Christian Soutou
Avec la participation d'Olivier Teste

SQL pour Oracle

4e édition

© Groupe Eyrolles, 2004, 2005, 2008, 2010, ISBN : 978-2-212-12794-2

Table des matières

Remerciements XIX

Avant-propos XXI

Guide de lecture XXII
Première partie : SQL de base XXII
Deuxième partie : PL/SQL XXII
Troisième partie : SQL avancé XXII
Annexe : bibliographie et webographie XXII
Conventions d'écriture XXIII
Contact avec l'auteur et site Web XXIII

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
Rachat de Sun (et de MySQL) 5
Offre du moment 6
Notion de schéma 8
Accès à Oracle depuis Windows 9
Détail d'un numéro de version 10
Installation d'Oracle 10
Mise en œuvre d'Oracle 9*i* 10
Désinstallation de la 9*i* 13
Mise en œuvre d'Oracle 10*g* 13
Désinstallation de la 10*g* 16
Mise en œuvre d'Oracle 10*g* Express Edition 17
Mise en œuvre d'Oracle 11*g* 17
Désinstallation de la 11*g* 22
Les interfaces SQL*Plus 22
Généralités 22
Premiers pas 27
Variables d'environnement 29
À propos des accents et jeux de caractères 30

Partie I SQL de base 33

1 Définition des données 35
Tables relationnelles 35
Création d'une table (CREATE TABLE) 35
Casse et commentaires 36
Premier exemple 37
Contraintes 37
Conventions recommandées 39
Types des colonnes 40
Structure d'une table (DESC) 43
Restrictions 44
Commentaires stockés (COMMENT) 44
Index 45
Classification 46
Index B-tree 46
Index bitmap 46
Index basés sur des fonctions 47
Création d'un index (CREATE INDEX) 47
Bilan 48
Tables organisées en index 49
Destruction d'un schéma 50

2 Manipulation des données 55
Insertions d'enregistrements (INSERT) 55
Syntaxe 55
Renseigner toutes les colonnes 56
Renseigner certaines colonnes 56
Ne pas respecter des contraintes 57
Dates/heures 57
Caractères Unicode 60
Données LOB 60
Séquences 61
Création d'une séquence (CREATE SEQUENCE) 62
Manipulation d'une séquence 64
Modification d'une séquence (ALTER SEQUENCE) 65
Visualisation d'une séquence 66
Suppression d'une séquence (DROP SEQUENCE) 67
Modifications de colonnes 67
Syntaxe (UPDATE) 67
Modification d'une colonne 68
Modification de plusieurs colonnes 68
Ne pas respecter des contraintes 68
Dates et intervalles 69

 © Éditions Eyrolles

Suppressions d’enregistrements . . . 74
Instruction DELETE . . . 74
Instruction TRUNCATE . . . 74
Intégrité référentielle . . . 75
Cohérences . . . 75
Contraintes côté « père » . . . 76
Contraintes côté « fils » . . . 76
Clés composites et nulles . . . 77
Cohérence du fils vers le père . . . 78
Cohérence du père vers le fils . . . 78
En résumé . . . 79
Flottants . . . 80
Valeurs spéciales . . . 81
Fonctions pour les flottants . . . 81

3 Évolution d’un schéma . . . 89
Renommer une table (RENAME) . . . 89
Modifications structurelles (ALTER TABLE) . . . 90
Ajout de colonnes . . . 90
Renommer des colonnes . . . 90
Modifier le type des colonnes . . . 91
Supprimer des colonnes . . . 92
Colonnes UNUSED . . . 92
Modifications comportementales . . . 93
Ajout de contraintes . . . 93
Suppression de contraintes . . . 94
Désactivation de contraintes . . . 96
Réactivation de contraintes . . . 98
Contraintes différées . . . 101
Directives DEFERRABLE et INITIALLY . . . 101
Instructions SET CONSTRAINT . . . 103
Instruction ALTER SESSION SET CONSTRAINTS . . . 103
Directives VALIDATE et NOVALIDATE . . . 103
Directive MODIFY CONSTRAINT . . . 105
Colonne virtuelle . . . 106
Table en lecture seule . . . 108

4 Interrogation des données . . . 113
Généralités . . . 113
Syntaxe (SELECT) . . . 114
Pseudo-table DUAL . . . 114
Projection (éléments du SELECT) . . . 115
Extraction de toutes les colonnes . . . 116
Extraction de certaines colonnes . . . 116

Alias. 117
Duplicatas 117
Expressions et valeurs nulles 118
Ordonnancement. 118
Concaténation 119
Pseudo-colonne ROWID 119
Pseudo-colonne ROWNUM 120
Insertion multiligne 120
Restriction (WHERE) 121
Opérateurs de comparaison 122
Opérateurs logiques 122
Opérateurs intégrés. 123
Fonctions 124
Caractères. 125
Numériques 128
Dates 129
Conversions 130
Autres fonctions. 132
Regroupements 132
Fonctions de groupe 133
Étude du GROUP BY et HAVING 134
Opérateurs ensemblistes 137
Restrictions 137
Exemple 138
Opérateur INTERSECT. 138
Opérateurs UNION et UNION ALL 139
Opérateur MINUS 139
Ordonner les résultats 140
Produit cartésien 141
Bilan 143
Jointures. 143
Classification 144
Jointure relationnelle 144
Jointures SQL2 144
Types de jointures. 145
Équijointure 145
Autojointure 147
Inéquijointure. 148
Jointures externes. 149
Jointures procédurales 154
Jointures mixtes 158
Sous-interrogations synchronisées 159
Autres directives SQL2 161

© Éditions Eyrolles

Division . . . 163
Définition . . . 164
Classification . . . 164
Division inexacte en SQL . . . 165
Division exacte en SQL . . . 166
Requêtes hiérarchiques . . . 166
Point de départ du parcours (START WITH) . . . 167
Parcours de l'arbre (CONNECT BY PRIOR) . . . 167
Indentation . . . 168
Élagage de l'arbre (WHERE et PRIOR) . . . 169
Jointures . . . 171
Ordonnancement . . . 171
Nouveautés 10*g* . . . 172
Mises à jour conditionnées (fusions) . . . 176
Syntaxe (MERGE) . . . 176
Exemple . . . 177
Nouveautés 10*g* . . . 177
Exemple . . . 178
Expressions régulières . . . 179
Quelques exemples . . . 181
Fonction REGEXP_LIKE . . . 181
Fonction REGEXP_REPLACE . . . 184
Fonction REGEXP_INSTR . . . 185
Fonction REGEXP_SUBSTR . . . 187
Nouveautés 11*g* . . . 188
Extractions diverses . . . 189
Directive WITH . . . 189
Fonction WIDTH_BUCKET . . . 191
Récursivité avec WITH (CTE) . . . 192
Pivots (PIVOT) . . . 201
Transpositions (UNPIVOT) . . . 205

5 Contrôle des données . . . 213
Gestion des utilisateurs . . . 214
Classification . . . 214
Création d'un utilisateur (CREATE USER) . . . 214
Modification d'un utilisateur (ALTER USER) . . . 216
Suppression d'un utilisateur (DROP USER) . . . 217
Profils . . . 218
Console Enterprise Manager . . . 221
Privilèges . . . 226
Privilèges système . . . 226
Privilèges objets . . . 229
Privilèges prédéfinis . . . 233

© Éditions Eyrolles

Rôles 234
Création d'un rôle (CREATE ROLE) 235
Rôles prédéfinis. 236
Console Enterprise Manager. 237
Révocation d'un rôle 238
Activation d'un rôle (SET ROLE). 239
Modification d'un rôle (ALTER ROLE). 240
Suppression d'un rôle (DROP ROLE). 241
Vues 241
Création d'une vue (CREATE VIEW) 242
Classification 244
Vues monotables. 244
Vues complexes 249
Autres utilisations de vues. 252
Transmission de droits 256
Modification d'une vue (ALTER VIEW) 256
Suppression d'une vue (DROP VIEW) 256
Synonymes. 257
Création d'un synonyme (CREATE SYNONYM) 257
Transmission de droits 259
Suppression d'un synonyme (DROP SYNONYM) 259
Dictionnaire des données 259
Constitution 260
Classification des vues 260
Démarche à suivre 261
Principales vues 263
Objets d'un schéma. 265
Structure d'une table 265
Recherche des contraintes d'une table 266
Composition des contraintes d'une table 266
Détails des contraintes référentielles 266
Recherche du code source d'un sous-programme 267
Recherche des utilisateurs d'une base de données 268
Rôles reçus 268

Partie II PL/SQL. 273

6 Bases du PL/SQL. 275

Généralités 275
Environnement client-serveur 275
Avantages 276
Structure d'un programme. 276
Portée des objets 277

 © Éditions Eyrolles

Jeu de caractères . . . 278
Identificateurs . . . 278
Commentaires . . . 279
Variables . . . 279
Variables scalaires . . . 280
Affectations . . . 280
Restrictions . . . 281
Variables %TYPE . . . 281
Variables %ROWTYPE . . . 282
Variables RECORD . . . 283
Variables tableaux (type TABLE) . . . 284
Résolution de noms . . . 286
Opérateurs . . . 286
Variables de substitution . . . 287
Variables de session . . . 288
Conventions recommandées . . . 288
Types de données PL/SQL . . . 289
Types prédéfinis . . . 289
Sous-types . . . 289
Conversions de types . . . 291
Nouveautés 11*g* . . . 291
Structures de contrôles . . . 292
Structures conditionnelles . . . 292
Structures répétitives . . . 295
Nouveautés 11*g* . . . 299
Interactions avec la base . . . 300
Extraire des données . . . 300
Manipuler des données . . . 302
Curseurs implicites . . . 304
Paquetage DBMS_OUTPUT . . . 305
Transactions . . . 308
Caractéristiques . . . 308
Début et fin d'une transaction . . . 309
Contrôle des transactions . . . 310
Transactions imbriquées . . . 311

7 Programmation avancée . . . 315

Sous-programmes . . . 315
Généralités . . . 315
Procédures cataloguées . . . 316
Fonctions cataloguées . . . 317
Codage d'un sous-programme PL/SQL . . . 318
Exemples . . . 318
Compilation . . . 321

