© Éditions Eyrolles

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 5-23 *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 5-37 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 5-38 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*.

© Éditions Eyrolles

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';
```


© Éditions Eyrolles

```
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
```

 © Éditions Eyrolles

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 5-39 Tarifs des postes

Type du poste	Tarif en F
TX	50
PCWS	100
PCNT	120
UNIX	200
NC	80
BeOS	400

© Éditions Eyrolles

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 = 300O). 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`, `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 :

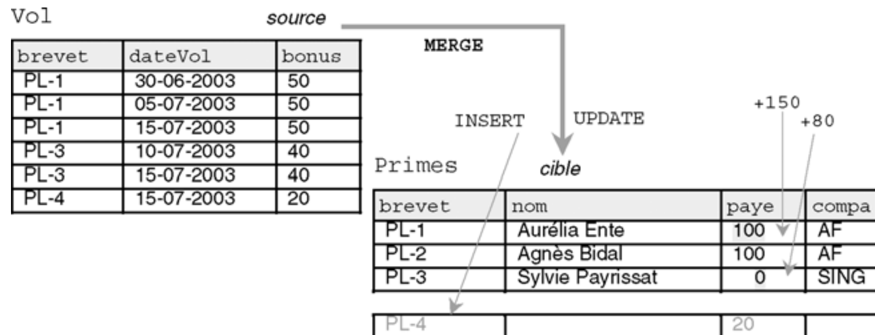
```
NOMSALLE         NOMPOSTE             ADRIP         NOMTYPEPOSTE
---------------- -------------------- ------------  ------------------
Salle 1          Poste 1              130.120.80.01 Terminal X-Window
Salle 1          Poste 2              130.120.80.02 Système Unix
…
```


© Éditions Eyrolles

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 5-24 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
...
```


© Éditions Eyrolles

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

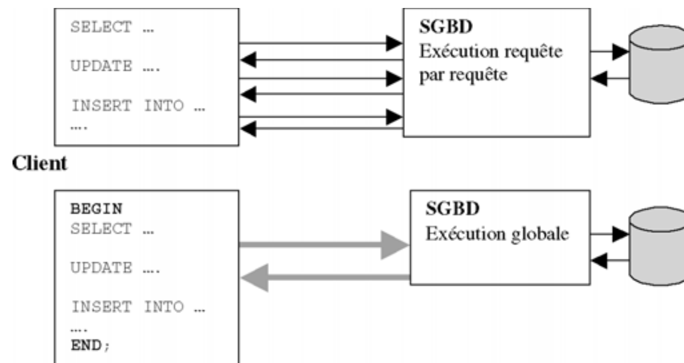
Les structures de contrôle habituelles d'un langage (`IF`, `WHILE`…) ne font pas partie intégrante de la norme SQL. Elles apparaissent dans une sous-partie optionnelle de la norme (ISO/IEC 9075-5:1996. *Flow-control statements*). Oracle les prend en compte dans PL/SQL. Nombre de concepts de PL/SQL proviennent 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 Trafic sur le réseau d'instructions SQL



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

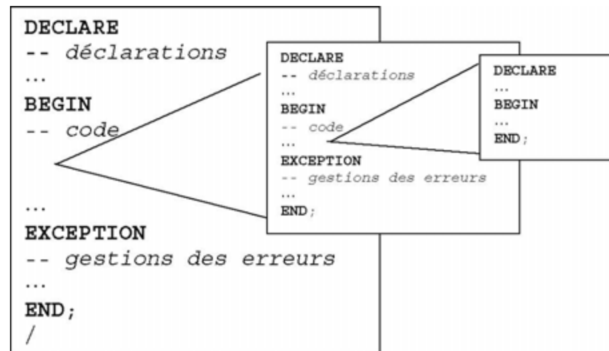
Un programme PL/SQL qui n'est pas nommé (aussi appelé bloc) est composé de trois sections comme le montre la figure suivante :

- `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.

 © Éditions Eyrolles

- `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

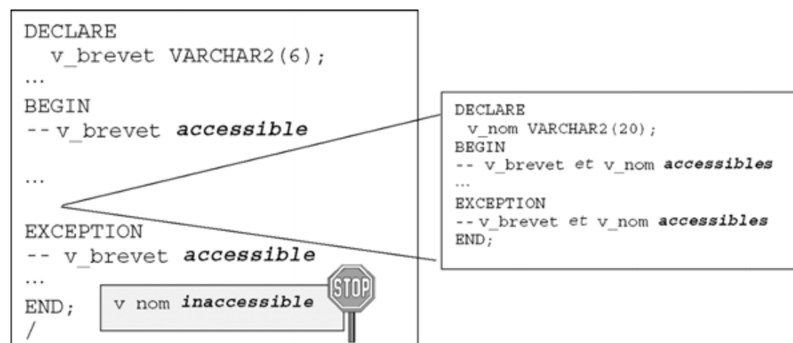


Portée des objets

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

<table>
<tr><th>Peu lisible</th><th>C'est mieux</th></tr>
<tr><td><pre>IF x>y THEN max:=x;ELSE max:=y;END IF;</pre></td><td><pre>IF x > y THEN
    max := x;
  ELSE
    max := y;
END IF;</pre></td></tr>
</table>

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`	

© Éditions Eyrolles

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 :

© Éditions Eyrolles

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.

© Éditions Eyrolles

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));
 	
  r_unA320   avionAirbus_rec;
  r_FGLFS   avionAirbus_rec;
BEGIN
	Déclaration du `RECORD` contenant quatre champs ; initialisation du champ `usine` par défaut.
	
	Déclaration de deux variables de type `RECORD`.

r_unA320.nserie   := 'A1';
r_unA320.nomAvion := 'A320-200';
r_unA320.nbHVol   := 2500.60;
r_FGLFS   := r_unA320;
…
	Initialisation des champs d'un `RECORD`.
	
	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));
```

© Éditions Eyrolles

```
  TYPE vols_rec IS RECORD
    (r_aéronef avionAirbus_rec , dateVol DATE,
     rty_coPilote Pilote%ROWTYPE, affretéPar 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 (*associative arrays*) permettent de définir et de manipuler des tableaux dynamiques (car définis sans dimension initiale). Un tableau est composé d'une clé primaire (de type BINARY_INTEGER) et d'une colonne (de type scalaire, TYPE, ROWTYPE ou RECORD) pour stocker chaque élément.

Syntaxe

La syntaxe générale pour déclarer un type de tableau et une variable tableau est la suivante :

```
TYPE nomTypeTableau IS TABLE OF
    {typeScalaire | variable%TYPE | table.colonne%TYPE} [NOT NULL]
    | table.%ROWTYPE
    [INDEX BY BINARY_INTEGER];
nomTableau nomTypeTableau;
```

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 7800). L'accès à des champs d'éléments complexes se fait à l'aide de la notation pointée (voir la dernière instruction).

 © Éditions Eyrolles

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

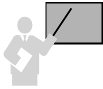
Préfixer les variables	Étiquette de bloc

`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.

© Éditions Eyrolles



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` (lecture de la variable au clavier) est détaillée dans le chapitre relatif à SQL*Plus. 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 (globales) définies sous SQL*Plus au niveau d'un bloc PL/SQL. 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*_**rec**	pilote_rec
Variable RECORD	**r**_*nomVariable*	r_pilote
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

© Éditions Eyrolles

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	Caractéristiques
CHARACTER	CHAR	Mêmes caractéristiques.
INTEGER	NUMBER(38,0)	Entiers sans décimales.
PLS_INTEGER	NUMBER	Déjà étudié.
BINARY_INTEGER	NUMBER	Déjà étudié.
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.

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.

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

<table>
<tr><th>Code PL/SQL</th><th>Commentaires</th></tr>
<tr><td>

```
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;
```

</td><td>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.</td></tr>
</table>

© Éditions Eyrolles

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).

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 ».

Nouveautés 11g

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` (-2147483648 à 2147483647), 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

<table>
<tr><th>Code SQL et PL/SQL</th><th>Exécution sous SQL*Plus</th></tr>
<tr><td><pre>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;
/</pre></td><td><pre>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</pre></td></tr>
</table>

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).

© Éditions Eyrolles

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.

© Éditions Eyrolles

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*

<table>
<tr><th>Condition simple</th><th>Condition composée</th></tr>
<tr><td><pre>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;</pre></td><td><pre>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;</pre></td></tr>
<tr><td>Somme = 5050</td><td>Trouvé 4 à l'indice : 11</td></tr>
</table>

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.

© Éditions Eyrolles

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.

© Éditions Eyrolles

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 ne faut pas modifier cette variable dans le corps de la boucle (ou alors si on veut modifier volontairement son pas).

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	Condition composée

<pre>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;</pre>
	<pre>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;</pre>

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	Code PL/SQL

`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);`
…
	`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;`

© Éditions Eyrolles

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…

Nouveauté 11g

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.

© Éditions Eyrolles

Si vous utilisez la directive `CONTINUE` dans une boucle `FOR` manipulant un curseur (étudié au chapitre 7), vous fermez automatiquement le curseur.

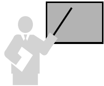
Interactions avec la base

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`.

© Éditions Eyrolles

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

<pre>DECLARE
 v_compa Pilote.compa%TYPE;
BEGIN
 SELECT compa INTO v_compa
        FROM Pilote;
END;</pre>
	<pre>DECLARE
	 rty_pilote Pilote%ROWTYPE;
	BEGIN
	 SELECT * INTO rty_pilote
	        FROM Pilote WHERE brevet = '$£*';
	END;</pre>

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

<pre>DECLARE
 v_nomEnMAJUSCULES Pilote.nom%TYPE;
BEGIN
 SELECT **UPPER**(nom)
 INTO v_nomEnMAJUSCULES
 FROM Pilote WHERE brevet = 'PL-1';
END;</pre>
	<pre>VARIABLE g_plusGrandHVol NUMBER;
	DECLARE
	BEGIN
	 SELECT **MAX**(nbHVol) INTO :g_plusGrandHVol
	  FROM Pilote;
	END;</pre>

© Éditions Eyrolles

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).

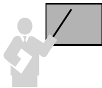
Dans le cas inverse, une exception qui précise la nature du problème est levée et peut être interceptée dans la section `EXCEPTION` (voir chapitre suivant). Si une telle partie n'existe pas dans le bloc de code qui contient l'instruction `INSERT`, la première exception fera s'interrompre le programme.

© Éditions Eyrolles

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

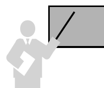
DECLARE
 v_duréeVol NUMBER(3,1) := 4.8;
BEGIN
 UPDATE Pilote
  SET nbHVol = nbHVol + v_duréeVol
  WHERE brevet= 'PL-6';
	Modification d'un enregistrement de la table Pilote en utilisant une variable.

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`).

 © Éditions Eyrolles

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 = 1,4142135623730950488016887242096980785 7 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.

 © Éditions Eyrolles

Tableau 6-40 Utilisation de GET_LINE

Code PL/SQL	Commentaires

<pre>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;</pre>
	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

<pre>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_LINE**(tab, v_resultat);</pre>
	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`	

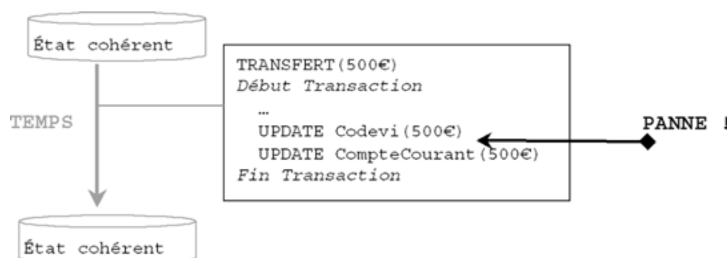
Transactions

Une transaction est un bloc d'instructions LMD faisant passer la base de données d'un état cohérent à un autre état cohérent. Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions contenues dans la transaction n'est effectuée, quel que soit l'endroit de la transaction où est intervenue l'erreur.

La grande majorité des transactions sous Oracle sont programmées en PL/SQL. Les langages plus évolués permettent de développer des transactions à travers des fonctions de leur API (la méthode `commit` est implémentée dans le paquetage `java.sql` par exemple).

Le modèle le plus simple et le plus frappant d'une transaction est celui du transfert d'un compte épargne vers un compte courant. Imaginez qu'après une panne votre compte épargne a été débité de la somme de 500 € sans que votre compte courant ait été crédité de ce même montant ! Vous ne seriez pas très content, sans doute, des services de votre banque. Le mécanisme transactionnel empêche cet épisode fâcheux en invalidant toutes les opérations faites depuis le début de la transaction si une panne survient au cours de cette même transaction.

Figure 6-4 *Transaction*



Caractéristiques

Une transaction assure :

- l'atomicité des instructions qui sont considérées comme une seule opération (principe du tout ou rien) ;
- la cohérence (passage d'un état cohérent de la base à un autre état cohérent) ;
- l'isolation des transactions entre elles (lecture consistante, mécanisme décrit plus loin) ;
- la durabilité des opérations (les mises à jour perdurent même si une panne se produit après la transaction).

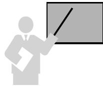
 © Éditions Eyrolles

D'autres transactions particulières existent, elles sont constituées par :

- un ordre SQL du LDD (CREATE, ALTER, DROP…) ;
- un ordre SQL du LCD (GRANT, REVOKE).

Début et fin d'une transaction

Il n'existe pas d'ordre PL/SQL ou SQL qui marque le début d'une transaction. Ainsi le BEGIN d'un programme PL/SQL n'est pas forcément synonyme de son commencement.



Une transaction débute à la première commande SQL rencontrée ou dès la fin de la transaction précédente.

Une transaction se termine explicitement par les instructions SQL COMMIT ou ROLLBACK. Elle se termine implicitement :

- à la première commande SQL du LDD ou du LCD rencontrée ;
- à la fin normale d'une session utilisateur avec déconnexion ;
- à la fin anormale d'une session utilisateur (sans déconnexion).

Le tableau suivant précise la validité de la transaction en fonction des événements possibles :

Tableau 6-42 Validité d'une transaction

Événement	Validité
COMMIT Commande SQL (LDD ou LCD) Fin normale d'une session.	Transaction validée.
ROLLBACK Fin anormale d'une session.	Transaction non validée.

Vous pouvez tester rapidement une partie de ces caractéristiques en écrivant le bloc suivant qui insère une ligne dans une de vos tables :

```
COMMIT;
BEGIN
   INSERT INTO TableàVous VALUES (…);
END;
/
SELECT * FROM TableàVous;
```

© Éditions Eyrolles

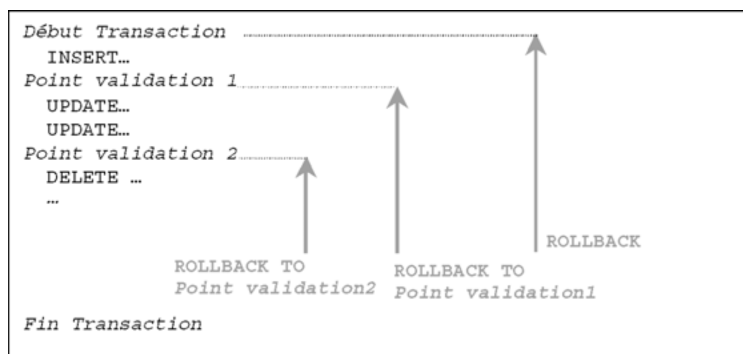
Exécutez ce bloc dans l'interface SQL*Plus, puis déconnectez-vous en cassant la fenêtre (icône en haut à droite). Reconnectez-vous et constatez que l'enregistrement n'est pas présent dans votre table. Relancez le bloc et sortez proprement de SQL*Plus avec `exit`. Reconnectez-vous et notez que l'enregistrement est présent désormais dans votre table. Comme M. Jourdain faisait de la prose, vous faisiez des transactions depuis peu sans le savoir.

Contrôle des transactions

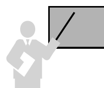
Il est intéressant de pouvoir découper une transaction en insérant des points de validation (*savepoints*) qui rendent possible l'annulation de tout ou partie des opérations d'une transaction.

La figure suivante illustre une transaction découpée en trois parties. L'instruction `ROLLBACK` peut s'écrire sous différentes formes. Ainsi `ROLLBACK TO Pointvalidation1` invalidera les `UPDATE` et le `DELETE` tout en laissant la possibilité de valider l'instruction `INSERT` (en fonction des commandes se trouvant après ce `ROLLBACK` restreint et de la manière dont la session se terminera).

Figure 6-5 *Points de validation*



Le tableau suivant décrit une transaction PL/SQL découpée en trois parties. Le programmeur aura le choix entre les instructions `ROLLBACK TO` indiquées en commentaire pour valider tout ou partie de la transaction. Il faudra finalement se décider entre `COMMIT` et `ROLLBACK` .



L'instruction PL/SQL `SAVEPOINT` déclare un point de validation.

Tableau 6-43 Transaction découpée

Code PL/SQL	Commentaires

`BEGIN`
`INSERT INTO Compagnie VALUES`
`('C2',2, 'Place Brassens', 'Blagnac', 'Easy Jet');`
	Première partie de la transaction.

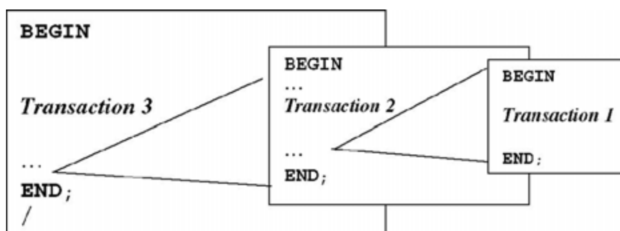
`SAVEPOINT P1;`
`UPDATE Compagnie SET nrue = 125`
`WHERE comp = 'AF';`
`UPDATE Compagnie SET ville = 'Castanet'`
`WHERE comp = 'C1';`
	Deuxième partie de la transaction.

`SAVEPOINT P2;` `DELETE FROM Compagnie WHERE comp = 'C1';`	Troisième partie de la transaction.
`-- ROLLBACK TO P1;`	Première partie à valider.
`-- ROLLBACK TO P2;`	Deuxième partie à valider.
`-- ROLLBACK TO P3`	Troisième partie à valider.
`ROLLBACK;`	Tout à invalider.
`COMMIT;` `END;`	Valide la ou les sous-parties.

Transactions imbriquées

Il est possible de programmer plusieurs transactions se déroulant dans des blocs imbriqués comme l'illustre la figure suivante. Les mécanismes d'atomicité, de cohérence, d'isolation et de durabilité sont aussi respectés.

Figure 6-6 Transactions imbriquées



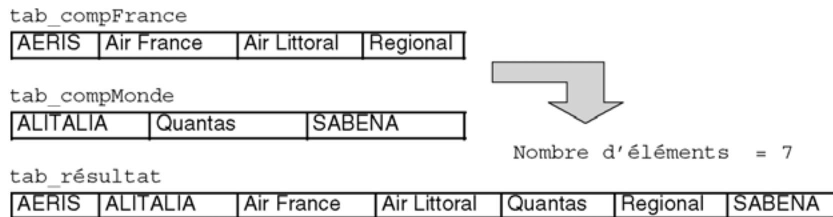
Exercices

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 6-7 *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
```


© Éditions Eyrolles

```
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        <-- Attente de 5 secondes à ce niveau
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.
```

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

Ce chapitre est consacré à des caractéristiques avancées du langage PL/SQL :

- définition de sous-programmes et de paquetages ;
- programmation des curseurs ;
- gestion des exceptions ;
- mise en place de déclencheurs ;
- utilisation du SQL dynamique.

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-utilisateurs.

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 :

© Éditions Eyrolles

- 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

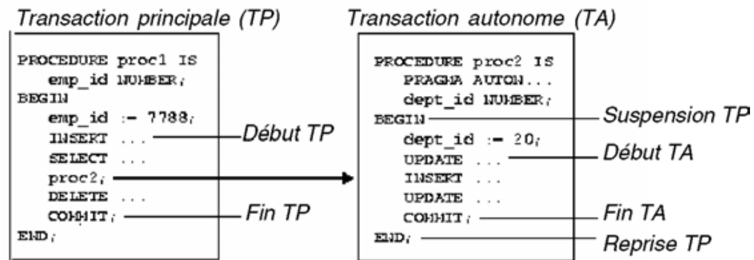
La syntaxe de création d'une procédure cataloguée est la suivante. Pour créer une procédure dans son propre schéma, le privilège `CREATE PROCEDURE` est requis (inclus dans le rôle `RESOURCE`). Pour créer une procédure dans un autre schéma, il faut posséder le privilège `CREATE ANY PROCEDURE`.

```
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 )] } };
```

 © Éditions Eyrolles

- 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
```

© Éditions Eyrolles

```
[ AUTHID { DEFINER | CURRENT_USER } ]
{IS | AS}
 { corpsduSousProgrammePL/SQL             |
   LANGUAGE {
   JAVA NAME 'nomMéthodeJava' |
   C [NAME nomSourceC] LIBRARY nomLibrairie [AGENT IN (paramètre)]
     [WITH CONTEXT] [PARAMETERS ( paramètres )] } };
```

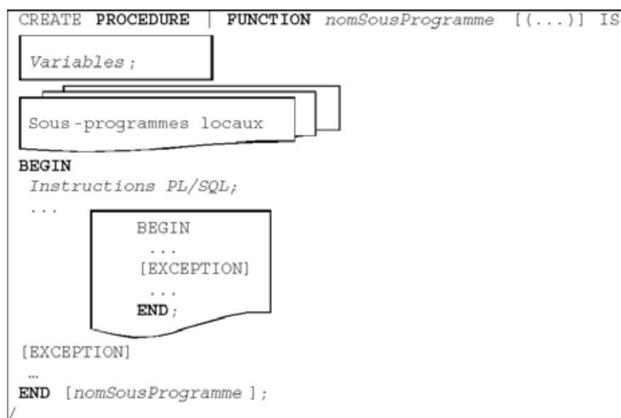
- `corpsduSousProgrammePL/SQL` contient la déclaration et les instructions de la fonction (il doit se trouver une instruction `RETURN` dans le code), toutes deux écrites en PL/SQL.

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 :

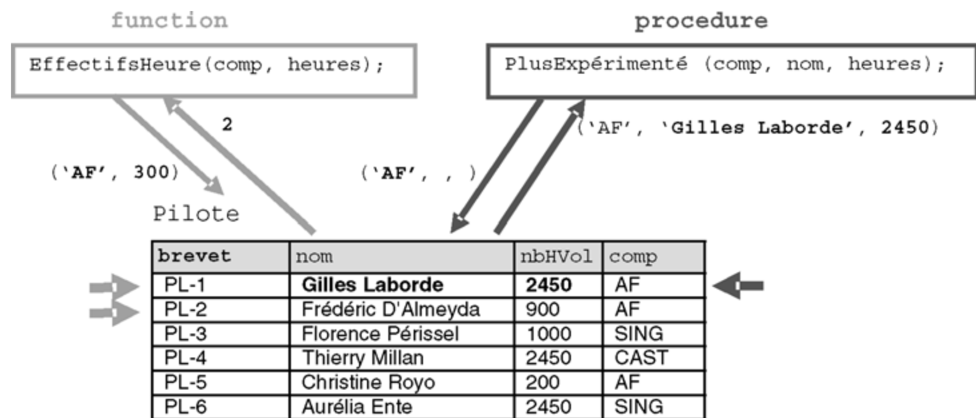
- la fonction `EffectifsHeure(comp,heures)` retourne le nombre de pilotes d'une compagnie donnée (premier paramètre) qui ont plus d'heures de vol que la valeur du

© Éditions Eyrolles

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*



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

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
   NUMBER)
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 ;
```


© Éditions Eyrolles

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.

© Éditions Eyrolles

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

<table>
<tr><th>Code PL/SQL</th><th>Commentaire</th></tr>
<tr><td><pre>DECLARE
 n NUMBER := 10;
BEGIN
 changeEtAffiche(n, n, n);
 DBMS_OUTPUT.PUT_LINE(n);
END;
/</pre></td><td>Affiche 20.</td></tr>
<tr><td><pre>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;</pre></td><td>Affiche 10.
Affiche 30.</td></tr>
<tr><td><pre>10
30
20
Procédure PL/SQL terminée avec succès.</pre></td><td>Résultat obtenu.</td></tr>
</table>

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.

 © Éditions Eyrolles

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.

© Éditions Eyrolles

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`).

© Éditions Eyrolles

Figure 7-4 Structure d'un paquetage

```
CREATE PACKAGE nomPaquetage AS
  PROCEDURE p2(...);
  FUNCTION f1(...) RETURN ...;
  Variables
  Exceptions
  ...                                   Public
END [nomPaquetage];              Spécification

CREATE PACKAGE BODY nomPaquetage AS
  PROCEDURE p1(...) IS
   BEGIN
    ...
  END p1;
  varPrive NUMBER;                Privé
  ...

  PROCEDURE p2(...) IS
   BEGIN
    ...
  END p2;
  FUNCTION f1(...) RETURN ... IS
   BEGIN
    ...
  END f1;
END [nomPaquetage];             Implémentation
```

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;
```


© Éditions Eyrolles

```
  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;
```

Curseurs

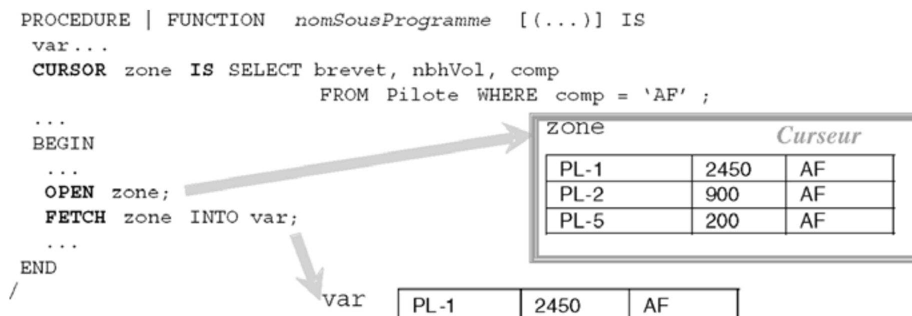
Au chapitre précédent nous avons parlé des curseurs implicites, ici nous allons étudier les curseurs explicites (que les programmeurs appellent *curseurs* tout simplement). Ils sont très utilisés, pour ne pas dire qu'ils sont présents, dans toute procédure d'une application importante. Le concept analogue 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 suivant :

© Éditions Eyrolles

Tableau 7-8 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 7-9 Parcours d'un curseur

<table>
<tr><th>Tant que</th><th>Répéter</th></tr>
<tr><td colspan="2"><pre>
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;
</pre></td></tr>
<tr><td><pre>
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;
…
</pre></td><td><pre>
BEGIN
 OPEN zone1;
 LOOP
   FETCH zone1 INTO var1,var2,var3;
   EXIT WHEN zone1%NOTFOUND;
   totalHeures := totalHeures + var2;
 END LOOP;
 CLOSE zone1;
…
</pre></td></tr>
</table>

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 7-10 Utilisation d'une variable structurée

<table>
<tr><th>Code PL/SQL</th><th>Commentaires</th></tr>
<tr><td><pre>
DECLARE
 CURSOR zone2 IS SELECT brevet, nom
              FROM Pilote WHERE NOT (comp='AF');
 enreg zone2%ROWTYPE;
</pre></td><td>Déclaration de la structure.</td></tr>
<tr><td><pre>
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;
/
</pre></td><td>Chargement de la structure.

Accès aux éléments de la structure.</td></tr>
<tr><td><pre>
nom : Florence Périssel (PL-3)
nom : Thierry Millan (PL-4)
nom : Aurélia Ente (PL-6)

Procédure PL/SQL terminée avec succès.
</pre></td><td>Résultat.</td></tr>
</table>

© Éditions Eyrolles

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 7-11 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 7-12 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 7-13 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 7-14 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.

© Éditions Eyrolles

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 7-15 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 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.

© Éditions Eyrolles

Tableau 7-16 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'.

© Éditions Eyrolles

Tableau 7-17 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 7-18 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.

 © Éditions Eyrolles

Tableau 7-18 Curseur typé *(suite)*

Code PL/SQL	Commentaires
<pre> **OPEN** zone7 **FOR** SELECT * FROM Pilote WHERE comp = 'AF'; FETCH zone7 INTO enreg; … CLOSE zone7; END; /</pre>	Autre chargement du curseur dynamique.

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.

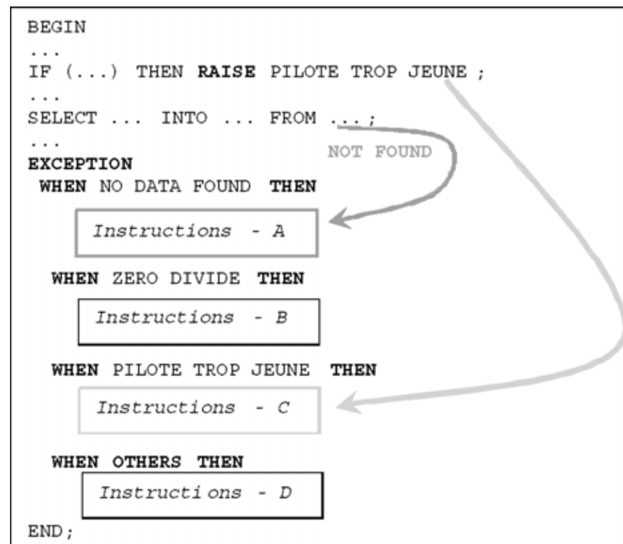
Généralités

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`).
- 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).

Figure 7-6 Principe général des exceptions



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).

© Éditions Eyrolles

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 7-19 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 -6592).

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 7-20 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');
```


© Éditions Eyrolles

```
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 ne se produit, en l'absence de la directive OTHERS dans le bloc d'exceptions, le programme se terminerait 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 7-21 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 7-21 Une exception traitée pour deux instructions

<table>
<tr><th>Code PL/SQL</th><th>Commentaires</th></tr>
<tr><td>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);</td><td>Requêtes déclenchant potentiellement une exception prévue.</td></tr>
<tr><td> requete := 2;
 SELECT nom INTO var1 FROM Pilote
           WHERE nbHVol = p_heures;
 DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' ||
           p_heures || ' heures est ' || var1);</td><td></td></tr>
<tr><td>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;</td><td>Aucun résultat.
Traitement pour savoir quelle requête a déclenché l'exception.</td></tr>
<tr><td> WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||
        SQLERRM || ' (' || SQLCODE || ')');
END;</td><td>Autre erreur.</td></tr>
</table>

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)
```


© Éditions Eyrolles

Tableau 7-22 Bloc d'exceptions imbriqué

<table>
<tr><th>Code PL/SQL</th><th>Commentaires</th></tr>
<tr><td><pre>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);
 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;</pre></td><td>Bloc imbriqué.

Gestion des exceptions de la première requête.</td></tr>
<tr><td><pre>SELECT nom INTO var1 FROM Pilote
          WHERE nbHVol = p_heures ;
 DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' || p_heures ||
            ' heures est ' || var1);</pre></td><td>Suite du traitement.</td></tr>
<tr><td><pre>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;</pre></td><td>Gestion des exceptions de la deuxième requête.</td></tr>
</table>

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 7-23 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 7-24 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, 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

© Éditions Eyrolles

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 7-25 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 || ')');`
`END;`
	Gestion des autres exceptions.

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

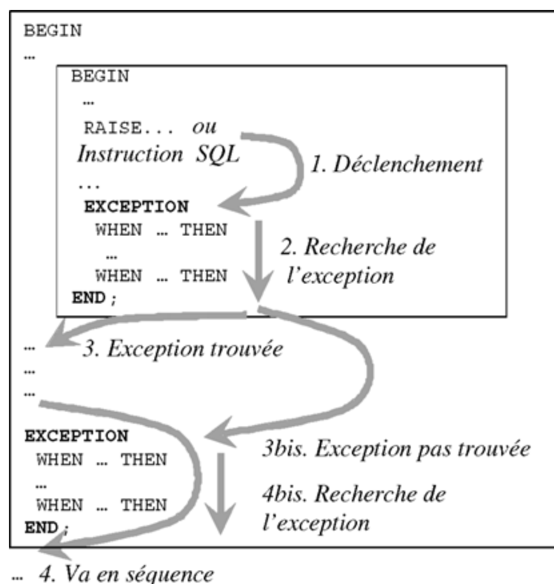
Nous avons vu jusqu'à présent que lorsqu'un bloc `EXCEPTION` traite correctement une exception (car il existe soit une entrée dans le bloc correspondant à l'exception, soit l'entrée `OTHERS`), l'exécution du traitement se poursuit en séquences après l'instruction `END` du bloc `EXCEPTION`.

© Éditions Eyrolles

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*



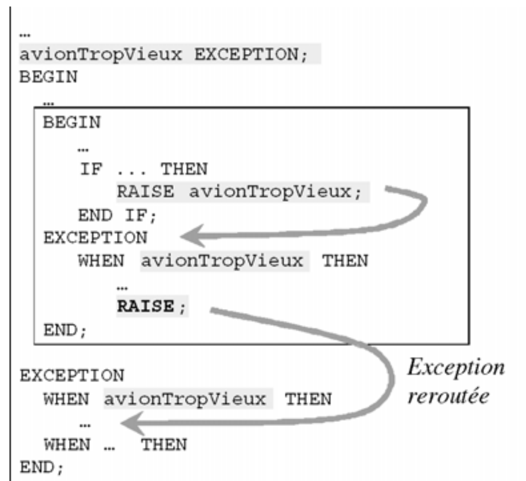
Notez que lorsque l'exception se propage à un bloc englobant, les actions exécutables restantes de ce bloc sont ignorées. Un des avantages de ce mécanisme est de pouvoir gérer des exceptions spécifiques dans leur propre bloc, tout en laissant le bloc englobant gérer les exceptions plus générales.

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).

 © Éditions Eyrolles

Figure 7-9 *Exception reroutée*



Procédure RAISE_APPLICATION ERROR

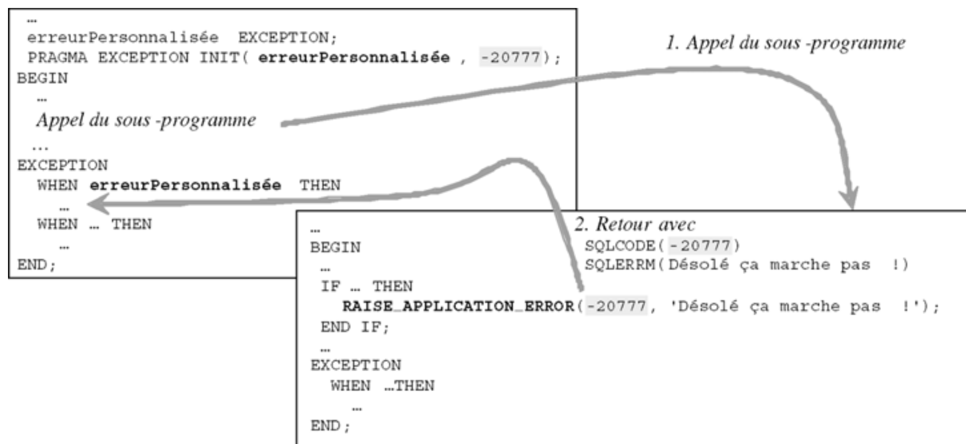
La procédure `RAISE_APPLICATION_ERROR` permet de définir ses propres messages et codes d'erreurs. Cette procédure évite le renvoi d'exceptions non traitées car le numéro d'erreur (inclus dans `RAISE_APPLICATION_ERROR`) sera communiqué à l'environnement appelant.

