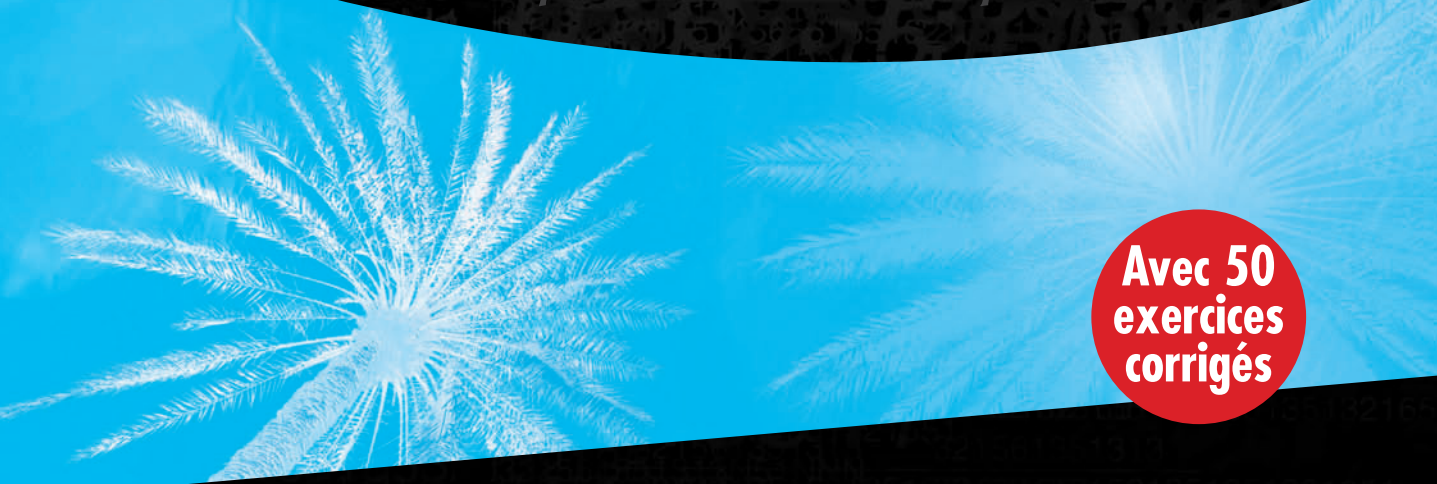
Christian Soutou

6e édition

SQL pour Oracle

Applications avec Java, PHP et XML

Optimisation des requêtes et schémas



Avec 50 exercices corrigés

EYROLLES

 ISBN : 978-2-212-13673-9

Table des matières

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.

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

Par ailleurs, sont disponibles en téléchargement sur la fiche de l'ouvrage (à l'adresse *www.editions-eyrolles.com*), quatre compléments qui traitent d'un usage d'Oracle moins courant :

- l'installation de versions qui ne sont plus supportées par l'éditeur (ou en fin de support) (complément 1 : Installation des versions 9*i* et 10*g*) ;
- la technologie SQLJ (complément 2 : L'approche SQLJ) ;
- les procédures externes (complément 3 : Procédures stockées et externes) ;
- les fonctions PL/SQL pour construire des pages HTML (complément 4 : PL/SQL Web Toolkit et PL/SQL Server Pages).

Guide de lecture

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

Première partie : SQL de base

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

Deuxième partie : PL/SQL

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

Troisième partie : SQL avancé

Cette partie intéressera les programmeurs qui envisagent d'exploiter une base Oracle en utilisant un langage de troisième ou quatrième génération (C, C++ ou Java), ou en employant une

interface Web. Le chapitre 8 est consacré à l'étude des mécanismes de base du précompilateur d'Oracle Pro*C/C++. Le chapitre 9 présente les principales fonctionnalités de l'API JDBC. Le chapitre 10 traite des deux principales API disponibles avec le langage PHP (OCI8 et PDO). Le chapitre 11 présente les fonctionnalités de XML DB et l'environnement *XML DB Repository*. Enfin, le chapitre 12 est dédié à l'optimisation des requêtes et des schémas relationnels.