Appels 321
À propos des paramètres 323
Récursivité. 324
Sous-programmes imbriqués 324
Recompilation d’un sous-programme 326
Destruction d’un sous-programme 326
Paquetages (packages) 326
Généralités 326
Spécification 327
Compilation 328
Implémentation 328
Appel 329
Surcharge 329
Recompilation 329
Destruction d’un paquetage 329
Curseurs 330
Comment retourner une table ? 330
Généralités 331
Instructions 331
Parcours d’un curseur 332
Utilisation de structures (%ROWTYPE) 333
Boucle FOR (gestion semi-automatique) 334
Utilisation de tableaux (type TABLE). 335
Utilisation de LIMIT et BULK COLLECT 336
Paramètres d’un curseur 337
Accès concurrents (FOR UPDATE et CURRENT OF) 338
Variables curseurs (REF CURSOR) 339
Fonctions table pipelined. 341
Exceptions 343
Généralités 343
Exception interne prédéfinie 345
Exception utilisateur 349
Utilisation du curseur implicite 351
Exception interne non prédéfinie 352
Propagation d’une exception 353
Procédure RAISE_APPLICATION ERROR 355
Déclencheurs 356
À quoi sert un déclencheur ?. 356
Généralités 357
Mécanisme général 357
Syntaxe 358
Déclencheurs LMD 359
Transactions autonomes 371
Déclencheurs LDD 372

© Éditions Eyrolles

Déclencheurs d'instances 372
Appels de sous-programmes 373
Gestion des déclencheurs 374
Ordre d'exécution 375
Tables mutantes 375
Nouveautés 11*g* 376
SQL dynamique 380
Classification 381
Utilisation de EXECUTE IMMEDIATE 381
Utilisation d'une variable curseur 383

Partie III SQL avancé 387

8 Le précompilateur Pro*C/C++ 389
Généralités 389
Ordres SQL intégrés 389
Variables 390
Variable indicatrice 391
Cas du VARCHAR 392
Zone de communication (SQLCA) 392
Connexion à une base 393
Gestion des exceptions 393
Transactions 394
Extraction d'un enregistrement 394
Mises à jour 396
Utilisation de curseurs 396
Variables scalaires 396
Variables tableaux 397
Utilisation de Microsoft Visual C++ 399

9 L'interface JDBC 401
Généralités 401
Classification des pilotes (drivers) 401
Les paquetages 402
Structure d'un programme 404
Variables d'environnement 404
Test de votre configuration 405
Connexion à une base 406
Base Access 406
Base Oracle 407
Déconnexion 409
Interface Connection 409

États d'une connexion 410
Interfaces disponibles 410
Méthodes génériques pour les paramètres. . . . 411
États simples (interface Statement) 411
Méthodes à utiliser 412
Correspondances de types. . . . 413
Interactions avec la base 414
Suppression de données 414
Ajout d'enregistrements. . . . 415
Modification d'enregistrements 415
Extraction de données 415
Curseurs statiques 416
Curseurs navigables 417
Curseurs modifiables 421
Suppressions. . . . 423
Modifications 424
Insertions. . . . 424
Restrictions 425
Interface ResultSetMetaData 426
Interface DatabaseMetaData. . . . 427
Instructions paramétrées (PreparedStatement) 428
Extraction de données (executeQuery). . . . 429
Mises à jour (executeUpdate) 430
Instruction LDD (execute) 430
Appels de sous-programmes 431
Appel d'une fonction 432
Appel d'une procédure 432
Transactions 433
Points de validation 434
Traitement des exceptions 435
Affichage des erreurs 436
Traitement des erreurs 436

10 L'approche SQLJ 439

Généralités 439
Blocs SQLJ 439
Précompilation. . . . 440
Configurations 440
Affectations (SET) 442
Intégration de SQL. . . . 442
Instructions du LDD. . . . 442
Instructions du LMD 443
Requêtes. . . . 443
À propos des itérateurs 447

 © Éditions Eyrolles

Transactions 450
Intégration de blocs PL/SQL 450
Points de validation 451
Appels de sous-programmes 452
Résultats scalaires 452
Résultats complexes 454
Traitement des exceptions 455
Définition des données 455
Manipulation des données 456
Interrogation des données 456
Contextes de connexion 457
SQL dynamique 459
Expression 459
Restrictions 460

11 Procédures stockées et externes 463
Procédures stockées Java 463
Stockage d'une procédure 464
Interactions avec la base 469
Déclencheurs 473
Procédures externes Java 474
Compilation de la classe 475
Création d'une librairie 475
Publication d'une procédure externe 475
Appel d'une procédure externe 476

12 Oracle et le Web 477
Configuration minimale d'Apache 477
PL/SQL Web Toolkit 479
Détail d'une URL 479
Paquetages HTP et HTF 480
Pose d'hyperliens 484
Formulaires 486
Tables 487
Listes 487
PL/SQL Server Pages 488
Généralités 488
Balises 489
Chargement d'un programme PSP 490
Appel 490
Interaction avec la base 490
Intégration de PHP 492
Configuration adoptée 492
API de PHP pour Oracle 495
Interactions avec la base 497

© Éditions Eyrolles

13 Oracle XML DB 511
Généralités 511
Comment disposer de XML DB ? 511
Le type de données XMLType 512
Modes de stockage 513
Stockages XMLType 514
Création d'une table 515
Répertoire de travail 517
Grammaire XML Schema 517
Annotation de la grammaire 518
Enregistrement de la grammaire 520
Stockage structuré (object-relational) 522
Stockage non structuré (CLOB) 536
Stockage non structuré (binary XML) 537
Autres fonctionnalités 541
Génération de contenus 541
Vues XMLType 542
Génération de grammaires annotées 545
Dictionnaire des données 547
XML DB Repository 549
Interfaces 549
Configuration 549
Paquetage XML_XDB 552
Accès par SQL 552
14 Optimisations 561
Cadre général 561
Les acteurs 562
Contexte et objectifs 562
Causes possibles 563
Présentation du jeu d'exemple 563
L'offre d'Oracle 11*g* 564
Les optimiseurs 565
L'estimateur 567
Traitement d'une instruction 568
Configuration de l'optimiseur (les hints) 570
Les statistiques destinées à l'optimiseur 571
Les histogrammes 571
Collecte 573
Outils de mesures de performances 576
Visualisation des plans d'exécution 577
L'outil tkprof 583
Utilisation de l'événement 10046 588
Paquetage DBMS_APPLICATION_INFO 589
Les vues dynamiques du dictionnaire 593

 © Éditions Eyrolles

L'utilitaire runstats de Tom Kyte 597
Bilan. 599
Organisation des données 600
Des contraintes au plus près des données 600
Indexation 601
Jointures 614
Variables de lien 622
Comment réaliser des fetchs multilignes ? 624
Clusters 625
Tables organisées en index 637
Comparatif 638
Les débordements 639
Création d'une IOT 639
Comparaison avec une table en heap 640
Limites 640
Partitionnement 640
La clé de partition 641
Partitions par intervalle 642
Intervalles automatiques 643
Partitions par hachage 644
Partitions par liste 645
Partitions par référence 646
Sous-partitions 647
Index partitionnés 648
Index partitionné local 649
Index partitionné global 650
Opérations sur les partitions et index 651
Partitionnement des tables IOT 651
Vues matérialisées 652
Réécriture de requêtes 653
Le rafraîchissement 654
Exemples 654
Dénormalisation 656
Colonnes calculées 657
Duplication de colonnes 657
Ajout de clés étrangères 658
Exemple de stratégie 658
Derniers conseils 659
Requêtes inefficaces 659
Les 10 commandements de F. Brouard 660

Annexe : Bibliographie et webographie. 663

Index. 665

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.

La troisième édition ajoutait 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.

Cette quatrième édition introduit quelques nouveautés SQL et PL/SQL (pivots, transpositions, requêtes pipe line, requêtes CTE et récursivité). Un nouveau chapitre est dédié à l’optimisation des requêtes et du schéma relationnel. Plusieurs aspects sont étudiés : le fonctionnement de l’optimiseur, l’utilisation de statistiques et quelques outils de mesure de performances. Différents mécanismes d’optimisation sont également présentés : contraintes, index, *clusters*, partitionnement, tables organisées en index, vues matérialisées et principes de dénormalisation.

© Éditions Eyrolles

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 jugé intéressant de mentionner ici.

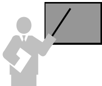
© Éditions Eyrolles

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).

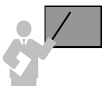


La contrainte référentielle concerne toujours deux tables – une table « père » aussi dite « maître » (*parent/referenced*) et une table « fils » (*child/dependent*) – possédant une ou plusieurs colonnes en commun. Pour la table « père », ces colonnes composent la clé primaire (ou candidate avec un index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

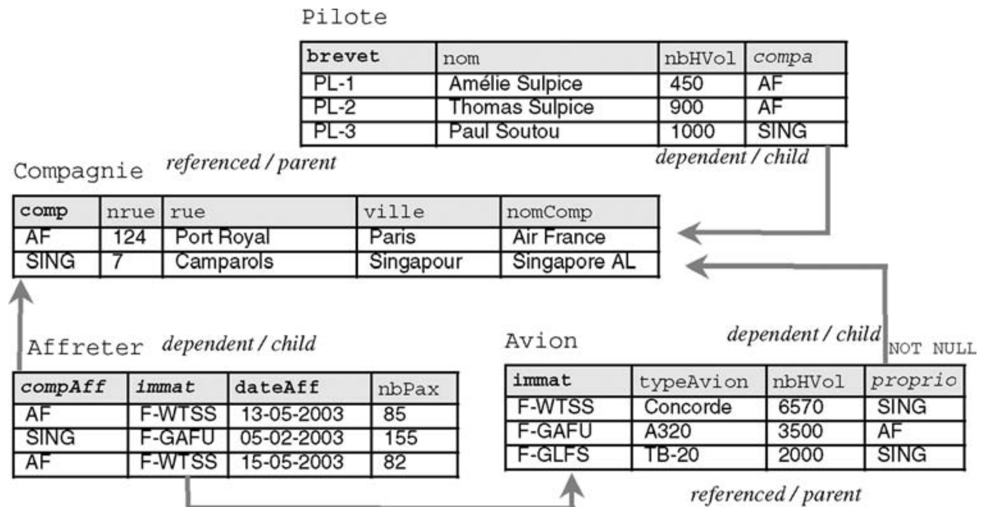


Deux types de problèmes sont automatiquement résolus par Oracle pour assurer l'intégrité référentielle :

- 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

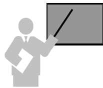
contrainte après la définition de la colonne. Cette écriture est préférable car elle convient aussi aux clés composées de plusieurs colonnes de par sa lisibilité.