```
RAISE_APPLICATION_ERROR(numéroErreur, message [, {TRUE | FALSE}]);
```

- *`numéroErreur`* : valeur définie par 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*

© Éditions Eyrolles

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

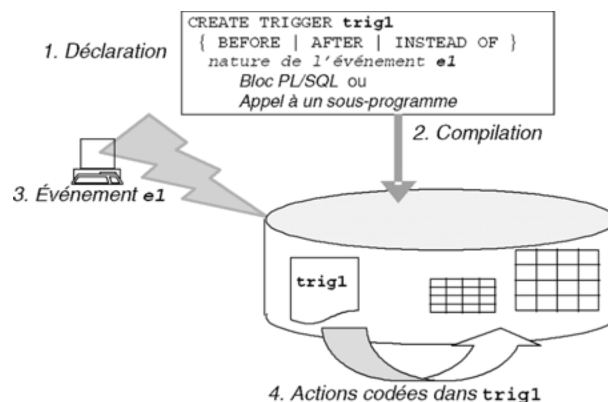
Les événements déclencheurs peuvent être :

- une instruction `INSERT`, `UPDATE`, ou `DELETE` sur une table (ou une vue). On parle de déclencheurs LMD ;
- une instruction `CREATE`, `ALTER`, ou `DROP` 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 (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

© Éditions Eyrolles

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 :

© Éditions Eyrolles

Tableau 7-26 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)

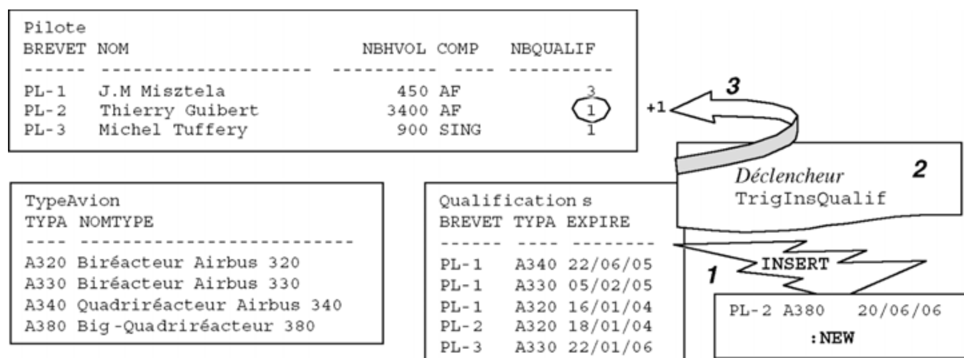
Un déclencheur de lignes est déclaré avec la directive `FOR EACH ROW`. Ce n'est que dans ce type de déclencheur qu'on a accès aux anciennes valeurs et aux nouvelles valeurs des colonnes de la ligne affectée par la mise à jour prévue par l'événement.

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.

 © Éditions Eyrolles

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 7-27 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!');`
` END IF;`
`END;`
`/`
	Renvoi d'une erreur utilisateur.

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 7-28 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 7-29 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.
	
	Renvoi d'une erreur utilisateur.
	
	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 7-30 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'`

© Éditions Eyrolles

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 INSERT` 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 7-31 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 7-32 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`

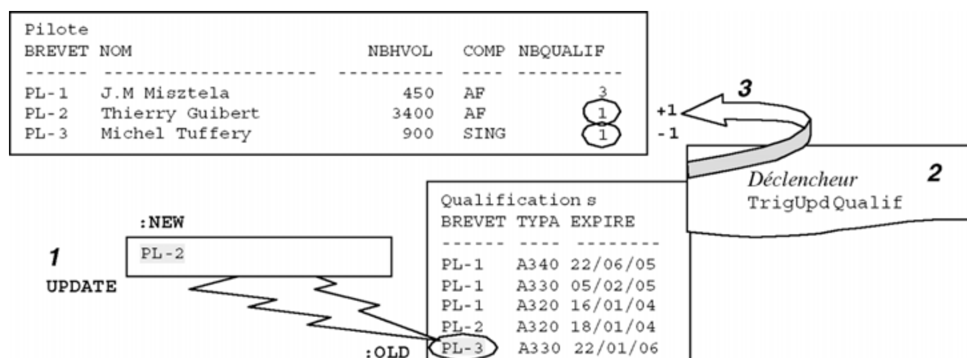
Pour tester le fait que l'instruction `UPDATE` n'affecte aucune ligne, il faudrait utiliser un curseur implicite (`SQL%FOUND`) et une erreur utilisateur (voir le paragraphe « Utilisation du curseur implicite » dans la section « Exceptions »).

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.

© Éditions Eyrolles

Tableau 7-33 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;`
` ELSE`
	Mise à jour des pilotes concernés par la modification de la qualification.

`    RAISE_APPLICATION_ERROR(-20100, 'Le pilote ' ||`
`      :NEW.brevet || ' a déjà 3 qualifications!');`
` END IF;`
`EXCEPTION`
` WHEN NO_DATA_FOUND THEN`
	Renvoi d'une erreur utilisateur.

`    RAISE_APPLICATION_ERROR(-20101, 'Pas de pilote`
`                 de code brevet ' || :NEW.brevet);`
` WHEN OTHERS THEN`
	Renvoi d'une erreur utilisateur.

`   RAISE;`
`END;`
`/`
	Retour de l'erreur courante.

Tableau 7-34 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 7-35 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 7-36 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 7-37 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

© Éditions Eyrolles

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 7-38 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.

© Éditions Eyrolles

Tableau 7-39 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 dans 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 7-40 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…) :

© Éditions Eyrolles

Tableau 7-41 Test du déclencheur

Événements déclencheurs	Sortie SQL*Plus
UPDATE Qualifications SET … INSERT INTO Qualifications VALUES… DELETE FROM Qualifications…	ERREUR à la ligne 1 : **ORA-20102**: Désolé pas de mises à jour des qualifs le week-end. ORA-06512: à "SOUTOU.PÉRIODEOKQUALIFS", ligne 3 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 (`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.

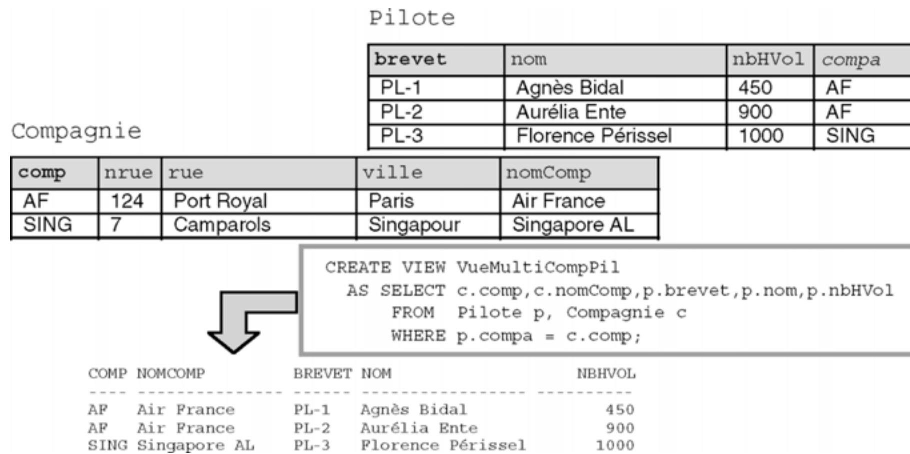
Il n'est pas possible de spécifier une liste de colonnes dans un déclencheur `INSTEAD OF UPDATE`, le déclencheur s'exécutera quelle que soit la colonne modifiée.

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 7-42 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;`
`IF v_pil = 0 THEN`
	Ajout dans la table `Compagnie`.

`INSERT INTO Pilote VALUES`
`(:NEW.brevet,:NEW.nom,:NEW.nbHVol,:NEW.comp);`
`END IF;`
`END IF;`
`END;`
`/`
	Ajout dans la table `Pilote`.

 © Éditions Eyrolles

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 7-43 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 7-44 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

© Éditions Eyrolles

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 7-45 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 7-46 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 7-47 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.

 © Éditions Eyrolles

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 7-48 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.

Nouveautés 11g

Trois nouvelles fonctionnalités de déclencheurs sont présentes avec la version 11*g*.

Activation et désactivation

Nous avons précédemment qu'il était possible de désactiver puis de réactiver des déclencheurs à l'aide des commandes ALTER TRIGGER et ALTER TABLE. Avant la version 11*g*, tout déclencheur créé était de fait actif (*enable*).

Il est désormais possible à l'aide de la directive DISABLE de déclarer un déclencheur en le désactivant dès sa création. En l'absence de cette directive ou en présence de la directive ENABLE, tout déclencheur est 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
```


© Éditions Eyrolles

```
  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';
```

 © Éditions Eyrolles

```
  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`.

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 7-49 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 (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).

© Éditions Eyrolles

Les instructions suivantes ne peuvent pas être prises en compte par un ordre SQL dynamique : `CLOSE`, `DECLARE`, `DESCRIBE`, `EXECUTE`, `FETCH`, `OPEN`, `PREPARE`, `SET`, `WHENEVER`.

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 7-50 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`.

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 7-51 Utilisations de EXECUTE IMMEDIATE

<table>
<tr><th>Code PL/SQL</th><th>Commentaires</th></tr>
<tr><td><pre>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';</pre></td><td>Déclaration des variables.</td></tr>
<tr><td><pre>BEGIN
 EXECUTE IMMEDIATE 'DELETE FROM Avion
        WHERE nbHVol > 1000 ';
 EXECUTE IMMEDIATE 'CREATE TABLE AvionChasse
      (immat VARCHAR2(6), prixEuros NUMBER)';</pre></td><td>Famille 1, sans paramètre.</td></tr>
<tr><td><pre> 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;</pre></td><td>Famille 2.
Insertion paramétrée.

Appel du sous-programme <code>sousProg</code>.</td></tr>
<tr><td><pre> ordreSQLdynamique := 'SELECT * FROM
                 Pilote WHERE brevet= :v1';
 EXECUTE IMMEDIATE ordreSQLdynamique INTO
               pilote_record USING v_brevet;
END;
/</pre></td><td>Famille 3.
Extraction monoligne paramétrée.</td></tr>
</table>

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.

© Éditions Eyrolles

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.

Tableau 7-52 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
	Famille 4, requête multiligne paramétrée. Passage du paramètre.

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;
/
	Parcours du curseur.

© Éditions Eyrolles

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 `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`) ;

© Éditions Eyrolles

- 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*".

© Éditions Eyrolles

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*).

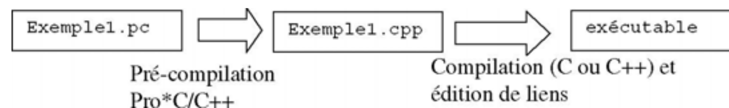
Il est possible d'intégrer le précompilateur Pro*C/C++ dans Microsoft Visual C++, de manière à précompiler, à compiler et à exécuter dans le même environnement de développement. La dernière section de ce chapitre traite de la configuration à mettre en œuvre.

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

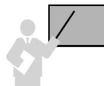
`BEGIN DECLARE SECTION`
…
`END DECLARE SECTION`
	Déclaration des variables hôtes (scalaires ou tableaux).

`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;`
	Curseur paramétré.

… `EXEC SQL FETCH curs INTO :tabNomcomp;`	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.

© Éditions Eyrolles

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	Commentaire
`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.

© Éditions Eyrolles

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;`	Préparation du déroutement en cas d'erreur.
`EXEC SQL CONNECT :utilisateur IDENTIFIED BY :pwd USING :descripteur;`	Connexion à la base.
`EXEC SQL WHENEVER NOT FOUND GOTO pasTrouve;`	Gestion de l'exception.
`EXEC SQL SELECT typeAvion, cap, comp INTO :typeav, :capacite, :codecomp FROM Avion WHERE immat = 'F-WTSS';`	Extraction d'un enregistrement.
`typeav.arr[typeav.len]     = '\0';` `codecomp.arr[codecomp.len] = '\0';`	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;`
 	
`EXEC SQL WHENEVER SQLERROR GOTO probleme;`
	Connexion.

`EXEC SQL INSERT INTO` `Compagnie` **`VALUES`** `('BAW', 'British Airways');`	Insertion.
`EXEC SQL COMMIT WORK;` `return;`	Validation.

`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`).

© Éditions Eyrolles

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.

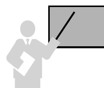
`finBoucle:`
` EXEC SQL CLOSE curs;`
` return;`
	Fin de curseur.

`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.

 © Éditions Eyrolles

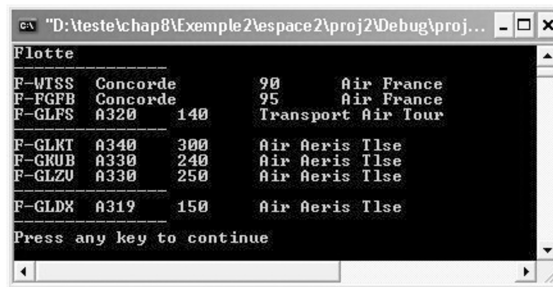
Tableau 8-12 Extraction dans des tableaux *(suite)*

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++

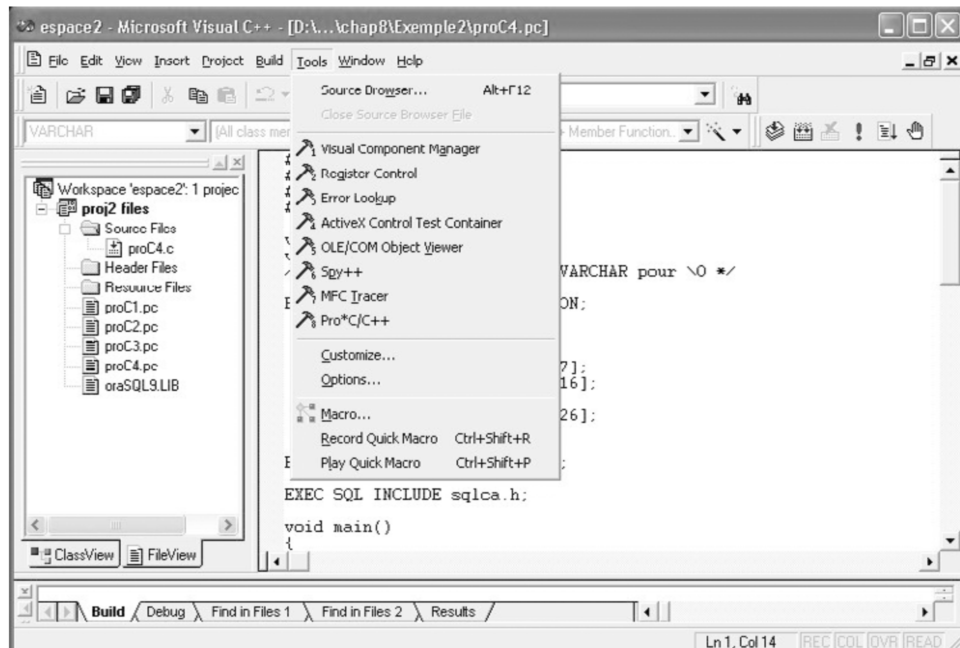
Afin de travailler avec Microsoft Visual C++, vous devez éventuellement avoir installé le précompilateur Pro*C/C++ en lançant à nouveau une installation à partir des extensions d'Oracle. La documentation à consulter est *Pro*C/C++ Precompiler Getting Started for Windows*, chapitre « Integrating Pro*C/C++ 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++*



 © Éditions Eyrolles

Chapitre 9

L'interface JDBC

L'interface JDBC (*Java DataBase Connectivity*) de Sun, appelée aussi « passerelle » ou « API », est composée d'un ensemble de classes permettant le dialogue entre une application Java et une source de données compatibles SQL (tables relationnelles en général). Ce chapitre détaille les caractéristiques de cette interface en incluant les nouvelles fonctionnalités de JDBC 3.0 (JDK 1.4).

Généralités

L'interface JDBC est conforme au niveau d'entrée de la norme SQL2 (*entry level*) et 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'applicatif Java peut être une classe ou une *applet* côté client, une *servlet*, un EJB (*Enterprise Java Beans*) ou une procédure cataloguée côté serveur.

Classification des pilotes (drivers)

Un pilote (*driver*) JDBC est une couche logicielle chargée d'assurer la liaison entre l'application Java (cliente) et le SGBD (serveur). Le site Web de Sun permet de télécharger des pilotes (voir annexe). La classification par Sun des pilotes JDBC distingue quatre types :

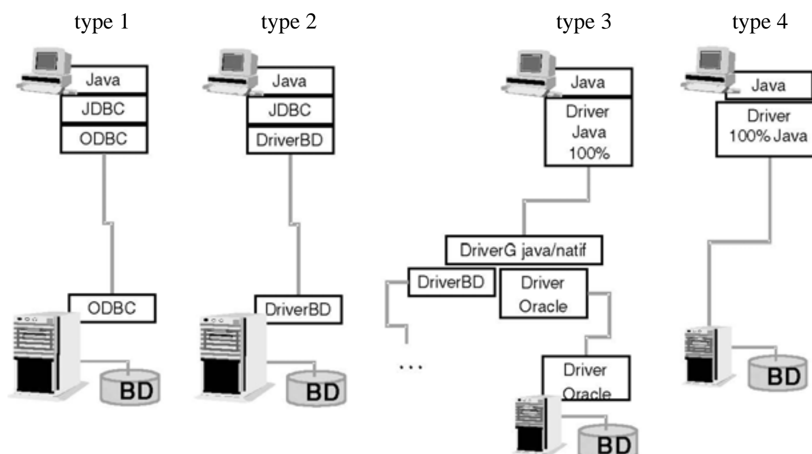
- Les pilotes de type 1 (*JDBC-ODBC Bridge*) utilisent la couche logicielle de Microsoft appelée ODBC (*Open DataBase Connectivity*). Le client est dit « épais » puisque le pilote JDBC convertit les appels Java en appels ODBC avant de les exécuter. Cette approche convient bien pour des sources de données 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*

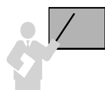


Le choix du pilote n'a pas d'influence majeure sur la programmation. Seules les phases de chargement du pilote et de connexion aux bases sont spécifiques, les autres instructions sont indépendantes du pilote. En d'autres termes, si vous avez une application déjà écrite et que vous décidez de changer 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

La version 3.0 de JDBC est composée de classes et d'interfaces situées dans le paquetage `java.sql`. Oracle propose également une API propriétaire (qui redéfinit et étend celle de Sun).

© Éditions Eyrolles



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 détaille la composition du paquetage `java.sql` de JDBC (3.0) de Sun :

Tableau 9-1 API JDBC 3.0 standard

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.

Le tableau suivant détaille la composition du paquetage `oracle.sql` de l'API d'Oracle :

Tableau 9-2 API JDBC 3.0 d'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.

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.

© Éditions Eyrolles

- La variable `PATH` doit contenir le chemin de la machine virtuelle Java pour compiler et exécuter des classes. Le JDK est en général installé dans `C:\j2sdk1.4.0`, les fichiers `javac` et `java` se trouve dans le sous-répertoire `bin`.
- La variable `CLASSPATH` doit inclure le paquetage JDBC pour Oracle à utiliser en fonction du pilote choisi par l'application *`Oracle_Home`*`\jdbc\lib\`*`paquetage`*. Le tableau suivant précise la configuration minimum à mettre en œuvre (nous utiliserons la dernière).

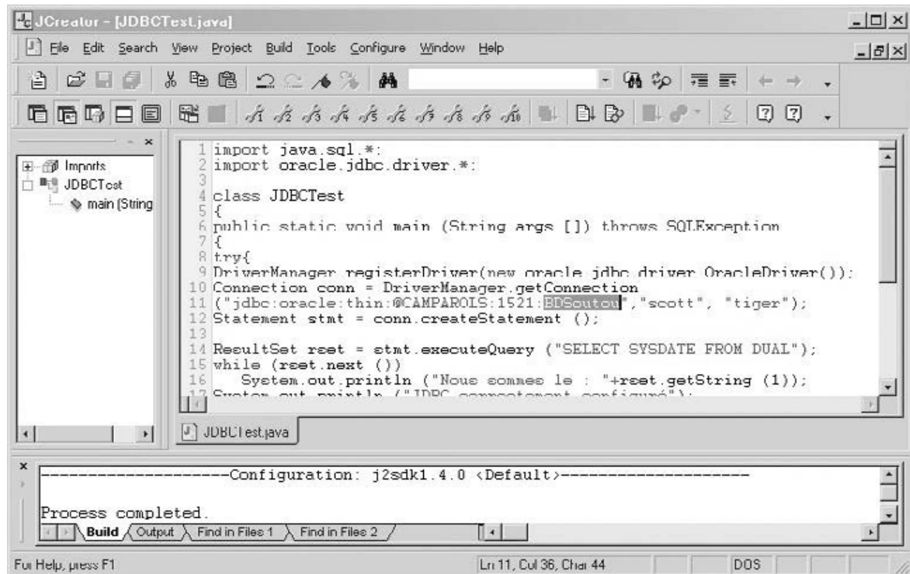
Tableau 9-4 Paquetages Oracle JDBC

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).

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 `Oracle_Home\jdbc\lib\ojdbc14.jar.`

Figure 9-2 *Interface JCreator*

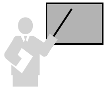


Cet exemple décrit le code nécessaire à la connexion à votre base (il faudra modifier le nom de la base, le nom et le mot de passe de l'utilisateur dans l'instruction surlignée de la figure précédente) et doit renvoyer les messages suivants :

```
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);
```

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.

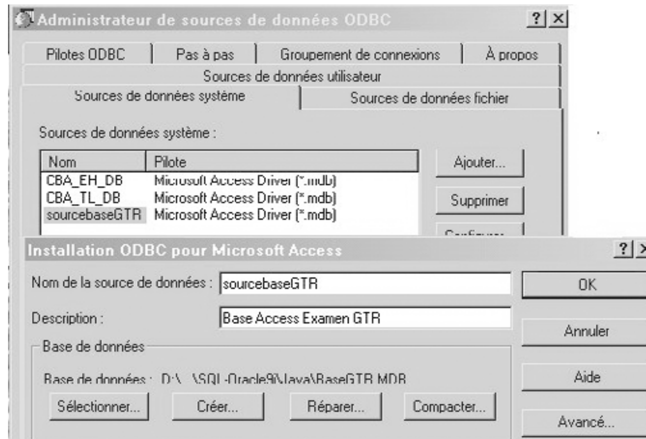
- *`protocole`* prend la valeur « `jdbc` » pour une connexion JDBC.
- *`sousProtocole`* indique la nature du pilote (« `odbc` » pour un pilote de type 1, « `oracle:thin` » pour un pilote Oracle de type 4, « `oracle:oci` » pour un pilote Oracle de type 2).
- *`infoConnexion`* donne les paramètres qui 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` :

© Éditions Eyrolles

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`
 	
` {try {Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");}`
`  catch (ClassNotFoundException ex)`
`      { System.out.println ("Problème au chargement"); }`
`  try {Connection conn = DriverManager.getConnection`
`                    ("jdbc:odbc:`**`sourcebaseGTR`**`","","");`
`  …     }`
	Classe ayant une méthode `main`.
	
	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);
```


© Éditions Eyrolles

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.

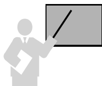
Si vous avez noté, lors de l'installation, le port UDP d'écoute du *listener* (en général 1521), le nom du service (nom de votre base), et si vous connaissez le nom du serveur, vous pouvez vous connecter sans problème a priori.

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","ingres");`	Connexion explicite.
`cx2 = DriverManager.getConnection(lienBD,"soutou","ingres");`	Connexion implicite.
`cx3 = DriverManager.getConnection ("jdbc:oracle:thin:@192.168.4.118:1521:BDSoutou", "soutou","ingres");`	Connexion avec adresse IP.
`cx4 = DriverManager.getConnection ("jdbc:oracle:thin:@camparols:1521:BDSoutou", "soutou","ingres");`	Connexion avec nom du serveur.
`… } 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.

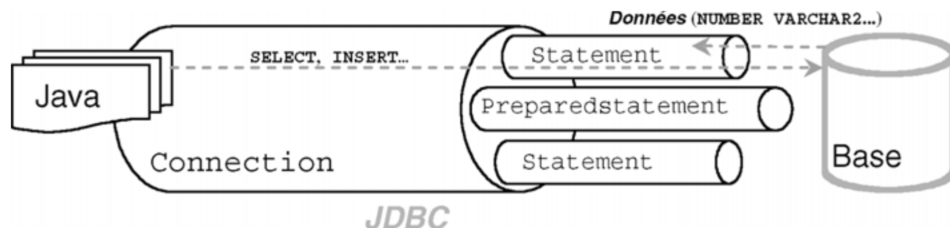
Tableau 9-9 Méthodes de l'interface Connection

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.

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.

 © Éditions Eyrolles

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) :

- `set`*`xxx`* 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) ;
- `get`*`xxx`* (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*) ;
- `update`*`xxx`* (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 9-10 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 9-11 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 9-12 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`

© Éditions Eyrolles

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 9-13 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 9-14 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
reference	REF	java.sql.Ref	oracle.sql.REF
collection	ARRAY	java.sql.Array	oracle.sql.ARRAY

Tableau 9-15 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 9-16 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 9-17 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.

© Éditions Eyrolles

Ajout d'enregistrements

Tableau 9-18 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.

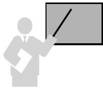
Modification d'enregistrements

Tableau 9-19 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 9-20 Méthodes principales de l'interface ResulSet

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 9-21 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.

 © Éditions Eyrolles

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 :`

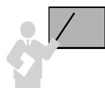
```
Statement createStatement(int typeCurseur, int modifCurseur)
```

Constantes

Les valeurs permises du premier paramètre (*typeCurseur*), et qui concernent le sens de parcours, sont présentées dans le tableau suivant :

Tableau 9-22 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 9-23 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

 © Éditions Eyrolles

pas positionné sur le premier enregistrement, et la méthode `next` le place selon le sens du parcours du curseur.

Tableau 9-24 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*

Avion

	immat	typeAvion	cap	comp
absolute(1) →	F-WTSS	Concorde	90	AF
	F-FGFB	Concorde	95	AF
relative(2) →	F-GLFS	A320	140	TAT
	F-GLKT	A340	300	AERI
	F-GKUB	A330	240	AERI
absolute(-1) →	F-GLZV	A330	250	AERI

curseurPosJava

Le code suivant (`SELECTPositions.java`) présente les méthodes qui permettent d'accéder directement à des enregistrements de ce curseur :

Tableau 9-25 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.

© Éditions Eyrolles

Tableau 9-25 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 « `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 9-26 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` :

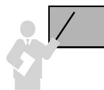
- Une requête ne doit pas contenir de jointure ni de regroupement.
- Écrivez « `SELECT a.* FROM` *table* `a…` » à la place de « `SELECT * FROM` *table*`…` » ;
- Une requête doit seulement extraire des colonnes (les fonctions monolignes et multilignes sont interdites).

© Éditions Eyrolles

Les principales méthodes relatives aux curseurs modifiables sont les suivantes :

Tableau 9-27 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*

immat	typeAvion	cap	comp
F-WTSS	Concorde	90	AF
F-FGFB	Concorde	95	AF
F-GLFS	A320	140	TAT
F-GLKT	A340	300	AERI
F-GKUB	A330→**A380**	240→**350**	AERI
F-GLZV	A330	250	AERI
F-LUTE	TB20	4	NULL

 © Éditions Eyrolles

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 9-28 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 9-29 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 9-30 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

© Éditions Eyrolles

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 9-31 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).

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 9-32 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 9-33 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 9-34 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 9-35 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`

© Éditions Eyrolles

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 9-36 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 9-37 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 9-38 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 9-39 Insertion 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.

© Éditions Eyrolles

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 `set`*`xxx`*. Les paramètres de sortie (définis `OUT` au niveau du sous-programme) sont extraits à l'aide des méthodes `get`*`xxx`*.

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 9-40 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 9-41 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 `get`*`xxx`*.

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 9-42 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 :

© Éditions Eyrolles

```
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 9-43 Appel d'une procédure

Code Java	Commentaires

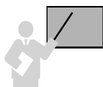
`try { …`
`String ordreSQL = "{call AugmenteCapacité(?,?)}";`
`CallableStatement étatAppelable =`
`cx.prepareCall(ordreSQL);`
	Création d'un état appelable.

`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 9-44 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 9-45 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.

 © Éditions Eyrolles

Tableau 9-46 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 9-47 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 9-48 Affichage des erreurs

Code Java	Commentaires

`import java.sql.*;`
`import oracle.jdbc.driver.*;`
`class Exceptions1`
`{public static void main(String args [])`
`throws SQLException`
`{try{`
`DriverManager.registerDriver(…);`
`Connection cx = DriverManager.getConnection(…);`
`…}`
	Classe principale.
	
	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.

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.

© Éditions Eyrolles

Tableau 9-49 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.

`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!");`
` else if (ex.getErrorCode() == **2291**)`
`     try {Connection cx =`
`        DriverManager.getConnection(……);`
`        cx.setAutoCommit(false);`
`        String ordreSQL2 = "INSERT INTO Compagnie`
`            VALUES ('NEW','Nouvelle Compagnie')";`
`       PreparedStatement étatPréparé2 =`
`            cx.prepareStatement(ordreSQL2);`
`      System.out.println(étatPréparé2.executeUpdate() +`
`            " compagnie insérée.");`
`       étatPréparé2 = cx.prepareStatement(ordreSQL);`
`      System.out.println(étatPréparé2.executeUpdate() +`
`           " avion inséré.");`
`       cx.commit();`
`       cx.close(); }`
`     catch (SQLException e)`
`       {System.err.println("Erreur : " + e); }`
`}`
	Gestion des erreurs.
	
	Clé étrangère absente
	
	Insertion d'une compagnie.
	
	Insertion d'un avion.
	
	Validation.
	
	Gestion des erreurs.

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.

 © Éditions Eyrolles

Chapitre 10

L'approche SQLJ

La technologie SQLJ (norme ISO) permet d'intégrer du code SQL dans un programme Java. Oracle est conforme aux spécifications de la norme. Alors que celle-ci inclut seulement des aspects statiques de SQL, nous verrons en fin de chapitre qu'Oracle dispose d'extensions pour programmer des instructions dynamiques.

Généralités

Les programmes SQLJ sont traduits en classes Java par l'intermédiaire d'un précompilateur (*Oracle SQLJ translator*) : classes qu'il faut compiler avant qu'une machine virtuelle puisse les interpréter.

Blocs SQLJ

Le précompilateur analyse une source d'extension `sqlj` qui est écrit comme une classe Java intégrant des instructions SQL à l'intérieur de blocs – entre accolades et préfixées de `#sql` comme le montre le source `ExempleSQLJ.sqlj` suivant. Par simplicité, nous n'avons précisé ni la connexion à la base ni l'appel à d'autres méthodes indispensables que nous aborderons dans cette section.

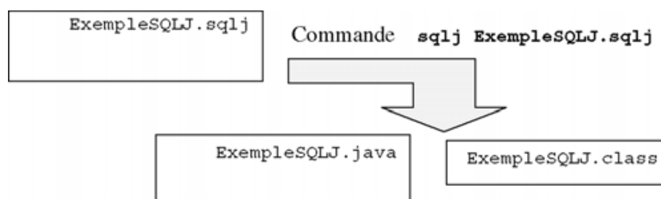
Figure 10-1 *Structure générale d'une source SQLJ*

```
import java.sql.*;                              ExempleSQLJ.sqlj
import oracle.sqlj.runtime.Oracle;
class ExempleSQLJ
{ public static void main (String args[])
  { try
    { ...
      #sql { DELETE FROM Pilote };
      #sql { INSERT INTO Pilote VALUES ('PL-1','Tanguy') };
      ... }
    catch (SQLException e)  { ...}  }
}
```

Précompilation

Comme l'illustre la figure suivante, le précompilateur traduit un source SQLJ (en s'appuyant sur la technologie JDBC) en produisant la classe Java (extension `java`) qu'il n'est pas conseillé de modifier, l'exécutable (extension `class`) contenant du code interprétable par les machines virtuelles Java, et, dans certains cas, des fichiers de configuration (extension `ser`) et d'autres classes résultant de la traduction des blocs SQL.

Figure 10-2 *Précompilation SQLJ*



Configurations

L'environnement de précompilation SQLJ nécessite de devoir configurer un certain nombre de variables et de fichiers.

Variables d'environnement

Il faut s'assurer que la variable `PATH` contient *`Oracle_Home`*`/bin` (répertoire d'installation d'Oracle) pour pouvoir invoquer le précompilateur (commande `sqlj`).

La variable `CLASSPATH` doit inclure le paquetage JDBC à utiliser (en fonction du pilote choisi par l'application) :

- *`Oracle_Home`*`\sqlj\lib\translator.zip` (ou `.jar`) pour inclure le précompilateur.
- *`Oracle_Home`*`\jdbc\lib\classes11.zip` ou `classes12.zip` (les versions `jar` sont aussi disponibles) pour le JDK selon la version (1.1.x ou 1.2.x et 1.3.x), ou `ojdbc14.jar` pour les versions 1.4.x.
- *`Oracle_Home`*`\sqlj\lib\runtime11.zip` ou `runtime12.zip` (les versions `jar` sont aussi disponibles) pour utiliser des pilotes JDBC Oracle couplés au JDK versions 1.1.x ou 1.2.x et supérieures.

D'autres librairies peuvent être aussi incluses :

- *`Oracle_Home`*`\sqlj\lib\runtime.zip` (ou `.jar`) pour utiliser des pilotes JDBC Oracle plus anciens, indépendamment du JDK.
- *`Oracle_Home`*`\sqlj\lib\runtime-nonoracle.zip` (ou `.jar`) pour utiliser des pilotes non Oracle, indépendamment du JDK.

© Éditions Eyrolles

Afin d'initialiser la variable CLASSPATH, écrivez la commande `set classpath=.;C:\oracle\ora92\sqlj\lib\translator.zip;autreChemin;…` dans le fichier `autoexec.bat` ou dans un fichier « .bat » que vous exécuterez à la demande.

Informations de connexions (connect.properties)

En plus des paramétrages des variables d'environnement, il faut configurer SQLJ pour désigner la base de données utilisée par défaut (en l'absence d'un autre contexte). Ces informations relatives à la connexion sont regroupées dans le fichier `connect.properties` (un exemplaire est situé dans le répertoire *Oracle_Home*\sqlj\demo).

Les lignes précédées du caractère « # » ne sont pas analysées. Dans l'exemple qui suit, on retrouve la chaîne de connexion (qu'Oracle appelle *URL*) étudiée dans la section relative à JDBC. Il faut aussi renseigner les variables `sqlj.url` pour désigner la base, `sqlj.user` et `sqlj.password` pour identifier le schéma cible.

```
#Pilote de type 4 sur une base locale   Fichier connect.properties
sqlj.url=jdbc:oracle:thin:@localhost:1521:BDsoutou
sqlj.user=soutou
sqlj.password=ingres
```

Exécution

Une fois que le source SQLJ a été précompilé, il faut exécuter la classe Java de même nom générée (exemple : `ExempleSQLJ.class`). L'exécution est lancée soit en ligne (`java ExempleSQLJ`), soit à l'aide d'un environnement de développement (dans *JCreator* par le menu `Build/Execute File`).

Test d'une configuration

Une fois que le fichier `connect.properties` est à jour en fonction des paramètres de votre base (port UDP, nom de l'instance, utilisateur, mot de passe), vous pouvez tester votre environnement en utilisant le fichier `TestSQLJ.sqlj`. Cet exemple crée une table, insère un enregistrement, extrait un enregistrement dans un curseur et détruit la table précédemment créée.

D'autres exemples se trouvent dans le répertoire *Oracle_Home*\sqlj\demo, citons :

- `TestInstallSQLJ.sqlj` pour tester le précompilateur et l'environnement. Exécutez la classe pour voir le message `Hello, SQLJ`!
- `TestInstallSQLChecker.sqlj` qui renvoie un message d'erreur à l'exécution. Il convient alors de modifier à la ligne 54, `ITEM_NAMAE` par `ITEM_NAME`. Compilez et exécutez pour voir apparaître `Hello, SQLJ Checker`!

Affectations (SET)

La clause `SET` permet d'affecter une valeur à une variable hôte Java dans un bloc SQLJ :

```
#sql { SET : [OUT] variableHôte = expression };
```

L'expression peut être numérique, arithmétique, être un appel d'une fonction, etc. Par défaut la variable hôte est de type `OUT`, il n'est pas possible de la qualifier par `IN` ou `INOUT`.

Le code suivant (`TestSET.sqlj`) décrit deux affectations via des fonctions d'Oracle : la variable `dat` reçoit la date du jour d'exécution du programme, la variable `i` reçoit les sommes de 750 et 250.

Tableau 10-1 Affectation de variables hôtes

Code SQLJ	Commentaires
`int i;` `java.sql.Date dat;`	Déclarations.

`#sql { SET :dat = sysdate };`
`#sql { SET :i = TO_NUMBER('750') +`
`TO_NUMBER('250') };`
	Affectations.

`System.out.println("dat = " + dat + ", i = " + i);`	`dat = 2003-07-24, i = 1000`

Intégration de SQL

Cette section décrit l'intégration d'instructions SQL dans un programme SQLJ. Nous verrons ensuite comment prendre en compte les différentes exceptions SQL pouvant survenir au cours de l'exécution du programme. Les instructions SQL sont disposées à l'intérieur de blocs délimités par des accolades et préfixés de la directive « `#sql` ».

Instructions du LDD

Le code suivant présente des exemples de définition de modification et de suppression d'une table par un programme SQLJ :

Tableau 10-2 LDD dans SQLJ

Code SQLJ	Commentaires
`try` `{Oracle.connect(Exemple.class,"connect.properties");`	Connexion à la base

`#sql { CREATE TABLE Avion (immat VARCHAR(6),`
`typeAvion VARCHAR(15), cap NUMBER(3),`
`CONSTRAINT pk_Avion PRIMARY KEY(immat)) };`
	Création d'une table.

`#sql { ALTER TABLE Avion` `ADD compa VARCHAR(4)};`	Ajout de la colonne `compa`.
`#sql { DROP TABLE Avion };`	Suppression d'une table.
`Oracle.close();`	Fermeture de la connexion.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

 © Éditions Eyrolles

Il est aussi possible de créer, modifier ou supprimer d'autres objets (index, vue, séquence, etc.).

Instructions du LMD

Le code suivant présente des exemples de manipulation de données par un programme SQLJ :

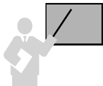
Tableau 10-3 LMD dans SQLJ

Code SQLJ	Commentaires
`try {` `  Oracle.connect(Exemple.class,"connect.properties");`	Connexion à la base.
`  #sql { INSERT INTO Avion VALUES` `            ('AV1', 'A340', 248, 'AF') };`	Ajout d'un avion.
`  #sql { UPDATE Avion SET cap = cap + 10` `                  WHERE typeAvion='A340' };`	Modification de la colonne `cap`.
`  #sql { DELETE FROM Avion WHERE cap < 50 };`	Suppression dans `Avion`.
`  Oracle.close();`	Fermeture de la connexion.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Il est aussi possible d'utiliser des variables hôtes (préfixées de « : ») au sein d'instructions SQL afin de paramétrer les ordres.

Requêtes

Contrairement aux instructions SQL précédentes, `SELECT` retourne un ou plusieurs enregistrements.



Si un seul enregistrement est retourné (extraction monoligne), il faut utiliser des variables hôtes et la directive `INTO` de l'instruction `SELECT`. Si plusieurs enregistrements sont retournés (extraction multiligne), il faudra utiliser des itérateurs (nom donné aux curseurs SQLJ).

Extraction monoligne

Comme dans le cas de PL/SQL, l'utilisation de la directive `INTO` de l'instruction `SELECT` permet d'extraire des valeurs de la base. Ces valeurs sont insérées dans des variables hôtes (préfixées de « : »). Il convient de déclarer ces variables Java au préalable en utilisant les conversions de type étudiées au chapitre 9.

La syntaxe générale d'une extraction monoligne avec SQLJ est la suivante :

```
#sql { SELECT col1 [,col2…] INTO :var1 [,:var2 …]
                FROM table1 [,table2…] WHERE condition … };
```

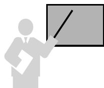
Le code suivant (`TestVarHotes.sqlj`) présente un exemple d'extraction monoligne :

Tableau 10-4 Extraction monoligne

Code SQLJ	Commentaires
`try {…` `String nc;`	Déclaration de la variable hôte.
`#sql { SELECT nomComp INTO :nc` `FROM Compagnie WHERE comp = 'AF' };`	Requête qui charge la variable.
`System.out.println("Nom de AF : " + nc);`	Affichage du résultat.
`…} catch(SQLException ex) { … }`	Gestion des erreurs.

Nature des variables hôtes

Il est possible de préciser la nature d'une variable hôte afin de spécifier si elle peut être utilisée en tant qu'entrée ou en tant que sortie d'un bloc SQLJ.



Une variable hôte peut être déclarée en tant qu'entrée d'un bloc SQLJ (préfixe `:IN`), en tant que sortie (préfixe `:OUT`), ou tant qu'entrée ou sortie (préfixe `:INOUT`).

Il est aussi possible de déclarer, d'une manière similaire, une méthode Java ou une expression combinant variables et méthodes Java.

Le code suivant (`TestVarHotes.sqlj`) présente un exemple de variables hôtes utilisées en entrée et en sortie :

Tableau 10-5 Nature des variables hôtes

Code SQLJ	Commentaires
`try {…` `String code_comp, nom_comp;`	Déclaration des variables hôtes.
`#sql { SELECT comp INTO :OUT code_comp` `FROM Avion WHERE immat = 'F-WTSS' };`	Requête qui charge une variable en sortie.

`#sql { SELECT nomComp INTO :OUT nom_comp`
`FROM Compagnie`
`WHERE comp = :IN code_comp };`
	Requête qui charge une variable en sortie en utilisant une variable en entrée.

`System.out.println` `("Compagnie de F-WTSS : " + nom_comp);`	Affichage du résultat.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

© Éditions Eyrolles

Extraction multiligne

La sélection de plusieurs lignes peut être réalisée de différentes manières :

- par des variables hôtes et un curseur Java (instance de la classe `ResultSet` recevant le résultat d'une requête) ;
- par itérateurs (nom des curseurs dans le vocable SQLJ). On distingue les itérateurs nommés (*named iterator*), pour lesquels on déclare le nom et le type de chaque colonne du curseur résultat, et les itérateurs positionnels (*positional iterator*) pour lesquels seul le type des colonnes du curseur résultat est déclaré.

Variables hôtes

Le code suivant (`TestVarHotes2.sqlj`) extrait la liste des compagnies par l'intermédiaire de variables hôtes utilisées en sortie (directive `:OUT`) et d'un curseur Java :

Tableau 10-6 Extraction multiple par variables hôtes

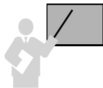
Code SQLJ	Commentaires
`try {…` `  ResultSet rs;`	Déclaration du curseur résultat.

`  #sql {BEGIN`
`        OPEN :OUT rs FOR SELECT comp, nomComp`
`                         FROM   Compagnie;`
`        END; };`
	Ouverture et chargement du curseur.

`  while(rs.next()) {`
`     System.out.print("Numéro: "+rs.getString(1));`
`     System.out.println(" Nom: "+rs.getString(2)); }`
	Affichage des résultats.

`  rs.close(); …`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Itérateurs nommés



Plusieurs étapes sont nécessaires pour utiliser un itérateur nommé : déclaration du type de l'itérateur, déclaration d'un itérateur, chargement par la requête et fermeture.

```
#sql iterator nomTypeItérateur (type1 colonne1 [,type2 colonne2…]);
nomTypeItérateur nomItérateur;
…
#sql nomItérateur = { SELECT … };
…
nomItérateur.close();
```

L'extraction d'une colonne d'un itérateur nommé est réalisée via l'instruction `nomCurseur.nomColonne()`. Le type du curseur est considéré comme une classe qu'il est possible d'alimenter en variables curseurs. Ces variables sont chargées par une requête.

Le code suivant (`TestVarHotes3.sqlj`) extrait la liste des compagnies par l'intermédiaire d'un itérateur nommé :

Tableau 10-7 Extraction multiple par un itérateur nommé

Code SQLJ	Commentaires

`try {…`
`#sql iterator CompagnieIter`
`(String comp, String nomComp);`
	Déclaration du type de l'itérateur nommé.

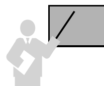
`CompagnieIter cp_iter;`	Déclaration d'un itérateur nommé.
`#sql cp_iter =` `{ SELECT comp, nomComp FROM Compagnie };`	Chargement de l'itérateur nommé.

`while(cp_iter.next()) {`
`System.out.print("Numéro: "+cp_iter.comp());`
`System.out.println(" Nom: "+cp_iter.nomComp());}`
	Affichage des résultats (parcours de l'itérateur nommé).

`cp_iter.close(); …`	Fermeture de l'itérateur nommé.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Les colonnes de l'itérateur doivent porter le même nom que les colonnes des tables.

Itérateurs positionnels



Plusieurs étapes sont nécessaires pour utiliser un itérateur positionnel : déclaration du type de l'itérateur, déclaration d'un itérateur, chargement par la requête et fermeture.