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.

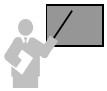
Conventions d'écriture et pictogrammes

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

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

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

Dans une instruction SQL, les symboles { et } désignent une liste d'éléments, et le symbole | un choix (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 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 (*christian.soutou@gmail.com*). Vous trouverez sur le site d'accompagnement, accessible par *www.editions-eyrolles.com*, les compléments et errata, ainsi que le code de tous les exemples et les exercices corrigés.

Chapitre 1

Définition des données

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

Tables relationnelles

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

Création d'une table (CREATE TABLE)

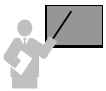
Pour pouvoir créer une table dans votre schéma, il faut que vous ayez reçu le privilège `CREATE TABLE`. Si vous avez le privilège `CREATE ANY TABLE`, vous pouvez créer des tables dans tout schéma. Le mécanisme des privilèges est décrit au chapitre « Contrôle des données ».

La syntaxe SQL simplifiée est la suivante :

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

- *schéma* : s'il est omis, il sera assimilé au nom de l'utilisateur connecté. S'il est précisé, il désigne soit l'utilisateur courant soit un autre utilisateur de la base (dans ce cas, il faut que l'utilisateur courant ait le droit de créer une table dans un autre schéma). Nous aborderons ces points dans le chapitre 5 et nous considérerons jusque-là que nous travaillons dans le schéma de l'utilisateur couramment connecté (ce sera votre configuration la plupart du temps).
- *nomTable* : peut comporter des lettres majuscules ou minuscules (accentuées ou pas), des chiffres et les symboles, par exemple : _, $ et #. Oracle est insensible à la casse et convertira au niveau du dictionnaire de données les noms de tables et de colonnes en majuscules.

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



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

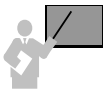
- *nomContraintei typeContraintei* : noms de la contrainte et son type (clé primaire, clé étrangère, etc.). Nous allons détailler dans le paragraphe suivant les différentes contraintes possibles.
- ; : symbole qui termine une instruction SQL d'Oracle. Le slash (/) peut également terminer une instruction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

Casse et commentaires

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

Tableau 1-1 Différentes écritures SQL

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



La casse a une incidence majeure dans les expressions de comparaison entre colonnes et valeurs, que ce soit dans une instruction SQL ou un test dans un programme. Ainsi, l'expression

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

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

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

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

Premier exemple

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

Figure 1-1 *Table à créer*

Compagnie

comp	nrue	rue	ville	nomComp

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

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

Contraintes

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

Les contraintes peuvent être déclarées de deux manières :

- En même temps que la colonne (valable pour les contraintes monocolonnes), ces contraintes sont dites « en ligne » (*inline constraints*). L'exemple précédent en déclare deux.
- Une fois la colonne déclarée, ces contraintes ne sont pas limitées à une colonne et peuvent être personnalisées par un nom (*out-of-line constraints*).

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

Quatre types de contraintes sont possibles :

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

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

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

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

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

Conventions recommandées

Adoptez les conventions d'écriture suivantes pour vos contraintes :

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

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

Figure 1-2 *Deux tables à créer*

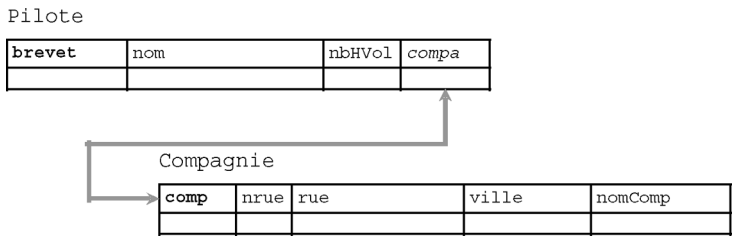


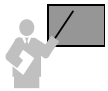
Tableau 1-3 Contraintes en ligne et nommées

Tables	Contraintes

`CREATE TABLE Compagnie`
`(comp VARCHAR2(4), nrue NUMBER(3),`
`rue VARCHAR2(20),`
`ville VARCHAR2(15) DEFAULT 'Paris',`
`nomComp VARCHAR2(15) NOT NULL,`
`CONSTRAINT pk_Compagnie PRIMARY KEY(comp));`
	Deux contraintes en ligne et une contrainte nommée de clé primaire.