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.

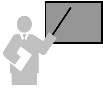
```
CREATE TABLE Compagnie …

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 sui vant 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 f aut insérer d'abord les compagnies, puis les pilotes (ou les a vions), 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 alternati ves 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 alternati ve est celle par déf aut. Dans notre e xemple, 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.

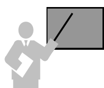
Conventions d'écriture

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`).

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));
```

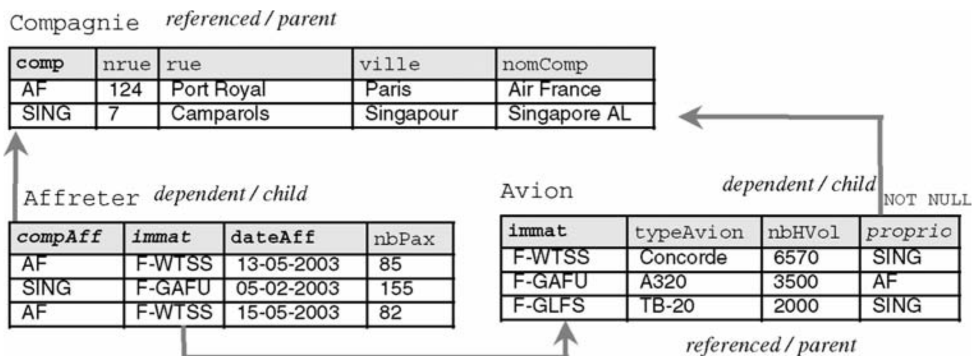
Clé primaire

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*



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

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*

Compagnie *referenced / parent*

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

Affreter *dependent / child*

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

Avion

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

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

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

Tableau 3-3 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 :

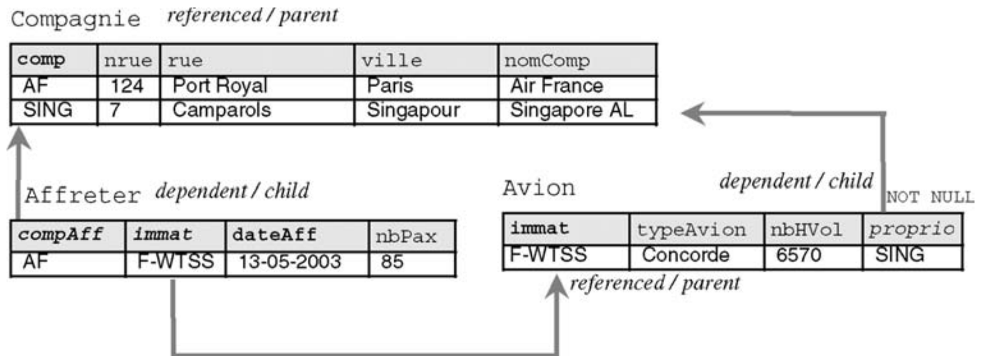
```
ALTER TABLE [schéma.]nomTable
   DISABLE [ VALIDATE | NOVALIDATE ] CONSTRAINT nomContrainte
   [CASCADE] [ { KEEP | DROP } INDEX ] ;
```

- `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.

 © Éditions Eyrolles

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 :

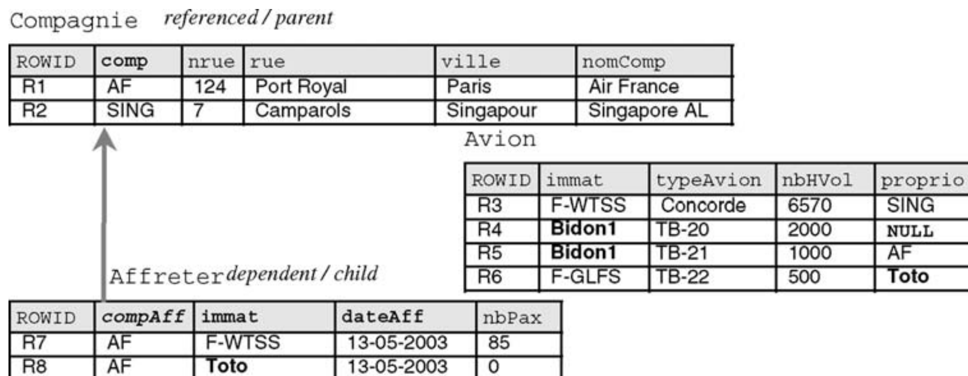
```
INSERT INTO Affreter VALUES ('AF', 'Toto', '13-05-2003', 0);
```

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 réacti ver les contraintes sans modifier les v aleurs 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'inde x 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'inde x) 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 :

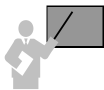
- La première, de type `ROWID`, contiendra les adresses des enregistrements ne respectant pas la contrainte ;
- la deuxième colonne de type VARCHAR2(30) contiendra le nom du propriétaire de la table ;
- la troisième colonne de type VARCHAR2(30) contiendra le nom de la table ;
- la quatrième, de type 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* R4 et R6) :

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

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

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

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

 © Éditions Eyrolles

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

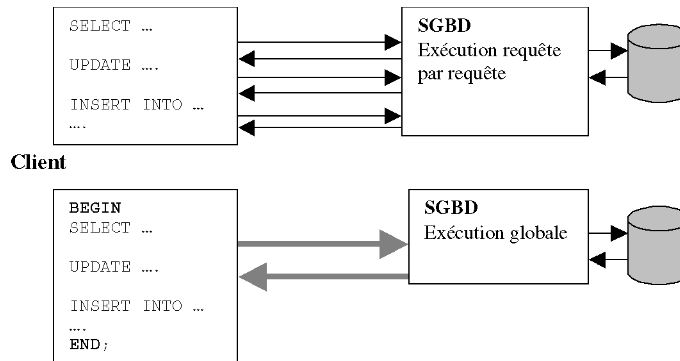
Les structures de contrôle habituelles d'un langage (`IF`, `WHILE`…) ne font pas partie intégrante de la norme SQL. Elles apparaissent dans une sous-partie optionnelle de la norme (ISO/IEC 9075-5:1996. *Flow-control statements*). Oracle les prend en compte dans PL/SQL. Nombre de concepts de PL/SQL proviennent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

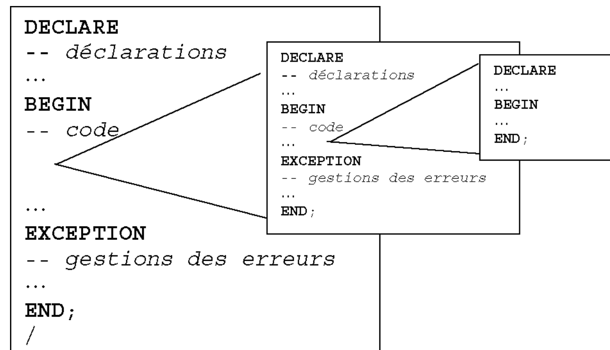
Un programme PL/SQL qui n'est pas nommé (aussi appelé bloc) est composé de trois sections comme le montre la figure suivante :

- `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*

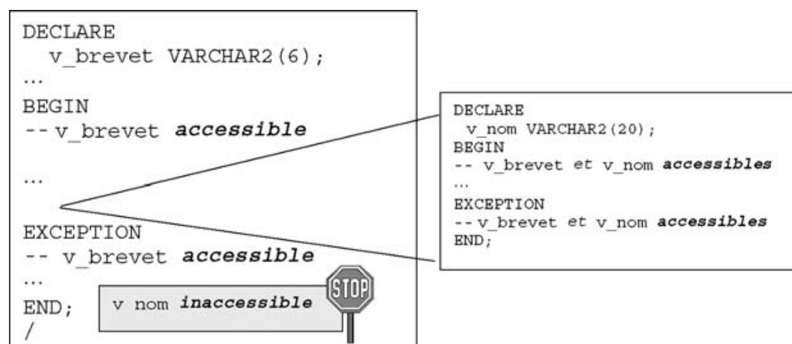


Portée des objets

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

Peu lisible	C'est mieux

`IF x>y THEN max:=x;ELSE max:=y;END IF;`	`IF x > y THEN`
	`    max := x;`
	`  ELSE`
	`    max := y;`
	`END IF;`

Identificateurs

Avant de parler des différents types de variables PL/SQL, décrivons comment il est possible de nommer des objets PL/SQL (variables, curseurs, exceptions, etc.).

Un identificateur commence par une lettre suivie (optionnel) de symboles (lettres, chiffres, `$`, `_`, `#`). Un identificateur peut contenir jusqu'à trente caractères. Les autres signes pourtant connus du langage sont interdits comme le montre le tableau suivant :

Tableau 6-2 Identificateurs

Autorisés	Interdits
`X`	`moi&toi` (symbole &)
`t2`	`debit-credit` (symbole -)
`téléphone#`	`on/off` (symbole /)
`code_brevet`	`code brevet` (symbole espace)
`codeBrevet`	
`oracle$nombre`	

© 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;`
`v_brevet Pilote.brevet%TYPE;`
`BEGIN`
	La structure `rty_pilote` est composée de toutes les colonnes de la table `Pilote`.

`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 xe 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 xe 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

Exceptions

Afin d'éviter qu'un programme s'arrête à la première erreur (requête ne retournant aucune ligne, valeur incorrecte à écrire dans la base, conflit de clés primaires, division par zéro, etc.), il est indispensable de prévoir tous les cas potentiels d'erreurs et d'associer à chacun de ces cas la programmation d'une exception PL/SQL. Dans le vocabulaire des programmeurs on dit qu'on *garde la main* pendant l'exécution du programme. Le mécanisme des exceptions (*handling errors*) est largement utilisé par tous les programmeurs car il est prépondérant dans la mise en œuvre des transactions.

Les exceptions peuvent se programmer dans un bloc PL/SQL, un sous-programme (fonction ou procédure cataloguée), dans un paquetage ou un déclencheur.