```
#sql iterator nomTypeItérateur (type1 [,type2 …]);
nomTypeItérateur nomItérateur;
…
#sql nomItérateur = { SELECT … };
…
nomItérateur.close();
```

Comme pour les curseurs PL/SQL, l'instruction `FETCH` permet de charger l'enregistrement courant dans le curseur. La méthode `endFetch` signale la fin du parcours. L'extraction d'une colonne est réalisée via l'instruction `nomCurseur.nomColonne()`.

 © Éditions Eyrolles

Le code suivant (`TestVarHotes4.sqlj`) extrait la liste des compagnies par l'intermédiaire d'un itérateur positionnel :

Tableau 10-8 Extraction multiple par un itérateur positionnel

Code SQLJ	Commentaires
`try {…` `#sql iterator CompagnieIter (String, String);`	Déclaration du type de l'itérateur positionnel.

`CompagnieIter cp_iter;`
`String numcp = null;`
`String nomcp = null;`
	Déclaration d'un itérateur positionnel.

`#sql cp_iter =` `{ SELECT comp, nomComp FROM Compagnie };`	Chargement de l'itérateur positionnel.

`while(true){`
`#sql { FETCH :cp_iter INTO :numcp, :nomcp };`
`if (cp_iter.endFetch()) break;`
`System.out.print("Numéro: "+numcp);`
`System.out.println(" Nom: "+nomcp);}`
	Affichage des résultats (parcours de l'itérateur positionnel).

`cp_iter.close(); …`	Fermeture de l'itérateur positionnel.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

À propos des itérateurs

Cette section résume et complète la description des fonctionnalités des itérateurs.

Méthodes

Les interfaces `ResultSetIterator` et `PositionedIterator` utilisées lors de la précompilation d'un itérateur (nommé ou positionnel) disposent des méthodes suivantes :

Tableau 10-9 Méthodes pour les itérateurs

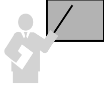
Méthodes	Fonctions
`close()`	Fermeture de l'itérateur.

`ResultSet getResultSet()`	Extrait le resultset équivalent à l'itérateur au format JDBC. Exemple :
	`#sql iter    = { SELECT * FROM Avion };`
	`ResultSet rs = iter.getResultSet();`

`boolean isClosed()`	Détermine l'état de l'itérateur (renvoie `true` si ouvert).
`boolean next()`	Déplacement vers le prochain enregistrement de l'itérateur (retourne `true` si cet enregistrement existe).
`boolean endFetch()`	Détermine si la fin de l'itérateur positionnel est atteinte.

Navigation dans un itérateur

Oracle répond à la norme SQLJ qui prévoit la possibilité de naviguer dans les itérateurs (*scrollable iterators*). Ce mécanisme est basé sur l'interface `Scrollable` de la spécification JDBC 2.0 (*scrollable result sets*).



Il faut suivre les étapes suivantes pour pouvoir naviguer dans un itérateur : déclaration d'une classe d'itérateurs navigables, création d'un itérateur, chargement par requête et exploitation des diverses méthodes disponibles avant de fermer l'itérateur.

```
#sql public static iterator classeItérateur implements
    sqlj.runtime.Scrollable(type1 colonne1 [,type2 colonne2…]);
classeItérateur iter;
…
#sql iter = { SELECT … };
… iter.méthodesNavigation();
```

Les principales méthodes relatives à la navigation dans un itérateur sont résumées dans le tableau suivant. Les quatre dernières méthodes retournent `false` si aucun enregistrement n'existe.

Tableau 10-10 Méthodes de navigation dans un itérateur

Méthodes	Fonctions
`void setFetchDirection(int)`	Précise la direction dans laquelle le parcours se fera. Les valeurs permises sont `FETCH_FORWARD` (par défaut si une direction a été donnée via l'interface `ExecutionContext`), `FETCH_REVERSE` ou `FETCH_UNKNOWN`.
`int getFetchDirection()`	Extrait la direction courante.
`boolean isBeforeFirst()`	Indique si le curseur est positionné avant le premier enregistrement.
`boolean isFirst()`	Indique si le curseur est positionné sur le premier enregistrement.
`boolean isLast()`	Indique si le curseur est positionné sur le dernier enregistrement.
`boolean isAfterLast()`	Indique si le curseur est positionné après le dernier enregistrement.

Le code suivant (`IterNavigable.sqlj`) utilise ces méthodes sur un itérateur reflétant l'état de la table `Compagnie` :

© Éditions Eyrolles

Tableau 10-11 Navigation dans un itérateur

Code SQLJ	Commentaires

`import oracle.sqlj.runtime.Oracle;`
`import java.sql.* ;`
`class IterNavigable {`
	Importations.

`#sql public static iterator ItérateurNavi`
`implements sqlj.runtime.Scrollable`
`(String comp, String nomComp);`
	Déclaration de la classe de l'itérateur.

`public static void main (String args[])`
`{try {Oracle.connect`
`(IterNavigable.class, "connect.properties");`
`ItérateurNavi compIter;`
`#sql compIter = { SELECT comp, nomComp`
`FROM Compagnie };`
	Déclaration et chargement de l'itérateur.

`if ( compIter.isBeforeFirst() )`
`System.out.println("Positionné au début");`
`if ( compIter.isFirst() )`
`System.out.println("Positionné sur le 1er déjà");`
	Test renvoyant true.
	
	Test renvoyant false.

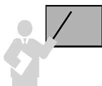
`while(compIter.next())`
`{if ( compIter.isFirst() )`
`System.out.println("1ère compagnie : ");`
`if ( compIter.isLast() )`
`System.out.println("Dernière compagnie : ");`
`System.out.print("Numéro: "+compIter.comp());`
`System.out.println("Nom: "+compIter.nomComp());}`
	Parcours de l'itérateur avec affichage du premier et du dernier élément.

`if ( compIter.isAfterLast() )`
`System.out.println("Positionné après la fin");`
`compIter.close(); Oracle.close(); }`
	Test renvoyant true.
	
	Fermeture de l'itérateur.

`} catch(SQLException ex) { … }`	Gestion des erreurs.

Sensibilité

Un itérateur peut être déclaré « sensible » (*sensitive*) aux changements de la base durant le temps de son existence. Il fournit ainsi une vue actualisée des données. Ce mécanisme est semblable à celui étudié au chapitre 9.



Pour déclarer un itérateur sensible, il faut ajouter une clause à la déclaration d'un itérateur :

```
#sql public static iterator classeItérateur implements
          sqlj.runtime.Scrollable(type1 colonne1 [,type2 colonne2…])
          with (sensitivity=SENSITIVE);
```

Transactions

Au même titre que les instructions SQL, il est possible d'intégrer un programme PL/SQL dans son intégralité dans un bloc SQLJ comme le montre la figure suivante. Des transactions peuvent ainsi être programmées et la gestion des exceptions au sein du bloc est également possible.

Figure 10-3 *Programmes PL/SQL sous SQLJ*

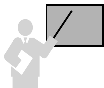
```
#sql { [ DECLARE … ]
        BEGIN
        …
          [DECLARE …
           BEGIN
           …
           EXCEPTION …
           END;]
        [ EXCEPTION … ]
        END; }
```

Intégration de blocs PL/SQL

Un simple bloc PL/SQL permet de remplacer la clause `SET` qui affecte une valeur à une variable hôte Java dans un bloc SQLJ :

```
#sql { BEGIN :OUT variableHôte:= expression; END };
```

Le code suivant (`TestBlocPL.sqlj`) décrit l'intégration d'une transaction dans un bloc SQLJ. La transaction consiste à insérer la compagnie 'BW' et un Concorde rattaché à cette compagnie. La capacité de cet avion sera diminuée de 10 places par rapport au Concorde de plus grande capacité de la base. Si aucun Concorde n'existe la transaction est annulée. Si d'autres erreurs surviennent il convient de les traiter dans le bloc d'exception de SQLJ (voir plus loin).



Il est possible de valider un ensemble d'instructions en écrivant le bloc « `#sql{COMMIT};` » ou au contraire d'invalider la transaction par « `#sql{ROLLBACK};` ».

© Éditions Eyrolles

Tableau 10-12 Intégration d'une transaction

Code SQLJ	Commentaires
`try {…`	
`#sql { DECLARE v_capacité NUMBER(3); pas_de_Concorde EXCEPTION;`	Déclarations.
`BEGIN SELECT MAX(cap) INTO v_capacité FROM Avion WHERE typeAvion = 'Concorde'; IF v_capacité IS NULL THEN RAISE pas_de_Concorde; END IF; v_capacité := v_capacité - 10;`	Extraction de la plus grande capacité des Concorde (si pas trouvé, annulation).
`INSERT INTO Compagnie VALUES ('BW', 'British Airways'); INSERT INTO Avion VALUES ('F-FGFB', 'Concorde', v_capacité, 'BW');`	Ajout de deux enregistrements.
`COMMIT; EXCEPTION WHEN pas_de_Concorde THEN ROLLBACK; END; };`	Validation ou invalidation.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

```
try {…
  #sql {
  DECLARE
    v_capacité NUMBER(3);
    pas_de_Concorde EXCEPTION;
  BEGIN
    SELECT MAX(cap) INTO v_capacité
           FROM Avion WHERE typeAvion = 'Concorde';
    IF v_capacité IS NULL THEN
       RAISE pas_de_Concorde;
    END IF;
    v_capacité :=  v_capacité - 10;
    INSERT INTO Compagnie VALUES
           ('BW', 'British Airways');
    INSERT INTO Avion VALUES
           ('F-FGFB', 'Concorde', v_capacité, 'BW');
    COMMIT;
  EXCEPTION
    WHEN pas_de_Concorde THEN ROLLBACK;
  END; };
} catch(SQLException ex) { … }
```

Points de validation

Comme avec JDBC, la technologie SQLJ d'Oracle intègre la programmation de points de validation (*savepoints*). Le tableau suivant décrit les instructions SQLJ de la norme concernée :

Tableau 10-13 Gestion des points de validation

Code SQLJ	Commentaires
`#sql { SET SAVEPOINT :nomPoint };`	Déclare un point de validation.
`#sql { ROLLBACK TO :nomPoint };`	Invalide la transaction depuis ce point.
`#sql { RELEASE :nomPoint };`	Annule le point de validation.

Oracle ne prend pas encore en charge la fonctionnalité d'annulation d'un point de validation.

Le code suivant (`TestSavePoint.sqlj`) décrit une transaction partagée en deux phases. L'instruction de non validation permet de valider seulement la première insertion :

Tableau 10-14 Point de validation

Web

Code SQLJ	Commentaires
`try {…`	

`String p1 = "P1";`
`#sql { INSERT INTO Avion VALUES`
`('F-TOTO', 'A340', 256, 'AF') };`
	Variable décrivant le point de validation.

`#sql { SET SAVEPOINT :p1 };`	Déclaration du point de validation.
`#sql { INSERT INTO Avion VALUES` `('F-TITI', 'A340', 256, 'AF') };`	
`#sql { ROLLBACK TO :p1 };` `#sql { COMMIT };`	Annulation de la dernière partie de la transaction.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Appels de sous-programmes

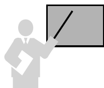
SQLJ permet d'inclure des appels à des procédures ou fonctions stockées qui peuvent être des éléments de paquetages applicatifs. Ces sous-programmes sont en général codés en PL/SQL mais peuvent être écrits depuis la version 8 en Java, C, C++, etc.

Nous distinguons deux cas :

- le sous-programme retourne un résultat scalaire (qui peut être composé d'une ou de plusieurs valeurs). Le cas particulier concerne le sous-programme qui ne retourne aucun résultat ;
- le sous-programme retourne un résultat complexe (traité par un curseur).

Résultats scalaires

Une fonction stockée nécessite une expression SQLJ qui reçoit un résultat et peut inclure des paramètres d'entrée. Une procédure stockée peut inclure des expressions SQLJ en paramètres d'entrée ou de sortie.



La syntaxe pour appeler un sous-programme retournant un résultat scalaire dans un bloc SQLJ est la suivante :

- `#sql { CALL nomProcédure(paramètres) }` pour une procédure ;

 © Éditions Eyrolles

- `#sql variableHôte = { VALUES(nomFonction(paramètres)) }` pour une fonction.
- L'expression *paramètres* désigne une liste de constantes ou de variables hôtes (`:IN v1, :OUT v2`…) si on désire faire passer des valeurs de variables Java en paramètres d'appel ou de réception.

Fonction

Intégrons dans un bloc SQLJ l'appel de la fonction PL/SQL `LeNomCompagnieEst` (décrite au chapitre 9, section « Appel d'une fonction ») qui retourne le nom de la compagnie d'un avion dont le numéro d'immatriculation est passé en paramètre. Si le numéro d'immatriculation n'est pas référencé dans la base, la valeur `NULL` est retournée. Le code suivant (`TestFonctPL.sqlj`) décrit l'appel de cette fonction par la directive `VALUES` :

Tableau 10-15 Intégration d'une fonction

Code SQLJ	Commentaires

`try {…`
`String immatAvion = "F-WTSS";`
`String nomCompF_WTSS;`
	Déclaration des variables hôtes d'entrée et de sortie.

`#sql nomCompF_WTSS =`
`{ VALUES( LeNomCompagnieEst(:IN immatAvion)) };`
`System.out.println("Résultat : "+nomCompF_WTSS);`
	Appel de la fonction (si numéro pas trouvé, `null` retourné).
	Affichage du résultat.

`} catch(SQLException ex) { … }`	Gestion des erreurs.

Procédure

Le code suivant (`TestProcPL.sqlj`) intègre l'appel de la procédure PL/SQL `AugmenteCapacité` (décrite au chapitre 9) qui augmente la capacité d'un avion dont le numéro est passé en paramètre. L'appel de la procédure est rendu possible par la directive `CALL`.

Tableau 10-16 Intégration d'une procédure

Code SQLJ	Commentaires

`try {…`
`String immatAvion = "F-WTSS";`
`int enPlus = 5;`
	Déclaration des variables hôtes d'entrée.

`#sql {` `CALL AugmenteCapacité(:IN immatAvion, :IN enPlus)};`	Appel de la procédure.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Si l'appel d'un sous-programme (fonction ou procédure) ne nécessite pas de paramètre, écrivez l'appel sans parenthèses : `CALL proc` au lieu de `CALL proc()` et `VALUES(fct)` au lieu `VALUES(fct())`.

Résultats complexes

Les sous-programmes PL/SQL retournent des résultats complexes par l'intermédiaire de variables curseurs (`REF CURSOR`). Ces résultats sont affectés à des itérateurs SQLJ ou des curseurs Java (instances de la classe `ResultSet`).

Considérons la fonction `retourneCompagnies` du paquetage `GestionAvs` qui retourne le code et le nom des compagnies par la variable curseur `résultat` :

Tableau 10-17 Tableau 10-16 : Intégration de l'appel d'une fonction

Spécification	Corps

`CREATE PACKAGE GestionAvs AS`
` TYPE Comp_Curtype IS REF CURSOR;`
` FUNCTION retourneCompagnies`
`          RETURN Comp_Curtype;`
`END GestionAvs;`
	`CREATE PACKAGE BODY GestionAvs AS`
	`FUNCTION retourneCompagnies`
	`         RETURN Comp_Curtype IS`
	` résultat Comp_Curtype;`
	` BEGIN`
	`  OPEN résultat FOR`
	`       SELECT comp, nomComp FROM Compagnie;`
	`  RETURN résultat;`
	` END;`
	`END GestionAvs;`

Le code suivant (`TestProcRefCur.sqlj`) illustre l'appel de cette fonction par l'intermédiaire d'un itérateur nommé :

Tableau 10-18 Intégration de l'appel d'une fonction

Code SQLJ	Commentaires
`#sql public static iterator` `          CompIter (String comp, String nomComp);`	Déclaration de la classe de l'itérateur.

`try {…`
` CompIter curs_comp;`
` #sql curs_comp =`
`     { VALUES( GestionAvs.retourneCompagnies ) };`
` while (curs_comp.next()) {`
`  System.out.print("Numéro: "+ curs_comp.comp());`
`  System.out.println("Non: "+ curs_comp.nomComp());}`
` curs_comp.close();`
	Déclaration d'un itérateur.
	
	Appel de la fonction.
	Parcours de l'itérateur.

`} catch(SQLException ex) { … }`	Gestion des erreurs.

© Éditions Eyrolles

Traitement des exceptions

Les exceptions qui ne sont pas traitées dans les blocs PL/SQL intégrés, ou celles qui sont volontairement retournées par un sous-programme ou un déclencheur, doivent être prises en compte au niveau du code Java (bloc `SQLException`). Ce bloc d'instructions permet de programmer des traitements en fonction des codes d'erreurs Oracle.

Définition des données

Écrivons le programme SQLJ qui crée une table en gérant la possibilité qu'elle existe déjà par le biais d'une exception (message d'Oracle : `ORA-00955: Ce nom d'objet existe déjà`).

Le code suivant (`ExceptLDD.sqlj`) crée la table `Temp` en gérant l'exception précédente. Dans ce cas la table est détruite puis recréée.

Tableau 10-19 Exception Oracle LDD

Code SQLJ	Commentaires
`import java.sql.*;` `import oracle.sqlj.runtime.Oracle;`	Importation des paquetages.

`class ExceptLDD`
`{ public static void main (String args[])`
`  { try { Oracle.connect`
`          (ExceptLDD.class, "connect.properties");`
`        #sql { CREATE TABLE`
`               Temp(colonne1 VARCHAR2(50)) }; }`
	Classe principale et méthode `main` contenant l'instruction LDD.

`    catch (SQLException e)`
`    { if (e.getErrorCode() == 955)`
`       {System.out.println("Table déjà existante!");`
`        try {  #sql { DROP TABLE Temp};`
`               #sql { CREATE TABLE`
`                      Temp(colonne1 VARCHAR2(50)) };`
`            } catch (SQLException ex)`
`            { System.err.println("Erreur : " + ex);}}`
`     else System.err.println("Autre erreur : " + e);}`
	Gestion de l'erreur.

`  finally`
`  { try { Oracle.close();`
`          System.out.println("Table Temp créée"); }`
`    catch (SQLException e)`
`        {System.err.println("Erreur : " + e); } } } }`
	Fin du traitement.

Manipulation des données

Le code suivant (ExceptLMD.sqlj) insère un enregistrement dans la table Avion en gérant quelques-unes des exceptions potentielles :

Tableau 10-20 Exceptions Oracle LMD

Code SQLJ	Commentaires
`import java.sql.*;` `import oracle.sqlj.runtime.Oracle;`	Importation des paquetages.

`class ExceptLMD`
`{ public static void main (String args[])`
`  { try { Oracle.connect`
`             (ExceptLMD.class, "connect.properties");`
`          #sql { INSERT INTO Avion VALUES`
`                ('AV1', 'A340', 248, 'AF') }; }`
	Classe principale et méthode `main` contenant l'instruction LMD.

`  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!");`
`   else if (ex.getErrorCode() == 2291)`
`      System.out.println("Compagnie inconnue!"); }`
	Gestion des erreurs potentielles.

`  finally`
`  { try { Oracle.close();`
`          System.out.println("Déconnexion OK."); }`
`    catch (SQLException e)`
`        {System.err.println("Erreur : " + e); } } } }`
	Fin du traitement.

Interrogation des données

Le code suivant (ExceptLID.sqlj) extrait un enregistrement de la table Avion. Les événements « aucune ligne sélectionnée » (getSQLState renvoie “2000”) et « plusieurs lignes extraites » (getSQLState renvoie "21000") sont pris en compte par des exceptions.

© Éditions Eyrolles

Tableau 10-21 Exceptions Oracle LID

Code SQLJ	Commentaires
`import java.sql.*;` `import oracle.sqlj.runtime.Oracle;`	Importation des paquetages.

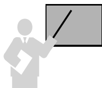
`class ExceptLID`
`{ public static void main (String args[])`
`  { try { Oracle.connect`
`        (ExceptLID.class, "connect.properties");`
`  String nc;`
`  #sql { SELECT nomComp INTO :nc`
`         FROM Compagnie WHERE comp = 'AF'};`
`  System.out.println("Nom de la compagnie : " + nc);}`
	Classe principale et méthode `main` contenant la requête et l'affichage du résultat si aucune exception n'est levée.

`   catch (SQLException ex)`
`    { if (ex.`**`getSQLState`**`() == "`**`2000`**`")`
`      System.out.println("Pas de compagnie!");`
`      else if (ex.`**`getSQLState`**`() == "`**`21000`**`")`
`      System.out.println`
`              ("Trop d'enregistrements renvoyés");`
`      else System.out.println`
`              ("Autre erreur : "+ex.getMessage()); }`
	Gestion des erreurs.

`  finally`
`  { try { Oracle.close();`
`          System.out.println("Déconnexion OK."); }`
`    catch (SQLException e)`
`        {System.err.println("Erreur : " + e); } } } }`
	Fin du traitement.

Contextes de connexion

Le fichier `connect.properties` décrit la connexion par défaut du programme SQLJ. Comme sous JDBC, SQLJ permet de travailler simultanément avec plusieurs contextes de connexion. Ces connexions peuvent concerner différents schémas d'une même base ou divers schémas d'instances distinctes.



Le travail avec un contexte de connexion s'effectue à plusieurs niveaux :

- de la classe de contexte de connexion (`#sql context MaConnexion`) ;
- de l'instance de la classe de contexte connexion pour chaque connexion (`cx = new MaConnexion (`*`chaîneConnexion`*`, `*`user`*`, `*`password`*`, `*`autoCommit`*`)`) ;
- du bloc SQLJ qui est précédé du contexte de connexion (`#sql [cx]{…}`). Ici les crochets ne signifient pas une option mais un symbole à utiliser impérativement.

Le code suivant (`TestCONTEXT.sqlj`) travaille avec deux connexions. La connexion `connexScott` extrait les enregistrements de la table `Compagnie`. La connexion courante

concerne un schéma qui contient la table `Avion`. Les avions de chaque compagnie sont affichés à l'aide de deux boucles imbriquées.

Tableau 10-22 Contextes

Code SQLJ	Commentaires
import java.sql.*; import oracle.sqlj.runtime.Oracle;	Importation des paquetages.
#sql context MaConnexion;	Déclaration de la classe de contexte de connexion.
class TestCONTEXT { public static void main (String args[]) { try { Oracle.connect (TestCONTEXT.class, "connect.properties");	Classe principale et méthode `main`.
MaConnexion **connexScott** = new MaConnexion ("jdbc:oracle:thin:@camparols:1521:BDSoutou", "scott", "tiger", false);	Déclaration de la deuxième connexion.
ResultSet cp; #sql [**connexScott**] { BEGIN OPEN :OUT cp FOR SELECT comp,nomComp FROM Compagnie; END; };	Requête dans la connexion *scott*/*tiger* sur *Camparols.*
while (cp.next()) {String codeComp = cp.getString(1); System.out.println ("Avions de la compagnie "+cp.getString(2));	Parcours des compagnies.

ResultSet av;
#sql { BEGIN
OPEN :OUT av FOR
SELECT immat, typeAvion
FROM Avion WHERE comp = :IN codeComp;
END; };
boolean trouve = false;
while (av.next()) {
System.out.println
("Immatriculation : "+av.getString(1)+
" Type : "+av.getString(2));
trouve=true; }
if (!trouve) System.out.println("Aucun avion");
av.close(); }
cp.close();
	Requête dans la connexion courante.

connexScott.close(); Oracle.close(); }	Fermeture des connexions.
catch (SQLException ex) { System.err.println("Erreur : " + ex); } } }	Fin du traitement.

© Éditions Eyrolles

SQL dynamique

Des extensions de SQLJ qui permettent de programmer des instructions SQL dynamiques sont proposées par Oracle. Ces aspects ne sont pas présents dans la norme SQLJ. Une instruction SQL dynamique n'est pas prédéfinie dans la source et peut évoluer au cours du programme. Les expressions du SQL dynamique qui sont intégrées au code SQLJ sont dites *meta bind expressions*.

Expression

Une expression SQL dynamique SQLJ contient un ou plusieurs identifiants Java (`String`) qui seront interprétés durant l'exécution. Une expression peut remplacer :

- le nom d'une table ;
- le nom d'une colonne dans un `SELECT` ;
- tout ou partie de conditions dans la clause or `WHERE` ;
- le nom d'un rôle, schéma, paquetage dans une instruction LDD ou LMD ;
- une valeur littérale dans une expression SQL.

Deux écritures sont possibles pour définir une expression SQL dynamique SQLJ. Un bloc peut contenir plusieurs expressions de ces types.

```
:{ expressionJavaBind }
:{ expressionJavaBind :: codeSQLquiRemplaceExécution }
```

Le code suivant (`SQLDynamique.sqlj`) illustre les deux écritures possibles d'une expression SQL dynamique. La première réalise une insertion dans une table dont le nom passe en paramètre. La deuxième permet de substituer à l'exécution un paramètre (ici la compilation génère une instruction d'insertion dans la table `Avion`, l'exécution insérera dans la table `Avion2`). Ce principe permet de compiler le programme sans que la table `Avion2` existe forcément dans le schéma.

Tableau 10-23 Expressions SQLJ pour SQL dynamique

Code SQLJ	Instruction SQL générée à l'exécution

`String x = "F-GAFU";`
`String y = "A380";`
`int nbPlaces = 345;`
`String nomTable = "Avion";`
`#sql { INSERT INTO :{`**`nomTable`**`}`
`VALUES (:x, :y, :nbPlaces, 'AF') };`
	`INSERT INTO `**`Avion`**` VALUES`
	`('F-GAFU', 'A380', 345, 'AF')`

`String nomTable2 = "Avion2";`
`#sql { INSERT INTO :{`**`nomTable2 :: Avion`**`}`
`VALUES ('F-ENTE', :y, 350, 'AF') };`
	`INSERT INTO `**`Avion2`**` VALUES`
	`('F-ENTE', 'A380', 350, 'AF')`

Restrictions

Une expression SQL dynamique SQLJ ne peut pas :

- être associée à un mode (`IN`, `OUT`, ou `INOUT`) ;
- être le premier mot-clé d'une instruction SQL dynamique ;
- inclure de clause `INTO` dans une requête ;
- apparaître dans une des clauses suivantes : `CALL`, `VALUES`, `COMMIT`, `ROLLBACK`, `FETCH INTO`, ou `CAST`.

© Éditions Eyrolles

Exercices

L'objectif de ces exercices est de manipuler des tables de votre base par des programmes SQLJ.

Exercice 10.1 Itérateur nommé

Écrire la classe Java `Consulte.sqlj` qui affiche les caractéristiques d'une salle dont le numéro est saisi au clavier. Programmer les méthodes :

- `void afficheSalle(String)` qui extrait l'enregistrement de la table `Salle` ;
- `void affichePostes(String)` qui extrait les postes présents dans la salle (utiliser un itérateur nommé).
- `main` qui saisit le numéro de salle et teste les méthodes de la classe.

Utiliser dans le `main` la méthode `lire` qui retourne une chaîne saisie au clavier :

```
private static String lire()
 {String c = "";
 try {BufferedReader entrée =
new BufferedReader(new InputStreamReader(System.in));
c = entree.readLine();}
 catch (java.io.IOException e)
   {System.out.println("Une erreur d'entree/sortie est survenue!!!");
    System.exit(0);}
return c;} }
```

La trace de l'exécution de cette classe pour la salle ''s01' est la suivante :

```
Saisir le numero d'une salle : s01
Salle 1 (3 postes, segment IP : 130.120.80)
Liste des postes :
Numero: p1 Nom Poste: Poste 1 IP: 130.120.80 Adr: 01Type Poste: TX
Numero: p2 Nom Poste: Poste 2 IP: 130.120.80 Adr: 02Type Poste: UNIX
Numero: p3 Nom Poste: Poste 3 IP: 130.120.80 Adr: 03Type Poste: TX
```

Exercice 10.2 Mise à jour de la base

Écrire la classe Java `Insere.sqlj` qui enregistre dans la table `Logiciel` une liste de logiciels contenus dans un tableau de chaîne de caractères (`String[][] valeurs`). Programmez les méthodes :

- `void insere(String[])` qui insère les enregistrements par une requête SQL paramétrée ;
- `main` qui charge le tableau et appelle `insere`.

Prendre en compte d'éventuelles exceptions (logiciel déjà présent, type inconnu, etc.) qui peuvent être levées durant une insertion. Effectuer les insertions à partir du tableau suivant :

valeurs					
v1	Forms	13-05-1995	4	UNIX	300
v2	SQLJ	15-09-1999	1	UNIX	560
v3	MySQL	12-04-1998	7	PCNT	0

 © Éditions Eyrolles

Chapitre 11

Procédures stockées et externes

Oracle propose deux mécanismes distincts mais complémentaires afin d'intégrer des programmes écrits en langages de troisième et quatrième générations (C, C++, Java,etc.) :

- Les procédures stockées (s*tored procedures*) au niveau de la base de données.
- Les procédures externes (*external procedures*) à la base de données qui sont appelables.

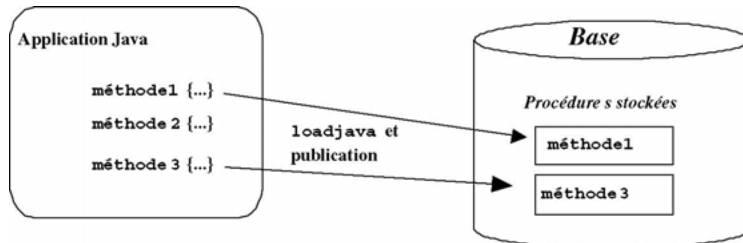
Ce chapitre présente ces deux mécanismes en utilisant le langage Java. Il est aisé de transposer les principes que nous présentons dans ce chapitre à d'autres langages évolués.

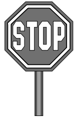
Procédures stockées Java

Une procédure stockée est une méthode d'une classe Java qui est compilée au niveau de la base de données. La machine virtuelle Java qui est présente dans le noyau exécute le fichier compilé (*byte-code*). Une procédure stockée peut accéder aux différents autres objets de la base (tables, vues, déclencheurs, etc.) par l'intermédiaire de la passerelle JDBC.

Les avantages à utiliser les procédures stockées sont multiples : le programmeur a la possibilité de récupérer son code qui peut avoir été développé il y a longtemps. D'autre part, l'optimisation et l'intégration de l'appel à ce code sont assurées par la base elle-même. La figure suivante illustre deux méthodes d'une classe Java qu'on transforme en procédure stockée de la base :

Figure 11-1 *Procédures stockées Java*





Seules les méthodes utilisant des directives des paquetages graphiques (*AWT* et *Swing*) ne peuvent pas être stockées dans la base.

Les trois contextes d'utilisation d'une procédure stockée sont les suivants :

- développement de fonctions ou procédures en relation avec les données de la base ;
- programmation de déclencheurs ;
- développement de méthodes associées à un type objet relationnel (extension objet du modèle relationnel d'Oracle non traitée ici).

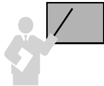
Stockage d'une procédure

Différentes étapes doivent être respectées afin d'intégrer une procédure stokée Java.

Développement de la classe

Il faut bien sûr disposer d'une classe Java qu'on aura éventuellement compilée auparavant afin de s'assurer de sa syntaxe. Le code suivant décrit la classe `PremierExemple` contenant la méthode `affiche` qui compte le nombre de lettres contenues dans la chaîne passée en paramètre.

```
public class PremierExemple {
   public static String affiche(String message)
   {  int nc=0;
      for (int i=0;i<message.length();i++)
        if (((message.charAt(i)>='a')&&(message.charAt(i)<='z'))
        || ((message.charAt(i)>='A')&&(message.charAt(i)<='Z')))
        nc++;
      return "Le message contient "+nc+" lettres!";}}
```



Seules les méthodes déclarées `public static` peuvent devenir des procédures stockées.

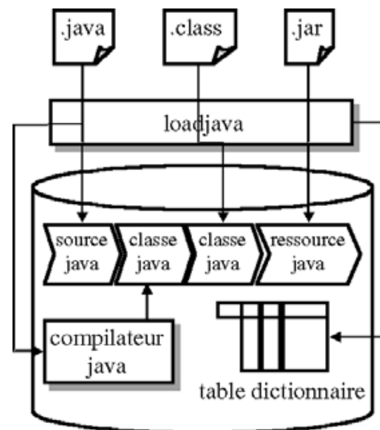
Chargement d'une méthode

La seconde étape consiste à faire appel à l'utilitaire `loadjava` qui permet de charger des ressources Java dans la base. Comme le montre la figure suivante, différents types de ressources Java peuvent être chargés :

- les fichiers sources (`.java`) ;
- les classes (`.class`) ;
- les archives (`.jar`).

© Éditions Eyrolles

Figure 11-2 Chargement des méthodes



La commande suivante réalise l'opération de chargement. L'instruction est passée en ligne de commande dans le répertoire contenant la source ; l'utilisateur, le mot de passe et la chaîne de connexion désignent le schéma cible. L'option `-verbose` permet de tracer les différentes actions effectuées. Le chargeur effectue la connexion à la base de données, compile le code source, enregistre la source et la classe compilée.

```
loadjava -user soutou/ingres@cxbdsoutou -verbose PremierExemple.java
arguments: '-user' 'soutou/ingres@cxbdsoutou' '-verbose' 'Premie-
rExemple.java'
created  : JAVA$CLASS$MD5$TABLE
creating : source PremierExemple
created  : CREATE$JAVA$LOB$TABLE
loading  : source PremierExemple
creating : PremierExemple
```

Vue du dictionnaire

La vue `USER_OBJECTS` permet d'extraire les différents objets Java entreposés dans la base (les types possibles sont : `JAVA CODE`, `JAVA CLASS` et `JAVA RESOURCE`). Nous retrouvons ainsi le nom de la classe chargée de notre exemple.

```
SQL> SELECT DBMS_JAVA.LONGNAME(OBJECT_NAME) FROM USER_OBJECTS
     WHERE  OBJECT_TYPE = 'JAVA CLASS';

DBMS_JAVA.LONGNAME(OBJECT_NAME)
-----------------------------------------
PremierExemple
```

Publication de la méthode

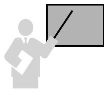
Cette étape se réalise sous l'interface de commande SQL*Plus. Chaque méthode Java qui est appelée doit être publiée. La publication attribue une signature PL/SQL à la méthode. La syntaxe SQL à utiliser pour publier une méthode Java est la suivante :

```
CREATE [OR REPLACE]
{ FUNCTION nomFonction [(paramètre1 [, paramètre2]…)] RETURN
typeSQL }
| PROCEDURE nomProcédure [(paramètre1 [, paramètre2]…)] }
{IS | AS} LANGUAGE JAVA
NAME 'nomClasse.nomMéthode([(paramètre1 [, paramètre2]…)]) [return
typeJava]';
```

Les paramètres SQL sont définis par un nom, un mode et un type SQL (*nomparamètre* [IN | OUT | IN OUT] *typeSQL*). L'instruction suivante publie la méthode `affiche` de la classe `PremierExemple` renommée au niveau de la base `PremierExemple_affiche` :

Tableau 11-1 Publication d'une méthode

Instruction SQL	Commentaires
`CREATE FUNCTION PremierExemple_affiche(mess VARCHAR2) RETURN VARCHAR2`	Signature de la procédure stockée et type de retour.
`AS LANGUAGE JAVA`	Langage utilisé.
`NAME 'PremierExemple.affiche(java.lang.String) return java.lang.String';` `/`	Identification de la méthode Java.



- le nom de la méthode publiée et les noms de ses paramètres ne sont pas nécessairement indentiques à la méthode source Java ;
- il faut respecter la correspondance des types Java et SQL (`String` et `VARCHAR2` par exemple) ;
- Java distingue les majuscules des minuscules, donc la casse doit être respectée (noms de classes, méthodes et paramètres Java). Par exemple, le mot `return` doit être en minuscules).

Appel de la méthode

L'appel d'une procédure stockée peut être réalisé :

- sous l'interface de commande SQL*Plus (*top level*) ;
- à partir d'une commande SQL (`SELECT`, `INSERT`, `UPDATE` ou `DELETE`) ;

 © Éditions Eyrolles

- à partir d'un programme PL/SQL (bloc, fonction ou procédure) ;
- à partir d'un déclencheur.

Sous SQL*Plus

Pour appeler la méthode publiée à partir de SQL*Plus, il faut utiliser la commande `CALL`. Dans le cas d'une fonction, il convient d'utiliser une variable globale de retour qu'il faudra définir auparavant (voir le chapitre 7).

```
CALL [nomPaquetage.]{nomProcédure([paramètre1[, paramètre2]…])
                    | nomFonction([paramètre1[,paramètre2]…]) INTO
                    :variable};
```

Le code suivant teste notre exemple :

Tableau 11-2 Appel d'une procédure stockée Java

Instruction SQL	Commentaires
`VARIABLE g_résultat VARCHAR2(50);`	Déclaration de la variable de retour.
`CALL PremierExemple_affiche('SQL, un langage évolué???') INTO :g_résultat;`	Appel de la méthode publiée.
`PRINT g_résultat;`	Affichage du résultat.

`G_RÉSULTAT`
`-------------------------------------`
`Le message contient 16 lettres!`	

Dans une commande SQL

La requête suivante invoque la procédure stockée. Nous utilisons la pseudo-table `DUAL` mais il est possible d'appeler une procédure stockée en passant en paramètres des valeurs de colonnes d'une table contenant des enregistrements.

```
SQL> SELECT PremierExemple_affiche('SQL, un langage évolué???')
FROM DUAL;

PREMIEREXEMPLE_AFFICHE('SQL,UNLANGAGEÉVOLUÉ???')
------------------------------------------------------------------
Le message contient 16 lettres!
```

Il est aussi possible d'appeler une procédure stockée dans une insertion, une modification ou une suppression.

Sous PL/SQL

L'appel de la même méthode sous PL/SQL se réalise classiquement comme s'il s'agissait d'une fonction cataloguée, elle-même écrite en PL/SQL.

```
DECLARE
   v_résultat VARCHAR2(50);
BEGIN
   v_résultat :=
PremierExemple_affiche('SQL, un langage évolué???') ;
END;
```

Dans un déclencheur

Le déclencheur suivant invoque une procédure stockée que nous décrirons dans le paragraphe « Déclencheurs » de cette section. Dans cet exemple, chaque suppression d'une compagnie déclenchera l'appel de la méthode Java correspondante en passant en paramètre le nom de la compagnie supprimée.

```
CREATE TRIGGER Ex_trig_Java
   AFTER DELETE ON Compagnie FOR EACH ROW
BEGIN
    DeuxièmeExemple_affiche(:OLD.nomcomp) ;
END;
```

Partage de la méthode

Une fois que la méthode a été publiée, elle devient un objet à part entière du schéma auquel on peut attribuer des prérogatives d'exécution (`GRANT EXECUTE ON PremierExemple_affiche TO Paul`).

Suppression de la méthode

Pour supprimer une méthode au niveau de la base, Oracle propose la procédure `dropjava` (issue du paquetage `DBMS_JAVA`) qu'on appelle soit en ligne de commande, soit sous PL/SQL ou SQL*Plus. Ainsi, le script suivant permet de supprimer les deux premiers exemples en utilisant deux écritures différentes (la deuxième écriture nécessite d'être connectée dans le schéma qui contient la procédure stockée).