`CREATE TABLE Pilote`
`(brevet VARCHAR2(6),`
`nom VARCHAR2(15) CONSTRAINT nn_nom NOT NULL,`
`nbHVol NUMBER(7,2), compa VARCHAR2(4),`
`CONSTRAINT pk_Pilote PRIMARY KEY(brevet),`
`CONSTRAINT ck_nbHVol CHECK (nbHVol BETWEEN 0 AND 20000),`
`CONSTRAINT un_nom UNIQUE (nom),`
`CONSTRAINT fk_Pil_compa_Comp FOREIGN KEY(compa)`
`REFERENCES Compagnie(comp));`
	Une contrainte en ligne nommée `(NOT NULL)` et quatre contraintes hors ligne nommées :
	• Clé primaire
	• `CHECK` (nombre d'heures de vol compris entre 0 et 20000)
	• `UNIQUE` (homonymes interdits)
	• Clé étrangère

Remarques



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

Types des colonnes

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

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

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

Caractères

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

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

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

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

N'utilisez le type `CHAR` que si vos données « remplissent bien » la taille définie par la colonne. En effet, la chaîne « Oracle » dans un `CHAR(500)` réservera 500 octets pour n'en stocker que 6 en réalité. Il est même conseillé de ne pas utiliser ce type.

Le type `VARCHAR` est obsolète (il permettait de gérer des chaînes de taille variable jusqu'à 2 000 caractères, et utilisait des valeurs `NULL` pour compléter la taille maximale de chaque donnée). Depuis la version 9*i*, ce type est devenu un synonyme de `VARCHAR2` (qui n'occupe pas d'espace supplémentaire à la taille de la donnée).

Tableau 1-4 Types de données caractères

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

Valeurs numériques

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

Tableau 1-5 Type de données numériques

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

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

Tableau 1-6 Représentation du nombre 7456123.89

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

Date/heure

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

Tableau 1-7 Types de données date/heure

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

Données binaires

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

Tableau 1-8 Types de données binaires

Type	Description	Commentaires pour une colonne
BLOB	Données binaires non structurées.	Jusqu'à 4 gigaoctets.
BFILE	Données binaires stockées dans un fichier externe à la base.	idem.
RAW(*size*)	Données binaires.	Jusqu'à 2 000 octets. Plus utilisé mais fourni pour assurer la compatibilité avec les anciennes applications.
LONG RAW	Données binaires.	Comme RAW, jusqu'à 2 gigaoctets.

Structure d'une table (DESC)

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

DESC[RIBE] [*schéma.*]*élément*

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

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

Tableau 1-9 Structure brute des tables

Table Compagnie

```
SQL> DESC Compagnie
 Nom           NULL ?    Type
 ------------------------------
 COMP          NOT NULL  VARCHAR2(4)
 NRUE                    NUMBER(3)
 RUE                     VARCHAR2(20)
 VILLE                   VARCHAR2(15)
 NOMCOMP       NOT NULL  VARCHAR2(15)
```

Table Pilote

```
SQL> DESC Pilote
 Nom           NULL ?    Type
 ------------------------------
 BREVET        NOT NULL  VARCHAR2(6)
 NOM           NOT NULL  VARCHAR2(15)
 NBHVOL                  NUMBER(7,2)
 COMPA                   VARCHAR2(4)
```

Les colonnes de type clé primaire sont définies par défaut `NOT NULL`.