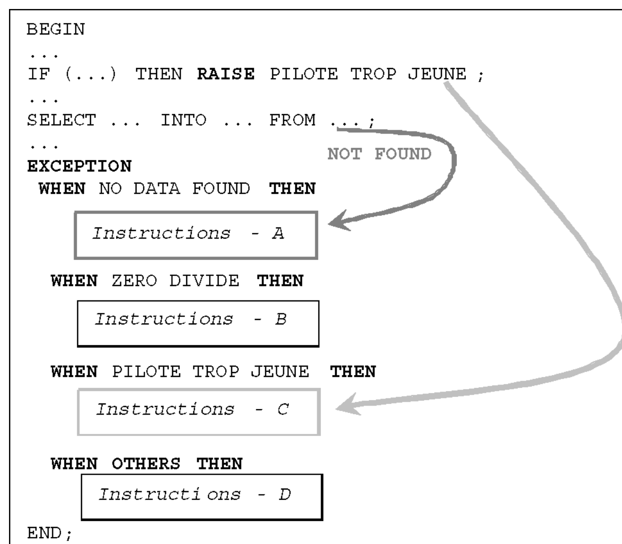
Généralités

Une exception PL/SQL correspond à une condition d'erreur et est associée à un identificateur. Une exception est détectée (aussi dite « levée ») au cours de l'exécution d'une partie de programme (entre un `BEGIN` et un `END`). Une fois levée, l'exception termine le corps principal des instructions et renvoie au bloc `EXCEPTION` du programme en question.

La figure suivante illustre les deux mécanismes qui peuvent déclencher une exception :

- Une erreur Oracle se produit, l'exception associée est déclenchée automatiquement (exemple du `SELECT` ne ramenant aucune ligne, ce qui déclenche l'exception `ORA-01403` d'identificateur `NO_DATA_FOUND`).

Figure 7-6 *Principe général des exceptions*



- Le programmeur désire dérouter volontairement (par l'intermédiaire de l'instruction RAISE) son programme dans le bloc des exceptions sous certaines conditions. L'exception est ici manuellement déclenchée et peut appartenir à l'utilisateur (ici la condition PILOTE_TROP_JEUNE) ou être prédéfinie au niveau d'Oracle (division par zéro d'identificateur ZERO_DIVIDE qui sera automatiquement déclenchée).

Si aucune erreur ne se produit, le bloc est ignoré et le traitement se termine (ou retourne à son appelant s'il s'agit d'un sous-programme).

La syntaxe générale d'un bloc d'exceptions est la suivante. Il est possible de grouper plusieurs exceptions pour programmer le même traitement. La dernière entrée (OTHERS) doit être éventuellement toujours placée en fin du bloc d'erreurs.

```
EXCEPTION
   WHEN exception1 [OR exception2 …] THEN
        instructions;
  [WHEN exception3 [OR exception4 …] THEN
        instructions;  ]
  [WHEN OTHERS THEN
        instructions; ]
```

Si une anomalie se produit, le bloc EXCEPTION s'exécute.

- Si le programme prend en compte l'erreur dans une entrée WHEN…, les instructions de cette entrée sont exécutées et le programme se termine.
- Si l'exception n'est pas prise en compte dans le bloc EXCEPTION :
  – il existe une section OTHERS où des instructions s'exécutent ;
  – il n'existe pas une section OTHERS et l'exception sera propagée au programme appelant (une section traite de la propagation des exceptions).

Études 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.

© Éditions Eyrolles

Exception interne prédéfinie

Les exceptions prédéfinies sont celles qui se produisent le plus souvent. Oracle affecte un nom de manière à les traiter plus facilement dans le bloc EXCEPTION. Le tableau suivant les décrit :

Tableau 7-22 Exceptions prédéfinies

Nom de l'exception	Numéro	Commentaires
ACCESS_INTO_NULL	ORA-06530	Affectation d'une valeur à un objet non initialisé.
CASE_NOT_FOUND	ORA-06592	Aucun des choix de la structure CASE sans ELSE n'est effectué.
COLLECTION_IS_NULL	ORA-06531	Utilisation d'une méthode autre que EXISTS sur une collection (*nested table* ou *varray*) non initialisée.
CURSOR_ALREADY_OPEN	ORA-06511	Ouverture d'un curseur déjà ouvert.
DUP_VAL_ON_INDEX	ORA-00001	Insertion d'une ligne en doublon (clé primaire).
INVALID_CURSOR	ORA-01001	Ouverture interdite sur un curseur.
INVALID_NUMBER	ORA-01722	Échec d'une conversion d'une chaîne de caractères en NUMBER.
LOGIN_DENIED	ORA-01017	Connexion incorrecte.
NO_DATA_FOUND	ORA-01403	Requête ne retournant aucun résultat.
NOT_LOGGED_ON	ORA-01012	Connexion inexistante.
PROGRAM_ERROR	ORA-06501	Problème PL/SQL interne (invitation au contact du support…).
ROWTYPE_MISMATCH	ORA-06504	Incompatibilité de types entre une variable externe et une variable PL/SQL.
SELF_IS_NULL	ORA-30625	Appel d'une méthode d'un type sur un objet NULL (extension objet).
STORAGE_ERROR	ORA-06500	Dépassement de capacité mémoire.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Référence à un indice incorrect d'une collection (*nested table* ou *varray*) ou variables de type TABLE.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	
SYS_INVALID_ROWID	ORA-01410	Échec d'une conversion d'une chaîne de caractères en ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Dépassement du délai alloué à une ressource.
TOO_MANY_ROWS	ORA-01422	Requête retournant plusieurs lignes.
VALUE_ERROR	ORA-06502	Erreur arithmétique (conversion, troncature, taille) d'un NUMBER.
ZERO_DIVIDE	ORA-01476	Division par zéro.

Le code d'erreur (SQLCODE) qui peut être récupéré par un programme d'application (Java par exemple sous JDBC), est inclus dans le numéro interne de l'erreur (pour la deuxième exception, il s'agit de -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 7-23 Deux exceptions traitées

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException1 (p_comp IN VARCHAR2) IS`
`  var1 Pilote.nom%TYPE;`
`BEGIN`
`  SELECT nom INTO var1 FROM Pilote`
`           WHERE comp = p_comp;`
`  DBMS_OUTPUT.PUT_LINE('Le pilote de la compagnie '`
`              || p_comp || ' est ' || var1);`
	Requête déclenchant potentiellement deux exceptions prévues.

`EXCEPTION`
` WHEN NO_DATA_FOUND THEN`
`      DBMS_OUTPUT.PUT_LINE('La compagnie ' ||`
`                 p_comp || ' n''a aucun pilote!');`
	Aucun résultat renvoyé.

` WHEN TOO_MANY_ROWS THEN`
`      DBMS_OUTPUT.PUT_LINE('La compagnie ' ||`
`                 p_comp || ' a plusieurs pilotes!');`
`END;`
	Plusieurs résultats renvoyés.

La trace de l'exécution de cette procédure est la suivante :

```
SQL> EXECUTE procException1('AF');
```


© 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 se produit, en l'absence de la directive OTHERS dans le bloc d'exceptions, le programme se termine anormalement en renvoyant l'erreur en question. Dans notre exemple, seule une erreur interne pourrait éventuellement se produire (PROGRAM_ERROR, STORAGE_ERROR, TIMEOUT_ON_RESOURCE).

Même erreur sur différentes instructions

Le tableau 7-21 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 7-24 Une exception traitée pour deux instructions

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException2`
`(p_brevet IN VARCHAR2, p_heures IN NUMBER) IS`
`var1 Pilote.nom%TYPE;`
`requete NUMBER := 1;`
`BEGIN`
`SELECT nom INTO var1 FROM Pilote`
`WHERE brevet = p_brevet;`
`DBMS_OUTPUT.PUT_LINE('Le pilote de ' ||`
`p_brevet || ' est ' || var1);`
	Requêtes déclenchant potentiellement une exception prévue.

`requete := 2;`
`SELECT nom INTO var1 FROM Pilote`
`WHERE nbHVol = p_heures;`
`DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' ||`
`p_heures || ' heures est ' || var1);`	

`EXCEPTION`
`WHEN` `NO_DATA_FOUND` **`THEN`**
`IF requete = 1 THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote de brevet : '`
`|| p_brevet);`
`ELSE`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote ayant ce`
`nombre d''heures de vol : ' || p_heures);`
`END IF;`
	Aucun résultat.
	Traitement pour savoir quelle requête a déclenché l'exception.

`WHEN` `OTHERS` **`THEN`**
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM` `|| ' (' ||` **`SQLCODE`** `|| ')');`
`END;`
	Autre erreur.

Imbrication de blocs d'erreurs

Le tableau suivant décrit une procédure qui inclut un bloc d'exceptions imbriqué au code principal. Ce mécanisme permet de poursuivre l'exécution après qu'Oracle a levé une exception. Dans cette procédure, les deux requêtes sont évaluées indépendamment du résultat retourné par chacune d'elles.

L'exécution suivante de cette procédure déclenche les deux exceptions. Le message d'erreur est contrôlé par le dernier cas d'exception, il ne s'agit pas d'une interruption anormale du programme.

```
SQL> EXECUTE procException3('PL-0', 2450);
Pas de pilote de brevet : PL-0
Erreur d'Oracle ORA-01422: l'extraction exacte ramène plus que le
nombre de lignes demandé (-1422)
```

 © Éditions Eyrolles

Tableau 7-25 Bloc d'exceptions imbriqué

Code PL/SQL	Commentaires
`CREATE PROCEDURE procException3 (p_brevet IN VARCHAR2, p_heures IN NUMBER) IS var1 Pilote.nom%TYPE; BEGIN`	
`BEGIN SELECT nom INTO var1 FROM Pilote WHERE brevet = p_brevet; DBMS_OUTPUT.PUT_LINE('Le pilote de ' || p_brevet || ' est ' || var1);`	Bloc imbriqué.
`EXCEPTION WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE('Pas de pilote de brevet : ' || p_brevet); WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM || ' (' || SQLCODE || ')'); END;`	Gestion des exceptions de la première requête.
`SELECT nom INTO var1 FROM Pilote WHERE nbHVol = p_heures ; DBMS_OUTPUT.PUT_LINE('Le pilote ayant ' || p_heures || ' heures est ' || var1);`	Suite du traitement.
`EXCEPTION WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE('Pas de pilote ayant ce nombre d''heures de vol : ' || p_heures); WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM || ' (' || SQLCODE || ')'); END;`	Gestion des exceptions de la deuxième requête.