```
dropjava -user soutou/ingres@cxbdsoutou PremierExemple.java
CALL DBMS_JAVA.DROPJAVA('DeuxièmeExemple.java');
```

Il est possible de supprimer de la même manière des classes (`.class`) ou des ressources (`.jar` ou `.zip`).

© Éditions Eyrolles

Interactions avec la base

Le mécanisme de communication entre la procédure stockée Java et la base de données est fondé sur la technologie JDBC que nous avons étudiée en détail au chapitre 9.

Connexion par défaut

La connexion par défaut est celle qui concerne l'utilisateur appelant la procédure. Le pilote JDBC utilisé est dit « interne » (*server-side internal JDBC driver*). Il est aussi possible de connecter un autre schéma sur une base locale ou distante en utilisant explicitement un autre pilote (*server-side JDBC Thin driver*, *client-side JDBC Thin* ou *JDBC OCI driver*).

Le code suivant décrit la classe `Compagnies` qui inclut la méthode `retourneCode`. Celles-ci a pour but de retourner le code de la compagnie dont le nom est passé en paramètre de la requête. La connexion par défaut est réalisée en passant le paramètre « `jdbc:default:connection:` » à l'appel de la méthode `getConnection`.

Tableau 11-3 Connexion par défaut

Code Java	Commentaires
`import java.sql.*;` `import oracle.jdbc.*;`	Importation des paquetages.

`public class Compagnies`
`{public static String retourneCode() throws SQLExcep-`
`tion`	

`{String codeComp = null;`
` try`
` { Connection cx = DriverManager.getConnection`
`                        ("jdbc:default:connection:");`
`  String sql = "SELECT comp FROM Compagnie`
`                WHERE nomComp ='Air France'";`
`  Statement stmt = cx.createStatement();`
`  ResultSet rs = stmt.executeQuery(sql);`
`  rs.next();`
`  codeComp = rs.getString(1);`
`  rs.close();`
`  stmt.close();`
	Établissement de la connexion par défaut.
	
	Extraction du code de la compagnie.

` } catch (SQLException e)` `    { System.err.print(e.getMessage());}`	Gestion des erreurs
` return codeComp; } }`	Retour du résultat.

Le tableau suivant synthétise les étapes à suivre pour pouvoir exploiter la méthode `retourneCode` en tant qu'objet de la base :

Tableau 11-4 Étapes à suivre

Ligne de commande	Commentaires
`javac Compagnies.java -classpath D:\oracle\ora92\jdbc\lib\ojdbc14.jar`	Compilation de la classe.

`loadjava -user soutou/ingres@cxbdsoutou -verbose Compagnies.class`	Chargement de la classe. Trace :
	`arguments: '-user' 'soutou/ingres@cxbdsoutou' '-verbose' 'Compagnies.class'`
	`creating : class Compagnies`
	`loading  : class Compagnies`

`CREATE FUNCTION JavaRetourneCodeComp`
`RETURN VARCHAR2 AS LANGUAGE JAVA`
`NAME 'Compagnies.retourneCode() return java.lang.String';`
`/`
	Publication de la méthode (sous SQL*Plus).

`SQL> VARIABLE le_code VARCHAR2(10);`
`SQL> CALL JavaRetourneCodeComp() INTO :le_code;`
`Appel terminé.`
 	
`SQL> PRINT le_code;`
`LE_CODE`
`--------------------------------`
`AF`
	Test de la procédure stockée (sous SQL*Plus).

Passage de paramètres

Chaque méthode d'une classe Java (même la méthode `main`) peut être publiée. Le code suivant présente une classe Java hébergeant la méthode `insereEtCompteComp` qui nécessite deux paramètres d'entrée. La classe contient aussi une méthode `main` qui affiche la valeur des paramètres d'entrée (quel que soit le nombre de ces derniers).

Nous allons à présent stocker ces deux méthodes en procédures au niveau de la base.

- la méthode `main` est publiée en spécifiant explicitement trois paramètres, par exemple ;
- la méthode `insereEtCompteComp` est publiée en respectant la correspondance des types SQL et Java.

Le tableau 11-6 synthétise les étapes à suivre pour pouvoir exploiter ces deux méthodes. Dans notre jeu d'exemple, trois compagnies sont déjà stockées dans la base.

Tableau 11-5 Classe Java ayant des paramètres en entrée

<table>
<tr><th>Code Java</th><th>Commentaires</th></tr>
<tr><td><pre>import java.sql.*;
import oracle.jdbc.*;
public class ExempleComplet {</pre></td><td></td></tr>
<tr><td><pre>public static void main (String[] args)
{System.out.println("Valeurs des paramètres : ");
 int i;
 for (i=0; i < args.length; i++)
    System.out.println("paramètre "+i+" = "+ args[i]);}</pre></td><td>Affichage des paramètres du main.</td></tr>
<tr><td><pre>public static int insereEtCompteComp(String param1,
                 String param2) throws SQLException
 {int retour = 0;
try
{Connection cx = DriverManager.getConnection
                 ("jdbc:default:connection:");
 String instruction =
           "INSERT INTO Compagnie VALUES (?,?)";
 PreparedStatement étatPréparé =
           cx.prepareStatement(instruction);
 étatPréparé.setString(1, param1);
 étatPréparé.setString(2, param2);
 étatPréparé.executeUpdate();
 instruction = "SELECT COUNT(comp) FROM Compagnie";
 étatPréparé = cx.prepareStatement(instruction);
 ResultSet rs = étatPréparé.executeQuery(instruction);
 rs.next();
 retour = rs.getInt(1);
 étatPréparé.close();
 cx.close();}</pre></td><td>Méthode insereEtCompteComp.

Insertion paramétrée.

Comptage des compagnies.</td></tr>
<tr><td><pre> catch (SQLException e)
    { System.err.println(e.getMessage());}
 return retour;} }</pre></td><td>Gestion des erreurs</td></tr>
</table>

La sortie par défaut d'une procédure cataloguée n'est pas l'écran. Pour rendre opérationnels vos affichages via les interfaces `System.out` et `System.err`, utilisez conjointement les procédures `SET SERVEROUTPUT ON SIZE n` de SQL*Plus et `SET_OUTPUT(n)` du paquetage `DBMS_JAVA`. La taille du buffer Java est au minimum (par défaut) de 2 000 octets; le maximum vaut 1 mégaoctet.

Les paramètres de sortie (déclarés en `OUT` or `IN OUT` au niveau de PL/SQL) doivent être déclarés au niveau de Java par une table d'un élément. Par exemple un paramètre `OUT` de type `NUMBER` devra être associé à un paramètre Java declaré en tant que `float[] tab`. L'affectation dans Java se fera au premier indice du tableau, soit `tab` [0].

Tableau 11-6 Passage de paramètres

Ligne de commande	Commentaires

`loadjava -user soutou/ingres@cxbdsoutou -verbose ExempleComplet.java`	Chargement et compilation de la classe. Trace :
	`arguments: '-user' 'soutou/ingres@cxbdsoutou' '-verbose' 'ExempleComplet.java'`
	`creating : source ExempleComplet`
	`loading : source ExempleComplet`
	`creating : ExempleComplet`

`CREATE PROCEDURE` **`ProgPrincipal`** `(p1 IN VARCHAR2, p2 IN VARCHAR2, p3 IN VARCHAR2)`
`AS LANGUAGE JAVA NAME`
`'ExempleComplet.main(java.lang.String[])';`
`/`
	Publication de la méthode `main`.

`CREATE FUNCTION` **`procédureDoubleEmploi`** `(p1 VARCHAR2, p2 VARCHAR2) RETURN NUMBER`
`AS LANGUAGE JAVA NAME`
`'ExempleComplet.insereEtCompteComp(java.lang.String,java.lang.String) return int';`
`/`
	Publication de la méthode `insereEtCompteComp`.

`SET SERVEROUTPUT ON SIZE 5000`
`CALL DBMS_JAVA.SET_OUTPUT(5000);`
`CALL` **`ProgPrincipal`**`('SQL', 'Oracle', 'Super!');`
	Test et résultats :
	`Valeurs des paramètres :`
	`paramètre 0 = SQL`
	`paramètre 1 = Oracle`
	`paramètre 2 = Super!`
	
	`Appel terminé.`

`SET SERVEROUTPUT ON SIZE 5000`
`CALL DBMS_JAVA.SET_OUTPUT(5000);`
`VARIABLE g_résultat NUMBER;`
`CALL` **`procédureDoubleEmploi`** `('SQL', 'Oracle AirLines') INTO :g_résultat;`
`PRINT g_résultat;`
	Test et résultats :
	`Appel terminé.`
	
	`SQL> PRINT g_résultat;`
	`G_RÉSULTAT`
	`----------`
	`4`

Une fonction qui possède des paramètres de sortie (`OUT` ou `IN OUT`) ne peut pas être appelée dans une instruction SQL.

 © Éditions Eyrolles

Paquetages

De manière analogue, il est possible de publier des méthodes Java en paquetages. La syntaxe est la suivante :

```
CREATE [OR REPLACE] PACKAGE nomPaquetage {IS | AS}
   Spécifications
END nomPaquetage;
CREATE [OR REPLACE] PACKAGE BODY nomPaquetage {IS | AS}
   Implémentations
END nomPaquetage ;
```

L'encapsulation dans un paquetage des deux méthodes de la précédente classe Java est réalisée de la manière suivante :

```
CREATE PACKAGE PaquetageJava IS
   PROCEDURE ProgPrincipal  (p1 IN VARCHAR2, p2 IN VARCHAR2, p3 IN
   VARCHAR2);
   FUNCTION procédureDoubleEmploi  (p1 VARCHAR2, p2 VARCHAR2)
   RETURN NUMBER;
END PaquetageJava;
/
CREATE PACKAGE BODY PaquetageJava IS
   PROCEDURE ProgPrincipal  (p1 IN VARCHAR2, p2 IN VARCHAR2, p3 IN
   VARCHAR2)
      AS LANGUAGE JAVA NAME 'ExempleComplet.main(java.lang.String[])';
   FUNCTION procédureDoubleEmploi  (p1 VARCHAR2, p2 VARCHAR2)
   RETURN NUMBER
      AS LANGUAGE JAVA
      NAME 'ExempleComplet.insereEtCompteComp(java.lang.String,java.
      lang.String)
      return int';
END PaquetageJava ;
/
```

L'appel de la méthode `main` est réalisé de la manière suivante :

```
SET SERVEROUTPUT ON SIZE 5000
CALL DBMS_JAVA.SET_OUTPUT(5000);
CALL PaquetageJava.ProgPrincipal('SQL', 'Oracle', 'Super!');
```

Déclencheurs

Oracle permet de programmer un déclencheur à l'aide d'une méthode Java stockée en tant que procédure. Ce mécanisme est aussi valable pour les déclencheurs de type `INSTEAD OF`.

L'exemple suivant décrit la méthode `affiche` de la classe Java `DeuxièmeExemple` qui affiche à l'écran le paramètre d'entrée dans un message prédéfini :

```
public class DeuxièmeExemple
{ public static void affiche(String param1)
  {System.out.println("La compagnie "+param1+" a été détruite."); } }
```

Le code du déclencheur a été décrit dans le paragraphe « Dans un déclencheur » de la section « Appel de la méthode ». L'appel de la procédure est déclenché lors de la suppression d'une compagnie. Le message affiché inclut le nom de la compagnie supprimée :

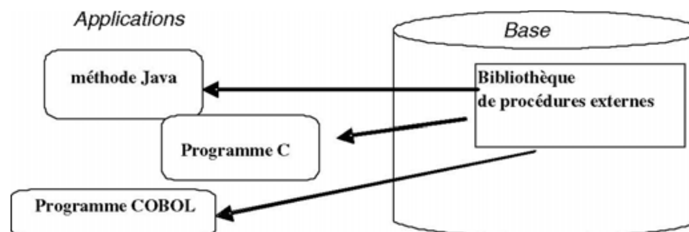
```
SQL> SET SERVEROUTPUT ON SIZE 5000
SQL> CALL DBMS_JAVA.SET_OUTPUT(5000);
Appel terminé.

SQL> DELETE FROM Compagnie WHERE comp = 'SQL5';
La compagnie Oracle AirLines5 a été détruite.
1 ligne supprimée.
```

Procédures externes Java

Le principe des procédures externes (*external procedures*) existe depuis la version 8 d'Oracle. Ce mécanisme offre la possibilité de faire appel à des programmes écrits dans des langages divers (C, C++, COBOL, Java, etc.). Ces programmes ne sont pas stockés dans l'environnement d'Oracle comme l'illustre la figure suivante :

Figure 11-3 *Procédures externes*



Comme dans le cas des procédures stockées, l'exploitation d'une procédure externe Java nécessite plusieurs phases :

- compilation du programme externe (ici de la classe Java) ;
- création d'une librairie ;
- publication de la procédure externe au travers d'une spécification PL/SQL ;
- appel de la procédure externe au travers de sa spécification.

 © Éditions Eyrolles

Compilation de la classe

Pour illustrer les mécanismes à mettre en œuvre, exploitons la méthode récursive `fib` de la classe Java `Fibonacci` qui calcule le résultat de la célèbre suite (1, 1, 2, 3, 5, 8, 13, 21… : chaque terme de la suite, à partir du deuxième, est la somme des deux termes qui le précédent) :

```
public class Fibonacci
{ public static int fib (int n)
   {if (n == 1 || n == 2)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);} }
```

La compilation de cette classe produit le fichier `Fibonacci.class` qui doit être placé dans un répertoire externe à Oracle (dans l'exemple `C:\WINDOWS\Temp`).

Création d'une librairie

Le chargement d'une procédure externe consiste à créer (si elle n'existe pas déjà) une librairie qui contiendra les exécutables par la commande `CREATE DIRECTORY`. La commande suivante définit la librairie de nom `répertoireProcExternes` qui référence le répertoire contenant les exécutables. Le privilège `CREATE ANY DIRECTORY` doit être acquis pour pouvoir exécuter cette commande.

```
CREATE DIRECTORY répertoireProcExternes AS 'C:\WINDOWS\Temp';
```

Le chargement de la classe est réalisé à l'aide de la commande SQL `CREATE JAVA`.

```
CREATE JAVA CLASS USING BFILE(répertoireProcExternes, 'Fibonacci.
class');
/
```

Publication d'une procédure externe

La publication d'une procédure externe Java est similaire à celle des procédures stockées. L'instruction suivante définit un point d'entrée de la méthode `fib` dans Oracle sous la forme de la signature de la fonction PL/SQL `fibonacciExterne` :

```
CREATE FUNCTION fibonacciExterne (n NUMBER) RETURN NUMBER
        AS LANGUAGE JAVA NAME 'Fibonacci.fib(int) return int' ;
/
```

Appel d'une procédure externe

Comme dans le cas des procédures stockées, l'appel d'une procédure externe peut être réalisé sous SQL*Plus, à partir d'une commande SQL, d'un programme PL/SQL (bloc, fonction ou procédure) ou d'un déclencheur.

Dans notre exemple, appelons la fonction sous SQL*Plus en demandant la somme des sept premiers termes, comme l'illustre la trace suivante :

```
SQL> VARIABLE n NUMBER
SQL> VARIABLE résultat NUMBER
SQL> EXECUTE :n := 7;
Procédure PL/SQL terminée avec succès.

SQL> CALL fibonacciExterne (:n) INTO :résultat ;
Appel terminé.
SQL> PRINT résultat
  RÉSULTAT
----------
        13
```


© Éditions Eyrolles

Chapitre 12

Oracle et le Web

En plus des servlets qui utilisent la technologie JDBC, Oracle propose différentes techniques qui permettent d'interfacer une base de données sur le Web :

- *PL/SQL Web Toolkit* offre la capacité d'écrire des programmes PL/SQL qui génèrent des pages HTML lorsqu'ils sont invoqués via un navigateur client.
- *PL/SQL Server Pages* (PSP) intègre des instructions PL/SQL dans des pages HTML (ou XML) par l'utilisation de balises spécifiques. Cette technique se rapproche des pages ASP (*Active Server Pages*) de Microsoft et de l'approche JSP (*Java Server Pages*) de Sun. Pour pouvoir développer des pages PSP, et si vous n'utilisez pas la configuration du Web, vous devez disposer d'un serveur 8.1.6 ou plus récent, et des cartouches PL/SQL d'Oracle sous Internet Application Server, WebDB, ou Oracle Application Server.
- *PHP* avec l'utilisation de l'API adéquate (fonctions `oci_`*xx*).

Le choix entre l'une de ces technologies sera guidé par le type d'application Web développée. Si l'application ne doit pas inclure un grand nombre d'instructions PL/SQL et contient beaucoup de code HTML, il vaut mieux utiliser *PL/SQL Server Pages*. Dans le cas suivant, *PL/SQL Web Toolkit* pourra être préféré. Si vous préférez le monde de l'Open Source, PHP sera un allié idéal pour concevoir vos sites interactifs.

Ce chapitre décrit, dans un premier temps, la configuration minimale d'Apache que vous pouvez adopter afin de pouvoir construire un serveur Web Oracle personnel. Après avoir étudié les caractéristiques de *PL/SQL Web Toolkit* et de *PL/SQL Server Pages*, nous détaillerons les moyens de faire interagir un programme PHP 5 avec une base Oracle.

Configuration minimale d'Apache

Oracle inclut un serveur Apache. Dans notre jeu d'exemple (Oracle9*i*) nous avons reconfiguré le numéro du port d'écoute à 77 (80 par défaut). Pour ce faire, modifiez les fichiers :

- `httpd.conf` sous `\oracle\ora92\Apache\Apache\conf\` (modifiez les entrées `Port 77` et `Listen 77`) ;
- `ports.ini` sous `\oracle\ora92\Apache\Apache\` (modifiez l'entrée `s_apachePort = 77`).

Arrêtez et redémarrez le service `OracleOraHome92HTTPServer` pour prendre en compte la nouvelle configuration.

Lancez l'explorateur en inscrivant le nom de votre machine suivi du numéro de port (exemple : `http://camparols:77/`), il vient :

Figure 12-1 *Menu général d'Apache sous Oracle*

Choissisez `Mod_plsql Configuration Menu`, puis `Paramètres des DAD de la passerelle`, enfin `Ajout d'un descripteur par défaut (configuration vide)`. Il vient l'écran suivant dans lequel vous devez nommer un DAD (*Data Access Descriptor*), ici `DADSOUTOU` :

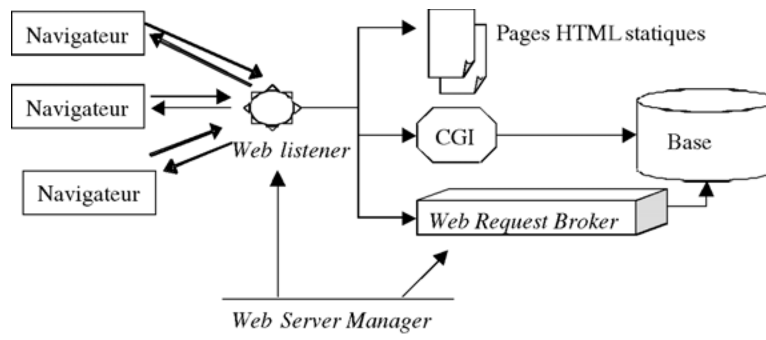
Figure 12-2 *Création d'un DAD*

Si vous voulez vous identifier à chaque démarrage, ne renseignez que le schéma et la chaîne de connexion à votre base locale. Si vous voulez ne pas avoir à ressaisir un nom d'utilisateur et un mot de passe, renseignez ces deux champs ainsi que la chaîne de connexion. Faites `OK`, Oracle doit signaler la création du nouveau DAD qui vous servira pour tester les exemples qui suivent.

PL/SQL Web Toolkit

L'architecture mise en place par Oracle pour les applications Web met en jeu différents composants. Le navigateur envoie une requête sous la forme d'une URL qui invoque une procédure PL/SQL. Le processus *Web listener* analyse et route l'appel vers le composant adapté (page statique, CGI, procédure PL/SQL, etc.) afin de générer une page HTML.

Figure 12-3 *Architecture du Web Oracle*



Le processus *Web Request Broker* est composé d'un ensemble de cartouches applicatives fournies en standard ou achetées à part (PL/SQL, Java, multimédia, etc.). Une cartouche contient du code qui permet la connexion à la base et l'exécution de procédures stockées.

Les paquetages HTP (*HyperText Procedures*) et HTF (*HyperText Functions*) regroupent un ensemble de sous-programmes permettant de générer des balises HTML. Il existe aussi d'autres fonctionnalités résidant dans le paquetage `OWA_UTIL` et `OWA` (*Oracle Web Agent*) que nous n'étudierons pas ici.

Détail d'une URL

Côté client, chaque URL doit être constituée de la manière suivante : `http://` *`adresseMachine.domaine:numPort/accèsPLSQL/programme`*. Le tableau ci-après détaille les composants d'une telle URL.

© Éditions Eyrolles

Tableau 12-1 Composition d'une URL

Composant	Commentaires
adresseMachine	Machine hébergeant les services Web.
domaine	Domaine internet.
numPort	Port UDP utilisé par le *Web listener*.
accèsPLSQL	Désigne le DAD, (*Data Access Descriptor*) qui réunit plusieurs paramètres (utilisateur, mot de passe, chaîne de connexion, pool de connexion, compatibilité SSO, etc.).
programme	Chemin d'accés à la procédure PL/SQL générant du HTML (de la forme [*nomSchéma*].[*nomPaquetage*].[*nomProcédure*]).

La figure suivante illustre le résulat généré par la procédure `NousSommesLe` détaillée plus loin. Les composant de l'URL sont : `http://camparols:77/pls/DADSOUTOU/NousSommesLe`.

- `camparols` est la machine qui héberge les services Web d'Oracle ;
- `77` est le port d'écoute UDP du *Web listener* ;
- `pls` précise la technologie utilisée (ici PL/SQL) ;
- `DADSOUTOU` désigne l'alias contenant les paramètres d'accés (utilisateur, mot de passe…) ;
- `NousSommesLe` est le nom de la procédure PL/SQL qui génère une page HTML.

***Figure 12-4** Appel de la procédure cataloguée* `NousSommesLe`

Paquetages HTP et HTF

Les tableaux suivants décrivent les principales procédures des paquetages HTP et HTF. Chaque procédure HTP possède une fonction HTF équivalente afin de pouvoir imbriquer deux balises.

Tableau 12-2 Balises générales

Procédures HTP	Balises HTML

`HTP.htmlOpen, HTP.htmlClose`
`HTP.headOpen, HTP.headClose`
`HTP.bodyOpen, HTP.bodyClose`
`HTP.comment`
	`<HTML>, </HTML>`
	`<HEAD>, </HEAD>`
	`<BODY>, </BODY>`
	`<!-- et -->`

© Éditions Eyrolles

Tableau 12-3 Balises d'en-tête

Procédures HTP	Balises HTML
HTP.base	<BASE>
HTP.title	<TITLE>
HTP.meta	<META>
HTP.script	<SCRIPT>
HTP.style	<STYLE>
HTP.isindex	<ISINDEX>

Tableau 12-4 Balises pour les applets Java

Procédures HTP	Balises HTML
HTP.appletopen, HTP.appletclose	<APPLET>, </APPLET>
HTP.param	<PARAM>

Tableau 12-5 Balises pour les listes

Procédures HTP	Balises HTML
HTP.olistOpen, HTP.olistClose	<OL>, </OL>
HTP.ulistOpen, HTP.ulistClose	<UL>, </UL>
HTP.dlistOpen, HTP.dlistClose	<DL>, </DL>
HTP.dlistTerm	<DT>
HTP.dlistDef	<DD>
HTP.dirlistOpen, HTP.dirlistClose	<DIR>, </DIR>
HTP.listHeader	<LH>
HTP.listingOpen, HTP.listingClose	<LISTING>, </LISTING>
HTP.menulistOpen, HTP.menulistClose	<MENU>, </MENU>
HTP.listItem	<LI>

Tableau 12-6 Balises pour les formulaires

Procédures HTP	Balises HTML
HTP.formOpen, HTP.formClose	<FORM>, </FORM>
HTP.formCheckbox	<INPUT TYPE="CHECKBOX">
HTP.formHidden	<INPUT TYPE="HIDDEN">
HTP.formImage	<INPUT TYPE="IMAGE">
HTP.formPassword	<INPUT TYPE="PASSWORD">
HTP.formRadio	<INPUT TYPE="RADIO">
HTP.formSelectOpen, HTP.formSelectClose	<SELECT>, </SELECT>
HTP.formSelectOption	<OPTION>
HTP.formText	<INPUT TYPE="TEXT">
HTP.formTextarea, HTP.formTextarea2	<TEXTAREA>
HTP.formTextareaOpen, HTP.formTextareaOpen2, HTP.formTextareaClose	<TEXTAREA>, </TEXTAREA>
HTP.formReset	<INPUT TYPE="RESET">
HTP.formSubmit	<INPUT TYPE="SUBMIT">

Tableau 12-7 Balises pour les tables

Procédures HTP	Balises HTML
HTP.tableOpen, HTP.tableClose	<TABLE>, </TABLE>
HTP.tableCaption	<CAPTION>
HTP.tableRowOpen, HTP.tableRowClose	<TR>, </TR>
HTP.tableHeader	<TH>
HTP.tableData	<TD>
HTF.format_cell	<TD>

Tableau 12-8 Balises images et ancres

Procédures HTP	Balises HTML
HTP.line, HTP.hr	<HR>
HTP.img, HTP.img2	<IMG>
HTP.anchor, HTP.anchor2	<A>
HTP.mapOpen, HTP.mapClose	<MAP>, </MAP>

Tableau 12-9 Balises de paragraphes

Procédures HTP	Balises HTML
HTP.header	<H1> à <H6>
HTP.para, HTP.paragraph	<P>
HTP.print, HTP.prn	génère du texte passé en paramètre.
HTP.prints, HTP.ps	génère du texte en caractères spéciaux.
HTP.preOpen, HTP.preClose	<PRE>, </PRE>
HTP.blockquoteOpen	<BLOCKQUOTE>,
HTP.blockquoteClose	</BLOCKQUOTE>
HTP.div	<DIV>
HTP.nl, HTP.br	
HTP.nobr	<NOBR>
HTP.wbr	<WBR>
HTP.plaintext	<PLAINTEXT>
HTP.address	<ADDRESS>
HTP.mailto	<A> avec l'attribut MAILTO
HTP.area	<AREA>
HTP.bgsound	<BGSOUND>

 © Éditions Eyrolles

Tableau 12-10 Balises de caractères

Procédures HTP	Balises HTML
HTP.basefont	<BASEFONT>
HTP.big	<BIG>
HTP.bold	<B>
HTP.center	<CENTER>, </CENTER>
HTP.centerOpen, HTP.centerClose	<CENTER>, </CENTER>
HTP.cite	<CITE>
HTP.code	<CODE>
HTP.dfn	<DFN>
HTP.get_download_files_list	<EM>
HTP.fontOpen, HTP.fontClose	<FONT>, </FONT>
HTP.italic	<I>
HTP.keyboard, HTP.kbd	<KBD>, </KBD>
HTP.s	<S>
HTP.sample	<SAMP>
HTP.small	<SMALL>
HTP.strike	<STRIKE>
HTP.strong	<STRONG>
HTP.sub	<SUB>
HTP.sup	<SUP>
HTP.teletype	<TT>
HTP.underline	<U>
HTP.variable	<VAR>

Tableau 12-11 Balises pour le multifenêtrage

Procédures HTP	Balises HTML
HTP.frame	<FRAME>
HTP.framesetOpen, HTP.framesetClose	<FRAMESET>,</FRAMESET>
HTP.noframesOpen, HTP.noframesClose	<NOFRAMES>,</NOFRAMES>

Générer du HTML avec HTP

Le code suivant (NousSommesLe.sql) décrit la procédure PL/SQL NousSommesLe dont l'appel est illustré dans la section précédente. Chaque procédure HTP génère une balise HTML.

Tableau 12-12 Génération d'une page HTML

<table>
<tr><th>Code PL/SQL</th><th>Page HTML générée</th></tr>
<tr><td>CREATE PROCEDURE NousSommesLe AS</td><td></td></tr>
<tr><td>BEGIN</td><td></td></tr>
<tr><td>HTP.htmlOpen;</td><td><HTML></td></tr>
<tr><td>HTP.headOpen;</td><td><HEAD></td></tr>
<tr><td>HTP.title('Quelle heure?');</td><td><TITLE>Quelle heure?</TITLE></td></tr>
<tr><td>HTP.headClose;</td><td></HEAD></td></tr>
<tr><td>HTP.bodyOpen;</td><td><BODY></td></tr>
<tr><td>HTP.header(1, 'Nous sommes le : '
||TO_CHAR(**SYSDATE**,'Day DD Month YYYY,
HH24:MM:SS'));</td><td><H1>Nous sommes le : Mercredi 03
septembre 2003, 10:09:44</H1>
</BODY></td></tr>
<tr><td>HTP.bodyClose;</td><td></HTML></td></tr>
<tr><td>HTP.htmlClose;</td><td></td></tr>
<tr><td>END;</td><td></td></tr>
</table>

Générer du HTML avec HTF

Les fonctions HTF sont utiles pour imbriquer deux balises HTML. Le résultat d'une fonction HTF est passé en paramètre d'une procédure HTP comme le montre le code suivant :

Tableau 12-13 Imbrication de balises HTML

Code PL/SQL	Page HTML générée
HTP.htmlOpen;	<HTML>
HTP.bodyOpen;	<BODY>
HTP.header(1, HTF.italic('Coucou'));	**<H1><I>Coucou</I></H1>**
HTP.bodyClose;	</BODY>
HTP.htmlClose;	</HTML>

Pose d'hyperliens

Il est possible de poser des hyperliens (ancres) statiques ou dynamiques. Les ancres statiques sont composées d'URL figées dans le code. Les ancres dynamiques sont construites au cours de l'exécution de la procédure.

Hyperliens statiques

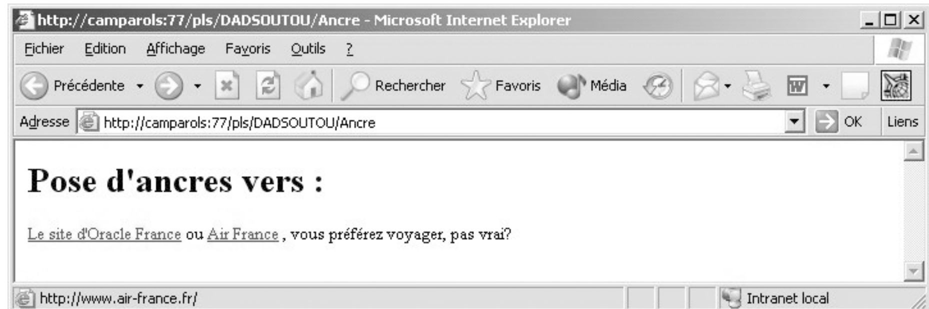
Le code suivant (Ancre.sql) présente l'exemple de deux ancres statiques. La première est studieuse, l'autre invite aux voyages :

Tableau 12-14 Pose d'ancres statiques

Code PL/SQL	Page HTML générée
CREATE OR REPLACE PROCEDURE Ancre AS	
BEGIN	
HTP.htmlOpen;	<HTML>
HTP.bodyOpen;	<BODY>
HTP.header(1,'Pose d''ancres vers : ');	<H1>Pose d'ancres vers : </H1>
HTP.**anchor**('http://www.oracle.fr', 'Le site d''Oracle France');	<A HREF="http://www.oracle.fr">Le site d'Oracle France</A>
HTP.print(' ou ');	ou
HTP.**anchor**('http://www.air-france.fr', 'Air France');	<A HREF="http://www.air-france.fr">Air France</A>
HTP.print(', vous préférez voyager, pas vrai? ');	, vous préférez voyager, pas vrai?
HTP.bodyClose;	</BODY>
HTP.htmlClose;	</HTML>
END;	

L'appel de cette procédure dans le navigateur produit l'affichage suivant :

Figure 12-5 *Pose d'hyperliens statiques*



Hyperliens dynamiques

Le code suivant (`affiche.sql`) décrit la construction dynamique d'une liste d'ancres (ici en fonction de la table `Avion`). Chaque ancre permettra d'appeler la procédure `afficheComp` (`afficheComp.sql`) qui affichera les caractéristiques de la compagnie propriétaire.

Tableau 12-15

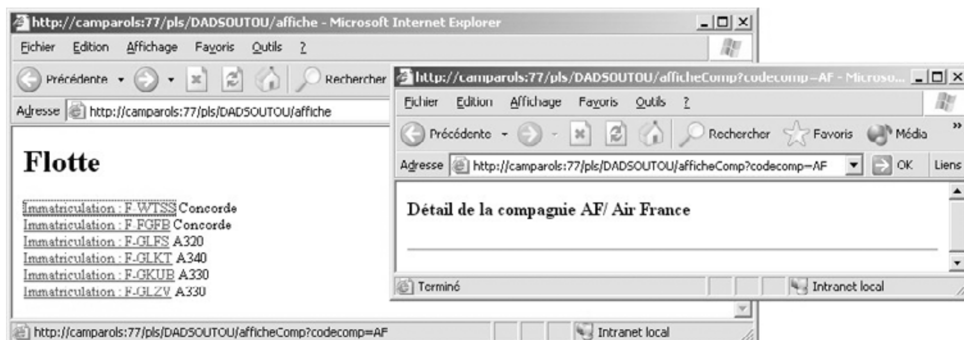
Code PL/SQL appelant	Code PL/SQL appelé

`CREATE PROCEDURE affiche IS`
`CURSOR curs IS SELECT immat, typeavion, comp FROM Avion;`
`BEGIN`
`HTP.htmlOpen;`
`HTP.bodyOpen;`
`HTP.header(1,'Flotte');`
`FOR ligne IN curs LOOP`
`  HTP.anchor('afficheComp?codecomp='|| TO_CHAR(ligne.comp),'Immatriculation : ' ||ligne.immat);`
`  HTP.print(' '||ligne.typeavion);`
`  HTP.br;`
`END LOOP;`
`HTP.bodyClose;`
`HTP.htmlClose;`
`END;`
	`CREATE PROCEDURE afficheComp(codecomp IN VARCHAR2 DEFAULT NULL) IS`
	` v_nom Compagnie.nomComp%TYPE;`
	`BEGIN`
	`HTP.htmlOpen;`
	`HTP.bodyOpen;`
	`SELECT nomComp INTO v_nom FROM Compagnie WHERE comp = codecomp;`
	`HTP.header(3,'Détail de la compagnie ' || codecomp ||'/ ' || v_nom );`
	`HTP.hr;`
	`HTP.bodyClose;`
	`HTP.htmlClose;`
	`END;`

L'appel de la procédure `affiche` et la sélection du premier avion déclenchent l'exécution de la procédure `afficheComp` comme le montre l'enchaînement des écrans suivants :

© Éditions Eyrolles

Figure 12-6 Hyperliens dynamiques



Formulaires

Le code suivant (`saisieComp.sql`) illustre la création d'un formulaire qui saisit le numéro d'une compagnie avant d'afficher ses avions par l'intermédiaire de la procédure `afficheAvions`.

Tableau 12-16 Formulaire

Code PL/SQL	Page HTML générée

```
CREATE PROCEDURE saisieComp(comp IN
VARCHAR2 DEFAULT NULL) IS
BEGIN
HTP.htmlOpen; HTP.bodyOpen;
HTP.header(1,'Avions d''une compa-
gnie','CENTER');
HTP.hr;
HTP.formOpen('saisieComp');
HTP.print('Numéro de la compagnie : ');
HTP.formText('comp');
HTP.hr;
HTP.formSubmit(cvalue=>'Executer');
HTP.formReset(cvalue=>''Annuler');
IF comp IS NOT NULL THEN
  afficheAvions(comp);
ELSE
  HTP.print('<BR>Saisir un code
compagnie <BR>');
END IF;
HTP.formClose;
HTP.bodyClose;
HTP.htmlClose;
END;
```

Figure 12-7 Formulaire de saisie

© Éditions Eyrolles

Tables

Les procédures HTP qui permettent de construire des tables sont intéressantes pour afficher des résultats de requêtes. Le code suivant (`afficheAvions.sql`) décrit la procédure appelée par le formulaire précédent. Un curseur permet de récupérer l'ensemble des avions appartenant à une compagnie dont le code est passé en paramètre (ici `AF`). La table produite est composée de trois colonnes.

Tableau 12-17 Création d'une table HTML

Code PL/SQL

```
CREATE PROCEDURE
afficheAvions(cp IN VARCHAR2)
IS
CURSOR curs IS SELECT immat,
typeAvion, cap FROM Avion WHERE
comp = cp;
begin
HTP.htmlOpen; HTP.bodyOpen;
HTP.line;
HTP.tableOpen('BORDER');
HTP.tableCaption('Flotte','CE
NTER');
HTP.tableRowOpen;
  HTP.tableHeader('Numero');
  HTP.tableHeader('Type');

HTP.tableHeader('Capacité');
HTP.tableRowClose;
FOR ligne IN curs LOOP
   HTP.tableRowOpen;

HTP.tableData(ligne.immat);
HTP.tableData(ligne.typeAvion);
   HTP.tableData(ligne.cap);
   HTP.tableRowClose;
END LOOP;
HTP.tableClose;
HTP.bodyClose; HTP.htmlClose;
END;
```

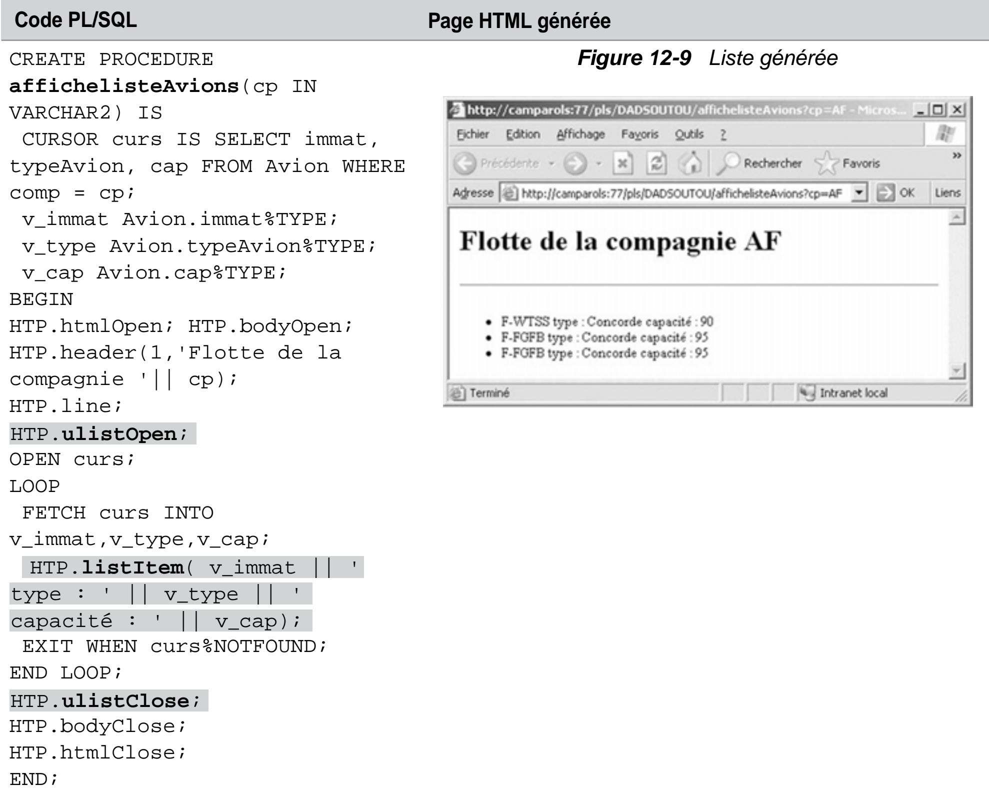
Page HTML générée

Figure 12-8 Table HTML

Listes

Le code suivant (`affichelisteAvions.sql`) décrit l'affichage des avions d'une compagnie dont le code passe en paramètre au niveau de l'URL (testée dans l'écran suivant avec le code 'AF'). La procédure génère une liste HTML.

Tableau 12-18 Création d'une liste HTML



Code PL/SQL	Page HTML générée

```
CREATE PROCEDURE
affichelisteAvions(cp IN
VARCHAR2) IS
 CURSOR curs IS SELECT immat,
typeAvion, cap FROM Avion WHERE
comp = cp;
 v_immat Avion.immat%TYPE;
 v_type Avion.typeAvion%TYPE;
 v_cap Avion.cap%TYPE;
BEGIN
HTP.htmlOpen; HTP.bodyOpen;
HTP.header(1,'Flotte de la
compagnie '|| cp);
HTP.line;
HTP.ulistOpen;
OPEN curs;
LOOP
 FETCH curs INTO
v_immat,v_type,v_cap;
  HTP.listItem( v_immat || '
type : ' || v_type || '
capacité : ' || v_cap);
 EXIT WHEN curs%NOTFOUND;
END LOOP;
HTP.ulistClose;
HTP.bodyClose;
HTP.htmlClose;
END;
```

Figure 12-9 Liste générée

PL/SQL Server Pages

La technologie PSP (*PL/SQL Server Pages*) permet d'intégrer un script PL/SQL dans une source HTML. Une fois écrit, le fichier d'extension `psp` doit être chargé dans la base comme une procédure stockée.

Généralités

Les fichiers PSP peuvent être composés :

- de balises HTML ;
- d'une procédure PL/SQL générant toutes les balises HTML (sans inclure de procédures HTP) ;
- d'instructions PL/SQL intégrées au code HTML.

 © Éditions Eyrolles

La structure générale d'un programme PSP est décrite dans la figure suivante. Les instructions PL/SQL se trouvent entre les balises <% et %>.

Figure 12-10 Structure générale d'un programme PSP

```
<%@ page language="PL/SQL" %>                        Exemple1.psp
<%@ plsql procedure="nomProcédure" %>
<%@ plsql parameter=" nomVariable" %>
…
<%! variableLocale typevariable; %>
…
<HTML>
<BODY>
  <% instructions PL/SQL %>
  < Balises HTML >
  <% ... %>
</BODY>
</HTML>
```

Balises

Le tableau suivant présente les principales balises propres à PSP :

Tableau 12-19 Imbrication de balises HTML

Directive	Signification
<%@ page language="PL/SQL" %>	Identification d'un fichier de type PSP.
<%@ plsql procedure="*nomProcedure*" %>	Nom de la procédure stokée.
<%@ plsql parameter="*nomVariable*" %>	Paramètre de la procédure stokée.
<%-- commentaire --%>	Commentaire non visible dans la page HTML produite.
<!-- commentaire -->	Commentaire visible dans la page HTML produite.
<%@ include file = " *nomFichier* " %>	Inclure le contenu d'un autre fichier.
<%! *nomvariable typevariable;* %> <%! *nomVariable typeVariable* := *valeurInitiale;* %>	Déclarer (et initialiser) une variable.
<% *instructions PL/SQL* %>	Bloc d'instruction PL/SQL.
<%= *expression* %>	Extraction d'une expression PL/SQL (exemple : valeur d'une variable).

Le type d'un paramètre d'une procédure stockée est VARCHAR2. Pour spécifier un autre type, utiliser la directive TYPE="*nomType*". Il est aussi possible de préciser une valeur par défaut avec la directive DEFAULT="*expression*".

Le contenu d'un fichier PSP est généralement traduit en page HTML. Il est toutefois possible de générer un résultat d'un type différent à l'aide de l'instruction `<%@ page contentType ="typeMIME" %>`. Le type définit la nature du fichier produit (`text/html`, `text/xml`, `text/plain`, `image/jpeg`…).

Une variable PL/SQL n'est visible que dans un seul bloc (entre `<%` et `%>`).

Chargement d'un programme PSP

Les sources PSP doivent être chargées (compilées) dans la base de données en tant que procédures stockées. Pour ce faire, Oracle fournit l'utilitaire `loadpsp`. Cet utilitaire s'utilise en ligne de commande de la manière suivante :

```
loadpsp -replace -user utilisateur/motdepasse@chaineconnexion
nomSource.psp
```

La procédure stockée générée (de nom *nomSource*) contient les instructions HTP et HTF qui permettent de construire la page HTML finale.

Appel

Après avoir chargé la source PSP dans la base, il est possible de l'appeler directement dans l'URL d'un navigateur (comme pour les procédures écrites sous PL/SQL *Web Toolkit*).