Si vous définissez `NOT NULL` sur une colonne avec une contrainte `CHECK`, vous ne verrez pas apparaître cette condition avec la commande `DESC` (mais la contrainte sera toutefois active). Ce moyen n'est pas conseillé et il est préférable de définir toutes vos contraintes `NOT NULL` en ligne (en les nommant ou pas).

Restrictions

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

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

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

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

Commentaires stockés (COMMENT)

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

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

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

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

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

Index

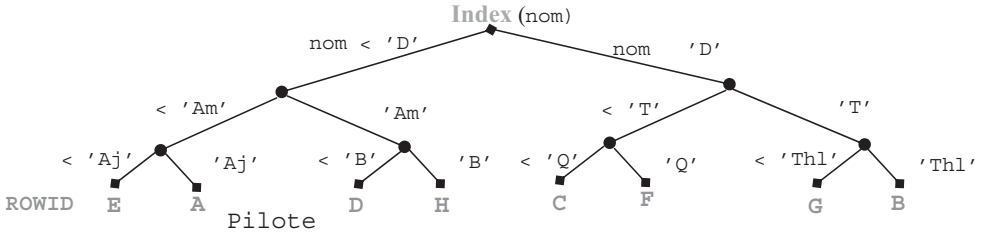
Cette section introduit simplement l'indexation qui sera approfondie à la section « Organisation des données » du chapitre 12 consacré aux mécanismes d'optimisation.

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

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

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

Figure 1-3 *Index sur la colonne nom*



Pilote

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

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

Classification

Plusieurs types d'index sont proposés par Oracle :

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

Index B-tree

La particularité de ce type d'index est qu'il conserve en permanence une arborescence symétrique (*balanced*). Toutes les feuilles sont à la même profondeur. Le temps de recherche est ainsi à peu près constant quel que soit l'enregistrement cherché. Le plus bas niveau de l'index (*leaf blocks*) contient les valeurs des colonnes indexées et le *rowid*. Toutes les feuilles de l'index sont chaînées entre elles. Pour les index non uniques (par exemple si on voulait définir un index sur la colonne `compa` de la table `Pilote`) le *rowid* est inclus dans la valeur de la colonne indexée. Ces index, premiers apparus, sont désormais très fiables et performants, ils ne se dégradent pas lors de la montée en charge de la table.

Index bitmap

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

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

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

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

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

Index *bitmap* sur `compa`

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

`Pilote`

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

Index basés sur des fonctions

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

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

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

Figure 1-5 *Index basé sur une fonction*

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

`CompteEpargne`

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

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

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

Création d'un index (CREATE INDEX)

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

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

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

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

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

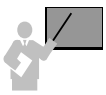
Créons plusieurs index sur la table des comptes bancaires. Le dernier (basé sur une fonction), doit faire apparaître les colonnes calculées dans ses paramètres après l'expression du calcul.

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

Tableau 1-10 Création d'index

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

Bilan



- Un index ralentit les rafraîchissements de la base (conséquence de la mise à jour de l'arbre ou des *bitmaps*). En revanche il accélère les accès.
- Il est conseillé de créer des index sur des colonnes (majoritairement des clés étrangères) utilisées dans les clauses de jointures (voir chapitre 4).

- Les index *bitmaps* sont conseillés quand il y a peu de valeurs distinctes de la (ou des) colonne(s) à indexer. Dans le cas inverse, utilisez un index *B-tree*.
- Les index sont pénalisants lorsqu'ils sont définis sur des colonnes très souvent modifiées ou si la table contient peu de lignes.