Exception utilisateur

Il est possible de définir ses propres exceptions. Cela pour bénéficier des blocs de traitements d'erreurs et aborder une erreur applicative comme une erreur renvoyée par la base. Cela améliore et facilite la maintenance et l'évolution des programmes car les erreurs applicatives peuvent très facilement être propagées aux programmes appelants.

Déclaration

La déclaration du nom de l'exception doit se trouver dans la section déclarative du sous-programme.

```
nomException EXCEPTION;
```

© Éditions Eyrolles

Déclenchement

Une exception utilisateur ne sera pas levée de la même manière qu'une exception interne. Le programme doit explicitement dérouter le traitement vers le bloc des exceptions par la directive RAISE. L'instruction RAISE permet également de déclencher des exceptions prédéfinies.

Dans notre exemple, programmons les deux exceptions suivantes :

- erreur_piloteTropJeune qui va interdire l'insertion des pilotes ayant moins de 200 heures de vol ;
- erreur_piloteTropExpérimenté qui va interdire l'insertion des pilotes ayant plus de 20 000 heures de vol.

Le tableau suivant décrit cette procédure qui intercepte ces deux erreurs applicatives :

Tableau 7-26 Exceptions utilisateur

Code PL/SQL	Commentaires
`CREATE PROCEDURE saisiePilote (p_brevet IN VARCHAR2,p_nom IN VARCHAR2, p_nbHVol IN NUMBER, p_comp IN VARCHAR2) IS erreur_piloteTropJeune EXCEPTION; erreur_piloteTropExpérimenté EXCEPTION;`	Déclaration de l'exception.
`BEGIN INSERT INTO Pilote (brevet,nom,nbHVol,comp) VALUES (p_brevet,p_nom,p_nbHVol,p_comp); IF p_nbHVol < 200 THEN RAISE erreur_piloteTropJeune; END IF; IF p_nbHVol > 20000 THEN RAISE erreur_piloteTropExpérimenté; END IF; COMMIT;`	Corps du traitement (validation).
`EXCEPTION WHEN erreur_piloteTropJeune THEN ROLLBACK; DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote manque d''expérience'); WHEN erreur_piloteTropExpérimenté THEN ROLLBACK; DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote a trop d''expérience'); WHEN OTHERS THEN ROLLBACK; DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM || '(' || SQLCODE || ')'); END;`	Gestion de l'exception. Gestion des autres exceptions.

La trace de l'exécution de cette procédure où l'on passe des valeurs en paramètres qui déclenchent les deux exceptions est la suivante :

```
SQL> EXECUTE saisiePilote('PL-9','Tuffery Michel', 199, 'AF');
Désolé, le pilote manque d'expérience
Procédure PL/SQL terminée avec succès.
SQL> EXECUTE saisiePilote('PL-9','Tuffery Michel', 20001, 'AF');
Désolé, le pilote a trop d'expérience
Procédure PL/SQL terminée avec succès.
```

Utilisation du curseur implicite

Étudiés dans le chapitre 6, les curseurs implicites permettent ici de pallier le fait qu'Oracle ne lève pas l'exception `NO_DATA_FOUND` pour les instructions `UPDATE` et `DELETE`. Ce qui est en théorie valable (aucune action sur la base peut ne pas être considérée comme une erreur), en pratique il est utile de connaître le code retour de l'instruction de mise à jour.

Considérons à nouv eau la procédure `détruitCompagnie` en prenant en compte l'erreur applicative `erreur_compagnieInexistante` qui intercepte une suppression non réalisée. Le test du curseur implicite de cette instruction déclenche l'exception utilisateur associée.

Tableau 7-27 Utilisation du curseur implicite

Code PL/SQL	Commentaires

`CREATE OR REPLACE PROCEDURE détruitCompagnie (p_comp IN VARCHAR2) IS`
`erreur_ilResteUnPilote EXCEPTION;`
`PRAGMA EXCEPTION_INIT(erreur_ilResteUnPilote , -2292);`
`erreur_compagnieInexistante EXCEPTION;`
	Déclaration des exceptions.

`BEGIN`
`DELETE FROM Compagnie WHERE comp = p_comp;`
`IF SQL%NOTFOUND THEN`
`RAISE erreur_compagnieInexistante;`
`END IF;`
`COMMIT;`
`DBMS_OUTPUT.PUT_LINE('Compagnie '||p_comp|| ' détruite.');`
	Corps du traitement (validation).

`EXCEPTION`
`WHEN erreur_ilResteUnPilote THEN`
`DBMS_OUTPUT.PUT_LINE ('Désolé, il reste encore un pilote à la compagnie ' || p_comp);`
`WHEN erreur_compagnieInexistante THEN`
`DBMS_OUTPUT.PUT_LINE ('La compagnie ' || p_comp || ' n''existe pas dans la base!');`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM || '(' || SQLCODE || ')');`
`END;`
	Gestion des exceptions.
	
	Gestion des autres exceptions.

L'exécution de cette procédure où l'on passe un code compagnie ine xistant fait maintenant dérouler la section des exceptions.

```
SQL> EXECUTE détruitCompagnie('rien');
La compagnie rien n'existe pas dans la base!
```

Exception interne non prédéfinie

Pour intercepter une erreur Oracle qui n'a pas été prédéfinie 'pour laquelle Oracle n'a pas associé de nom), et être ainsi plus précis qu'avec la clause `OTHERS`, il faut utiliser la directive `PRAGMA EXCEPTION_INIT`. Celle-ci indique au compilateur d'associer un nom d'exception, que vous aurez choisi, à un code d'erreur Oracle existant. La directive `PRAGMA` (appelée aussi pseudo-instruction) est un mot-clé signifiant que l'instruction est destinée au compilateur (elle n'est pas traitée au moment de l'exécution).

Déclaration

Deux commandes sont nécessaires dans la section déclarati ve 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 le vée de la même manière qu'une e xception 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 7-28 Exception interne non prédéfinie

Code PL/SQL	Commentaires

`CREATE PROCEDURE détruitCompagnie(p_comp IN VARCHAR2) IS`
`  erreur_ilResteUnPilote EXCEPTION;`
`  PRAGMA EXCEPTION_INIT(erreur_ilResteUnPilote , -2292);`
	Déclaration de l'exception.

`BEGIN`
`  DELETE FROM Compagnie WHERE comp = p_comp;`
`  COMMIT;`
`  DBMS_OUTPUT.PUT_LINE ('Compagnie ' || p_comp ||`
`            ' détruite.');`
	Corps du traitement (validation).

`EXCEPTION`
`  WHEN erreur_ilResteUnPilote THEN`
`    DBMS_OUTPUT.PUT_LINE ('Désolé, il reste encore un`
`           pilote à la compagnie ' || p_comp);`
	Gestion de l'exception.

`  WHEN OTHERS THEN`
`    DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM`
`            || '(' || SQLCODE || ')');`
`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*

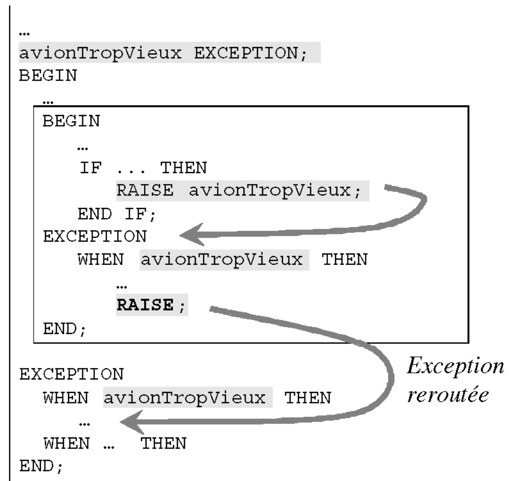
BEGIN
…
BEGIN
…
RAISE... *ou*
Instruction SQL
...
EXCEPTION
WHEN … THEN
…
WHEN … THEN
END ;
1. Déclenchement
2. Recherche de l'exception
… *3. Exception trouvée*
…
…
EXCEPTION
WHEN … THEN
…
WHEN … THEN
END ;
3bis. Exception pas trouvée
4bis. Recherche de l'exception
… *4. Va en séquence*

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

Exceptions reroutées (reraise)

Il est, dans certains cas, intéressant d'exécuter plusieurs blocs d'erreurs pour la même exception. On déclenche plusieurs fois l'exception (*exception reraised*). Le principe consiste à utiliser la directive `RAISE` sans spécifier le nom de l'exception à traiter de nouveau (voir la figure suivante dans laquelle l'exception `avionTropVieux` est reroutée). Si l'exception ne peut être traitée dans le bloc englobant, alors elle est propagée à l'environnement appelant ou englobant (voir section précédente).

Figure 7-9 Exception reroutée



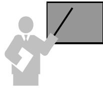
Procédure RAISE_APPLICATION_ERROR

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

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

- *numéroErreur* : valeur définie 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 :



L'opérateur `IS NULL` permet de tester une expression avec la valeur `NULL`. Toute expression arithmétique contenant une valeur nulle est évaluée à `NULL`.

Le tableau suivant illustre quelques utilisations possibles d'opérateurs logiques :

Tableau 6-13 Utilisation d'opérateurs

Code PL/SQL	Commentaires

DECLARE
 v_compteur NUMBER(3) DEFAULT 0;
 v_booléen BOOLEAN;
 v_nombre NUMBER(3);
BEGIN	

v_compteur := v_compteur+1;	Incrémentation, opérateur +
v_booléen := (v_compteur = v_nombre);	v_booléen reçoit NULL du fait de la condition.
v_booléen := (v_nombre IS NULL);	v_booléen reçoit TRUE car la condition est vraie.

Variables de substitution

Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*Plus. Ces variables sont dites de substitution. On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable du symbole « & » (avec ou sans guillemets simples suivant qu'il s'agit d'un nombre ou pas).

Le tableau suivant illustre un exemple de deux variables de substitution. La directive `ACCEPT` permet la saisie au clavier de variables dans l'interface SQL*Plus. Elle doit être utilisée conjointement à la directive `PROMPT` toutes deux placées en amont d'un bloc PL/SQL qui devra être exécuté via la commande `START`. Dans cet exemple on extrait le nom et le nombre d'heures de vol d'un pilote. Son numéro de brevet et la durée du vol sont lus au clavier et la durée est ajoutée au nombre d'heures de vol du pilote. Il est à noter qu'il ne faut pas déclarer des variables de substitution.