```
http://adresseMachine.domaine:numPort/appliPLSQL/accès
```

- *appliPLSQL* : alias de l'application et de la cartouche. Les paramètres sont constitués par le DAD (*Data Access Descriptor*) qui identifie l'accès à la base ;
- *accès* Chemin d'accés à la procédure PL/SQL avec d'éventuels paramètres (de la forme *nomSchéma.nomProcédure?paramètres=valeurs*).

Il est aussi possible de l'appeler dans une page HTML ou dans une autre procédure PSP (hyperlien ou via la méthode `POST` d'un formulaire).

Interaction avec la base

L'utilisation d'instructions PL/SQL dans un script PSP permet de manipuler facilement des données de la base. Les balises HTML permettront de développer l'interface graphique (affichages et saisies).

Formulaire

Les lignes suivantes (`codecompagnie.psp`) décrivent le code de la procédure PSP `codecompagnie` qui effectue, par l'intermédiaire d'un formulaire, la saisie du code d'une compagnie aérienne. L'affichage des avions de cette compagnie sera réalisé par la procédure

listeavions appelée par la méthode POST du formulaire (quand l'utilisateur actionnera le bouton Valider). La procédure codecompagnie ne contient pas d'instructions PL/SQL.

Tableau 12-20 Formulaire généré par une page PSP

Code PSP	Commentaires
<%@ page language="PL/SQL" %>	
<%@ plsql procedure="**codecompagnie**" %>	Nom de de la procédure stockée.
<HTML><BODY>	
<H1>Liste des avions </H1> <HR>	
<FORM METHOD="**POST**" ACTION = "**listeavions**">	
<p>Saisir le code d'une compagnie :	Définition de la procédure appelée.
<INPUT TYPE = "text" NAME = "**codecomp**"> 	
<INPUT TYPE = "submit" VALUE = "Valider">	Définition du paramètre.
<INPUT TYPE = "reset" VALUE = "Annuler">	
</FORM></BODY></HTML>	

Après compilation de ces deux procédures, l'appel de codecompagnie produit l'écran suivant. Supposons que l'utilisateur saisisse 'AF' dans la zone de texte.

Figure 12-11 Appel de la procédure codecompagnie

Affichage des résultats

Le code suivant (listeavions.psp) décrit la procédure listeavions qui réalise l'affichage des avions d'une compagnie dont le code passe en paramètre :

Tableau 12-21 Source PSP définissant un formulaire

Code PSP	Commentaires
<%@ page language="PL/SQL" %>	Nom de la procédure stockée.
<%@ plsql procedure="**listeavions**" %>	Paramètre de la procédure.
<%@ plsql parameter="**codecomp**" %>	
<HTML><BODY>	
<H1> Avions de la compagnie <%= **codecomp** %> </H1> 	Affichage du paramètre.
<% FOR ligne IN (SELECT immat, typeAvion, cap FROM Avion WHERE comp = **codecomp**) LOOP %>	Curseur parcourant la table Avion.
<LI>	
Immatriculation : <%= ligne.immat %> - <%= ligne.typeAvion %> - <%= ligne.cap %> places 	
</LI> <% END LOOP; %></BODY></HTML>	

L'activation du bouton `Valider` au niveau de la page précédente produit l'écran suivant :

Figure 12-12 *Affichage des résultats (appel de listeavions)*

Intégration de PHP

Configuration adoptée

Plusieurs configurations sont possibles en fonction de la version de PHP utilisée, de la version d'Apache et de celle d'Oracle. Nous avons opté pour faire interagir un programme PHP5 avec une base Oracle 10*g* sous Apache 1.3. Pour ceux qui travaillent sous Oracle 9*i*, l'étape d'installation d'Apache n'est pas nécessaire, il faudra seulement modifier les fichiers de configuration. Je décris ici une procédure minimale sans plus d'explication. Vous trouverez sur le Web de nombreuses ressources à ce sujet.

© Éditions Eyrolles

Les logiciels

Récupérez et installez Apache 1.3 (`www.apache.org`). Lancer Apache.exe s'il n'est pas automatiquement lancé après l'installation. Testez le service dans le navigateur en fonction du nom de serveur que vous avez spécifié à l'installation (`http://camparols` dans mon cas).

Installez PHP 5 (`http://www.php.net/downloads.php`) en dézippant le fichier téléchargé dans un répertoire personnel (`C:\PHP` dans mon cas).

Les fichiers de configuration

Dans le fichier `httpd.conf` (situé dans `C:\Program Files\Apache Group\Apache\conf\httpd.conf` dans mon cas), modifiez ou ajoutez les lignes suivantes (le « # » désigne un commentaire) :

```
# modif pour Oracle et PHP ici 9999 par exemple
Port 9999
…
#Ajout pour Oracle et PHP
LoadModule php5_module "c:/php/php5apache.dll"
…
# Ajout pour Oracle et PHP
AddModule mod_php5.c
SEtEnv PHPRC C:/php
AddType application/x-httpd-php .php
…
# Ajout pour Oracle et PHP
DirectoryIndex index.html index.php
…
# Ajout pour Oracle : répertoire contenant les sources php (pas
d'accent dans les noms de répertoire)
DocumentRoot "D:/dev/PHP-Oracle"
```

Dans le fichier `php.ini` (se trouvant dans `C:\WINDOWS` dans mon cas), modifiez ou ajoutez les lignes suivantes (le « ; » désigne un commentaire) :

```
; Ajout pour Oracle et PHP
extension_dir = "C:\PHP\ext"
; enlever le commentaire sur la ligne :
extension=php_oci8.dll
```

© Éditions Eyrolles

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).

Web

```
<html> <head> <title>test Apache 1.3 PHP5 </title> </head>
   <body> Test de la configuration Apache1.3 - PHP5
      <?php
             phpinfo();
      ?>
   </body> </html>
```

Pour tester votre serveur, arrêter puis relancer le. Dans le navigateur saisir l'URL de votre serveur sur le port concerné (`http://camparols:9999` dans mon cas) qui doit lancer le programme `index.php`. Vous devez voir le message « Test de la configuration Apache1.3 - PHP5 » suivi de la configuration actuelle de PHP (résultat de la fonction système PHP `phpinfo`).

Test d'Apache, de PHP et d'Oracle

Il faut que le *listener* et la base10*g* soient lancés (vérifiez dans `Services` à partir du panneau de configuration). Écrire le programme `cx1.php` et disposez le dans le répertoire contenant les sources PHP. Inscrivez votre nom d'utilisateur Oracle, le mot de passe et le nom ou la description du service (se trouvant dans le fichier `tnsnames.ora` qui réfère à votre instance Oracle). Il est à noter 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)`.

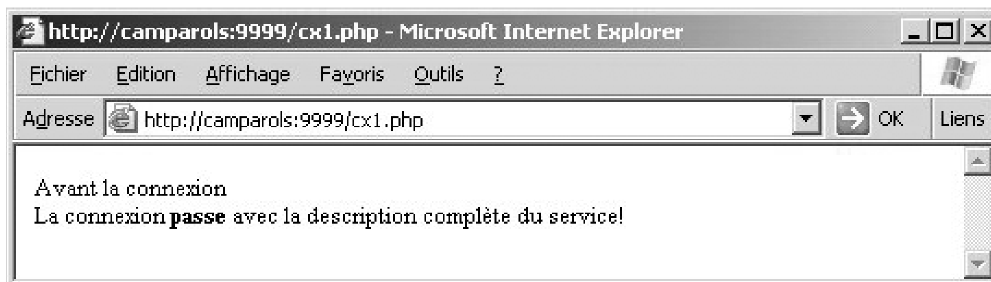
Web

```
<?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);
?>
```


© Éditions Eyrolles

Tester votre programme dans le navigateur (`http://camparols:9999/cx1.php` dans mon cas). Vous devez obtenir le résultat suivant.

Figure 12-13 *Test d'une connexion*



API de PHP pour Oracle

Les extensions Oracle des premières versions de PHP (fonctions préfixées par `ora`) sont désormais obsolètes (elles étaient valables avec les bases 7 et 8.0). À partir de la version 8*i*, sont apparues des fonctions PHP (entre 50 et 100 fonctions suivant la version de PHP utilisée) préfixées par `oci8`. Ces fonctions permettaient la gestion des types avancés (LOB, descripteurs de fichiers, objets, collections et ROWID) et amélioraient la manipulation des métadonnées (vues du dictionnaire des données).

Depuis PHP5, les noms de fonctions de cette dernière extension ont changé. Le préfixe est désormais `oci_`. Ainsi `ociexecute` est devenu `oci_execute`, `ocilogon` est devenu `oci_connect`, etc. Les anciens noms sont encore supportés mais il est plus prudent de ne plus les utiliser. Avec cette nouvelle API, la fonctionnalité de lecture des lignes (*fetch*) est plus puissante et de nouvelles fonctions apparaissent : préfixées par `lob->` pour les LOB et `collection->` pour les collections. Nous n'étudierons pas ces deux dernières familles de fonctions qui sont très spécifiques.

Étudions à présent les fonctions de base de l'API de PHP5 pour Oracle.

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 12-22 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()`	Ferme la connexion courante.

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 12-23 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.

© Éditions Eyrolles

Tableau 12-24 Constantes prédéfinies

Constante	Commentaire
`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`.
`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 12-25 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 12-26 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 tableau suivant décrit le code PHP qui insère 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 12-27 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 »).

 © Éditions Eyrolles

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 12-28 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 tableau suivant décrit le code PHP utilisant oci_fetch_array afin d'extraire les avions de la compagnie de code 'AF'. 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 ».

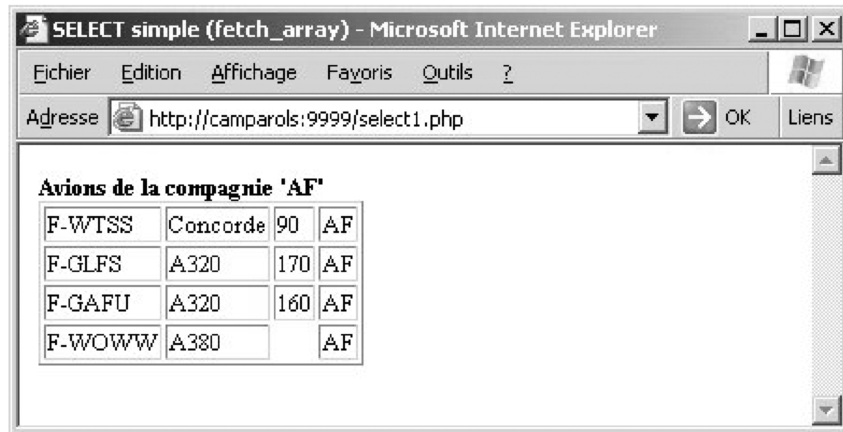
Tableau 12-29 Fonction oci_fetch_array

Code PHP	Commentaires
$requete = "SELECT * FROM Avion WHERE	Création de la requête.
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>" ;}	
print "</TR> "; }	
print "</TABLE> ";	Affichage.

Le résultat est le suivant (en supposant que la compagnie 'AF' dispose de 4 avions dont un est affecté d'une capacité nulle).

 © Éditions Eyrolles

Figure 12-14 Exemple avec oci_fetch_array



Le tableau suivant décrit le code PHP utilisant `oci_fetch_all` pour extraire tous les avions (sauf les 2 premiers grâce au troisième paramètre de la fonction d'extraction). 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 12-30 Fonction oci_fetch_all

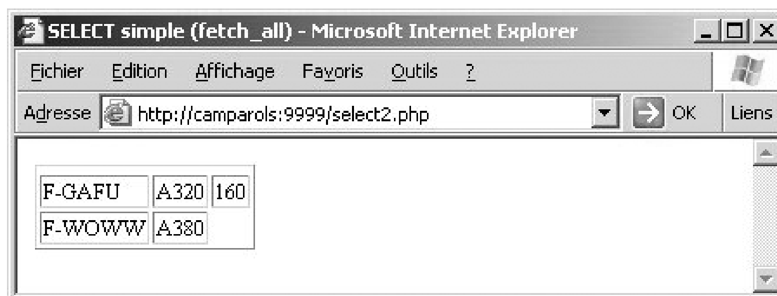
Code PHP	Commentaires
$requete = "SELECT immat, typeavion, capacite FROM Avion";	Création de la requête.
$ordre = oci_parse ($cx, $requete); oci_execute ($ordre); $nblignes = **oci_fetch_all**($ordre, $tabresults, 2); if ($nblignes > 0) { print "<table border=1>\n"; print"<tr>\n";	Exécution de la requête. Obtention du nombre de lignes et chargement.

for ($i = 0; $i < $nblignes; $i++) {
reset($tabresults);
print "<tr>\n";
while ($col = each($tabresults)) {
	Parcours du curseur.

$donnée = $col['value'];
print "<td>$donnée[$i]</td>\n";}
print "</tr>\n"; }
print "</table>\n";
}
else
print "Pas de données
\n"; }
	Parcours des colonnes.
	Affichage des colonnes.

Le résultat est le suivant (en supposant toujours que la base ne stocke que 4 avions).

Figure 12-15 *Exemple avec oci_fetch_all*



Tableaux associatifs

Ci-dessous, un exemple d'utilisation d'un tableau associatif.

```
$query = "SELECT immat, compa FROM Avions";
$statement = oci_parse($cx, $query);
oci_execute ($statement);
while ($row = oci_fetch_array ($statement, OCI_BOTH))
   {// $row[0 et $row['IMMAT'] désignent la même donnée
    // $row[1] et $row['COMPA'] désignent la même donnée }
```

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`.

Tableau 12-31 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.

© Éditions Eyrolles

Le tableau suivant décrit le code PHP utilisant `oci_define_by_name` afin d'extraire l'immatriculation et le type de tous les avions. Les deux variables PHP sont définies avant d'exécuter l'ordre.

Tableau 12-32 Fonction oci_define_by_name

Code PHP	Commentaires
`$requete   = "SELECT * FROM Avion";`	Création de la requête.
`$ordre = oci_parse ($cx, $requete);`	
`oci_define_by_name``($ordre, "IMMAT", $immatriculation);`	Définition des variables PHP.
`oci_define_by_name``($ordre, "TYPEAVION", $typav);`	
`oci_execute ($ordre);`	Exécution de la requête.
`print "<B>Liste des avions</B>";`	
`print "<TABLE BORDER=1> ";`	
`while (oci_fetch_array($ordre))`	Chargement et parcours
`       {print "<TR> <TD> $immatriculation </TD>" ;`	du curseur.
`       print "     <TD> $typav     </TD> </TR> "; }`	
`print "</TABLE> ";`	Affichage des colonnes.

Le tableau suivant décrit le code PHP utilisant `oci_bind_by_name` pour faire passer deux paramètres (variables PHP) lors de l'insertion d'une nouvelle compagnie.

Tableau 12-33 Fonction oci_bind_by_name

Code PHP	Commentaires
`$codeComp = "CAST";`	Affectation des variables
`$nomComp  = "Castanet Air";`	PHP.
`$insert2 = "INSERT INTO Compagnie VALUES (:v1, :v2) ";`	Définition de l'ordre
`$ordre   = oci_parse ($cx, $insert2);`	paramétré.
`oci_bind_by_name``($ordre, ":v1", $codeComp, -1);`	Association avec les
`oci_bind_by_name``($ordre, ":v2", $nomComp,  -1);`	variables PHP.
`oci_execute($ordre);`	Exécution de l'ordre.
`oci_commit($cx);`	

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`).

© Éditions Eyrolles

Tableau 12-34 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 tableau suivant décrit le code entier PHP utilisant `oci_error` (sans paramètre) pour débuter une transaction. Dans cet exemple, la connexion ne se déroule pas correctement du fait d'un nom erroné du serveur.

Tableau 12-35 Fonction oci_error (sans paramètre)

Code PHP	Commentaires
`<html> <head> <title>Erreur connexion </title> </head>`	

`<body>`
`<?php`
`$service = "(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP) (HOST = `**`toto`**`) (PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = bdcs10g)))";`
`$utilisateur = "soutou";`
`$mdp = "iut";`
	Début du code PHP.

`$cx = @oci_connect($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`**`'];` ` print " <B>Code    : </B>" . $taberr['`**`code`**`'];`	Affichage 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.

 © Éditions Eyrolles

Figure 12-16 Problème de connexion décelé à l'aide de oci_error

Le tableau suivant décrit le code PHP utilisant `oci_error` (avec paramètre) afin de récupérer une erreur survenue lors d'une extraction éronnée de données. 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 12-36 Fonction oci_error (avec paramètre)

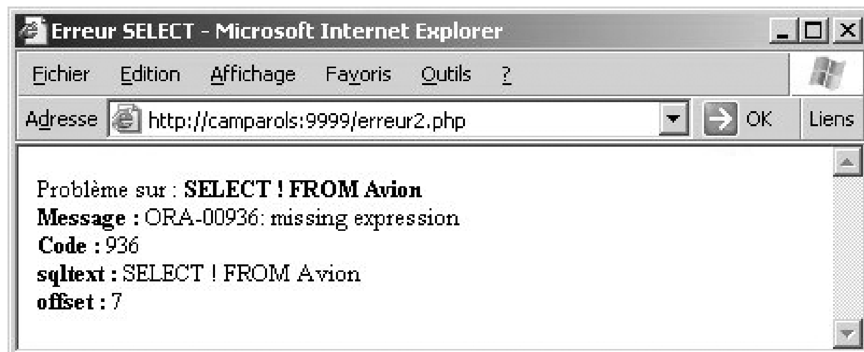
Code PHP	Commentaires

`$requetePB = "SELECT ! FROM Avion";`
`$ordre = oci_parse($cx,$requetePB);`
`if (!@oci_execute($ordre))`
	Analyse et exécution de l'ordre (erroné).

`{` `  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'];}`	Récupération de l'erreur. Affichage détaillé du tableau associatif contenant le résultat de l'erreur.

Le résultat est le suivant.

Figure 12-17 Erreur de syntaxe SQL décelé à l'aide oci_error

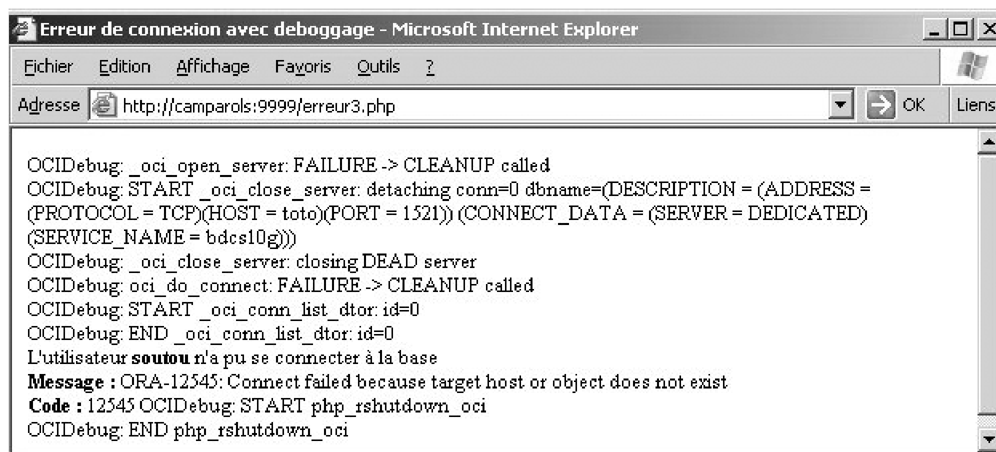


Supposons à présent qu'on active le déboggage interne lors d'une erreur de connexion (ajout de l'appel à `oci_internal_debug` à notre premier exemple d'erreur de connexion `erreur1.php`).

```
// active le deboggage
   oci_internal_debug(1);
   $cx = @oci_connect($utilisateur ,$mdp, $service);
   …
```

Le résultat est le suivant, il détaille les appels aux différents fonctions *OCI* d'Oracle.

Figure 12-18 *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 tableau suivant décrit le code PHP qui appelle cette procédure afin d'augmenter de 50 la capacité des avions de la compagnie de code 'AF'. 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 12-37 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 12-38 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* ».
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 tableau suivant décrit le code entier PHP utilisant oci_server_version et oci_field_is_null afin d’extraire les avions de capacité nulle (colonne capacite valant NULL).

Tableau 12-39 Affichage de la version et test de nullité

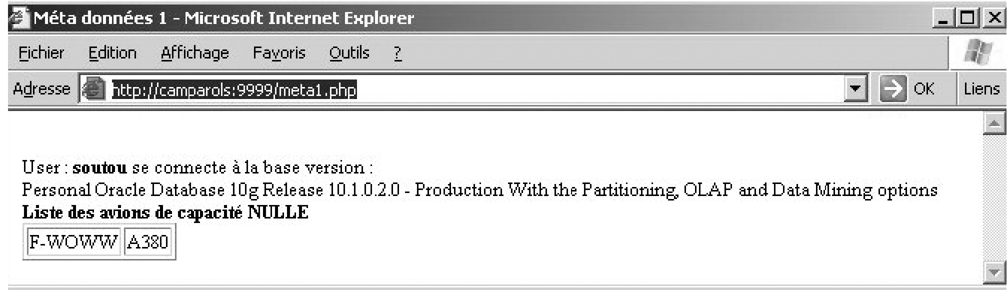
Code PHP	Commentaires
$cx = oci_connect($utilisateur ,$mdp, $service);	
print " User : <B>$utilisateur</B> se connecte à la base version : "; print **oci_server_version**($cx);	Affichage de la version de la base utilisée.

$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.

 © Éditions Eyrolles

Le résultat est le suivant.

Figure 12-19 Affichage de la version de la base et test de nullité d'une colonne



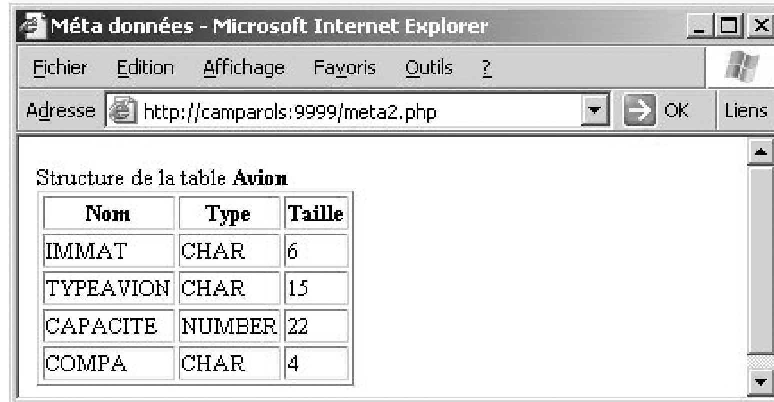
Le tableau suivant décrit le code entier PHP utilisant `oci_num_fields`, `oci_field_name`, `oci_field_type` et `oci_field_size` afin d'extraire la structure complète (en terme de colonnes) d'une table.

Tableau 12-40 Extraction de la structure d'une table

Code PHP	Commentaires
`$ordre = oci_parse($cx, "SELECT * FROM Avion");`	Extraction des avions de la base.
`oci_execute($ordre);`	
`print "Structure de la table <B>Avion</B>";`	
`print "<table border=1>";`	
`print "<tr><th>Nom</th>";`	
`print "<th>Type</th>";`	
`print "<th>Taille</th></tr>";`	
`$ncols = oci_num_fields($ordre);`	Extraction du nombre de colonnes.
`for ($i = 1; $i <= $ncols; $i++)`	
`{$col_nom  = oci_field_name($ordre, $i);`	Nom, type et taille de la colonne extraite.
`$col_type = oci_field_type($ordre, $i);`	
`$col_size = oci_field_size($ordre, $i);`	
`print "<tr><td>$col_nom</td>";`	Affichage des informations extraites.
`print "<td>$col_type</td>";`	
`print "<td>$col_size</td></tr>";`	
`}`	
`print "</table>\n";`	

Le résultat est le suivant.

Figure 12-20 Extraction de la structure d'une table



Nom	Type	Taille
IMMAT	CHAR	6
TYPEAVION	CHAR	15
CAPACITE	NUMBER	22
COMPA	CHAR	4

© Éditions Eyrolles

Chapitre 13

Oracle XML DB

Généralités

XML a été pris en compte il y a une dizaine d'années par Oracle8*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 données `XMLType` est dédié à la gestion de contenus XML. Grâce à la *Release 2* de la version 9*i*, il est possible d'y associer des grammaires *XML Schema* et de travailler avec le produit *XML Repository*. Depuis la version 10*g*, des efforts ont surtout été réalisés en ce qui concerne l'évolution des grammaires *XML Schema*. La version 11*g* propose le mode de stockage *binary XML* et un accès par *Web Services*.

XML DB est le nom de la technologie Oracle permettant de gérer du contenu XML en base (stockage, mises à jour et extractions). Alors que la plupart des SGBD natifs XML ne permettent que la persistance, XML DB offre en plus de nombreuses fonctionnalités (contrôle des transactions, intégrité des données, réplication et indexation).

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*. Sans XML DB, on peut toutefois stocker du contenu XML (sans pouvoir bien le manipuler par la suite) soit par une API (*XML Developer's Kit*), soit en utilisant des colonnes de type *Large Object Binary* : `CLOB`, `BLOB`, `BFILE` ou même `VARCHAR`.

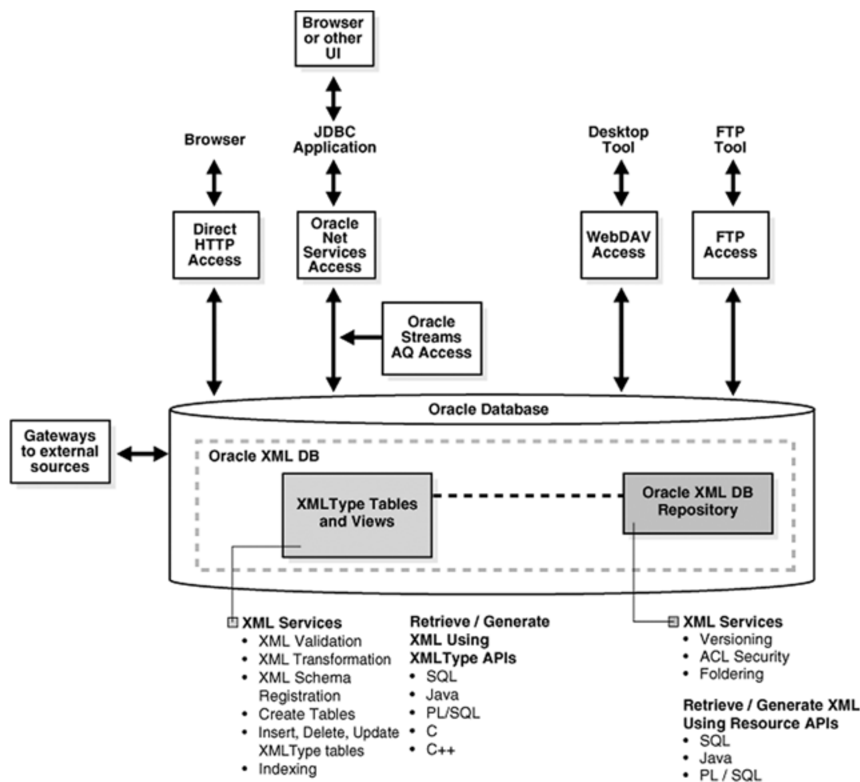
Ce chapitre présente les principales fonctionnalités de ce produit. Certaines figures sont extraites de la documentation *Oracle XML DB Developer's Guide*.

Comment disposer de XML DB ?

XML DB est opérationnel si vous avez choisi les options par défaut (*general purpose*) lors de l'installation. Dans le doute, vous pouvez interroger le dictionnaire de données pour vérifier la présence de l'utilisateur `XDB` (le compte doit être déverrouillé) `SELECT USERNAME, ACCOUNT_STATUS FROM DBA_USERS` ou de la vue `RESOURCE_VIEW`.

La figure suivante illustre l'architecture générale de XML DB.

Figure 13-1 *Architecture de XML DB*



Le type de données XMLType

Le type de données `XMLType` fournit de nombreuses fonctionnalités. La plupart sont relatives à XML (validation de schéma XML et transformation XSL), d'autres concernent SQL :

- définition d'une colonne d'une table (jouant le rôle d'un type de données) ;
- définition d'une table (jouant le rôle d'un type d'objet) ;
- déclaration de variables PL/SQL ;
- appel de procédures PL/SQL cataloguées.

Par défaut, une table (ou colonne) `XMLType` peut contenir n'importe quel document XML bien formé. De plus, le document peut être contraint selon une spécification *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 du document XML en fonction du mode de stockage choisi.

Modes de stockage

Suivant le contenu XML à stocker, Oracle fournit différents mécanismes pour enregistrer et indexer les données. Le contenu d'un document XML peut être :

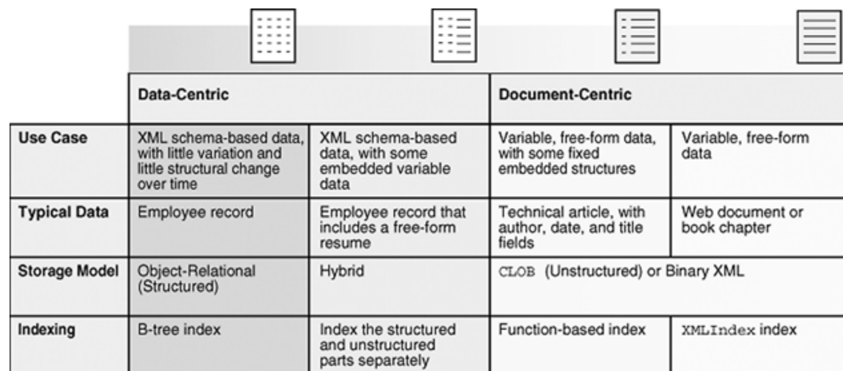
- orienté « données » (*data-centric*), structure régulière des données à granularité fine (la plus petite unité indépendante de donnée est située au niveau d'un élément `PCDATA` ou d'un attribut), contenant peu ou aucun contenu mixte (éléments mêlant contenu, balises et attributs) ;
- orienté « document » (*document-centric*), structure moins régulière des données qui présentent une granularité importante avec de nombreux contenus mixtes. L'ordre dans lequel les éléments apparaissent est très significatif (page Web, par exemple).

Plusieurs modes de stockage d'un type `XMLType` sont possibles :

- Stockage non structuré – les données sont enregistrées en tant que `CLOB`, l'index est de nature *function-based*.
- Stockage binaire (*binary XML*) – les données sont aussi enregistrées en tant que `CLOB` mais dans un format binaire conçu pour des données XML. L'index est de nature `XMLIndex` (les contenus XML doivent être associés à des grammaires *XML Schema*).
- Stockage structuré ou hybride – les données sont enregistrées comme un ensemble d'objets (au sens *object-relational* d'Oracle [SOU 04]), l'index est de nature *B-tree*.

A priori, les documents orientés données devraient être stockés selon un mode structuré ou hybride. Le stockage non structuré conviendrait davantage aux autres documents. Quel que soit le mode de stockage, les fonctions d'extraction de XML DB produisent les mêmes résultats. En revanche, les possibilités de mise à jour diffèrent. La figure suivante résume les préconisations d'Oracle en fonction du type de contenu XML.

Figure 13-2 *Préconisations d'Oracle du mode de stockage*



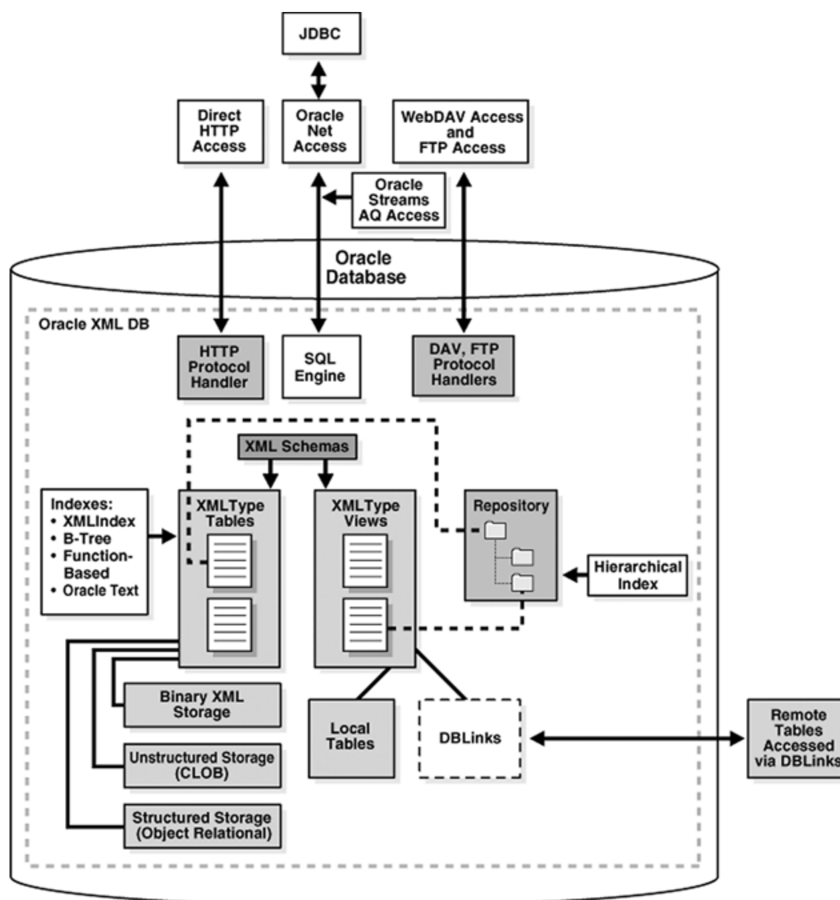
	Data-Centric		Document-Centric	
Use Case	XML schema-based data, with little variation and little structural change over time	XML schema-based data, with some embedded variable data	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Employee record that includes a free-form resume	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Hybrid	`CLOB` (Unstructured) or Binary XML	
Indexing	B-tree index	Index the structured and unstructured parts separately	Function-based index	`XMLIndex` index

© Éditions Eyrolles

Stockages XMLType

La figure 13-3 présente les différents accès et modes de stockage possibles.

Figure 13-3 *Stockage avec XMLType*



Le tableau 13-1 résume les points forts et les faiblesses (notés « + » et « - » respectivement) de chaque mode de stockage. Le mode *binary XML* semble le plus polyvalent. D'une manière générale, si le contenu XML à stocker ne doit pas être associé à une grammaire, choisissez le format CLOB.

© Éditions Eyrolles

Tableau 13-1 Comparatif des modes de stockage

	Structuré (objet-relationnel)	CLOB	Binary XML
Extraction	– (composition structurée de contenus XML)	+	++
Espace disque	++	– (prise en compte de caractères non significatifs et répétition de balises)	+
Flexibilité des données	– (stockage de documents conformes à une grammaire)	+	+
Flexibilité des grammaires *XML Schema*	– (données et grammaires stockées indépendamment, il est impossible d'avoir plusieurs grammaires pour une table `XMLType`)	+ (données et grammaires stockées ensemble, il est impossible d'avoir plusieurs grammaires pour une table `XMLType`)	++ (données et grammaires stockées ensemble, plusieurs grammaires possibles pour une table `XMLType`)
Mises à jour (UPDATE, DELETE)	++	– (une modification d'un contenu entraîne la mise à jour de tout le document)	+
Requête de type *XPath*	++	–	+
SQL contrainte	+	– (impossible)	+
Optimisation des opérations XML	+	–	+
Validation après insertion	Partielle (données XML seulement)	Partielle (données XML basées sur des grammaires)	Totale (données XML basées sur des grammaires)

Création d'une table

La syntaxe simplifiée de création d'une table `XMLType` est la suivante

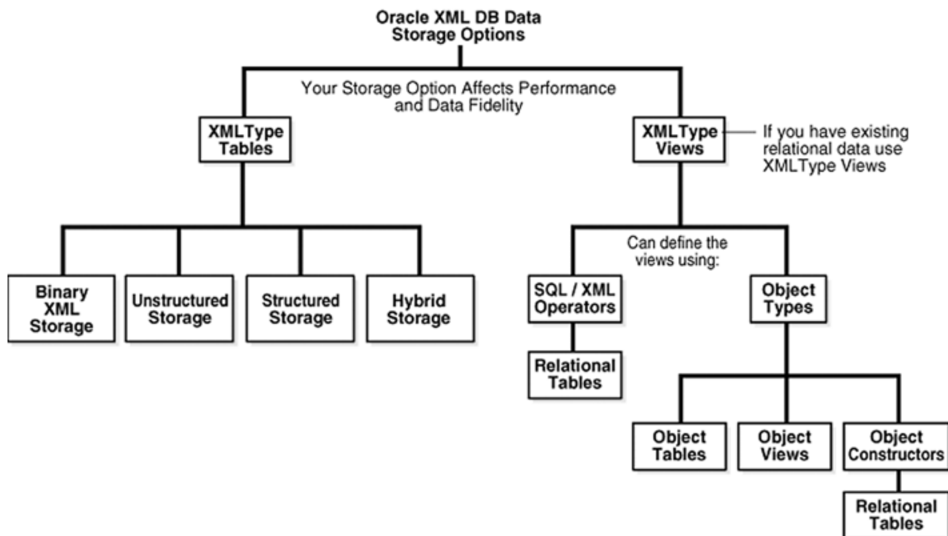
```
CREATE TABLE [schéma.] nomTable OF XMLTYPE
  XMLTYPE [STORE AS
 { OBJECT RELATIONAL | CLOB | BINARY XML }
 [ [ XMLSCHEMA nomXMLSchema ]
      ELEMENT { élément | nomXMLSchema # élément …} ]
 [ VIRTUAL COLUMNS ( colonne1 AS (expression1),...) ] ];
```

- La clause `OBJECT RELATIONAL` doit être combinée à l'option `XMLSCHEMA` qui associe une grammaire *XML Schema*.
- La clause `CLOB` doit être associée soit à la spécification de paramètres de stockage, soit à l'option `XMLSCHEMA`.
- La clause `VIRTUAL COLUMNS` est plutôt réservée au mode `BINARY XML`. Elle sert à construire des index ou des contraintes (définies ultérieurement par `ALTER TABLE`).
- En l'absence de la clause `STORE AS`, le mode de stockage par défaut est structuré (objet-relationnel).

Ces options peuvent également permettre de définir une colonne `XMLType`. On utilise une colonne plutôt qu'une table quand le contenu XML est fortement associé à des données relationnelles (par exemple, si on veut stocker la date d'ajout et de dernière mise à jour pour chaque document). Un exemple de colonne `XMLType` est présenté à la section « Stockage non structuré (`CLOB`) ».

La figure suivante illustre les différents conteneurs de données qu'il est possible de mettre en œuvre avec XML DB.

Figure 13-4 *Conteneur de données*



Décrivons à présent une méthodologie de travail pour étudier les différentes fonctionnalités de stockage et de manipulation de documents XML. Nous considérerons, dans un premier temps une table de stockage structuré. Nous aborderons ensuite les particularités des autres modes de stockage (*binary XML* et `CLOB`).

© Éditions Eyrolles

Répertoire de travail

Si vous n'utilisez pas l'environnement *XML DB Repository*, la création d'un répertoire logique qui référence celui qui contient les documents XML est nécessaire. Pensez également à positionner certaines variables d'environnement SQL*Plus (`SET LONG 10000` et `SET PAGESIZE 100`) pour que vos états de sortie ne soient pas tronqués du fait des valeurs par défaut.