Tables organisées en index

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

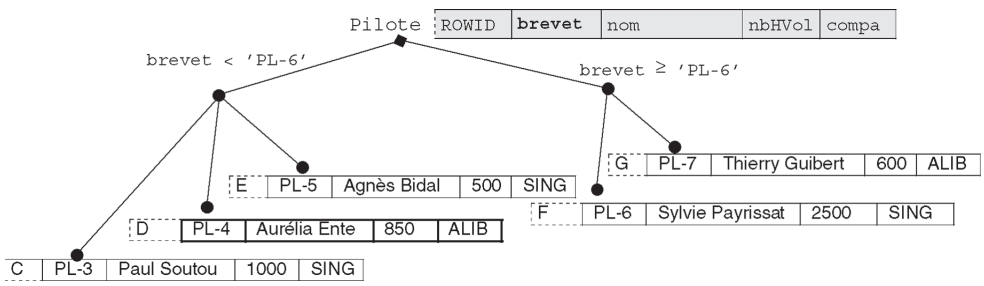
Apparu en version 8, ce type de tables est particulièrement utile pour les applications qui doivent extraire des informations basées essentiellement sur les clés primaires ou des éléments plus complexes (textes, images et sons). Le tableau suivant détaille les différences entre les deux types de tables.

Tableau 1-11 Caractéristiques des tables

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

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

Figure 1-6 *Table organisée en index*



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

Dans notre exemple la syntaxe à utiliser est la suivante :

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

Les autres options de la directive `ORGANIZATION` sont :

- `HEAP` qui indique que les données ne sont pas stockées dans un ordre particulier (option par défaut) ;
- `EXTERNAL` qui précise que la table est en lecture seule et est située à l'extérieur de la base (sous la forme d'un fichier ASCII par exemple).

Utilisation de SQL Developer Data Modeler

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

Dans l'arborescence de gauche, par un clic droit sur l'élément `Modèles relationnels`, choisir `Nouveau modèle relationnel`. Une fois dans le modèle relationnel, les icônes indiquées vous permettront de créer vos tables et toutes les liaisons entre elles (clés étrangères) ; voir figure 1-7.

En considérant l'exemple illustré par la figure 1-2 de la section « Conventions recommandées », deux tables doivent être définies ainsi qu'une clé étrangère.