Tableau 6-14 Variables de substitution

Code PL/SQL	Sous SQL*Plus

ACCEPT s_brevet PROMPT 'Entrer code Brevet : '
ACCEPT s_duréeVol PROMPT 'Entrer durée du vol : '
DECLARE
 v_nom Pilote.nom%TYPE;
 v_nbHVol Pilote.nbHVol%TYPE;
BEGIN
 SELECT nom, nbHVol INTO v_nom, v_nbHVol
 FROM Pilote WHERE brevet = '&s_brevet';
 v_nbHVol := v_nbHVol + &s_duréeVol;
 DBMS_OUTPUT.PUT_LINE
 ('Total heures vol : ' || v_nbHVol ||
 ' de ' || v_nom);
END;
	Entrer code Brevet : PL-2
	Entrer durée du vol : 27
	Total heures vol : 927 de
	Didier Linxe
	
	Procédure PL/SQL terminée
	avec succès.

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

Document XML

```
<?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>
```

Grammaire *XML Schema* (compagnies.xsd)

```
<?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>
```

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`.

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

<table>
<tr><th>Code SQL</th><th>Résultats</th></tr>
<tr><td><pre>SELECT OBJECT_NAME
   FROM USER_OBJECTS
   WHERE OBJECT_TYPE='TYPE';</pre></td><td><pre>OBJECT_NAME

COMPAGNIE_TYPE
PILOTE_VRY
PILOTE_TYPE
PILOTES_TYPE</pre></td></tr>
<tr><td><pre>DESCRIBE COMPAGNIE_TYPE;</pre></td><td><pre>COMPAGNIE_TYPE est NOT FINAL
Nom   NULL ?   Type
----------------- -------- ------------------
SYS_XDBPD$ XDB.XDB$RAW_LIST_T
COMP   VARCHAR2(6 CHAR)
PILOTES   PILOTES_TYPE
NOMCOMP   VARCHAR2(40 CHAR)</pre></td></tr>
<tr><td><pre>DESCRIBE PILOTES_TYPE</pre></td><td><pre>PILOTES_TYPE est NOT FINAL
Nom   NULL ?   Type
----------------- -------- ------------------
SYS_XDBPD$ XDB.XDB$RAW_LIST_T
PILOTE   PILOTE_VRY</pre></td></tr>
<tr><td><pre>DESCRIBE PILOTE_VRY;</pre></td><td><pre>PILOTE_VRY VARRAY(2147483647) OF PILOTE_TYPE
PILOTE_TYPE est NOT FINAL
Nom   NULL ?   Type
----------------- -------- ------------------
SYS_XDBPD$ XDB.XDB$RAW_LIST_T
BREVET   VARCHAR2(4 CHAR)
NOM   VARCHAR2(30 CHAR)
SALAIRE   NUMBER(8,2)</pre></td></tr>
</table>

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 compagnie.xml.</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.

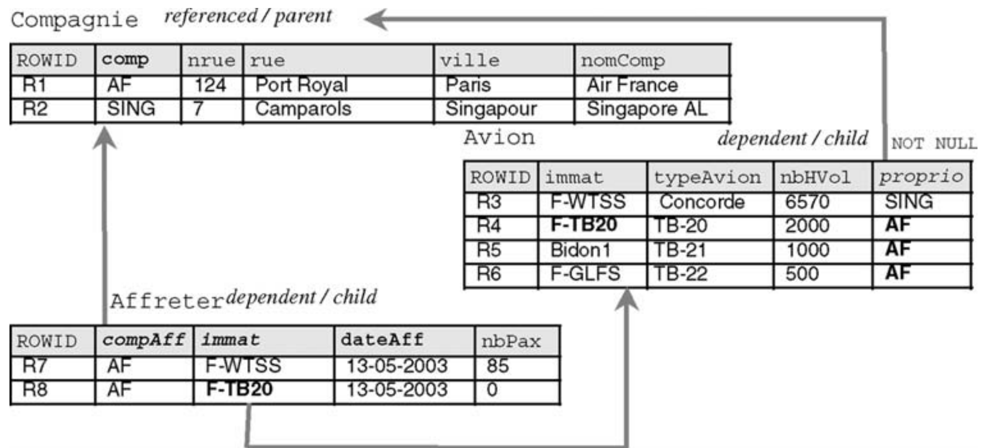
Tableau 13-7 Insertion de contenu partiellement valide

<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 pilotes 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 pilotes.</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>

```
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 *referenced / parent*

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

Avion *dependent / child* — NOT NULL

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 *dependent / child*

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

Directives DEFERRABLE et INITIALLY

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

© Éditions Eyrolles

Définissez NOT NULL sur le plus de colonnes possibles pour renseigner l'optimiseur.

Préférez toujours la seconde écriture (*in line constraint*), pour que l'optimiseur puisse intégrer cette information, alors qu'il ignorera la contrainte déclarée avec CHECK.

Les colonnes UNIQUE

Pour toute contrainte UNIQUE, un index (unique) est créé. Une contrainte UNIQUE diffère d'une contrainte PRIMARY KEY par le fait que les valeurs NULL sont autorisées ; elle n'a donc pas vocation à identifier toute ligne.

Définissez UNIQUE sur les colonnes potentiellement uniques de sorte que l'optimiseur puisse bénéficier d'un index supplémentaire (la désactivation d'une contrainte UNIQUE provoque la suppression de l'index).

Le tableau suivant présente la déclaration d'une contrainte UNIQUE (création implicite d'un index de nom un_nom_prenom_tel) et sa désactivation (suppression implicite d'un index). Du fait qu'il existe des homonymes au sein des adhérents, la contrainte UNIQUE minimale à mettre en œuvre est composée du nom, prénom et numéro de téléphone.

L'index généré sera bénéfique pour les extractions dont un prédicat est basé sur le nom, le prénom et le numéro de téléphone, et sur un accès aux trois colonnes simultanément.

Tableau 14-32 Déclaration de UNIQUE

Déclaration de la contrainte	Désactivation

ALTER TABLE Adherent
  ADD CONSTRAINT un_nom_prenom_tel
  UNIQUE (nom,prenom,tel);
	ALTER TABLE Adherent
	  DISABLE CONSTRAINT un_nom_prenom_tel;

L'index multicolonne (nom+prenom+tel) sera bénéfique pour les extractions dont un prédicat est basé sur le nom, le prénom et le numéro, et sur un accès aux trois colonnes simultanément.

Indexation

Les différents types d'index ont été brièvement présentés au chapitre 1. Sans index, toute recherche s'apparente à un parcours séquentiel de toute la table. Ainsi pour *n* lignes, le nombre moyen de lectures est égal *n*/2, ce qui est très pénalisant dès que le volume de données devient important. De plus, ce nombre d'accès croît proportionnellement avec le nombre de lignes (100 fois plus de lignes implique un temps d'accès 100 fois plus long).

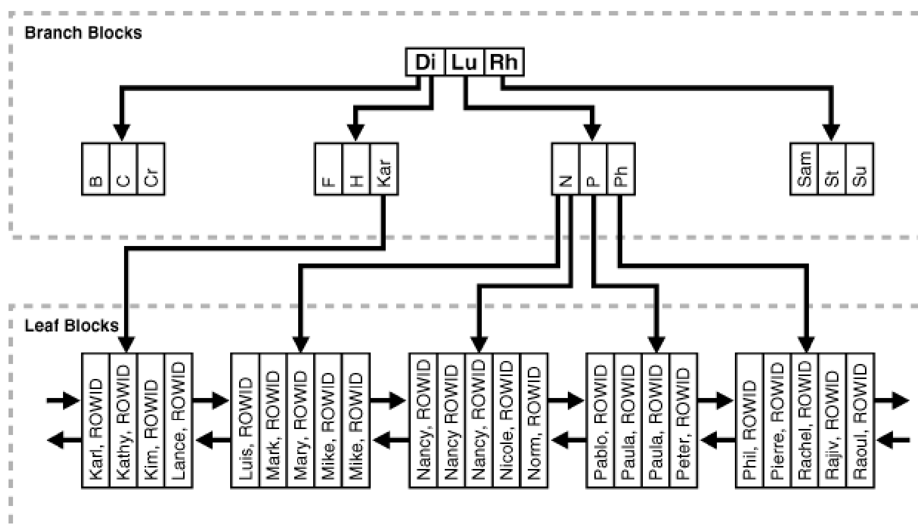
Étudions les cas d'utilisation des index d'Oracle de sorte à rendre une requête plus optimale.

Index B-tree

Les index *B-tree* (*B* comme *Balanced*) sont constitués comme des arbres dont les noeuds aiguillent vers des sous-noeuds (suivant la valeur recherchée) jusqu'aux blocs feuille (*leaf blocks*) qui contiennent toutes les valeurs de l'index et les adresses de ligne (*rowid*) identifiant le segment de données associé. Les blocs feuille sont doublement chaînés de sorte que l'index puisse être parcouru dans les deux sens sans passer par la racine.

Ce mécanisme est bien plus performant qu'un accès séquentiel car pour *n* lignes, le nombre moyen de lectures n'est plus proportionnel à *n* mais à *log*(*n*). La taille maximale d'une entrée d'index est environ égale à la moitié de la taille des blocs de données (soit de l'ordre de 4 000 pointeurs pour une taille de bloc de 8 Ko).

Figure 14-7 *Index B-tree (doc. Oracle)*



Un index *B-tree* est conçu automatiquement lors de la création de la clé primaire d'une table et d'une contrainte `UNIQUE`. Les arbres *B-tree* présentent de nombreux avantages :

- Malgré les mises à jour de la table, ils restent équilibrés (les feuilles blocs sont au même niveau). En conséquence, quelle que soit la valeur cherchée, le temps de parcours est sensiblement identique. Les blocs intermédiares sont remplis, en moyenne, au trois quart de leur capacité.
- Les performances d'extraction, répondant à la majorité des prédicats des requêtes, sont excellentes, notamment les comparaisons d'égalité et d'intervalles.

 © Éditions Eyrolles

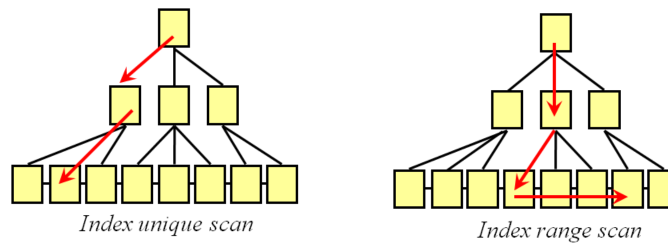
- Les répercussions des mises à jour sont efficaces et ne se dégradent pas en fonction d'une forte augmentation de la taille des tables.

Nous ne traiterons pas ici des caractéristiques physiques des index (partitions, compression, pourcentages des tailles de blocs, etc.).

Les principales opérations que l'optimiseur réalise sur un index sont les suivantes :

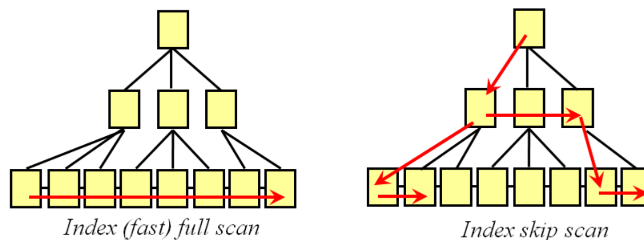
- *unique scan* passe par la racine de l'arbre ; généralement toutes les colonnes de l'index sont concernées par une égalité dans le prédicat `WHERE`. Il s'agit en principe de la manière la plus optimale, mais qui n'est pas toujours utilisée par l'optimiseur au profit de *range scan*.
- *index range scan* passe par la racine de l'arbre et accède séquentiellement aux blocs feuille (doublement chaînées). Opération très utilisée par l'optimiseur, notamment lorsque une colonne de l'index est concernée par une inégalité dans le prédicat `WHERE`, et que l'index n'est pas unique. Dans tous ces cas, l'optimiseur juge qu'il est plus rapide de parcourir les feuilles de l'index plutôt que l'index lui-même.

Figure 14-8 *Accès direct et par parcours par intervalles d'un index B-tree*

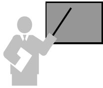


- *index full scan* et *index fast full scan* sont une alternative au parcours *full table scan* quand l'index contient toutes les colonnes necéssaires à la requête et qu'au moins une de ces colonnes est `NOT NULL`. Il ne peut pas être utilisé sur un index bitmap ; le parcours de l'index entier est plus rapide car il se réalise en mode lecture multibloc et peut être parallélisé.
- *index skip scan* (concerne les index multicolonnes) utilise l'index alors que la (ou les) première(s) colonne(s) de l'index n'est (ne sont) pas présente(s) dans le prédicat `WHERE`.

Figure 14-9 *Parcours séquentiel et par saut d'un index B-tree*



© Éditions Eyrolles



Généralement, afin d'isoler le stockage physique des index, on utilise un *tablespace* dédié (qui peut se trouver sur un autre disque que celui des données). Il est aussi d'usage de créer un index pour chaque clé étrangère afin de rendre plus efficace les jointures.

Le tableau suivant présente d'une part la création de l'espace de stockage pour héberger les index et, d'autre part, la création d'index affectés à cet espace (une clé primaire et une clé étrangère non unique).

Tableau 14-33 Création d'index en association avec un tablespace

Création de l'espace	Création d'index

`CREATE TABLESPACE tbs_index`
`DATAFILE 'tbs_index.dat'`
`SIZE 500M REUSE`
`AUTOEXTEND ON`
`NEXT 500K MAXSIZE 2000M;`
	`--colonne "clé primaire"`
	`ALTER TABLE Adherent`
	`ADD CONSTRAINT pk_Adherent PRIMARY KEY (adhid)`
	`USING INDEX TABLESPACE tbs_index``;`
	…
	`--colonne "clé étrangère"`
	`CREATE INDEX idx_Pratique_adhid`
	`ON Pratique (adhid)` **`TABLESPACE tbs_index`**`;`

Pour se convaincre de l'utilité des index, exécutez la requête avec et sans index (il s'agit d'une division) qui extrait les adhérents inscrits à tous les sports. L'adhérente la plus sportive est, sans conteste, Céline Larrazet et il faut 36 secondes sans indexage pour découvrir l'identité de la championne alors que la réponse est quasi-instantanée en présence d'index sur les clés étrangères. Les chiffres sont éloquents même pour une volumétrie réduite (24 000 adhérents dans 1 800 blocs) : sans index, on recense 20 fois plus d'accès aux blocs et de nombreux tris.

Tableau 14-34 Performances d'une extraction avec et sans index

Avec index	Sans index