```
CREATE DIRECTORY repxml AS
'C:\Donnees\Livres\Livres-Eyrolles\SQLpourOracle3\sourcesXML';
```

Grammaire XML Schema

Considérons le document `compagnies.xml` présenté dans le tableau 13-2. La grammaire est ici générée à l'aide de l'outil *XML Schema Generator* (`http://www.xmlforasp.net/`).

Tableau 13-2 Exemple de contenu et de sa grammaire

<table>
<tr><th>Document XML</th><th>Grammaire XML Schema (compagnies.xsd)</th></tr>
<tr><td><pre>
<?xml version="1.0"
  encoding="ISO-8859-1"?>
<compagnie>
  <comp>AB</comp>
  <pilotes>
    <pilote brevet="PL-1">
      <nom>C. Sigaudes</nom>
      <salaire>4000</salaire>
  </pilote>
    <pilote brevet="PL-2">
      <nom>P. Filloux</nom>
      <salaire>5000</salaire>
    </pilote>
  </pilotes>
  <nomComp>Air Blagnac
  </nomComp>
</compagnie>
</pre></td><td><pre>
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unquali-
fied"
 elementFormDefault="qualified" version="1.0"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="compagnie"
               type="compagnieType" />
  <xsd:complexType name="compagnieType">
    <xsd:sequence>
      <xsd:element name="comp"
                   type="xsd:string" />
      <xsd:element name="pilotes"
                   type="pilotesType" />
      <xsd:element name="nomComp"
                   type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="pilotesType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded"
                   name="pilote"
                   type="piloteType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="piloteType">
    <xsd:sequence>
      <xsd:element name="nom"
                   type="xsd:string" />
      <xsd:element name="salaire"
                   type="xsd:decimal" />
    </xsd:sequence>
    <xsd:attribute name="brevet"
                   type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
</pre></td></tr>
</table>

Annotation de la grammaire

Il est nécessaire d'annoter la grammaire pour faire correspondre le modèle de documents XML (éléments et attributs) avec les colonnes du SGBD (nom et type). L'espace de noms utilisé par Oracle est `http://xmlns.oracle.com/xdb`. Préfixés par `xdb`, de nombreux éléments sont proposés pour rendre la grammaire compatible à XML DB et l'enrichir de caractéristiques concernant le SGBD. Le tableau suivant présente les principaux éléments d'annotation d'Oracle. Tous ne sont pas forcément applicables aux différents modes de stockage (le mode `CLOB` est le plus restrictif).

Tableau 13-3 Éléments d'annotation

Nom de l'élément	Commentaires et exemple
`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.

Considérons les annotations suivantes apportées à la grammaire initiale. Les types et colonnes sont notés en majuscules pour mieux les différencier des éléments et attributs XML et car c'est ainsi qu'Oracle les stocke en interne. Déclarons le code et le nom de la compagnie obligatoires. Il convient également de déclarer que si une collection de pilotes existe, celle-ci n'est pas vide (`minOccurs="1"` pour l'élément `pilote`).

© Éditions Eyrolles

Tableau 13-4 Grammaire annotée

XML Schema (compagniesannote.xsd)	Commentaires
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" **xmlns:xdb="http://xmlns.oracle.com/xdb"** **xdb:storeVarrayAsTable="true"** version="1.0"> <xsd:element name="compagnie" type="compagnieType"/>	Espaces de noms. Élément du premier niveau.
<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="NOMCOMP"**/> </xsd:sequence> </xsd:complexType>	Éléments du second niveau.
<xsd:complexType name="pilotesType" **xdb:SQLType="PILOTES_TYPE"**> <xsd:sequence> <xsd:element **minOccurs="1"** maxOccurs="unbounded" name="pilote" type="piloteType" **xdb:SQLName="PILOTE"** **xdb:SQLCollType="PILOTE_VRY"**/> </xsd:sequence> </xsd:complexType>	Composition de la collection.
<xsd:complexType name="piloteType" **xdb:SQLType="PILOTE_TYPE"**> <xsd:sequence> <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" **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>	Élément de la collection. Précision de la taille de la colonne BREVET.

© Éditions Eyrolles

Tableau 13-4 Grammaire annotée *(suite)*

XML Schema (compagniesannote.xsd)	Commentaires

`<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:fractionDigits value="2"/>`
`<xsd:totalDigits value="6"/> </xsd:restriction>`
`</xsd:simpleType>`
`</xsd:schema>`
	Typage des autres colonnes.

Enregistrement de la grammaire

La phase suivante enregistre la grammaire dans la base (*repository*) à l'aide de la procédure REGISTERSCHEMA du paquetage DBMS_XMLSCHEMA. Décommentez la première instruction si vous souhaitez relancer à la demande cet enregistrement (après avoir modifié votre grammaire, par exemple).

```
BEGIN
-- DBMS_XMLSCHEMA.DELETESCHEMA(
--  'http://www.soutou.net/compagnies.xsd',
-- DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);
-- DBMS_XMLSCHEMA.REGISTERSCHEMA(
    SCHEMAURL => 'http://www.soutou.net/compagnies.xsd',
    SCHEMADOC => BFILENAME('REPXML', 'compagniesannote.xsd'),
    LOCAL => TRUE, GENTYPES => TRUE, GENTABLES => FALSE,
    CSID => NLS_CHARSET_ID('AL32UTF8'));
END;
/
```


© Éditions Eyrolles

- `SCHEMAURL` spécifie l'URL logique de la grammaire.
- `SCHEMADOC` référence le fichier lui-même (notez le nom du répertoire logique en majuscules dans la fonction `BFILENAME`).
- `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 (dans le cas de stockage *binary XML*, affectez la valeur `FALSE`).
- `GENTABLES` permet de générer une table (cela évite de la créer à part) dont le nom doit se trouver dans la grammaire en tant qu'attribut de l'élément racine `xdb:defaultTable="…"`.
- `CSID` indique le jeu de caractères associé (`AL32UTF8` est approprié au type de données `XMLType` et équivaut au standard UTF-8).

Après la vérification de la grammaire, Oracle génère les types suivants. La colonne `SYS_XDBPD$` est réservée à un usage interne (*positional descriptor*).

Tableau 13-5 Structures obtenues

Code SQL	Résultats

`SELECT OBJECT_NAME`
`FROM USER_OBJECTS`
`WHERE OBJECT_TYPE='TYPE';`
	`OBJECT_NAME`
	`------------------------`
	`COMPAGNIE_TYPE`
	`PILOTE_VRY`
	`PILOTE_TYPE`
	`PILOTES_TYPE`

`DESCRIBE COMPAGNIE_TYPE;`	`COMPAGNIE_TYPE est NOT FINAL`
	`Nom NULL ? Type`
	`----------------- -------- ------------------`
	`SYS_XDBPD$ XDB.XDB$RAW_LIST_T`
	`COMP VARCHAR2(6 CHAR)`
	`PILOTES PILOTES_TYPE`
	`NOMCOMP VARCHAR2(40 CHAR)`

`DESCRIBE PILOTES_TYPE`	`PILOTES_TYPE est NOT FINAL`
	`Nom NULL ? Type`
	`----------------- -------- ------------------`
	`SYS_XDBPD$ XDB.XDB$RAW_LIST_T`
	`PILOTE PILOTE_VRY`

`DESCRIBE PILOTE_VRY;`	`PILOTE_VRY VARRAY(2147483647) OF PILOTE_TYPE`
	`PILOTE_TYPE est NOT FINAL`
	`Nom NULL ? Type`
	`----------------- -------- ------------------`
	`SYS_XDBPD$ XDB.XDB$RAW_LIST_T`
	`BREVET VARCHAR2(4 CHAR)`
	`NOM VARCHAR2(30 CHAR)`
	`SALAIRE NUMBER(8,2)`

© Éditions Eyrolles

Stockage structuré (object-relational)

Une fois la grammaire enregistrée, il est possible de créer explicitement une table (ou colonne) pour stocker des documents XML respectant cette grammaire. Il faut renseigner la grammaire, le nom de l'élément racine et de(s) éventuelle(s) collection(s) *varray*. La table de stockage de la collection est nommée ici `pilote_table`.

```
CREATE TABLE compagnie_OR_xmlschema OF XMLType
       XMLTYPE STORE AS OBJECT RELATIONAL
       XMLSCHEMA "http://www.soutou.net/compagnies.xsd"
       ELEMENT "compagnie"
       VARRAY "XMLDATA"."PILOTES"."PILOTE"
       STORE AS TABLE pilote_table
           ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)));
```

Par défaut, chaque collection (élément XML ayant un attribut `maxOccurs` > 1) est sérialisée en tant que `LOB` (approprié pour la gestion de plusieurs documents mais mal adapté à la mise à jour d'éléments particuliers d'une collection donnée). La clause `VARRAY` définit une table de stockage pour chaque collection (bien adaptée à la mise à jour). La directive `XMLDATA` précise un chemin dans une arborescence XML. L'affichage de la structure de cette table rappelle en partie ses caractéristiques.

```
DESCRIBE compagnie_OR_xmlschema;
 Nom                                           NULL ?   Type
 --------------------------------------------- -------- --------------
 TABLE of SYS.XMLTYPE(XMLSchema "http://www.soutou.net/
 compagnies.xsd" Element "compagnie") STORAGE Object-relational
 TYPE "COMPAGNIE_TYPE"
```

Validation partielle

Bien que la grammaire soit associée à la table, il est néanmoins toujours possible de stocker du contenu XML ne respectant que partiellement la grammaire (une compagnie sans pilote ou sans nom, etc.). En revanche, il n'est pas possible d'insérer du contenu XML plus riche (éléments ou attributs) ou dont l'élément racine n'est pas celui défini dans la grammaire (ici compagnie).

Le tableau 13-6 présente des insertions tout à fait valides par rapport à la grammaire. Dans la première instruction, on retrouve la fonction d'extraction d'un fichier. Le constructeur `XMLType` transforme un document XML en `CLOB`, `BFILE` ou `VARCHAR`. Dans la seconde insertion, la fonction `CREATEXML` retourne un type `XMLType`.

 © Éditions Eyrolles

Tableau 13-6 Insertion de contenu entièrement valide

<table>
<tr><th>Code SQL</th><th>Commentaires</th></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema VALUES
 (XMLType(BFILENAME('REPXML','compagnie.xml'),
              NLS_CHARSET_ID('AL32UTF8')));</pre></td><td>Insertion du fichier <code>compagnie.xml</code>.</td></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema VALUES
 (XMLTYPE.CREATEXML('<?xml version="1.0"
                     encoding="ISO-8859-1"?>
  <compagnie>
    <comp>AC</comp>
    <pilotes>
        <pilote brevet="PL-3">
        <nom>G. Diffis</nom>
        <salaire>5000</salaire>
      </pilote>
      <pilote brevet="PL-4">
        <nom>S. Lacombe</nom>
      </pilote>
    </pilotes>
    <nomComp>Castanet Lines</nomComp>
</compagnie>'));</pre></td><td>Insertion d'un document passé directement en paramètre.</td></tr>
</table>

Le tableau 13-7 illustre des insertions qui ne respectent la grammaire que partiellement.

Tableau 13-7 Insertion de contenu partiellement valide

<table>
<tr><th>Code SQL</th><th>Commentaires</th></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema VALUES
 (XMLTYPE.CREATEXML(
   '<?xml version="1.0" encoding="ISO-8859-1"?>
    <compagnie>
        <comp>NoPil</comp>
        <pilotes></pilotes>
        <nomComp>No pilot</nomComp>
   </compagnie>'));</pre></td><td>Collection <code>pilotes</code> vide.</td></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema VALUES
 (XMLTYPE.CREATEXML(
   '<?xml version="1.0" encoding="ISO-8859-1"?>
    <compagnie>
      <comp>PasPil</comp>
      <nomComp>Pas de pilote</nomComp>
   </compagnie>'));</pre></td><td>Absence de collection <code>pilotes</code>.</td></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema VALUES
 (XMLTYPE.CREATEXML(
    '<?xml version="1.0" encoding="ISO-8859-1"?>
     <compagnie> <comp>seul!</comp> </compagnie>'));</pre></td><td>Compagnie sans nom et sans pilote.</td></tr>
</table>

La fonction `ISSCHEMAVALID` retourne `1` si l'objet est valide par rapport à sa grammaire, sinon `0`. La requête du tableau 13-8 est bien utile pour vérifier le contenu de la table. Les autres fonctions de la requête seront expliquées plus loin.

Tableau 13-8 Détermination des documents invalides

Code SQL	Résultat	
SELECT **EXTRACT** (c.OBJECT_VALUE,'compagnie/comp') "Comp", c.**ISSCHEMAVALID**() FROM compagnie_OR_xmlschema c;	Comp ------------------- <comp>AB</comp> <comp>AC</comp> <comp>NoPil</comp> <comp>PasPil</comp> <comp>seul!</comp>	C.ISSCHEMAVALID() ----------------- 1 1 0 0 0

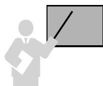
Le tableau 13-9 suivant illustre des tentatives d'insertions qui ne respectent pas la grammaire. Différents messages d'Oracle sont retournés en fonction de l'erreur. Pour les ajouts provenant de fichiers extérieurs invalides, les erreurs seraient similaires.

Tableau 13-9 Insertion de contenu invalide

Code SQL	Commentaires
INSERT INTO compagnie_OR_xmlschema VALUES (XMLTYPE.CREATEXML('<?xml version="1.0" encoding="ISO-8859-1"?> <compagnie> <comp>PB</comp> <pilote brevet="PL-5"> <nom>L. Schneider</nom> <salaire>8000</salaire> </pilote> <nomComp>Manque élément pilotes</nomComp> </compagnie>')); ERREUR ORA-30937: Absence de définition de schéma pour 'pilote' (espace de noms '##local') dans le parent '/compagnie'	Il manque la collection `pilotes`.
INSERT INTO compagnie_OR_xmlschema VALUE (XMLTYPE.CREATEXML('<?xml version="1.0" encoding="ISO-8859-1"?> <compagnie> <comp>1234567</comp> <pilotes></pilotes> <nomComp>Trop long le code comp!</nomComp> </compagnie>')); ERREUR ORA-30951: L'élément ou l'attribut (Xpath 1234567) dépasse la longueur maximale	Erreur de type du code de la compagnie.

© Éditions Eyrolles

Validation totale



Pour que la validation soit complète (*full compliant*), il faut ajouter à une table de stockage de type objet-relationnelle une contrainte de type CHECK ou programmer un déclencheur de type BEFORE INSERT (qui réalise la même fonctionnalité).

Les performances d'insertion sont affectées par ce type de validation.

La fonction XMLISVALID permet de programmer la contrainte de validité de l'objet en paramètre. Assurez-vous que la table ne contient pas d'enregistrements ne respectant pas la grammaire sinon Oracle retournera l'erreur ORA-02293: impossible de valider (…) - violation d'une contrainte de contrôle. Pour supprimer les documents invalides, exécutez "DELETE FROM … c WHERE c.ISSCHEMAVALID()=0";

Une fois cette contrainte existante, seuls les documents vraiment compatibles avec la grammaire pourront être stockés en base. L'erreur retournée, le cas échéant, sera invariablement ORA-02290: violation de contraintes (…) de vérification.

Tableau 13-10 Mécanismes de validation

<table>
<tr><th>Contrainte de vérification</th><th>Déclencheur</th></tr>
<tr><td><pre>ALTER TABLE compagnie_OR_xmlschema
  ADD CONSTRAINT valide_compagnie
  CHECK (XMLISVALID(OBJECT_VALUE)
    = 1);</pre></td><td><pre>CREATE TRIGGER valide_compagnie
  BEFORE INSERT ON compagnie_OR_xmlschema
FOR EACH ROW
BEGIN
  IF (:NEW.OBJECT_VALUE IS NOT NULL) THEN
      :NEW.OBJECT_VALUE.SCHEMAVALIDATE();
  END IF;
END;</pre></td></tr>
</table>

Le déclencheur de type BEFORE INSERT aura l'avantage de renseigner davantage l'erreur. Son code est simple et la fonction SCHEMAVALIDATE retourne une exception fournissant des informations précises sur l'erreur comme le montre le tableau 13-11.

Tableau 13-11 Insertion de contenu invalide

<table>
<tr><th>Code SQL</th><th>Résultats</th></tr>
<tr><td><pre>INSERT INTO compagnie_OR_xmlschema
  VALUES(XMLTYPE.CREATEXML('<?xml
  version="1.0" encoding="ISO-8859-1"?>
  <compagnie>
       <comp>NoPil</comp>
       <pilotes></pilotes>
       <nomComp>No pilot</nomComp>
  </compagnie>'));</pre></td><td><pre>ERREUR ORA-31154: document XML non
valide
ORA-19202: Une erreur s'est produite
lors du traitement la fonction XML
(LSX-00213: Seulement 0 occurrences
de l'élément "pilote"; minimum : 1)…
ORA-04088: erreur lors d'exécution
du déclencheur 'SOUTOU.VALIDE_
COMPAGNIE'</pre></td></tr>
</table>

Contraintes

Bien que le mécanisme d'une grammaire *XML Schema* soit puissant, il n'est pas possible de créer une contrainte d'unicité ou de référence. De plus, alors que les contraintes XML concernent individuellement chaque document, une contrainte SQL va permettre d'étendre une restriction à plusieurs documents (peuplant une table ou une colonne).

Deux mécanismes doivent être mis en œuvre : la pseudo-colonne `XMLDATA` qui permet d'adresser une colonne au niveau d'un type `XMLType` et la table de stockage définie dans la directive `VARRAY` pour opérer sur des collections.

Unicité

Le tableau suivant décrit l'ajout de différentes contraintes. Assurez-vous que les contenus XML respectent chaque contrainte. Si ce n'est pas le cas pour la première contrainte, par exemple, Oracle retourne l'erreur `ORA-00001: violation de contrainte unique (...UNIQUE_NOMCOMP)`. Pour les contraintes de vérification, l'erreur `ORA-02290: violation de contraintes ...` est retournée.

Tableau 13-12 Ajout de contraintes

Code SQL	Contraintes
`ALTER TABLE compagnie_OR_xmlschema ADD CONSTRAINT unique_nomcomp **UNIQUE** (**XMLDATA**."NOMCOMP");`	Unicité du nom de la compagnie.
`ALTER TABLE compagnie_OR_xmlschema ADD CONSTRAINT pk_compagnie_OR **PRIMARY KEY**(**XMLDATA**."COMP");`	Clé primaire sur le code compagnie.
`ALTER TABLE compagnie_OR_xmlschema ADD CONSTRAINT nn_nomcomp **CHECK** ((**XMLDATA**."NOMCOMP") IS **NOT NULL**);`	Non-nullité du nom de la compagnie.
`ALTER TABLE compagnie_OR_xmlschema ADD CONSTRAINT debut_comp **CHECK** ((**XMLDATA**."COMP") LIKE 'A%' OR (**XMLDATA**."COMP") LIKE 'E%');`	Le code de la compagnie débute par la lettre `A` ou la lettre `E`.

Intégrité référentielle

Le tableau 13-13 décrit la mise en œuvre d'une contrainte de référence vers une table relationnelle. Assurez-vous que les données relationnelles recensent toutes les compagnies du contenu XML. Si ce n'est pas le cas, Oracle retourne l'erreur classique `ORA-02291: violation de contrainte d'intégrité (…) - clé parent introuvable`.

Tableau 13-13 Intégrité référentielle

Table et données relationnelles	Contrainte référentielle
CREATE TABLE compagnie_R1 (codecomp VARCHAR2(6) PRIMARY KEY, budget NUMBER); INSERT INTO compagnie_R1 VALUES ('AB',30090); INSERT INTO compagnie_R1 VALUES ('AC',500000); INSERT INTO compagnie_R1 VALUES ('EA',900000); INSERT INTO compagnie_R1 VALUES ('EF',700000); INSERT INTO compagnie_R1 VALUES ('EG',40000);	ALTER TABLE compagnie_OR_xmlschema ADD CONSTRAINT fk_comp **FOREIGN KEY** (**XMLDATA.**"COMP") **REFERENCES** compagnie_R1(codecomp);

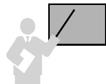
Éléments composés

La directive XMLDATA est nécessaire pour de définir une contrainte sur un élément précis. Le tableau 13-14 décrit les mécanismes à mettre en œuvre pour rendre l'élément nomAv unique parmi tous les documents qui seront stockés dans la table.

Tableau 13-14 Contrainte d'élément composé

Exemple d'un document et extrait de sa grammaire	Table associée et contrainte
<avion> <immat>F-GOWW</immat> <**typav**> <**nomAv**>A320</nomAv> <prixAv>40000</prixAv> </typav> </avion>	CREATE TABLE avion_OR_xmlschema OF XMLType XMLTYPE STORE AS OBJECT RELATIONAL XMLSCHEMA "http://www.soutou.net/avions.xsd" ELEMENT "avion";
… <xsd:complexType name="avionType" xdb:SQLType="AVION_TYPE"> <xsd:sequence> <xsd:element name="immat" type="immattype" xdb:SQLName="IMMAT"/> <xsd:element name="**typav**" type="typavType" xdb:SQLName="**TYPEAV_T**"/> </xsd:sequence> </xsd:complexType> <xsd:complexType name="typavType"> <xsd:sequence> <xsd:element name="**nomAv**" type="nomAvtype" xdb:SQLName="**NOMAV**" xdb:SQLType="VARCHAR2"/>…	ALTER TABLE avion_OR_xmlschema ADD CONSTRAINT unique_nomav **UNIQUE** (**XMLDATA.** "**TYPEAV_T".**"**NOMAV**");

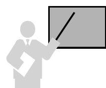
Collections



Il est possible de définir une contrainte sur un élément ou un attribut d'une collection en utilisant le nom de la table de stockage définie dans la clause `STORE AS` de la directive `VARRAY`.

Le code suivant ajoute une contrainte d'unicité sur le numéro de brevet du pilote au niveau de chaque document XML. Le nom de la table de stockage du *varray* est utilisé, de même que la directive `NESTED_TABLE_ID` qui identifie chaque ligne de la collection.

```
ALTER TABLE pilote_table
      ADD CONSTRAINT un_brevet UNIQUE (NESTED_TABLE_ID, "BREVET");
```



Pour étendre cette contrainte au niveau de tous les documents XML (toutes les compagnies), il ne faut pas mentionner la directive `NESTED_TABLE_ID` et se placer au niveau de la table de stockage elle-même (qui constitue l'union de toutes les collections).

Tableau 13-15 Contrainte sur une collection

Code SQL	Commentaires

`ALTER TABLE pilote_table`
`ADD CONSTRAINT un_brevet_tous`
`UNIQUE (BREVET);`
	Un pilote n'est embauché que par une seule compagnie.

`ALTER TABLE pilote_table`
`ADD CONSTRAINT ck_salaire`
`CHECK (SALAIRE IS NULL OR SALAIRE > 600);`
	Les salaires sont soit non renseignés soit, supérieurs à 600 €.

Extractions

Oracle fournit plusieurs fonctions SQL utiles à l'extraction de types `XMLType`, citons `XMLQUERY`, `XMLTABLE`, `XMLEXISTS` et `XMLCAST`. D'autres fonctions doivent y être associées comme `EXTRACT`, `EXTRACTVALUE`, `OBJECT_VALUE` et `EXISTSNODE`.

Éléments (nœuds)

Le tableau 13-16 présente quelques extractions (code des compagnies, nom des pilotes et détails de tous les deuxièmes pilotes). La fonction `EXTRACT` admet un paramètre ciblant un nœud et retourne une instance `XMLType`. La directive `OBJECT_VALUE` extrait la valeur d'une colonne XMLType (remplace la fonction `VALUE` utilisée dans les versions antérieures à 10*g* R1 et associée à la pseudo-colonne `SYS_NC_ROWINFO$`).

© Éditions Eyrolles

Tableau 13-16 Extractions d'éléments XML

Code SQL	Résultats
SELECT **EXTRACT**(**OBJECT_VALUE**, 'compagnie/comp') FROM compagnie_OR_xmlschema;	EXTRACT(OBJECT_VALUE,'compagnie/comp') -------------------------------------- <comp>AB</comp> <comp>AC</comp>
SELECT **EXTRACT**(**OBJECT_VALUE**, 'compagnie/pilotes/pilote/nom') "Nom de tous les pilotes" FROM compagnie_OR_xmlschema;	Nom de tous les pilotes ------------------------- <nom>C. Sigaudes</nom> <nom>P. Filloux</nom> <nom>G. Diffis</nom> <nom>S. Lacombe</nom>
SELECT **EXTRACT**(**OBJECT_VALUE**, 'compagnie/pilotes/pilote[2]') "Deuxièmes pilotes" FROM compagnie_OR_xmlschema;	Deuxièmes pilotes ---------------------------------- <pilote brevet="PL-2"> <nom>P. Filloux</nom> <salaire>5000</salaire> </pilote> <pilote brevet="PL-4"> <nom>S. Lacombe</nom> </pilote>

Attributs et textes

Le tableau 13-17 présente l'utilisation de la fonction XMLQUERY qui admet en premier paramètre une expression (XMLType ou scalaire SQL) et retourne le résultat de l'évaluation *XQuery* en reliant d'éventuelles variables. La directive PASSING BY VALUE permet de programmer la substitution dans l'expression *XQuery*. La clause AS précise la nature du résultat extrait (ici une chaîne de 9 caractères). Si le résultat est un ensemble vide, la fonction retourne NULL.

La deuxième requête extrait les numéros de brevet mais l'état de sortie n'est pas des plus satisfaisants. Il faut utiliser la fonction EXTRACTVALUE en la couplant à la directive XMLTABLE (qui construit une table virtuelle à partir de contenu XML). La dernière requête utilise une fonction d'agrégat pour calculer la somme des salaires.

Tableau 13-17 Extractions d'attributs et textes XML

Code SQL	Résultats
SELECT **XMLCAST**(**XMLQUERY**('$obj/compagnie/comp/text()' **PASSING BY VALUE** OBJECT_VALUE AS "obj" RETURNING CONTENT) AS VARCHAR2(9)) "Code comp" FROM compagnie_OR_xmlschema;	Code comp --------- AB AC

Tableau 13-17 Extractions d'attributs et textes XML *(suite)*

Code SQL	Résultats
`SELECT XMLCAST(XMLQUERY( '$obj/compagnie/pilotes/pilote/@brevet' PASSING BY VALUE OBJECT_VALUE AS "obj" RETURNING CONTENT) AS VARCHAR2(9)) "Brevet" FROM compagnie_OR_xmlschema;`	Brevet --------- PL-1PL-2 PL-3PL-4
`SELECT EXTRACTVALUE (acoll.COLUMN_VALUE,'/pilote/@brevet') "Brev" FROM compagnie_OR_xmlschema c, XMLTABLE('/compagnie/pilotes/pilote' PASSING c.OBJECT_VALUE) acoll;`	Brev ---- PL-1 PL-2 PL-3 PL-4
`SELECT SUM(TO_NUMBER(EXTRACTVALUE (acoll.COLUMN_VALUE,'/pilote/salaire'))) "Masse salariale" FROM compagnie_OR_xmlschema c, XMLTABLE('/compagnie/pilotes/pilote' PASSING c.OBJECT_VALUE) acoll;`	Masse salariale --------------- 14000

Prédicats d'extraction

Le tableau 13-18 présente quelques prédicats. La première requête extrait le nom du pilote de numéro `PL-2`. La fonction `XMLEXISTS` évalue une expression *XQuery* passée en paramètre (compagnies qui s'acquittent d'un salaire valant 5 000). La fonction booléenne `EXISTSNODE` vérifie l'expression *XPath* en paramètre (pilotes ayant un salaire inférieur à 5 000). La dernière requête compose (à l'aide d'alias, de `COLUMNS` et des variables de liens dans la clause `PASSING`) des tables virtuelles utiles à des fonctions d'agrégations (masse salariale de chaque compagnie).

Tableau 13-18 Prédicats d'extraction

Code SQL	Résultats
`SELECT XMLCAST(XMLQUERY('$obj/compagnie/pilotes/pilote [@brevet="PL-2"]/nom/text()' PASSING BY VALUE OBJECT_VALUE AS "obj" RETURNING CONTENT) AS VARCHAR2(15)) "Nom de PL-2" FROM compagnie_OR_xmlschema;`	Nom de PL-2 --------------- P. Filloux
`SELECT COUNT(*) "Nombre de compagnies" FROM compagnie_OR_xmlschema WHERE XMLEXISTS('$obj/compagnie/pilotes/pilote/salaire [number()=5000]' PASSING OBJECT_VALUE AS "obj");`	Nombre de compagnies -------------------- 2

© Éditions Eyrolles

Tableau 13-18 Prédicats d'extraction *(suite)*

Code SQL	Résultats

SELECT
 EXTRACT(OBJECT_VALUE,'compagnie/comp')"comp",
 EXTRACT(OBJECT_VALUE,'compagnie/nomComp')
  "nom"
 FROM compagnie_OR_xmlschema
 WHERE
  XMLEXISTS('$obj/compagnie/pilotes/pilote/
  salaire [number()<5000]'
  PASSING OBJECT_VALUE AS "obj");
	comp
	----------------------------- 	
	nom
	----------------------------- 	
	<comp>AB</comp>
	<nomComp>Air Blagnac
	</nomComp>

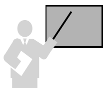
SELECT
  EXTRACT(OBJECT_VALUE,'compagnie/comp')"comp"
  FROM compagnie_OR_xmlschema
  WHERE **EXISTSNODE**(OBJECT_VALUE,
       'compagnie/pilotes/pilote/salaire
        [number()<5000]')=1;
	comp
	----------------------------- 	
	<comp>AB</comp>

SELECT acomp.colcmp,SUM(TO_NUMBER(li.sal))
  FROM compagnie_OR_xmlschema c,
   XMLTABLE('/compagnie' PASSING
   c.OBJECT_VALUE
   COLUMNS
  colcmp VARCHAR2(6) PATH 'comp',
  pils XMLType   PATH 'pilotes/pilote')
   acomp,
   XMLTABLE('pilote' PASSING acomp.pils
    COLUMNS sal NUMBER PATH 'salaire') li
  GROUP BY acomp.colcmp;
	COLCMP SUM(TO_NUMBER(LI.SAL))
	------ ----------------------
	AC 5000
	AB 9000

Mises à jour

Il est possible de modifier tout le contenu (`UPDATE` classique) ou seulement une partie (fragment) d'un document XML. Le code suivant remplace la compagnie de code `AB` par le contenu du fichier passé en paramètre (sous réserve qu'il soit bien formé et qu'il respecte la grammaire et les éventuelles contraintes d'intégrité).

```
UPDATE compagnie_OR_xmlschema
  SET OBJECT_VALUE =
  XMLTYPE(BFILENAME('REPXML','autrecompagnie.xml'),
    NLS_CHARSET_ID('AL32UTF8'))
  WHERE EXISTSNODE(OBJECT_VALUE, 'compagnie/comp[text()="AB"]')=1;
```



Le mode de stockage structuré permet l'utilisation de fonctions adaptées à la mise à jour de fragments XML, citons principalement `UPDATEXML`, `INSERTCHILDXML`, `INSERTXMLBEFORE`, `APPENDCHILDXML` et `DELETEXML`.

© Éditions Eyrolles

La fonction UPDATEXML*(XMLType_instance, expressionXPath, expression)* met à jour le contenu désigné par le premier paramètre et ciblé par le deuxième paramètre par l'expression du troisième paramètre. Le premier exemple du tableau suivant modifie le salaire d'un pilote.

La fonction INSERTCHILDXML*(XMLType_instance, expressionXPath, expressionFils, expression)* insère un nœud fils (troisième paramètre) à l'emplacement désigné par le deuxième paramètre et valué par le dernier paramètre. La deuxième modification ajoute un pilote pour une compagnie.

La fonction APPENDCHILDXML*(XMLType_instance, expressionXPath, expression)* insère un nouveau nœud après l'élément désigné par le deuxième paramètre. La troisième modification ajoute l'élément salaire au pilote qui en était démuni.

La fonction DELETEXML*(XMLType_instance, expressionXPath)* supprime un ou plusieurs nœuds à l'emplacement désigné par le deuxième paramètre. La dernière modification élimine le premier pilote de la compagnie de code AC.

Tableau 13-19 Mises à jour

Code SQL	Résultats (collection pilotes)
UPDATE compagnie_OR_xmlschema SET OBJECT_VALUE = **UPDATEXML**(OBJECT_VALUE, '/compagnie/pilotes/pilote/salaire/text()', '6000') WHERE EXISTSNODE(OBJECT_VALUE, '/compagnie/pilotes/pilote/nom[text() = "G. Diffis"]')=1;	<pilote brevet="PL-3"> <nom>G. Diffis</nom> <salaire>**6000**</salaire> </pilote> …
UPDATE compagnie_OR_xmlschema SET OBJECT_VALUE = **INSERTCHILDXML**(OBJECT_VALUE,'/compagnie/ pilotes','pilote', XMLType('<pilote brevet="PL-5"> <nom>L. Schneider</nom> <salaire>8000</salaire> </pilote>')) WHERE EXISTSNODE(OBJECT_VALUE, '/compagnie[comp="AC"]') = 1;	<pilote brevet="PL-3"> <nom>G. Diffis</nom> <salaire>6000</salaire> </pilote> <pilote brevet="PL-4"> <nom>S. Lacombe</nom> </pilote> **<pilote brevet="PL-5">** **<nom>L. Schneider</nom>** **<salaire>8000</salaire>** **</pilote>**
UPDATE compagnie_OR_xmlschema SET OBJECT_VALUE = **APPENDCHILDXML**(OBJECT_VALUE, '/compagnie/pilotes/pilote[2]', XMLType('<salaire>4000</salaire>')) WHERE EXISTSNODE(OBJECT_VALUE, '/compagnie[comp="AC"]') = 1;	… <pilote brevet="PL-4"> <nom>S. Lacombe</nom> **<salaire>4000</salaire>** </pilote> …

© Éditions Eyrolles

Tableau 13-19 Mises à jour *(suite)*

Code SQL	Résultats (collection pilotes)

`UPDATE compagnie_OR_xmlschema`
`  SET OBJECT_VALUE = **DELETEXML**(OBJECT_VALUE,`
`        '/compagnie/pilotes/pilote[1]')`
`  WHERE EXISTSNODE(OBJECT_VALUE,`
`        '/compagnie[comp="AC"]') = 1;`
	`<pilote brevet="PL-4">`
	`  <nom>S. Lacombe</nom>`
	`   <salaire>4000</salaire>`
	`</pilote>`
	`<pilote brevet="PL-5">`
	`  <nom>L. Schneider</nom>`
	`  <salaire>8000</salaire>`
	`</pilote>`

Vues relationnelles

Les vues relationnelles fournissent un accès classique à du contenu XML. Ce mécanisme intéressera les adeptes de SQL ne maîtrisant pas forcément XML. De plus, ces vues permettent d'« aplatir » les collections pour les réfractaires du modèle objet (il y en a hélas tellement encore !). Les fonctions `EXTRACTVALUE` et `XMLTABLE` qui combinent des expressions *XPath* avec SQL permettent de définir des vues (*mapping* entre colonnes de la vue et les éléments XML).

Le code suivant déclare une vue relationnelle du contenu XML qui décrit les compagnies. La clause `FROM` contient trois tables : la première héberge les documents XML, la deuxième (virtuelle) compose les colonnes du premier niveau (code et nom de chaque compagnie), la troisième table (virtuelle) définit les éléments de chaque collection (pilotes).

Tableau 13-20 Vue relationnelle

Code SQL	Commentaires
`**CREATE VIEW** compagnie_detail_vue AS`	
`  SELECT acomp.colcmp, acomp.nomcmp, ligne.*`	Colonnes de la vue.
`  FROM compagnie_OR_xmlschema c,`	

`    **XMLTABLE**('/compagnie' **PASSING** c.OBJECT_VALUE`
`      **COLUMNS** colcmp VARCHAR2(6)  PATH 'comp',`
`              nomcmp VARCHAR2(20) PATH 'nomComp',`
`              pils XMLType PATH 'pilotes/pilote') acomp,`
	Définition de la table virtuelle « maître ».

`    **XMLTABLE**('pilote' **PASSING** acomp.pils`
`      **COLUMNS** brevetpil VARCHAR2(4) PATH '@brevet',`
`              nompil    VARCHAR2(20) PATH 'nom',`
`              salpil    NUMBER PATH 'salaire') ligne;`
	Définition de la table virtuelle « fils ».

Une fois cette vue créée, les connaisseurs de SQL y trouveront leur compte quelle que soit l'extraction souhaitée. Le tableau 13-21 présente quelques extractions classiques (on suppose que la table contient le contenu XML initial).

© Éditions Eyrolles

Tableau 13-21 Extraction de contenu XML par vue

<table>
<tr><th>Requête</th><th>Code SQL</th></tr>
<tr><td>Liste des compagnies.</td><td><pre>SELECT DISTINCT v.colcmp, v.nomcmp
       FROM compagnie_detail_vue v;
COLCMP NOMCMP
------ --------------------
AB   Air Blagnac
AC   Castanet Lines</pre></td></tr>
<tr><td>Détails des pilotes
de la compagnie de code AB.</td><td><pre>SELECT v.brevetpil, v.nompil, v.salpil
       FROM compagnie_detail_vue v
       WHERE v.colcmp = 'AB';
BREV NOMPIL   SALPIL
---- -------------------- ----------
PL-1 C. Sigaudes   4000
PL-2 P. Filloux   5000</pre></td></tr>
<tr><td>Code des compagnies
et détails de chaque pilote.</td><td><pre>SELECT v.colcmp, v.brevetpil, v.nompil, v.salpil
       FROM compagnie_detail_vue v
       ORDER BY 1,2;
COLCMP BREV NOMPIL   SALPIL
------ ---- -------------------- ----------
AB   PL-1 C. Sigaudes   4000
AB   PL-2 P. Filloux   5000
AC   PL-3 G. Diffis   5000
AC   PL-4 S. Lacombe</pre></td></tr>
<tr><td>Nombre de pilotes salariés
par compagnie.</td><td><pre>SELECT v.colcmp, COUNT(v.brevetpil)
                "Nombre de pilotes salariés"
       FROM compagnie_detail_vue v
       WHERE v.salpil IS NOT NULL
       GROUP BY v.colcmp
       ORDER BY 1;
COLCMP Nombre de pilotes salariés
------ --------------------------
AB   2
AC   1</pre></td></tr>
</table>

Création d'une table par annotations

Le code suivant décrit la modification qu'il faudrait apporter à la grammaire annotée décrivant les compagnies pour générer automatiquement la table `compagniedefaut`, similaire à la précédente en ce qui concerne les contraintes de clés et la définition de la collection. Seul l'élément racine est à modifier. Pensez à positionner l'option `GENTABLES` à `TRUE` afin de générer la table et ses contraintes.

 © Éditions Eyrolles

```
<xsd:element name="compagnie" type="compagnieType"
 xdb:defaultTable="COMPAGNIEDEFAUT"
 xdb:columnProps="CONSTRAINT compagniedefault_pk
                             PRIMARY KEY (XMLDATA.COMP),
                   CONSTRAINT fk_compdef FOREIGN KEY (XMLDATA.COMP)
                             REFERENCES compagnie_R1(codecomp)"
 xdb:tableProps="VARRAY XMLDATA.PILOTES.PILOTE
                  STORE AS TABLE pilote_table ((PRIMARY KEY
                        (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))"/>
```

Stockage non structuré (CLOB)

La table suivante contient une colonne de stockage non structuré associée à une grammaire semblable à celle précédemment définie. Une spécification du `LOB` est également précisée.

```
CREATE TABLE compagnie_col_xmlschema_CLOB
 (cle VARCHAR2(10) PRIMARY KEY, col_clob_xml XMLType)
  XMLTYPE col_clob_xml STORE AS CLOB
 (TABLESPACE users STORAGE (INITIAL 100K NEXT 50K) CHUNK 800
  CACHE LOGGING)
  XMLSCHEMA "http://www.soutou.net/compagnies2.xsd"
  ELEMENT "compagnie";
```

Validation

Le mécanisme de validation est similaire à celui du mode de stockage objet. Le contenu XML doit seulement respecter initialement et en partie la grammaire. La validation totale se traite aussi par une contrainte ou un déclencheur (remplacez la directive `OBJECT_VALUE` par le nom de la colonne `col_clob_xml`).

Contraintes

Aucune contrainte supplémentaire sur le contenu XML ne peut être mise en œuvre avec SQL.

© Éditions Eyrolles

Extractions

Les requêtes incluant des fonctions `EXTRACT` et `EXISTSNODE` sont bien plus coûteuses que si elles interrogeaient une structure objet (du fait de la construction en mémoire d'un arbre DOM à chaque exécution). Toutes les interrogations précédentes s'adaptent toutefois en remplaçant pour chaque requête le nom de la table par `compagnie_col_xmlschema_CLOB c` (avec un alias) et la directive `OBJECT_VALUE` par la colonne `c.col_clob_xml`.

Mises à jour

Le mode de stockage non structuré ne permet la mise à jour qu'au niveau du document entier (`UPDATE` classique). Ainsi, la modification suivante substitue le document XML relatif à la compagnie de code `AB` par le contenu XML du fichier passé en paramètre.

```
UPDATE compagnie_col_xmlschema_CLOB c
       SET c.col_clob_xml =
       XMLTYPE(BFILENAME('REPXML','autrecompagnie.xml'),
                        NLS_CHARSET_ID('AL32UTF8'))
       WHERE EXISTSNODE(c.col_clob_xml,
                      'compagnie/comp[text()="AB"]') = 1;
```

Vues relationnelles

Pour des raisons de performances, il est déconseillé de composer des vues relationnelles de contenu XML stocké en `CLOB`. Néanmoins, il est possible de définir des vues relationnelles de la même manière que pour le mode de stockage structuré (remplacez la directive `OBJECT VALUE` par la colonne `col_clob_xml` dans l'exemple).

Stockage non structuré (binary XML)

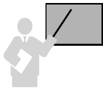
Le mode de stockage *binary XML* est le plus avancé en ce qui concerne l'encodage du contenu en fonction des grammaires. Considérons dans un premier temps ce type de stockage sans grammaire associée et pour lequel il est nécessaire de définir une ou plusieurs colonnes virtuelles afin de pouvoir mettre en place des contraintes. En effet, le contenu XML n'est pas directement transposé (au niveau de la structure) comme pour le mode de stockage objet-relationnel.

Une colonne virtuelle est basée sur une expression *XPath* qui doit retourner une valeur scalaire par contenu XML (élément ou attribut). La table suivante n'est rattachée à aucune grammaire mais déclare deux colonnes virtuelles. L'encodage de tous les documents utilise le mode neutre (*non-schema-based encoding*).

 © Éditions Eyrolles

```
CREATE TABLE compagnie_binaryXML OF XMLType
  XMLTYPE STORE AS BINARY XML
  VIRTUAL COLUMNS
  (vircolcomp AS (EXTRACTVALUE(OBJECT_VALUE,'/compagnie/comp')),
   vircolnomcomp AS
      (EXTRACTVALUE(OBJECT_VALUE,'/compagnie/nomComp')));
```

Contraintes



Seules des contraintes fonctionnelles (unique et clé étrangère) sont opérationnelles. Attention, bien qu'Oracle permette de déclarer tout type de contrainte, elles n'auront pas l'effet escompté.

Comme le montre le tableau 13-22, les contraintes sont définies naturellement à l'aide des colonnes virtuelles.

Tableau 13-22 Ajout de contraintes (mode binary XML)

Contraintes	Commentaires
`ALTER TABLE compagnie_binaryXML ADD CONSTRAINT unnomcomp_compagnie_binaryXML UNIQUE(vircolnomcomp);`	Unicité du nom de la compagnie.
`ALTER TABLE compagnie_binaryXML ADD CONSTRAINT pk_compagnie_binaryXML PRIMARY KEY(vircolcomp);`	Clé primaire sur le code compagnie.
`ALTER TABLE compagnie_binaryXML ADD CONSTRAINT fk_comp_binary FOREIGN KEY (vircolcomp) REFERENCES compagnie_R1(codecomp);`	Intégrité référentielle vers une table relationnelle.

Il n'est pas possible d'ajouter une colonne virtuelle à l'aide de l'instruction `ALTER TABLE`.

Extractions

Le mode *binary XML* est le plus performant pour l'extraction de contenus XML. Toutes les requêtes étudiées sur la table structurée (objet-relationnelle) s'écrivent à l'identique en remplaçant simplement le nom de la table pour chaque requête.

Mises à jour

Le mode *binary XML* est moins performant pour la mise à jour de contenus XML que le mode structuré. Néanmoins, toutes les mises à jour décrites sur la table objet-relationnelle s'écrivent à l'identique (modifiez seulement le nom de la table).

Vues relationnelles

Les vues relationnelles se définissent de la même manière que pour le mode de stockage structuré.

Options de grammaire

Le mode de stockage *binary XML* est le plus avancé en ce qui concerne l'association de grammaires *XML Schema*.

Une grammaire associée à une table (ou colonne) `XMLType` *binary XML* ne peut être utilisée par une autre table (ou colonne) adoptant le mode de stockage structuré ou `CLOB`. L'inverse est également vrai : l'utilisation d'un autre mode de stockage que *binary XML* pour une grammaire *XML Schema* impose de n'utiliser cette grammaire que pour le mode de stockage structuré ou `CLOB`.

Le mode de stockage *binary XML* encode le contenu XML en fonction de la grammaire associée (plusieurs grammaires peuvent être associées à une table ou colonne). Possibilité est également donnée d'encoder ou pas du contenu même s'il est associé à une grammaire. Trois choix s'offrent alors à vous (l'option `ALLOW ANYSCHEMA` n'est pas recommandée si votre grammaire est susceptible d'évoluer dans le temps).

- Encoder le contenu sans tenir compte d'une grammaire (*non-schema-based*). Le contenu XML pourra néanmoins être valide avec la grammaire, sans y être contraint. Toute grammaire associée sera ignorée à propos de l'encodage et le contenu ne sera pas automatiquement validé lors d'une insertion ou d'une mise à jour. Il sera également possible de valider explicitement du contenu.
- Encoder le contenu en tenant compte d'une seule grammaire. Tous les documents devront être valides (*full compliant*). Il est possible de préciser que tous les documents ne respectant pas la grammaire (*non-schema-based documents*) peuvent être stockés dans la même colonne.
- Encoder le contenu en tenant compte d'une grammaire parmi plusieurs. Dans ce cas, la même grammaire peut être déclinée en plusieurs versions de manière à stocker le contenu en accord avec une version particulière. Les documents ne respectant pas la grammaire peuvent être stockés dans la même colonne.

© Éditions Eyrolles

Tableau 13-23 Options de grammaire

Clause SQL	Commentaires
`STORE AS BINARY XML`	Encode tous les documents en utilisant le mode neutre (*non-schema-based encoding*).
`STORE AS BINARY XML XMLSCHEMA ... [DISALLOW NONSCHEMA]`	Encode tous les documents en utilisant la grammaire référencée. L'insertion ou la mise à jour de contenu non valide déclenche une erreur (`ORA-31011 : échec d'analyse XML`).
`STORE AS BINARY XML XMLSCHEMA ... ALLOW NONSCHEMA`	Encode tous les documents associés à une grammaire en utilisant cette même grammaire. Encode tous les documents non associés à une grammaire en utilisant le mode neutre. L'insertion ou la mise à jour de contenu non valide à sa grammaire déclenche une erreur.
`STORE AS BINARY XML ALLOW ANYSCHEMA`	Chaque grammaire peut être utilisée pour encoder le contenu XML à insérer ou à mettre à jour. L'insertion ou la mise à jour de contenu non valide à sa grammaire ou qui ne référence aucune grammaire déclenche une erreur.
`STORE AS BINARY XML ALLOW ANYSCHEMA ALLOW NONSCHEMA`	Chaque grammaire peut être utilisée pour encoder le contenu XML à insérer ou mettre à jour. Encode tous les documents non associés à une grammaire en utilisant le mode neutre. L'insertion ou la mise à jour de contenu non valide à sa grammaire ou qui ne référence aucune grammaire déclenche une erreur.

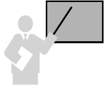
Lors de l'enregistrement de la grammaire, positionnez `GENTYPES` à `FALSE` et renseignez l'option `REGISTER_BINARYXML`. La grammaire suivante est sensiblement identique à la première (supprimez les annotations `SQLTYPE` et les types scalaires comme `VARCHAR2`, etc.).

```
DBMS_XMLSCHEMA.REGISTERSCHEMA(
    SCHEMAURL => 'http://www.soutou.net/compagnies3.xsd',
    SCHEMADOC => BFILENAME('REPXML','compagniesannotebinXML.xsd'),
    LOCAL => TRUE, GENTYPES => FALSE, GENTABLES => FALSE,
    OPTIONS => DBMS_XMLSCHEMA.REGISTER_BINARYXML);
```

La table suivante qui est rattachée à une grammaire, déclare une colonne virtuelle et encodera le contenu selon la grammaire.

```
CREATE TABLE compagnie_binaryXML_grammaire OF XMLType
  XMLTYPE STORE AS BINARY XML
   XMLSCHEMA "http://www.soutou.net/compagnies3.xsd"
   ELEMENT "compagnie"
   DISALLOW NONSCHEMA
  VIRTUAL COLUMNS
     (vircolcomp AS (EXTRACTVALUE(OBJECT_VALUE,'/compagnie/comp')));
```

© Éditions Eyrolles



Le mode de stockage *binary XML* associé à une grammaire fournit une validation complète (*full compliant*).

Autres fonctionnalités

Cette section décrit quelques fonctionnalités qu'il est intéressant de connaître.

Génération de contenus

Plusieurs mécanismes permettent de générer du contenu XML à partir de données relationnelles (tables). Les fonctions les plus intéressantes sont celles de la norme ANSI. Citons `XMLELEMENT` (créé un élément), `XMLATTRIBUTES` (ajoute un attribut à un élément), `XMLFOREST` (créé une arborescence) et `XMLAGG` (peuple une collection).

Considérons les données des 3 tables suivantes (un avion appartient à une compagnie et peut être affrété par plusieurs).

Web

```
compagnie_R
CODEC  NOMCOMPA
------ -------------------
AF     Air France
EJ     Easy Jet
AB     Air Blagnac

affreter_R
NA     CODEC  DATE_A     NB_PASSAGERS
------ ------ ---------- ------------
F-GODF EJ     08-12-2007          120
F-GODF AF     08-12-2007          150
F-PROG AF     08-12-2007          130
F-PROG AF     09-12-2007          110

avion_R
NA     TYPAV    CAPACITE PROPRIO
------ ------ ---------- -------
F-GODF A320          170 AB
F-PROG A318          140 AF
F-WOWW A380          490 EJ
```


© Éditions Eyrolles

Le tableau 13-24 décrit la génération d'une arborescence XML décrivant les affrètements ordonnés par compagnie.

Tableau 13-24 Génération de contenus XML

Code SQL

```sql
SELECT XMLElement
 ("Affretement",
  XMLAttributes(c.codec AS "comp"),
  XMLForest(c.nomCompa AS "nomComp"),
  XMLElement("Vols",
    (SELECT XMLAgg(XMLElement
    ("Vol",
     XMLForest(
         TO_CHAR(af.date_a,'DD/MM/YYYY')
                    AS "date",
         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) )) AS "Contenu_XML"
 FROM compagnie_R c ORDER BY nomCompa;
```

Résultat

```
Contenu_XML
--------------------------------
<Affretement comp="AB">
  <nomComp>Air Blagnac</nomComp>
  <Vols></Vols>
</Affretement>
<Affretement comp="AF">
  <nomComp>Air France</nomComp>
  <Vols>
   <Vol>
     <date>08/12/2007</date>
     <avion>A320</avion>
     <passagers>150</passagers>
   </Vol>
   <Vol>
     <date>08/12/2007</date>
     <avion>A318</avion>
     <passagers>130</passagers>
   </Vol>
   <Vol>
      <date>09/12/2007</date>
      <avion>A318</avion>
      <passagers>110</passagers>
   </Vol>
  </Vols>
</Affretement>
<Affretement comp="EJ">
  <nomComp>Easy Jet</nomComp>
  <Vols>
   <Vol>
    <date>08/12/2007</date>
    <avion>A320</avion>
    <passagers>120</passagers>
   </Vol>
  </Vols>
</Affretement>
```

Vues XMLType

Concernant les données qui sont stockées dans des tables relationnelles ou objet-relationnelles, les vues XMLType permettent de composer du contenu XML contraint ou pas par une grammaire préalablement enregistrée.

Sans grammaire

Le tableau 13-25 présente la déclaration et l'interrogation de la vue XMLType qui fusionne des données des trois tables relationnelles précédentes. La requête de définition inclut en plus un identifiant objet (par exemple ici, le nom de la compagnie). Les extractions retournent le nombre d'affrètements stockés puis le détail d'un affrètement.

Tableau 13-25 Vue XMLType

Création de la vue	Interrogations de la vue

<pre>**CREATE VIEW** compagnie_xml OF **XMLType**
  WITH OBJECT ID
 (SUBSTR(EXTRACTVALUE(
   OBJECT_VALUE,'/Affretement/nomComp'),
   1,30))
  AS SELECT XMLElement("Affretement",
     XMLAttributes(c.codec AS "comp"),
     XMLForest(c.nomCompa AS "nomComp"),
     XMLElement("Vols",
     (SELECT XMLAgg(XMLElement("Vol",
      XMLForest(TO_CHAR(af.date_a,
         'DD/MM/YYYY') AS "date",
         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)) AS "Contenu_XML"
  FROM compagnie_R c;</pre>
	<pre>SELECT COUNT(OBJECT_VALUE)
	       FROM compagnie_xml ;
	COUNT(OBJECT_VALUE)
	------------------- 	
	                  3
	
	SELECT OBJECT_VALUE
	  FROM compagnie_xml
	  WHERE EXISTSNODE(OBJECT_VALUE,
	  '/Affretement[@comp="EJ"]')=1;
	OBJECT_VALUE
	-------------------------------- 	
	<Affretement comp="EJ">
	<nomComp>Easy Jet</nomComp>
	 <Vols>
	  <Vol>
	    <date>08/12/2007</date>
	    <avion>A320</avion>
	    <passagers>120</passagers>
	  </Vol>
	 </Vols>
	</Affretement></pre>

À partir d'un type objet

La fonction SYS_XMLGEN génère une instance XMLType à partir d'un type objet. Le tableau 13-26 décrit la création d'un type objet décrivant les affrètements (en utilisant un attribut). La requête de définition de la vue XMLType (tous les affrètements de plus de 110 passagers dans un format XML) fait intervenir la fonction XMLFORMAT qui compose l'élément racine.

© Éditions Eyrolles

Tableau 13-26 Vue XMLType à partir d'un type

Création du type et de la vue	Interrogation de la vue

`CREATE TYPE affreter_type AS OBJECT`
`("@na" CHAR(6), codec VARCHAR(6),`
`  date_a DATE, nb_passagers NUMBER);`
`/`
 	
`CREATE VIEW affreter_view_xml OF XMLType`
`  WITH OBJECT ID (EXTRACT(OBJECT_VALUE,`
`  '/Affetement/@na').GETNUMBERVAL()) AS`
`  SELECT SYS_XMLGEN(affreter_type(a.na,`
`    a.codec, a.date_a, a.nb_passagers),`
`    XMLFORMAT('Affretement'))`
`  FROM affreter_R a`
`  WHERE a.nb_passagers > 110;`
	`SELECT OBJECT_VALUE`
	`  FROM affreter_view_xml`
	`  WHERE EXISTSNODE(OBJECT_VALUE,`
	`'Affretement/CODEC[text()="EJ"]')=1;`
	
	`OBJECT_VALUE`
	`-----------------------------------`
	`<?xml version="1.0"?>`
	`<Affretement na="F-GODF">`
	` <CODEC>EJ</CODEC>`
	` <DATE_A>08/12/07</DATE_A>`
	` <NB_PASSAGERS>120</NB_PASSAGERS>`
	`</Affretement>`

Association d'une grammaire

Une vue `XMLType` peut être associée à une grammaire pour contraindre davantage les données extraites. Considérons la simple grammaire caractérisant l'élément `avioncomp` et définissons une vue `XMLType` peuplée à partir des données (tous les avions) des tables relationnelles.

Tableau 13-27 Structure et grammaire de la vue

Structure XML	Grammaire

`<?xml version="1.0"`
` encoding="ISO-8859-1"?>`
`<avioncomp na = "…">`
` <nomAv> … </nomAv>`
` <capacite> … </capacite>`
` <compav>`
`   <comp> … </comp>`
`   <nomcomp> … </nomcomp>`
` </compav>`
`</avioncomp>`
	`<xsd:schema attributeFormDefault="unqualified"`
	`  elementFormDefault="qualified" version="1.0"`
	`  xmlns:xsd="http://www.w3.org/2001/XMLSchema">`
	` <xsd:element name="avioncomp" type="avionType"/>`
	` <xsd:complexType name="avionType">`
	`   <xsd:sequence>`
	`     <xsd:element name="nomAv" type="xsd:string"/>`
	`     <xsd:element name="capacite" type="xsd:int"/>`
	`     <xsd:element name="compav" type="compavType"/>`
	`   </xsd:sequence>`
	`   <xsd:attribute name="na" type="xsd:string"/>`
	` </xsd:complexType>`
	` <xsd:complexType name="compavType">`
	`   <xsd:sequence>`
	`     <xsd:element name="comp" type="xsd:string"/>`
	`     <xsd:element name="nomcomp" type="xsd:string"/>`
	`   </xsd:sequence>`
	` </xsd:complexType>`
	`</xsd:schema>`

Le tableau 13-28 décrit la définition et l'extraction complète de la vue. La fonction `GETNUMBERVAL` permet d'affecter une valeur numérique à chaque enregistrement extrait et définir ainsi avec `WITH OBJECT ID` l'identifiant de la vue. On retrouve les fonctions de la norme ANSI qui génèrent du contenu XML.

Tableau 13-28 Vue XMLType associée à une grammaire

Création de la vue

```
CREATE VIEW avicomp_view_xml OF XMLType
  XMLSCHEMA
  "http://www.soutou.net/Avioncomps.xsd"
  ELEMENT "avioncomp" WITH OBJECT ID
(EXTRACT (OBJECT_VALUE,
      '/AvionComp/@immat').GETNUMBERVAL())
  AS SELECT XMLElement
  ("AvionComp",
   XMLAttributes(av.na AS "immat"),
   XMLForest(av.typav AS "nomav",
            av.capacite AS "nbplaces"),
   XMLElement
  ("compav", XMLForest(c.codec AS "comp",
             c.nomcompa AS "nomcomp")))
  FROM avion_R av, compagnie_R c
  WHERE av.proprio = c.codec;
```

Contenu de la vue

```
SELECT OBJECT_VALUE
      FROM avicomp_view_xml;
OBJECT_VALUE
-------------------------------
<AvionComp immat="F-GODF">
  <nomav>A320</nomav>
  <nbplaces>170</nbplaces>
  <compav>
    <comp>AB</comp>
    <nomcomp>Air Blagnac</nomcomp>
  </compav>
</AvionComp>
<AvionComp immat="F-PROG">
  <nomav>A318</nomav>
  <nbplaces>140</nbplaces>
  <compav>
    <comp>AF</comp>
    <nomcomp>Air France</nomcomp>
  </compav>
</AvionComp>
<AvionComp immat="F-WOWW">
  <nomav>A380</nomav>
  <nbplaces>490</nbplaces>
  <compav>
    <comp>EJ</comp>
    <nomcomp>Easy Jet</nomcomp>
  </compav>
</AvionComp>
```

Génération de grammaires annotées

La fonction `GENERATESCHEMA` du paquetage `DBMS_XMLSCHEMA` permet de générer une grammaire annotée *XML Schema*. Les paramètres sont à inscrire en majuscules. Ils décrivent le nom du schéma d'Oracle qui contient le type objet-relationnel et le nom du type lui-même. Générons la grammaire annotée décrivant le type `societe_type`.

Il restera à ajouter d'éventuelles annotations qui contraindront ou préciseront le stockage des sociétés. Ce mécanisme fonctionne également pour les collections objet (`AS TABLE OF`).

© Éditions Eyrolles

Tableau 13-29 Génération d'une grammaire annotée

Web

Code SQL

```
CREATE TYPE adresse_type
  AS OBJECT (nrue CHAR(3),
  nomrue VARCHAR2(20),
  ville VARCHAR2(20))
/

CREATE TYPE societe_type
  AS OBJECT (siret VARCHAR2(15),
  creation DATE,
  adresse_t adresse_type,
  effectif NUMBER)
/

SELECT
  DBMS_XMLSCHEMA.GENERATESCHEMA(
  'SOUTOU','SOCIETE_TYPE')
  FROM DUAL;
```

Résultats (en partie)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xdb="http://xmlns.oracle.com/xdb"
xsi:schemaLocation="http://xmlns.oracle.com/xdb
http://xmlns.oracle.com/xdb/XDBSchema.xsd">

<xsd:element name="SOCIETE_TYPE" type="SOCIETE_TYPEType"
xdb:SQLType="SOCIETE_TYPE" xdb:SQLSchema="SOUTOU"/>

 <xsd:complexType name="SOCIETE_TYPEType"
 xdb:SQLType="SOCIETE_TYPE" xdb:SQLSchema="SOUTOU"
 xdb:maintainDOM="false">
  <xsd:sequence>
   <xsd:element name="SIRET" xdb:SQLName="SIRET"
    xdb:SQLType="VARCHAR2">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:maxLength value="15"/>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
   <xsd:element name="CREATION" type="xsd:date"
    xdb:SQLName="CREATION" xdb:SQLType="DATE"/>
   <xsd:element name="ADRESSE_T" type="ADRESSE_TYPEType"
    xdb:SQLName="ADRESSE_T"
 xdb:SQLSchema="SOUTOU" xdb:SQLType="ADRESSE_TYPE"/>
   <xsd:element name="EFFECTIF" type="xsd:double"
    xdb:SQLName="EFFECTIF" xdb:SQLType="NUMBER"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="ADRESSE_TYPEType"
  xdb:SQLType="ADRESSE_TYPE" xdb:SQLSchema="SOUTOU"
  xdb:maintainDOM="false">
  <xsd:sequence>
   <xsd:element name="NRUE" xdb:SQLName="NRUE"
    xdb:SQLType="CHAR">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:length value="3"/>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
   …
</xsd:schema>
```

Dictionnaire des données

Le dictionnaire des données propose un certain nombre de vues (préfixées par USER pour les objets du schéma courant, ALL pour les objets sur lesquels on a des privilèges et DBA pour tous les objets et quel que soit le schéma) qui intéresseront les utilisateurs de XML DB.

Tables XMLType

Au niveau d'un utilisateur, la vue USER_XML_TABLES décrit les tables XMLType en ce qui concerne le type de stockage et les options de grammaire.

Tableau 13-30 Nature des tables XMLType

<table>
<tr><th>Code SQL</th><th>Résultats</th></tr>
<tr><td><pre>SELECT TABLE_NAME,
  XMLSCHEMA,
  STORAGE_TYPE
  FROM USER_XML_
  TABLES;</pre></td><td><pre>TABLE_NAME

XMLSCHEMA   STORAGE_TYPE
------------------------------------ --------------
AVION_OR_XMLSCHEMA
http://www.soutou.net/avions.xsd   OBJECT-RELATIONAL

COMPAGNIE_BINARYXML_GRAMMAIRE
http://www.soutou.net/compagnies3.xsd BINARY

COMPAGNIE_OR_XMLSCHEMA
http://www.soutou.net/compagnies.xsd OBJECT-RELATIONAL</pre></td></tr>
<tr><td><pre>SELECT TABLE_NAME,
  ELEMENT_NAME,
  ANYSCHEMA,
  NONSCHEMA
  FROM USER_XML_
  TABLES;</pre></td><td><pre>TABLE_NAME   ELEMENT_NAME   ANY NON
----------------------------- ---------------- --- ---
COMPAGNIE_BINARYXML   NO YES
AVION_OR_XMLSCHEMA   avion
COMPAGNIE_BINARYXML_GRAMMAIRE compagnie   NO NO
COMPAGNIE_OR_XMLSCHEMA   compagnie</pre></td></tr>
</table>

Colonnes XMLType

Sur le même principe, la vue USER_XML_TAB_COLS décrit les colonnes XMLType en ce qui concerne le type de stockage et la grammaire.

© Éditions Eyrolles

Tableau 13-31 Nature des colonnes XMLType

Code SQL	Résultats
SELECT COLUMN_NAME, XMLSCHEMA, ELEMENT_NAME, STORAGE_TYPE FROM **USER_XML_TAB_COLS**;	COLUMN_NAME XMLSCHEMA ---------------- ------------------------- ELEMENT_NAME STORAGE_TYPE ---------------- ----------------- SYS_NC_ROWINFO$ http://www.soutou.net/compagnies3.xsd compagnie BINARY SYS_NC_ROWINFO$ http://www.soutou.net/compagnies.xsd compagnie OBJECT-RELATIONAL SYS_NC_ROWINFO$ http://www.soutou.net/avions.xsd avion OBJECT-RELATIONAL COL_CLOB_XML http://www.soutou.net/compagnies2.xsd Compagnie CLOB

Grammaires XML Schema

Sur le même principe, USER_XML_SCHEMAS renseigne à propos des grammaires *XML Schema*. La colonne XMLSCHEMA de cette vue contient le code complet de la grammaire.

Tableau 13-32 Nature des grammaires XML Schema

Code SQL	Résultats
SELECT SCHEMA_URL, LOCAL, BINARY FROM **USER_XML_SCHEMAS**;	SCHEMA_URL LOC BIN ------------------------------------- --- --- http://www.soutou.net/avions.xsd YES NO http://www.soutou.net/compagnies.xsd YES NO http://www.soutou.net/compagnies2.xsd YES NO http://www.soutou.net/compagnies3.xsd YES YES

Vues XMLType

Les vues XMLType sont renseignées via USER_XML_VIEWS. La requête suivante extrait les caractéristiques des vues XMLType du schéma courant.

Tableau 13-33 Nature des vues XMLType

Code SQL	Résultats
SELECT VIEW_NAME, XMLSCHEMA, ELEMENT_NAME FROM **USER_XML_VIEWS**;	VIEW_NAME XMLSCHEMA ---------------- ------------------------------------ ELEMENT_NAME -------------- COMPAGNIE_XML AFFRETER_VIEW_XML AVICOMP_VIEW_XML http://www.soutou.net/Avioncomps.xsd avioncomp

XML DB Repository

XML DB Repository est un environnement partagé de contenus (XML ou autre) basé sur le concept de système de gestion de fichiers (répertoires). Les contenus non XML sont stockés en tant que CLOB. L'environnement est compatible avec la norme DAV *(Distributed Authoring and Versioning*) et utilise un serveur WebDAV.

Toutes les informations de cet environnement sont stockées dans le schéma de l'utilisateur XDB (initialement verrouillé à l'installation de la base). Une action dans la console ou l'exécution du script catqm.sql situé dans ORACLE_HOME\rdbms\admin rend opérationnel cet utilisateur.

Interfaces

Les moyens de travailler avec *XML DB Repository* sont divers :

- protocoles http(s), WebDAV ou FTP pour les ajouts, mises à jour et suppressions de contenus ;
- directement via les tables en utilisant SQL et le paquetage XML_XDB ;
- gestion de versions en utilisant le paquetage XML_XDB_VERSION ;
- par l'API Java (*Content Connector*).

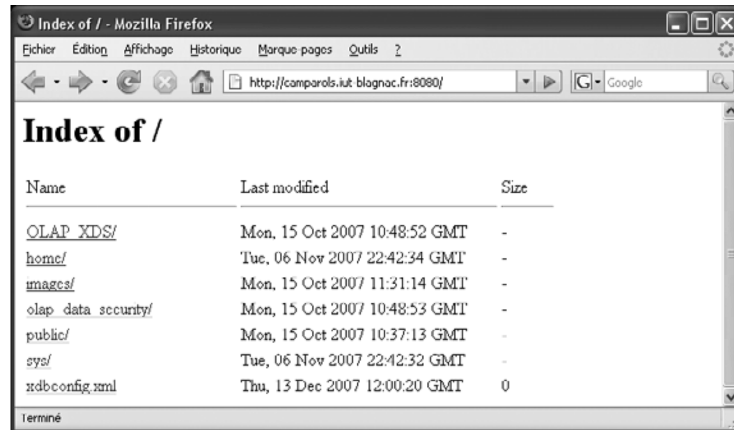
Configuration

Pour configurer cet environnement, assurez-vous que votre instance est bien associée à un service (la commande lsnrctl status doit retourner Le service "instanceOracleXDB" comporte 1 instance(s)…). Affectez un numéro de port (ici 8080) dans l'élément <http-port> du fichier xdbconfig.xml.11.0 du répertoire ORACLE_HOME\xml.

L'écran suivant illustre un accès http (accès réglementé par un compte et un mot de passe Oracle) à l'arborescence principale de *XML DB Repository.*

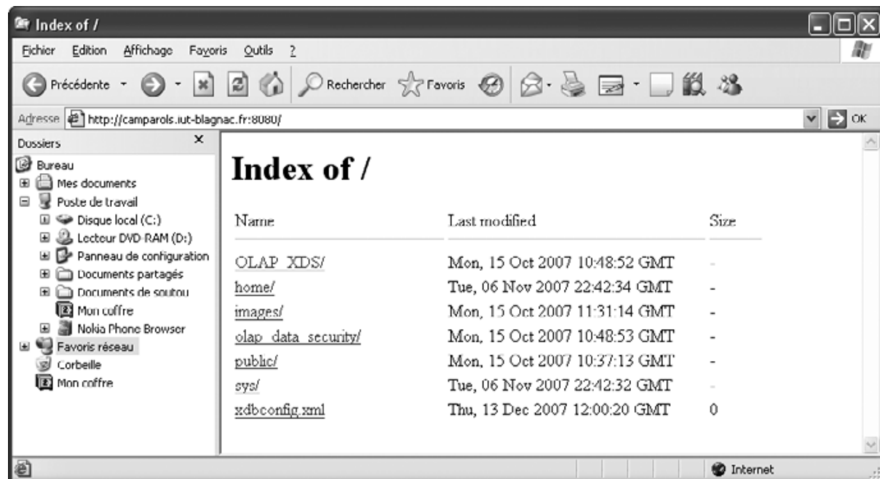
© Éditions Eyrolles

Figure 13-5 *Accès par http*



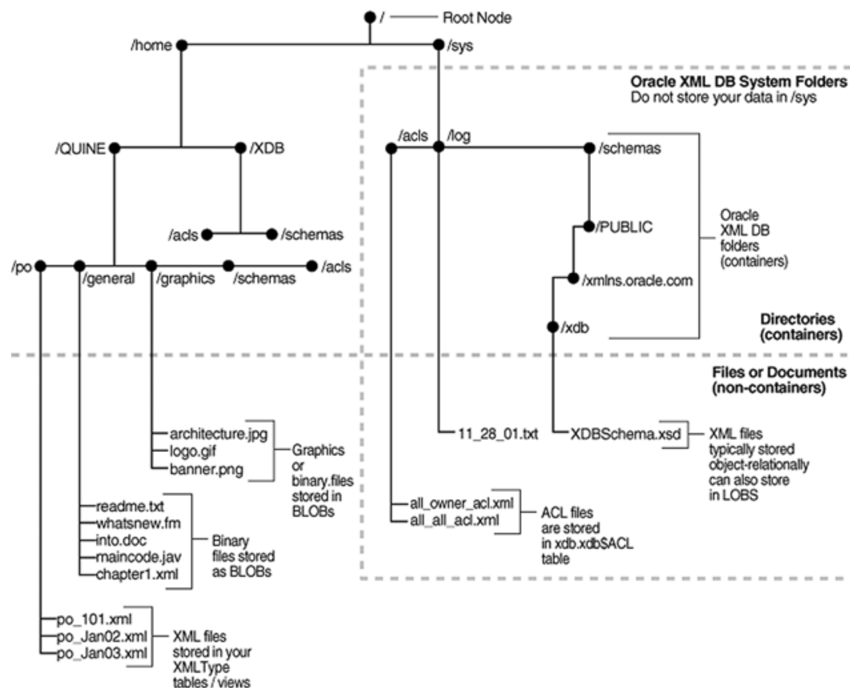
La figure 13-6 illustre le même accès par l'explorateur Windows. Vous pouvez également définir cet accès au niveau de vos favoris réseau. Une fois connecté, vous pouvez créer des répertoires et déposer vos documents par glisser-déposer.

Figure 13-6 *Accès par l'explorateur Windows*



La figure 13-7 décrit l'arborescence qu'Oracle préconise afin d'utiliser correctement le système de gestion de fichiers de *XML DB Repository.*

Figure 13-7 Arborescence du système de gestion de fichiers



Les fichiers grammaire qui ont été créés par l'instruction `REGISTERSCHEMA` se trouvent dans l'arborescence `sys/schemas` suivi du nom d'utilisateur et de l'URL logique de chaque grammaire.

Figure 13-8 Liste des grammaires

Index of /sys/schemas/SOUTOU/www.soutou.net/ - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://camparols.iut-blagnac.fr:8080/sys/schemas/SOUTOU/www.soutou.net/

Index of /sys/schemas/SOUTOU/www.soutou.net/

Name	Last modified	Size
Avioncomps.xsd	Fri, 21 Dec 2007 17:39:41 GMT	0
avions.xsd	Fri, 21 Dec 2007 16:24:37 GMT	0
compagnies.xsd	Fri, 21 Dec 2007 16:22:37 GMT	0
compagnies2.xsd	Fri, 21 Dec 2007 16:25:06 GMT	0
compagnies3.xsd	Fri, 21 Dec 2007 15:45:02 GMT	0

Terminé

 © Éditions Eyrolles

Paquetage XML_XDB

Le paquetage XML_XDB propose de nombreuses fonctions pour manipuler le système de gestion de fichiers, citons :

- CREATEFOLDER qui crée un répertoire ;
- DELETERESOURCE qui supprime une ressource (document ou répertoire) ;
- EXISTSRESOURCE qui teste l'existence d'un répertoire ;
- CREATERESOURCE qui crée une ressource (document ou répertoire).

Le tableau 13-34 dépose la ressource (document A-Faire.txt contenant trois lignes de texte et situé dans le répertoire référencé par le nom logique REPXML) dans l'arborescence /home/SOUTOU/general. Il est vrai que quelques clics seulement auraient aussi pu faire l'affaire !

Tableau 13-34 Dépose d'une ressource

Code SQL	Commentaires

`DECLARE`
`v_resultat BOOLEAN;`
`BEGIN`
`IF (NOT DBMS_XDB.`**`EXISTSRESOURCE`**`('/home/SOUTOU')) THEN`
`v_resultat := DBMS_XDB.`**`CREATEFOLDER`**`('/home/SOUTOU');`
`END IF;`
	Création de /home/SOUTOU si nécessaire.

`IF (NOT DBMS_XDB.`**`EXISTSRESOURCE`**`('/home/SOUTOU/general')) THEN`
`v_resultat:=DBMS_XDB.`**`CREATEFOLDER`**`('/home/SOUTOU/general');`
`END IF;`
	Création du sous-répertoire general si nécessaire.

`IF (DBMS_XDB.`**`EXISTSRESOURCE`**`('/home/SOUTOU/general/note.txt')) THEN`
`DBMS_XDB.`**`DELETERESOURCE`**`('/home/SOUTOU/general/note.txt',4);`
`END IF;`
	Suppression de la ressource.

`v_resultat := DBMS_XDB.`**`CREATERESOURCE`**`(`
`'/home/SOUTOU/general/note.txt',`
`BFILENAME('REPXML','A-Faire.txt'),NLS_CHARSET_ID('AL32UTF8'));`
`COMMIT;`
`END;`
	Transfert de la ressource.

Accès par SQL

Deux vues permettent d'accéder aux ressources contenues dans *XML DB Repository* : RESOURCE_VIEW et PATH_VIEW. Toutes deux possèdent une colonne virtuelle nommée RES de type XMLType permettant d'extraire et de mettre à jour des contenus en utilisant la notation pointée (alias SQL). Chaque ligne de la vue RESOURCE_VIEW concerne un unique chemin dans l'arborescence. Chaque ligne de la vue PATH_VIEW concerne une unique ressource. Une ressource peut être associée à plusieurs chemins de répertoires (liens).

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 XML stocké en base. Le tableau suivant décrit les principaux éléments définis dans la grammaire XDBResource.xsd.

Tableau 13-35 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.

Les fonctions EQUALS_PATH (trouve une ressource située dans un répertoire) et UNDER_PATH (retourne les sous-répertoires d'un répertoire donné) doivent également être utilisées. Le tableau suivant présente quelques extractions. La fonction CONVERT du paquetage DBMS_XMLGEN convertit un contenu XML selon un format donné et retourne un CLOB.

 © Éditions Eyrolles

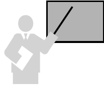
Tableau 13-36 Extractions SQL

Requêtes SQL	Commentaires et résultats
SELECT XDBURIType ('/home/SOUTOU/general/note.txt').GETCLOB() "A faire..." FROM DUAL;	Extraction sous la forme d'un CLOB du contenu d'une ressource. A faire... ------------------------------ - Finir article Programmez - Contacter l'éditeur Eyrolles - Acheter 3 baguettes
SELECT COUNT(*) FROM **RESOURCE_VIEW** rv WHERE **UNDER_PATH** (rv.RES,'/home/SOUTOU/general') = 1;	Nombre de ressources dans un répertoire donné. COUNT(*) ---------- 2
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE EXTRACTVALUE(RES, '/Resource/DisplayName') LIKE '%.txt';	Chemins contenant une ressource d'extension .txt. ANY_PATH ------------------------------ /home/SOUTOU/general/note.txt
SELECT EXTRACT(rv.RES, '/Resource/CreationDate') "Date création" FROM **RESOURCE_VIEW** rv WHERE **EQUALS_PATH** (rv.RES, '/home/SOUTOU/general/note.txt')=1;	Date de création de la ressource note.txt. Date création -------------------------------- <CreationDate xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd">2007-12-22T17:48:09.015000 </CreationDate>
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE UNDER_PATH(RES, 2, '/home/SOUTOU') = 1 AND EXISTSNODE(RES, '/Resource[@Container="true"]')=1;	Répertoires seuls (à partir de /home/SOUTOU, profondeur maximale : 2). ANY_PATH -------------------- /home/SOUTOU/general /home/SOUTOU/xsd
SELECT DBMS_XMLGEN.CONVERT(EXTRACT (rv.RES,'/Resource/Contents/text/text()', 'xmlns="http://xmlns.oracle.com/xdb/ XDBResource.xsd"').GETCLOBVAL(),1) "Contenu" FROM **RESOURCE_VIEW** rv WHERE **EQUALS_PATH**(rv.RES, '/home/SOUTOU/general/note.txt')=1;	Extraction du contenu d'une ressource en utilisant la grammaire. Contenu ------------------------------ - Finir article Programmez - Contacter l'éditeur Eyrolles - Acheter 3 baguettes
SELECT alias.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') alias WHERE **EQUALS_PATH**(rv.RES,'/home/SOUTOU/general/compagnie.xml')=1;	Extraction du nom des pilotes. NOM ---------------- C. Sigaudes P. Filloux

Contenus XML basés sur une grammaire

L'accès au contenu de documents associés à une grammaire (*schema-based XML documents*) peut se programmer de deux manières :

- par la vue `RESOURCE_VIEW` (comme précédemment même si le contenu n'est contraint par aucune grammaire ;
- par jointure entre la table créée par défaut au niveau de la grammaire et la vue `RESOURCE_VIEW`.



Annotez votre grammaire avec les attributs `xdb:schemaURL = "votreURL.xsd"` et `xdb:defaultTable="NOM_TABLE_EN_MAJUSCULES"` avant de la stocker dans un répertoire de *XML DB Repository.*

Enregistrez ensuite votre grammaire avec la version `CLOB` de la fonction `REGISTERSCHEMA` en utilisant la directive `XDBURITYPE`.

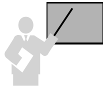
Le tableau 13-37 présente l'annotation de la grammaire (fichier `compagniesRepository.xsd`) puis le stockage et l'enregistrement de cette grammaire dans *XML DB Repository* (on suppose que le répertoire `/home/SOUTOU/xsd/` existe).