```
SELECT a.civilite, a.prenom, a.nom, a.tel FROM Adherent a
       WHERE NOT EXISTS
            (SELECT spid FROM Sport
             MINUS  SELECT spid FROM Pratique WHERE adhid = a.adhid)
       AND NOT EXISTS
           (SELECT spid FROM Pratique WHERE adhid = a.adhid
            MINUS SELECT spid FROM Sport);

CIVILITE     PRENOM                NOM                   TEL
------------ --------------------- --------------------- --------------
Mme.         CELINE               LARRAZET               05-62-18-04-76
```

Avec index	Sans index
`36    secondes`	`3 centièmes de secondes`
`114   recursive calls`	`533  recursive calls`
`72734 consistent gets`	`1442719  consistent gets`
`0     sorts (memory)`	`48080  sorts (memory)`

 © Éditions Eyrolles

Bien que les index *B-tree* soient majoritairement employés, ils ne conviennent pas aux conditions suivantes :

- Données de faible cardinalité : on considère qu'une colonne disposant de moins de 200 valeurs disctinctes n'est pas une bonne candidate à un index *B-tree* (par exemple, la civilité qui ne comporte que 3 valeurs). Les index *bitmap* sont une alternative à cette limitation.
- Quand l'accès aux données s'effectue par une fonction SQL (*built-in function*), l'index *B-tree* n'est pas utilisé (par exemple `WHERE UPPER(prenom)='PAUL'` n'emploiera pas l'index sur `prenom`). Le fait de créér un index sur cette fonction est une alternative à cette limitation.

Avant la version 9*i*, l'optimiseur optait systématiquement pour un parcours entier de la table (*full scan*) si les requêtes contenaient des prédicats basés sur des expressions. Depuis, Oracle a répondu à ces problématiques à l'aide de mécanismes complémentaires avec les index *bitmap* et ceux basés sur des expressions ou des fonctions (*functions based index*).

Index et expressions (built-in function)

Si vous utilisez des fonctions caractères (`UPPER`, `SUBSTR`, `RTRIM`, etc.) ou des fonctions numériques (`MOD`, `ROUND`, `TRUNC`, etc.) dans le prédicat de vos requêtes, n'espérerez pas utiliser vos index.

Le tableau 14-35 présente les résultats de différentes requêtes selon deux stratégies d'indexage. La volumétrie de la table `Adherentbis` est de plus d'un million d'adhérents (88 Mo de donnnées occupant près de 90 000 blocs). Pour chaque requête, sont donnés : le type de parcours de l'index (*table access full* : l'index n'est pas utilisé), le nombre de blocs lus (*b*) et le coût (*c*).

Les remarques que l'on peut déduire à propos de la première stratégie d'indexage sont les suivantes :

- Les fonctions `ROUND` et `UPPER` rendent inopérants les index définis pourtant sur les colonnes concernées.
- Les index sur les colonnes numériques sont plus performants que les index sur les colonnes chaînes de caractères.

Concernant la deuxième stratégie d'indexage, les fonctions `ROUND` et `UPPER` rendent opérationnels les index, mais les conditions simples sur les colonnes entraînent un parcours entier de la table.

© Éditions Eyrolles

Tableau 14-35 Utilisation d’index B-tree sur des expressions de colonnes

Index existants	Prédicats et résultats	
CREATE INDEX idx_nom ON Adherentbis (**nom**) TABLESPACE tbs_index; CREATE INDEX idx_solde ON Adherentbis (**solde**) TABLESPACE tbs_index;	WHERE **nom='DUCLOS'** AND civilite='Mr.' AND tel LIKE '+33%' *Index range scan* (578 b – 110 c)	WHERE UPPER(**nom**)='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%' *Table access full* (10236 *b* – 2797 *c*)
	WHERE ROUND(**solde**,1)=9030.8 *Table access full* (10235 *b* – 2802 *c*)	WHERE **solde=9030.75** *Index range scan* (3 *b* – 3 *c*)
CREATE INDEX idx_UPPERnom ON Adherent- bis(**UPPER(nom)**) TABLESPACE tbs_index; CREATE INDEX idx_ROUND- solde ON Adherentbis (**ROUND(solde,1)**) TABLESPACE tbs_index;	WHERE **nom**='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%' *Table access full* (10229 *b* – 2795 *c*)	WHERE **UPPER(nom)='DUCLOS'** AND civilite='Mr.' AND tel LIKE '+33%' *Index range scan* (578 *b* – 107 *c*)
	WHERE **ROUND(solde,1)=9030.8** *Index range scan* (3 *b* – 3 *c*)	WHERE **solde**=9030.75 *Table access full* (10229 *b* – 2795 *c*)

Index et valeurs NULL

Le principe de fonctionnement des index *B-tree* ne permet pas une recherche directe (*unique scan*) sur une absence de valeur (NULL) ; en conséquence, si un index existe sur une colonne non nulle, il ne sera pas utilisé au mieux lors de la recherche des NULL (prédicat IS NULL ou IS NOT NULL).

La valeur NULL n’est pas à éviter à tout prix, elle est parfois bien utile pour exprimer « je ne sais pas encore » ou « sans objet ». Il est aussi pratique de vouloir indexer une colonne qui contiendra des valeurs NULL. Plusieurs solutions existent, elles sont basées sur la création d’un index :

- Sur une fonction déterministe qui retourne un entier quand la colonne est nulle.
- Composé dont une colonne n’est jamais nulle.
- Basé sur la fonction NVL2(chaine, valeur_si_NOT_null, valeur_si_null), qui retourne une des deux valeurs suivant que chaine est nulle ou non.

Appliquez ces différentes solutions à votre base de sorte à déterminer la plus performante. Le tableau suivant présente quelques résultats d’après la recherche du nombre d’adhérents en fonction de leur numéro de téléphone donné (NULL, valeur, NOT NULL). Concernant les données, 37 485 adhérents n’ont pas de numéro de téléphone (soit 3 % de la population). Pour

La table suivante organisée en index est partionnée selon la date des contrats (avant 2000, entre 2000 et 2010 et après).

Tableau 14-67 Création d'une table IOT partitionnée

Code SQL	Commentaires

`CREATE TABLE Historique_partition_iot`
` (opeid        CHAR(4) NOT NULL,`
`  adhid        NUMBER(10) NOT NULL,`
`  date_contrat DATE NOT NULL,`
`  categorie    VARCHAR(15) NOT NULL,`
	Colonnes de la table.

`  reliquat     NUMBER(7) NOT NULL,`
`  CONSTRAINT   pk_Historique_partition_iot`
`               PRIMARY KEY (opeid,adhid,date_contrat))`
	Définition de la clé primaire. Le segment d'index se trouvera dans le *tablespace* `tbs_index`.

`  ORGANIZATION INDEX`
`       TABLESPACE tbs_index`
`       INCLUDING  date_contrat`
`  OVERFLOW TABLESPACE users`
	Les 3 premières colonnes seront dans le segment d'index, le débordement dans le *tablespace* `users`.

`PARTITION BY RANGE (date_contrat)`
`       (`**`PARTITION`**` avant_2000 VALUES LESS THAN`