Tableau 13-37 Gestion de la grammaire pour XML DB Repository

Début de la grammaire	Stockage puis enregistrement

`<xsd:schema`
`  xmlns:xsd="http://www.w3.org/2001/`
`    XMLSchema"`
`  xmlns:xdb="http://xmlns.oracle.com/xdb"`
`  xdb:storeVarrayAsTable="true"`
`  `**`xdb:schemaURL`**`="http://www.soutou.net/`
`                compRepository.xsd">`
`  <xsd:element name="compagnie"`
`      type="compagnieType"`
`      `**`xdb:defaultTable`**`="TABCOMPREPO"/>`
`  <xsd:complexType name="compagnieType">`
`…`
	`DECLARE`
	`  v_resultat BOOLEAN;`
	`BEGIN`
	`-- stockage`
	`  v_resultat :=`
	`  DBMS_XDB.CREATERESOURCE`
	` ('/home/SOUTOU/xsd/`
	`  compagniesRepository.xsd',`
	`  BFILENAME('REPXML',`
	`  'compagniesRepository.xsd'),`
	`   NLS_CHARSET_ID('AL32UTF8'));`
	`-- enregistrement`
	`  DBMS_XMLSCHEMA.`**`REGISTERSCHEMA`**
	`  (SCHEMAURL =>'http://www.soutou`
	`   .net/compRepository.xsd',`
	`   SCHEMADOC => `**`XDBURITYPE`**`('/home/`
	`   SOUTOU/xsd/compagniesRepository`
	`   .xsd').GETCLOB(),`
	`   LOCAL => TRUE, GENTYPES =>TRUE,`
	`   GENTABLES=>TRUE, FORCE=>TRUE);`
	`END;`
	`/`

 © Éditions Eyrolles



Chaque contenu XML (ressource au vocable *XML DB Repository*) doit inclure l'espace de noms `xsi` au niveau de l'élément racine, et l'attribut `noNamespaceSchemaLocation` au niveau de l'élément racine (ici, `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` et `xsi:noNamespaceSchemaLocation="http://www.soutou.net/compRepository.xsd"`).

Il est désormais possible d'extraire les emplacements (chemins de répertoires) qui contiennent une ressource dont l'élément racine est précisément défini dans cette grammaire (on suppose que les documents `comp1.xml` et `comp2.xml` respectent la grammaire et les conditions précédemment énoncées).

Tableau 13-38 Localisation d'emplacements

Requête SQL	Résultat
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE EXISTSNODE(RES, '/**Resource**[**SchemaElement**= "http://www.soutou.net/ compRepository.xsd#compagnie"]')=1;	ANY_PATH -------------------------------- /home/SOUTOU/comp2.xml /home/SOUTOU/general/comp1.xml

Le stockage des documents basés sur une grammaire alimente automatiquement la table par défaut définie dans l'élément racine de la grammaire. Chaque ligne de cette table est accessible par une référence physique (élément `XMLRef` de la vue `RESOURCE_VIEW`). Cet élément doit être utilisé dans une jointure avec la table par défaut `XMLType`.

Le tableau suivant présente les possibilités de cette technique. La première requête extrait, sous la forme de `CLOB`, les deux premiers contenus associés à cette grammaire (avec leur emplacement). La seconde requête extrait la référence associée au contenu d'une ressource donnée (non exploitable sans jointure). La dernière extrait une partie d'un contenu donné (ici le nom des pilotes).

Tableau 13-39 Extractions par jointure avec la vue RESOURCE_VIEW

Requêtes SQL	Résultats
SELECT ANY_PATH, EXTRACT(VALUE(c), '/compagnie/pilotes').GETCLOBVAL() "CLOB" FROM RESOURCE_VIEW r, TABCOMPREPO c WHERE EXTRACTVALUE(r.res,'/Resource/ **XMLRef'**) = **REF**(c) AND ROWNUM < 3;	ANY_PATH ----------------------------------- CLOB ----------------------------------- /home/SOUTOU/general/comp1.xml <pilotes> <pilote brevet="PL-9"> <nom>J. Nouveau</nom> <salaire>9000</salaire> </pilote> </pilotes> /home/SOUTOU/comp2.xml <pilotes> <pilote brevet="P-10"> <nom>G. Diffis</nom> <salaire>19000</salaire> </pilote> <pilote brevet="P-11"> <nom>S. Lacombe</nom> <salaire>23000</salaire> </pilote> </pilotes>
SELECT EXTRACTVALUE(res, 'Resource/**XMLRef'**) FROM RESOURCE_VIEW WHERE ANY_PATH = '/home/SOUTOU/ general/comp1.xml';	EXTRACTVALUE(RES,'RESOURCE/XMLREF') ----------------------------------- 0000280209399615823EE346498E0533F29 3D703E3417C1AFC651D49F892B9646B6F17 1956010001B40000
SELECT alias.NOM FROM RESOURCE_VIEW rv, TABCOMPREPO c, XMLTable('/compagnie/pilotes/pilote' PASSING c.OBJECT_VALUE COLUMNS nom VARCHAR2(20) PATH 'nom') alias WHERE EQUALS_PATH (rv.RES,'/home/SOUTOU/ comp2.xml') = 1 AND **REF**(c)=EXTRACTVALUE (rv.RES,'/Resource/**XMLRef'**);	NOM -------------------- G. Diffis S. Lacombe

© Éditions Eyrolles

Vue PATH_VIEW

Cette vue est composée de cinq colonnes :

- PATH (VARCHAR2) qui indique un chemin (absolu) d'une ressource ;
- RES (XMLType) qui décrit une ressource du répertoire décrit dans PATH ;
- LINK (XMLType) qui décrit un lien vers la ressource ;
- RESID (RAW) qui contient l'identifiant d'une ressource.

L'interrogation de cette vue associe les fonctions suivantes :

- DEPTH(*correlation*) qui retourne la profondeur (niveau du répertoire) relativement à un répertoire donné (de départ). L'entier en paramètre permet de corréler cette fonction à UNDER_PATH. La fonction PATH fournit aussi un tel paramétrage.
- UNDER_PATH(*col_resource, profondeurMax, chemindepart, correlation)* qui détermine si une ressource est présente sous un chemin passé en paramètre sous la contrainte d'un nombre de niveaux et en corrélation avec les fonctions DEPTH ou PATH.

Tableau 13-40 Interrogation de la vue PATH_VIEW

<table>
<tr><th>Requête SQL</th><th>Résultat</th></tr>
<tr><td>

```
SELECT PATH, DEPTH(1),
DEPTH(2)
 FROM PATH_VIEW
  WHERE (UNDER_PATH(RES,
    3, '/sys/schemas/
    SOUTOU',1) = 1
  OR UNDER_PATH(RES,3,
  '/home/SOUTOU',2) = 1);
```

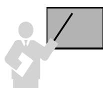
</td><td>

```
PATH                                DEPTH(1) DEPTH(2)
----------------------------------- -------- --------
/home/SOUTOU/general/comp1.xml                      2
/home/SOUTOU/comp2.xml                              1
/home/SOUTOU/xsd/
 compagniesRepository.xsd                           2
/sys/schemas/SOUTOU/www.soutou.net         1
/sys/schemas/SOUTOU/www.soutou.net/        2
 compRepository.xsd
/home/SOUTOU/general                                1
/home/SOUTOU/general/note.txt                       2
/home/SOUTOU/xsd                                    1
```

</td></tr>
</table>

Mises à jour

Les mises à jour de ressources ou de contenus (modifications et suppressions) peuvent également se programmer par UPDATE et DELETE à l'aide de ces vues. Le tableau 13-41 présente quelques mises à jour (toutes effectives après COMMIT).



La suppression d'une ressource par DELETE (deuxième instruction dans le tableau suivant) n'est possible que si la ressource est « finale » (répertoire vide ou document). Dans le cas contraire, il faut d'abord supprimer tous les contenus d'un répertoire pour pouvoir ensuite supprimer le répertoire.

© Éditions Eyrolles

Tableau 13-41 Mises à jour de ressources et de contenus

<table>
<tr><th>Code SQL</th><th>Commentaires</th></tr>
<tr><td><pre>UPDATE RESOURCE_VIEW r
    SET r.RES = UPDATEXML(r.RES,
          '/Resource/DisplayName/text()','Mom_memo')
    WHERE EQUALS_PATH(r.RES,
          '/home/SOUTOU/general/note.txt') = 1;</pre></td><td>Modification du nom logique d’une ressource.</td></tr>
<tr><td><pre>DELETE FROM RESOURCE_VIEW
       WHERE EQUALS_PATH(RES,
       '/home/SOUTOU/general/note.txt') = 1;</pre></td><td>Suppression de la ressource <code>note.txt</code>.</td></tr>
<tr><td><pre>UPDATE PATH_VIEW
       SET PATH   = '/home/SOUTOU/comp1.xml'
       WHERE PATH = '/home/SOUTOU/general/comp1.xml';</pre></td><td>Déplacement de la ressource <code>comp1.xml</code> vers le répertoire <code>/home/SOUTOU</code>.</td></tr>
<tr><td><pre>UPDATE RESOURCE_VIEW
  SET res = UPDATEXML(RES,
  '/r:Resource/r:Contents/compagnie/nomComp/text()',
  'Compagnie Euralair','xmlns:r="http://xmlns.oracle.com/
   xdb/XDBResource.xsd"')
  WHERE EQUALS_PATH(RES,'/home/SOUTOU/comp1.xml')=1;</pre></td><td>Modification du nom de la compagnie du document <code>comp1.xml</code>.</td></tr>
<tr><td><pre>UPDATE TABCOMPREPO c
  SET c.OBJECT_VALUE = UPDATEXML(
           c.OBJECT_VALUE,'/compagnie/nomComp/text()',
          'Nouveau nom : Euralair')
  WHERE REF(c) =
    (SELECT EXTRACTVALUE(rv.RES,'/Resource/XMLRef')
     FROM RESOURCE_VIEW rv
     WHERE EQUALS_PATH(rv.RES,'/home/SOUTOU/comp1.xml')=1);</pre></td><td>Modification du nom de la compagnie du document <code>comp1.xml</code> en utilisant la table de stockage définie par défaut au niveau de la grammaire.</td></tr>
</table>

© Éditions Eyrolles

Annexe

Bibliographie et webographie

Dans les livres

[BOU 99] N. Boudjilda, *Bases de données et systèmes d'informations – Le modèle relationnel : langages, systèmes et méthodes,* Dunod, 1999.

[BRO 06] S. Brouard, *SQL*, Pearson Education, 2006.

[DEL 00] P. Delmal, *SQL2-SQL3, Applications à Oracle*, De Boeck Université, 2000.

[DEL 98] D. Deléglise, *Oracle et le Web,* Eyrolles, 1998.

[GEN 03] J. Gennick, *First Expressions*, Oracle Magazine, pp. 95-98, sept-oct. 2003.

[MAR 94] C. Marée, G. Ledant, *SQL2 Initiation Programmation*, Armand Colin, 1994.

[MAT 00] P. Mathieu, *Des bases de données à l'Internet*, Vuibert, 2000.

[SOU 04] C. Soutou, *Programmer objet sous Oracle*, Vuibert, 2004.

Sur le Web

À propos d'Oracle

Le site de l'éditeur : http://www.oracle.com

Tutoriaux, FAQ, forum : http://oracle.developpez.com/

Plein d'articles et d'exemples : http://oracle-base.com

Quelques trucs à propos de la version 11g : http://dbmsdirect.blogspot.com/2007/10/oracle-11g-database-hints.html

À propos de SQL

Des ressources : http://sqlpro.developpez.com

La norme : http://www.iso.org/iso/fr

© Éditions Eyrolles

Index

Symboles

%FOUND 311
%ISOPEN 311
%NOTFOUND 311
%ROWCOUNT 311
%ROWTYPE 262
%TYPE 261
(+) 146
:NEW 336
:OLD 339

A

ABS 124
absolute 394
ACCEPT 267
ACOS 124
ADD CONSTRAINT 89
ADD_MONTHS 125
ADMIN_OPTION 248
ADMINISTER DATABASE TRIGGER 334
AFTER 336
afterLast 394
alias
  colonne 113
  en-tête 115
  table 113
  vue 225
ALL 151
ALL_USERS 244, 248
ALLOW ANYSCHEMA 515
ALLOW NONSCHEMA 515
ALTER PROFILE 200
ALTER ROLE 221
ALTER SEQUENCE 62
ALTER TABLE
  ADD 86
  ADD CONSTRAINT 89
  DISABLE CONSTRAINT 92
  DROP COLUMN 88
  DROP CONSTRAINT 90
  ENABLE CONSTRAINT 94
  MODIFY 87
  MODIFY CONSTRAINT 101
  RENAME COLUMN 86
  SET UNUSED COLUMN 88
ALTER TRIGGER 350
ALTER VIEW 236
AND 118
ANY 151
ANY_PATH 528
Apache 453, 469
APPENDCHILDXML 508
AS SELECT 116
ASC 114
ASCII 121
associative array 264
ATAN 124
AUTHID 297
autojointure 143
AVG 129

B

BEFORE 336
beforeFirst 394
BEGIN 256
BETWEEN 119
BFILE 38, 57
BFILENAME 57
BIN_TO_NUM 127
binary XML 489, 512
BINARY_INTEGER 269
BLANK_TRIMMING 87
BLOB 38
block label 266

C

CACHE 59
CALL 443
CallableStatement 407
cancelRowUpdates 398
CASCADE 91, 92, 198, 200
CASCADE CONSTRAINTS 46, 212
CASE 274
CAST 127
CEIL 124
CHAR 36
CHARACTER 270
CHARARR 285
CHARTOROWID 127
CHR 121
CLASSPATH 381
clé
    candidate 3
    étrangère 3
    primaire 3
CLOB 36, 492, 511
colonne 2
    XMLType 511
    virtuelle 102
COLUMN 164
COMMENT 40
COMMENT ANY TABLE 213
commentaire
    PL/SQL 259
    SQL 32
COMMIT 289
COMPOSE 127
CONCAT 121
concaténation 115
CONNECT 26, 217
CONNECT BY 163
CONNECT_TIME 199
Connection 385
CONSTANT 260
CONSTRAINT_TYPE 246
CONTINUE 279, 369
contrainte 34
    CHECK 34
    FOREIGN KEY 34
    in-line 34
    out-of-line 34
    PRIMARY KEY 34
    référentielle 71
    UNIQUE 34
conversions 126
CONVERT 127, 528
COS 124
COSH 124
COUNT 129
CPU_PER_CALL 199
CPU_PER_SESSION 199
CREATE DIRECTORY 451, 493
CREATE FUNCTION 297
CREATE INDEX 43
CREATE JAVA 451
CREATE PACKAGE 307
CREATE PACKAGE BODY 308
CREATE PROCEDURE 296
CREATE PROFILE 198
CREATE ROLE 215
CREATE SEQUENCE 58
CREATE SYNONYM 237
CREATE TABLE 31
CREATE TRIGGER 334
CREATE USER 195
CREATE VIEW 223
CREATEFOLDER 527
CREATERESOURCE 527
createStatement 386
CREATEXML 498
CROSS JOIN 159
CSID 497
CURRENT_DATE 55, 125
CURRVAL 60
curseur
    explicite 310
    implicite 284, 327
CURSOR 311
CYCLE 58

© Éditions Eyrolles

D

DAD 454
data dictionary 239
DatabaseMetaData 403
DATE 38, 53
DAV 524
DBA 194, 217
DBA_ROLE_PRIVS 244
DBA_ROLES 244
DBMS_JAVA.DROPJAVA 444
DBMS_OUTPUT 285
DBMS_RANDOM 285
DBMS_ROWID 285
DBMS_SQL 285
DBMS_XMLQUERY 487
DBMS_XMLSAVE 487
DBMS_XMLSCHEMA 496
DBTIMEZONE 55
DEC 270
DECIMAL 270
DECLARE 256
DECLARE SECTION 366
déclencheur 332
DECODE 128
DEFAULT 52
DEFAULT ROLE 197
DEFAULT TABLESPACE 195
DEFERRABLE 97
DEFERRED 98
DELETE 70
DELETE() 265
DELETERESOURCE 527
deleteRow 398
DELETESCHEMA 496
DELETEXML 508
DELETING 343
DEPTH 533
DESC 39, 114
descripteur de connexion 20
DICT 241
DICTIONARY 241
dictionnaire des données 239
DISABLE 352
DISABLE ALL TRIGGERS 350
DISABLE CONSTRAINT 92
DISALLOW NONSCHEMA 515
DISTINCT 113
division 159
DO 369
DOUBLE PRECISION 270
DriverManager 382
DROP COLUMN 88
DROP CONSTRAINT 90
DROP FUNCTION 306
DROP INDEX 47, 92
DROP PACKAGE 309
DROP PACKAGE BODY 309
DROP PROCEDURE 306
DROP PROFILE 200
DROP ROLE 221
DROP SEQUENCE 63
DROP TABLE 46
DROP TRIGGER 350
DROP UNUSED COLUMNS 88
DROP USER 198
DROP VIEW 236
dropjava 444
DUAL 110

E

ECHO 27
ELEMENT 498
ENABLE ALL TRIGGERS 350
ENABLE CONSTRAINT 94
endFetch() 423
Enterprise Manager 201
ENTRYID 233
EQUALS_PATH 528
équijointure 141
étiquette 266, 369
EXCEPTION 257
exception
  JDBC 411
  PL/SQL 319
  SQLJ 431
EXCEPTIONS INTO 95

EXECUTE 301
execute 387, 405, 407
EXECUTE IMMEDIATE 357
executeQuery 387, 407
executeUpdate 387, 405, 407
EXISTS 156
EXISTS() 265
EXISTSNODE 506
EXISTSRESOURCE 527
EXIT 276
EXP 124
EXP_FULL_DATABASE 217
EXPIRE 195
expression 114
external procedures 439
EXTRACT 68, 125, 504
EXTRACTVALUE 505

F

FAILED_LOGIN_ATTEMPTS 199
FETCH 311
FIRST 265
first 394
FLOAT 270
FLOOR 124
FOLLOWS 352
FOR 277
FOR EACH ROW 335
FOR UPDATE 315
FORCE 213, 223
formulaire 462
FROM 112
FULL OUTER JOIN 148

G

GENERATED ALWAYS 102
GENERATESCHEMA 520
GENTABLES 497
GENTYPES 497
GET 26
GET_LINES 285
getColumnCount 402
getColumnName 402
getColumns 403
getColumnType 402
getColumnTypeName 402
getConcurrency 398
getConnection 387
getDatabaseProductName 403
getDatabaseProductVersion 403
getErrorCode 411
getFetchDirection 394, 424
getMessage 411
getMetaData 392
getNextException 411
GETNUMBERVAL 520
getPrecision 402
getResultSet 423
getResultSetConcurrency 398
getResultSetType 398
getSavepointId 410
getSavepointName 410
getScale 402
getSchemaName 402
getSQLState 411
getTableName 402
getTables 403
getter methods 387
getType 398
getUpdateCount 387
getUserName 403
GOTO 369
grammaire XML Schema 493
GRANT 207, 210
GRANT ANY OBJECT PRIVILEGE 213
GRANT ANY PRIVILEGE 213
GRANTED_ROLE 248
GREATEST 128
GROUP BY 128

H

HAVING 128
HEAP 45
HOST 19
HTF 455
HTP 455

© Éditions Eyrolles

I

IDENTIFIED 215
IDENTIFIED BY 195
IDLE_TIME 199
IF 272
IMMEDIATE 98
IMP_FULL_DATABASE 217
IN 119, 151
IN OUT 300
INCREMENT BY 58
index 40
  bitmap 42
  B-tree 41
  function based 42
  UNIQUE 43
INDEX BY BINARY_INTEGER 264
inéquijointure 144
INITCAP 121
INITIALLY 97
INNER JOIN 142
INSERT 51
INSERTCHILDXML 508
INSERTING 343
insertRow 398
INSTEAD OF 345
INSTR 122
INT 270
INTEGER 270
intégrité réferentielle 71
INTERSECT 134
INTERVAL DAY TO SECOND 38, 55
INTERVAL YEAR TO MONTH 38, 55
INTO 280
IS NULL 120, 267
IS TABLE OF 264
isAfterLast 394, 424
isBeforeFirst 394, 424
isClosed 423
isFirst 394, 424
isLast 394, 424
isNullable 402
*iSQL*Plus* 22
ISSCHEMAVALID 500
itérateur 421

J

JDBC 377
JOIN 142
jointure 139
  equi join 141
  externe 145
  inner join 141
  mixte 154
  naturelle 157
  outer join 145
  procédurale 150
  relationnelle 140
  self join 143
  SQL2 140

K

KEEP INDEX 92
key preserved 231
KILL SESSION 197

L

LANGUAGE 233
LAST 265
last 394
LAST_DAY 125
LEAST 128
LENGTH 122
LEVEL 164
LIKE 120
LINESIZE 27
LINK 533
LMD 51
LN 124
loadjava 440
LOB 2, 57
LOCAL 497
LOCALTIMESTAMP 55
LOG 124
LOGOFF 349
LOGON 349
LONG RAW 38
LOOP 276

© Éditions Eyrolles

LOWER 122
LPAD 122, 164
LTRIM 122

M

MAX 129
MAXVALUE 58
MERGE 172, 173
MIN 129
MINUS 135
MINVALUE 58
MOD 124
MODIFY 87
MODIFY CONSTRAINT 101
MONTHS_BETWEEN 125
moveToCurrentRow 398
moveToInsertRow 398
mutating tables 351

N

NATURAL 270
NATURAL JOIN 157
NATURALN 270
NCHAR 36, 56
NCLOB 36
nested subprogram 304
NESTED_TABLE_ID 504
NEW_LINE 285
NEW_TIME 125
NEXT 265
next() 392, 423
NEXT_DAY 125
NEXTVAL 60
NOCACHE 59
NOCOPY 297
NOCYCLE 59
NOFORCE 223
NOMAXVALUE 58
NOMINVALUE 58
NOORDER 59
NOT 118
NOT EXISTS 157
NOT IDENTIFIED 215
NOT IN 151
NOT NULL 32, 34
NOVALIDATE 100
NOWAIT 316
NULL 32, 52
NULLIF 128
NULLS FIRST 114
NULLS LAST 114
NUMBER 37
NUMERIC 270
NUMTODSINTERVAL 67, 68
NUMTOYMINTERVAL 68
NVARCHAR2 36
NVL 128

O

O7_DICTIONARY_ACCESSIBILITY 206
OBJECT RELATIONAL 492
OBJECT_VALUE 504
OCI 384
oci_bind_by_name 478
oci_cancel 476
oci_close 472
oci_commit 474
oci_connect 471
OCI_DEFAULT 473
oci_define_by_name 478
oci_error 479
oci_execute 473
oci_fetch_all 475, 477
oci_fetch_array 475, 476
oci_fetch_assoc 475
oci_fetch_object 475
oci_fetch_row 475
OCI_FETCHSTATEMENT_BY_ROW 473
oci_field_is_null 483
oci_field_name 483, 485
oci_field_precision 483
oci_field_scale 483
oci_field_size 484, 485
oci_field_type 484, 485

© Éditions Eyrolles

oci_free_statement 476
oci_internal_debug 479, 482
oci_new_connect 471
OCI_NUM 473
oci_num_fields 476, 483, 485
oci_num_rows 484
oci_parse 473
oci_password_change 484
oci_pconnect 472
OCI_RETURN_NULLS 473
oci_rollback 474
oci_server_version 483
oci_set_prefetch 476
oci_statement_type 484
ON DELETE CASCADE 75
ON DELETE SET NULL 75
OPEN 311
OPEN FOR 317
OR 118
OR REPLACE 223
ORA-00001 53, 321
ORA-00051 321
ORA-01001 321
ORA-01012 321
ORA-01017 321
ORA-01402 235
ORA-01403 281, 321
ORA-01410 321
ORA-01422 281, 321
ORA-01426 269, 271
ORA-01476 321
ORA-01722 321
ORA-01732 230
ORA-01733 227, 232
ORA-01752 227
ORA-01776 230
ORA-02273 91
ORA-02290 53, 501
ORA-02291 53
ORA-02293 501
ORA-02297 92
ORA-04091 351
ORA-06500 321
ORA-06501 321
ORA-06502 270, 321
ORA-06504 321
ORA-06511 321
ORA-06530 321
ORA-06531 321
ORA-06532 321
ORA-06533 321
ORA-06592 321
ORA-12081 104
ORA-30625 321
ORA-30937 500
ORA-30951 500
ORA-31154 501
OracleNet 6
ORDER 59
ORDER BY 114
ORGANIZATION INDEX 45
OTHERS 320
OUTER JOIN 147

P

PAGESIZE 27
paquetage 306
PASSING BY VALUE 505
PASSWORD_GRACE_TIME 199
PASSWORD_LIFE_TIME 199
PASSWORD_LOCK_TIME 199
PASSWORD_REUSE_MAX 199
PASSWORD_REUSE_TIME 199
PATH 533
PATH_VIEW 527, 533
PHP 468
PL/SQL 6, 255
  curseurs 310
  exceptions 319
  fonction cataloguée 295
  paquetage 306
  procédure cataloguée 295
  sous-programme 295
  variable curseur 317
PL/SQL Web Toolkit 453
PLS_INTEGER 269
PLS-00218 270

© Éditions Eyrolles

POSITION 246
POSITIVE 270
POSITIVEN 270
POST 467
POWER 124
PRAGMA AUTONOMOUS_TRANSACTION 297
PRAGMA EXCEPTION_INIT 328
précompilateur 6
prepareCall 386
PreparedStatement 405
prepareStatement 386
previous 394
PRINT 268
PRIOR 164, 265
PRIVATE_SGA 199
privilège 206
  objet 209
  système 206
Pro*C/C++ 365
produit cartésien 137, 159
PSP 453, 464
PUBLIC 207, 209
PUT 285
PUT_LINE 285

Q

QUOTA 195

R

R_CONSTRAINT_NAME 246
RAISE 326
RAISE_APPLICATION_ERROR 331
RAW 38
REAL 270
RECORD 263
récursivité 304
REF CURSOR 317, 359, 430
REFERENCING 342
REGEXP_COUNT 185
registerOutParameter 407
REGISTERSCHEMA 496
relative 394
RELEASE 27
releaseSavepoint 410
RENAME 85
RENAME COLUMN 86
RENAME TO 85
REPLACE 122
requête 109
  hiérarchique 162
reraise 330
RES 528
RESID 528, 533
RESOURCE 217, 296
RESOURCE_VIEW 527
ResultSet 391
ResultSetMetaData 402
RETURN 299, 313
REVERSE 277
REVOKE 209, 212, 219
rôle 214
ROLE_ROLE_PRIVS 244
ROLE_SYS_PRIVS 244
ROLE_TAB_PRIVS 244
ROLLBACK 289
ROLLBACK TO... 290
ROUND 124, 125
row 2
row trigger 335
ROWID 45
rowid 115
ROWIDTOCHAR 127
ROWNUM 116
RPAD 122
RTRIM 123

S

SAVE 26
Savepoint 410
  JDBC 410
  PL/SQL 290
  SQLJ 427
SCHEMA 348
schéma 7

 © Éditions Eyrolles

SCHEMADOC 497
SCHEMAURL 497
SCHEMAVALIDATE 501
Scrollable 424
SELECT 110
  fonctions 120
SELECT ANY DICTIONARY 213
SELECT... INTO 280
SEQUEL 1
séquence 57
SERVERERROR 349
SERVEROUT 27
SESSION_ROLES 244
SESSIONID 233
SESSIONS_PER_USER 199
SESSIONTIMEZONE 55
SET 26
SET CONSTRAINT 99
SET CONSTRAINTS 99
SET ROLE 219
SET UNUSED COLUMN 88
SET_OUTPUT 447
setAutoCommit 386
setFetchDirection 394, 424
setMaxRows 387
setNull 405
setSavepoint 410
setter methods 387
SHOW 27
SHOW ERRORS 301
SHUTDOWN 349
SIBLINGS 114, 167
SIGN 124
SIGNTYPE 270
SIMPLE_DOUBLE 271
SIMPLE_FLOAT 271
SIMPLE_INTEGER 271
SIN 124
SINH 124
SMALLINT 270
SOUNDEX 123
sous-interrogation 150
  dans le FROM 154
  synchronisée 155
SPOOL 26
SQL Developer 24
SQL dynamique 356
SQL%FOUND 284
SQL%NOTFOUND 284
SQL%ROWCOUNT 284
SQL*Plus 19
SQL*Plus Worksheet 21
SQL2 1
SQL3 1
SQLCA 368
SQLCODE 322
sqlerrd 374
SQLERRM 323
sqlerrml 368
SQLException 411
SQLJ 415
SQRT 124
START 26
START WITH 58, 163
STARTUP 349
Statement 387
statement trigger 344
STDDEV 129
STOP 369
stored procedures 439
substitution 267
SUBSTR 123
SUBTYPE 270
SUM 129
supportsSavepoints 403
supportsTransactions 403
SUSPEND 349
SYN 244
synonyme 237
SYS 196
SYS_XMLGEN 518
SYSDATE 66, 125
SYSDBA 196, 213
SYSOPER 196, 213
SYSTEM 196
SYSTIMESTAMP 55

© Éditions Eyrolles

T

table 2, 31
  dominante 145
  EXTERNAL 46
  fils 71
  heap-organized 44
  index-organized 44
  key preserved 231
  père 71
  subordonnée 146
  XMLType 498
table lecture seule 104
tableau
  PL/SQL 264
  Pro*C/C++ 373
tablespace 193
TABS 244
TAN 124
TANH 124
TEMP 195
TEMPORARY TABLESPACE 195
TERMINAL 233
TERMOUT 27
TIME 27
TIMESTAMP 38, 54
tnsnames.ora 20
TO_CHAR 68, 127
TO_DATE 54, 67, 68
TO_DSINTERVAL 127
TO_NUMBER 127
TO_YMINTERVAL 127
transaction 288
TRANSLATE 123
TRIM 123
TRUNC 125
TRUNCATE 70

U

UID 233
UNDER_PATH 528, 533
Unicode 56
UNION 135
UNION ALL 135
UNIQUE 113
UNISTR 56, 127
UPDATE 63
updater methods 387
updateRow 398
UPDATEXML 508
UPDATING 343
UPPER 123
USER 27, 233
user 194
USER_ALL_TABLES 244
USER_COL_COMMENTS 244
USER_COL_GRANTS 244
USER_COL_GRANTS_MADE 244
USER_COL_PRIVS_MADE 244
USER_COL_PRIVS_RECD 244
USER_CONS_COLUMNS 244
USER_CONSTRAINTS 244, 246
USER_ERRORS 244, 301
USER_IND_COLUMNS 244
USER_IND_EXPRESSIONS 244
USER_INDEXES 244
USER_OBJECTS 244, 245
USER_ROLE_PRIVS 244, 248
USER_SEQUENCES 242
USER_SOURCE 244, 247
USER_STORED_SETTINGS 244
USER_SYNONYMS 244
USER_TAB_COLUMNS 244, 245
USER_TAB_COMMENTS 244
USER_TAB_GRANTS 244
USER_TAB_GRANTS_MADE 244
USER_TAB_GRANTS_RECD 244
USER_TABLES 244
USER_UNUSED_COL_TABS 244
USER_UPDATABLE_COLUMNS 232
USER_USERS 244
USER_VIEWS 244
USER_XML_SCHEMAS 523
USER_XML_TAB_COLS 522
USER_XML_TABLES 522
USER_XML_VIEWS 523
USERS 195
USING 158

 © Éditions Eyrolles

V

VALIDATE 99
VALUE_ERROR 260
VARCHAR2 36
VARIABLE 268
variable curseur 317
VARIANCE 129
VARRAY 498
VIRTUAL 102
VIRTUAL COLUMNS 492, 513
vue 222
  monotable 224
  XMLType 517

W

WAIT 316
wasNull 407
WebDAV 524
WHEN 341
WHENEVER 369
WHILE 275
WIDTH_BUCKET 125, 187
WITH 185
WITH ADMIN OPTION 207
WITH CHECK OPTION 224
WITH GRANT OPTION 210
WITH OBJECT ID 520
WITH READ ONLY 224, 226

X

XDB 487
xdb:columnProps 494, 511
xdb:defaultTable 494, 511
xdb:defaultTableSchema 494
xdb:SQLCollType 494
xdb:SQLName 494
xdb:SQLType 494
xdb:storeVarrayAsTable 494
xdb:tableProps 494, 511
XML DB Repository 524
XML_DB 487
XML_XDB 527
XMLATTRIBUTES 516
XMLDATA 498
XMLELEMENT 516
XMLEXISTS 506
XMLFOREST 516
XMLFORMAT 518
XMLISVALID 501
XMLQUERY 505
XMLSCHEMA 492, 498
XMLTABLE 505
XMLTYPE 491
XMLType 488

Oracle Technology Network Developer License Terms

To accept this license, you must agree to all of the following terms :

ELIGIBILITY EXPORT RESTRICTIONS

- I am not a citizen, national or resident of, and am not under the control of, the government of: Cuba, Iran, Sudan, Iraq, Libya, North Korea, Syria, nor any other country to which the United States has prohibited export.
- I will not download or otherwise export or re-export the Programs, directly or indirectly, to the above mentioned countries nor to citizens, nationals or residents of those countries.
- I am not listed on the United States Department of Treasury lists of Specially Designated Nationals, Specially Designated Terrorists, and Specially Designated Narcotic Traffickers, nor am I listed on the United States Department of Commerce Table of Denial Orders.
- I will not download or otherwise export or re-export the Programs, directly or indirectly, to persons on the above mentioned lists.
- I will not use the Programs for, and will not allow the Programs to be used for, any purposes prohibited by United States law, including, without limitation, for the development, design, manufacture or production of nuclear, chemical or biological weapons of mass destruction.

Note: You are bound by the Oracle Technology Network ("OTN") License Agreement terms. The OTN License Agreement terms also apply to all updates you receive under your Technology Track subscription.

The OTN License Agreement terms below supercede any shrinkwrap license on the OTN Technology Track software CDs and previous OTN License terms (including the Oracle Program License as modified by the OTN Program Use Certificate).

ORACLE TECHNOLOGY NETWORK DEVELOPMENT LICENSE AGREEMENT

"We," "us," and "our" refers to Oracle Corporation. "You" and "your" refers to the individual or entity that has ordered the programs from Oracle. "Programs" refers to the software product which you have ordered and program documentation. "License" refers to your right to use the programs under the terms of this agreement. This agreement is governed by the substantive and procedural laws of California. You and Oracle agree to submit to the exclusive jurisdiction of, and venue in, the courts of California in any dispute relating to this agreement.

We are willing to license the programs to you only upon the condition that you accept all of the terms contained in this agreement. Read the terms and conditions of the Agreement carefully. By opening this package, you agree to be bound by the terms and conditions of the Agreement.

License Rights

We grant you a nonexclusive, nontransferable limited license to use the programs only for purposes of developing and prototyping your applications, and not for any other purpose. If you use the applications you develop under this license for any internal data processing or for any commercial or production purposes, or you want to use the programs for any purpose other than as permitted under this agreement, you must contact us, or an Oracle reseller, to obtain the appropriate license. We may audit your use of the programs. Program documentation is either shipped with the programs, or documentation may accessed online at http://otn.oracle.com/docs.

Ownership and Restrictions

We retain all ownership and intellectual property rights in the programs. The programs may be installed on one computer only, and used by one person in the operating environment identified by us. You may make one copy of the programs for backup purposes.

You may not:

- use the programs for your own internal data processing or for any commercial or production purposes, or use the programs for any purpose except the development and prototyping of your applications;
- use the applications you develop with the programs for any internal data processing or commercial or production purposes without securing an appropriate license from us;
- remove or modify any program markings or any notice of our proprietary rights;
- make the programs available in any manner to any third party;
- use the programs to provide third party training;
- assign this agreement or give or transfer the programs or an interest in them to another individual or entity;
- cause or permit reverse engineering or decompilation of the programs;
- disclose results of any program benchmark tests without our prior consent; or,
- use any Oracle name, trademark or logo.

Export

You agree that U.S. export control laws and other applicable export and import laws govern your use of the programs, including technical data. You agree that neither the programs nor any direct product thereof will be exported, directly, or indirectly, in violation of these laws, or will be used for any purpose prohibited by these laws including, without limitation, nuclear, chemical, or biological weapons proliferation.

Disclaimer of Warranty and Exclusive Remedies

THE PROGRAMS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.WE FURTHER DISCLAIM ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL WE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. OUR ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

Trial Programs Included With Orders

We may include additional programs with an order which may be used for trial purposes only.You will have 30 days from the delivery date to evaluate these programs. Any use of these programs after the 30 day trial period requires you to obtain the applicable license. Programs licensed for trial purposes are provided "as is" and we do not provide technical support or any warranties for these programs.

No Technical Support

Our technical support organization will not provide technical support, phone support, or updates to you for the programs licensed under this agreement.

End of Agreement

You may terminate this agreement by destroying all copies of the programs. We have the right to terminate your right to use the programs if you fail to comply with any of the terms of this agreement, in which case you shall destroy all copies of the programs.

Relationship Between the Parties

The relationship between you and us is that of licensee/licensor. Neither party will represent that it has any authority to assume or create any obligation, express or implied, on behalf of the other party, nor to represent the other party as agent, employee, franchisee, or in any other capacity. Nothing in this agreement shall be construed to limit either party's right to independently develop or distribute software that is functionally similar to the other party's products, so long as proprietary information of the other party is not included in such software.

Restricted Rights

Use, duplication or disclosure by the United States government is subject to the restrictions as set forth in the Rights in Technical Data and Computer Software Clauses in DFARS 252.227-7013(c)(1)(ii) and FAR 52.227-19(c)(2) as applicable. The manufacturer is Oracle Corporation, 500 Oracle Parkway, Redwood City, California 94065, U.S.A.

Open Source

"Open Source" software - software available without charge for use, modification and distribution - is often licensed under terms that require the user to make the user's modifications to the Open Source software or any software that the user 'combines' with the Open Source software freely available in source code form. If you use Open Source software in conjunction with the programs, you must ensure that your use does not: (i) create, or purport to create, obligations of us with respect to the Oracle programs; or (ii) grant, or purport to grant, to any third party any rights to or immunities under our intellectual property or proprietary rights in the Oracle programs. For example, you may not develop a software program using an Oracle program and an Open Source program where such use results in a program file(s) that contains code from both the Oracle program and the Open Source program (including without limitation libraries) if the Open Source program is licensed under a license that requires any "modifications" be made freely available. You also may not combine the Oracle program with programs licensed under the GNU General Public License ("GPL") in any manner that could cause, or could be interpreted or asserted to cause, the Oracle program or any modifications thereto to become subject to the terms of the GPL.

Entire Agreement

You agree that this agreement is the complete agreement for the programs and licenses, and this agreement supersedes all prior or contemporaneous agreements or representations. If any term of this agreement is found to be invalid or unenforceable, the remaining provisions will remain effective.

Should you have any questions concerning this License Agreement, or if you desire to contact Oracle for any reason, please write:Oracle Corporation500 Oracle Parkway,Redwood City, CA 94065

Christian Soutou est maître de conférences à l'université Toulouse Le Mirail et dirige le département Réseaux et Télécoms de l'IUT de Blagnac. Il intervient autour des bases de données et des technologies de l'information (XML, services Web et SOA) en licence et master professionnels. Il est également l'auteur des ouvrages *Apprendre SQL avec MySQL* et *UML 2 pour les bases de données*, parus aux éditions Eyrolles.

Apprendre SQL par l'exemple

Tout particulièrement conçu pour les étudiants et les débutants, cet ouvrage permet d'acquérir les notions essentielles du langage SQL par le biais 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 11 *g*, 10 *g* et 9*i* d'Oracle, il permet également de se familiariser avec les principales fonctionnalités de ce logiciel, ainsi qu'avec les API les plus utilisées (JBDC et PHP). Mise à jour et augmentée, la troisième édition de cet ouvrage consacre en outre un chapitre à l'interopérabilité entre SQL et XML (documents et grammaires) et à la gestion de ressources avec XML DB Repository.

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 !

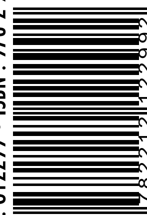
Cet ouvrage décrit en détail la procédure d'installation des versions 11 *g*, 10 *g*, 10 *g* Express et 9*i* d'Oracle. Ces versions peuvent être téléchargées gratuitement sur le site d'Oracle : destinées à des fins non commerciales, elles sont complètes et sans limitation de durée.

Au sommaire

Introduction. Installation d'Oracle. Les interfaces SQL *Plus. **Partie I : SQL de base.** Définition des données. Manipulation des données. Évolution d'un schéma. Interrogation des données. Contrôle des données. **Partie II : PL/ SQL.** Bases du PL/SQL. Programmation avancée. **Partie III : SQL avancé.** Le précompilateur Pro*C/C++. L'interface JDBC. L'approche SQLJ. Procédures stockées et externes. Oracle et le Web (Web Toolkit, PSP, API PHP). Oracle XML DB.

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 : G12299 • ISBN : 978-2-212-12299-2

www.editions-eyrolles.com
Groupe Eyrolles | Diffusion Geodif | Distribution Sodis