`            (TO_DATE('01/01/2000','DD/MM/YYYY'))`
`             TABLESPACE tbs_part1,`
`        `**`PARTITION`**` de_2000_a_2010 VALUES LESS THAN`
`            (TO_DATE('01/01/2011','DD/MM/YYYY'))`
`             TABLESPACE tbs_part2,`
`        `**`PARTITION`**` apres VALUES LESS THAN (MAXVALUE)`
`             TABLESPACE tbs_part3);`
	Le partitionnement est défini sur la date du contrat et chaque partition se trouve dans un *tablespace* distinct.

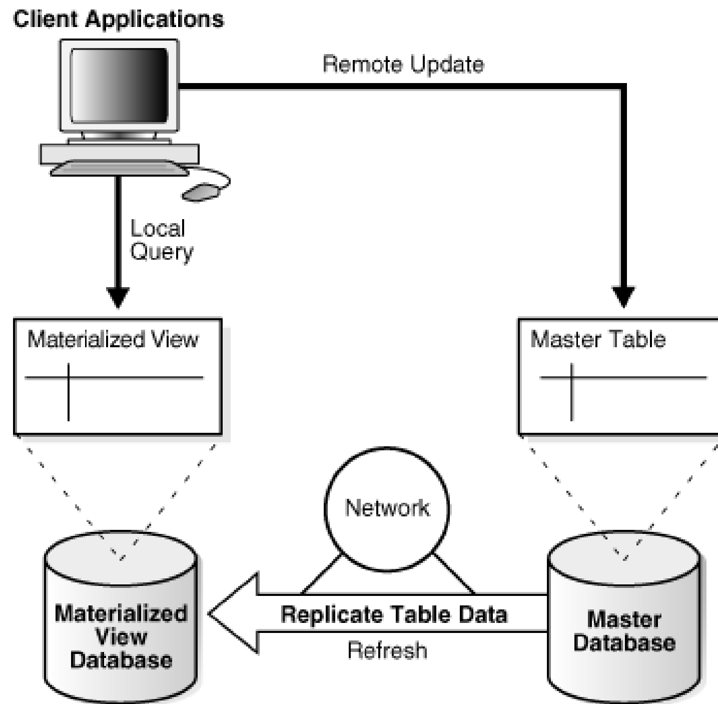
Vues matérialisées

Les vues (dématérialisées) étudiées au chapitre 5 permettent de simplifier l'écriture de certaines requêtes particulièrement complexes mais ne garantissent rien en regard des performances. Dans le pire des cas, une vue peut être consommatrice de ressources si d'autres vues sont impliquées en cascade dans la requête.

Les vues matérialisées (*materialized views*, anciennement *snapshots*) sont formées à partir de requêtes dont le résultat est stocké (comme les lignes d'une table). Une requête composant une vue matérialisée peut concerner des tables, vues et vues matérialisées. Dans un contexte de réplication, l'utilisation première de ces vues, une vue matérialisée s'appelle *master table*. Dans un contexte de *data warehouse*, une vue matérialisée est nommée *detail table*.

© Éditions Eyrolles

Figure 14-24 Vues matérialisées © doc. Oracle

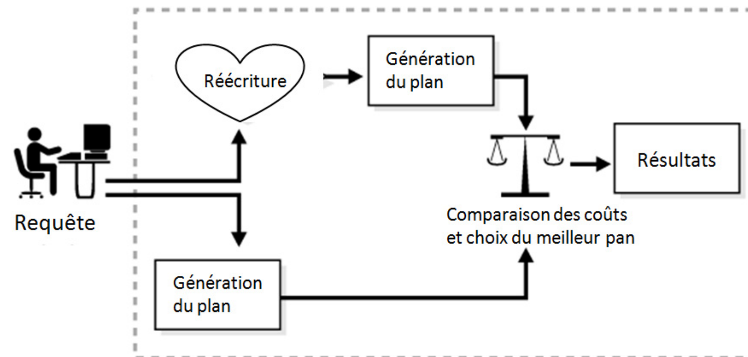


Sans aborder les a vantages de ces vues dans une architecture répartie ou d'entrepôts de données, du point de vue des performances, les vues matérialisées répondent parf aitement à l'amélioration des jointures du fait du stockage de lignes précalculées et de la possibilité de réécriture de requêtes (*query rewrite*). De plus, le partitionnement et l'inde xation sont possibles.

Réécriture de requêtes

La réécriture de requêtes est une technique d'optimisation qui transforme une requête complexe émise sur une table v olumineuse en une requête sémantiquement équi valente interrogeant la vue matérialisée. Dès qu'il est plus intéressant d'utiliser la vue matérialisée parce qu'elle contient des résultats déjà calculés (agrég ats et jointures), toute requête est réécrite (d'une manière transparente pour l'utilisateur) et utilise la vue à la place de la table. Aucun code n'est à ajouter dans l'instruction SQL qui ne référence que la (ou les) tables interrogées.

Figure 14-25 Réécriture de requêtes



Le rafraîchissement

Du fait du stockage redondant des données (dans les tables et dans la vue matérialisée), des méthodes de synchronisation (*refresh*) sont disponibles. La méthode de rafraîchissement peut être incrémentale (*fast refresh*) ou complète (*complete refresh*).

Le rafraîchissement incrémental évite de reconstruire entièrement la vue matérialisée. Cependant, ce mécanisme doit s'opérer relativement rapidement (à la demande ou périodiquement) pour garantir l'intégrité des données. Chaque table est associée à un journal d'opérations (*materialized view log*) qui recense toutes les modifications effectuées sur la table.

Le rafraîchissement complet se produit à la création de la vue matérialisée (définie avec `BUILD IMMEDIATE`). Bien que ce procédé puisse être couteux si les volumes de données manipulés sont importants, les requêtes interrogeant ces tables seront bien plus performantes.

Exemples

Dans un contexte de réplication, les vues matérialisées permettent de maintenir sur une base locale des copies de données distantes. Ces copies peuvent être modifiables sous réserve d'utiliser l'option *Advanced Replication*. En général, ces vues sont basées sur la clé primaire des tables (ou les *rowid*). Dans un contexte d'entrepôts de données, les vues matérialisées composent généralement des regroupements (agrégations) et des jointures.

Utilisons une vue matérialisée pour préparer les extractions d'adeptes de l'escrime et du tennis de table et comparons quelques avec une solution classique.

 © Éditions Eyrolles

Tableau 14-68 Création d'une vue matérialisée

Code SQL	Commentaires

`CREATE MATERIALIZED VIEW adh_escrime_pingpong`
`TABLESPACE tbs_cluster`
`BUILD IMMEDIATE`
`REFRESH COMPLETE`
`ENABLE QUERY REWRITE`
`AS SELECT  a.adhid, s.spid, s.splibelle,`
`a.nom, a.prenom, a.tel, a.date_nais, a.solde`
`FROM  Adherentbis a, Sport s, Pratiquebis p`
`WHERE s.splibelle IN ('Escrime','Ping-pong')`
`AND   a.adhid = p.adhid`
`AND   s.spid  = p.spid`
`ORDER BY a.adhid,a.nom;`
	Création de la vue matérialisée située dans le tablespace `tbs_cluster`.
	La construction est immédiate et la vue est éligible à la réécriture de requête (`ENABLE QUERY REWRITE`).

Les requêtes de jointure entre adhérents et l'escrime ou le tennis de table utiliseront la vue à la place des tables d'une manière bien plus efficace. L'opérateur que vous verrez apparaître dans vos plans d'exécution est `MAT_VIEW REWRITE ACCESS FULL`.

Le rafraîchissement automatique nécessite de créer un journal d'opérations par table interrogée. Les exemples suivants décrivent des vues matérialisées qui seront mises à jour automatiquement. La première sera actualisée dès la modification de la table `Sport`. La deuxième le sera tous les lundis à 15h00.

Tableau 14-69 Rafraîchissement automatique

Code SQL	Commentaires

`CREATE MATERIALIZED VIEW LOG`
`ON Sport WITH PRIMARY KEY, ROWID;`
`CREATE MATERIALIZED VIEW catalogue_sports`
`REFRESH FAST`
`ON COMMIT`
`WITH PRIMARY KEY`
`AS SELECT spid, splibelle FROM Sport;`
	Création du journal des opérations.
	Création de la vue matérialisée avec une contrainte *primary key* (colonne clé primaire de la table).
	Rafraîchissement incrémental (*fast*) après modifications sur la table.

`CREATE MATERIALIZED VIEW LOG`
`ON Adherentbis`
`WITH PRIMARY KEY, ROWID`
`PURGE REPEAT INTERVAL '5' DAY;`
`CREATE MATERIALIZED VIEW Adherent_hommes`
`REFRESH FAST`
`START WITH ROUND(SYSDATE + 1) + 11/24`
`NEXT NEXT_DAY(TRUNC(SYS-DATE),'LUNDI')+15/24`
`WITH PRIMARY KEY`
`AS   SELECT a.adhid, a.nom, a.prenom,`
`a.tel, a.date_nais, a.solde`
`FROM  Adherentbis a`
`WHERE a.civilite = 'Mr.';`
	Création du journal des opérations qui sera vidé tous les 5 jours.
	
	Création de la vue matérialisée avec une contrainte *primary key* (colonne clé primaire de la table).
	Rafraîchissement incrémental, première actualisation : le lendemain à 11h puis tous les lundis à 15h.

© Éditions Eyrolles