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

Figure 1-7 *Création d'un modèle relationnel avec Data Modeler*

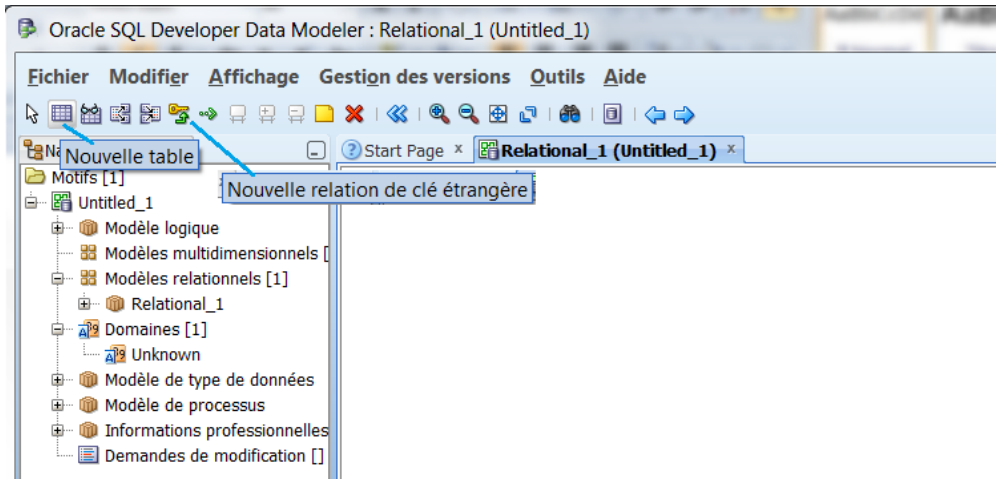
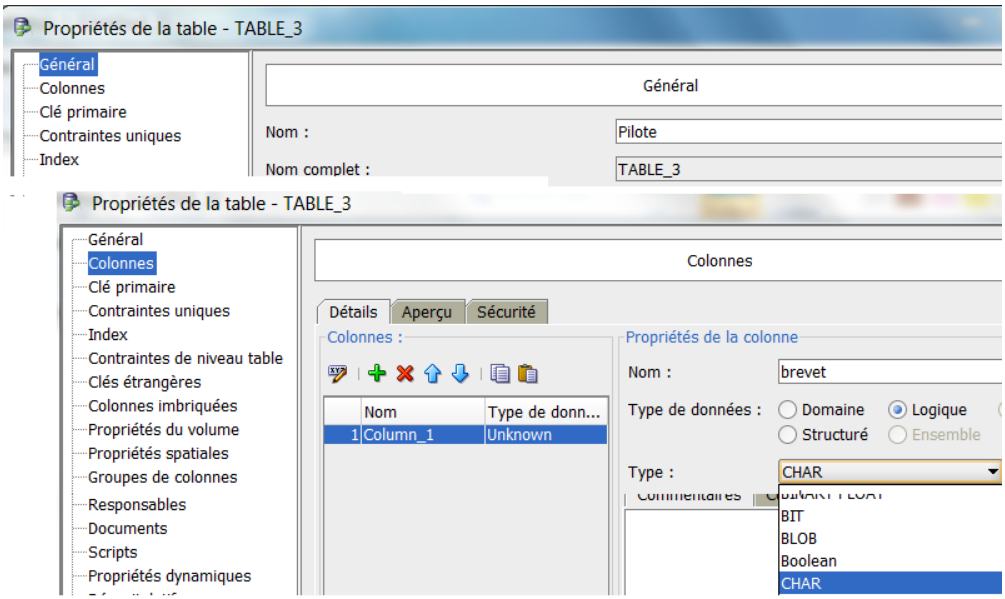
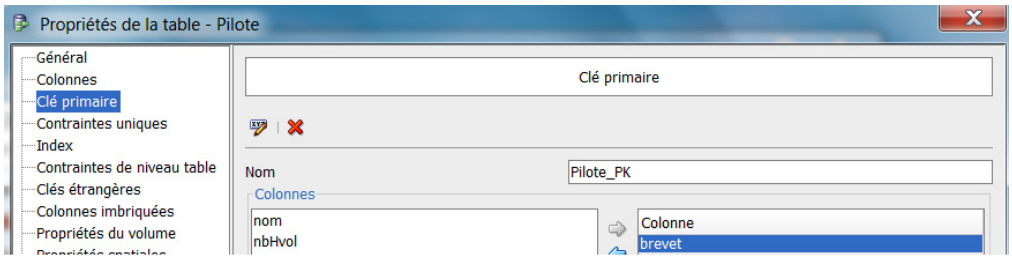


Figure 1-8 *Colonnes d'une table avec Data Modeler*



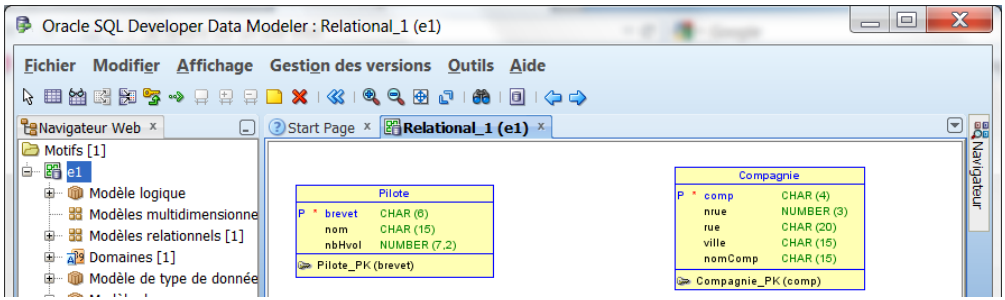
Définissez ensuite la clé primaire de chaque table (colonne `brevet` de la table `Pilote` et colonne `comp` de la table `Compagnie`).

Figure 1-9 *Clé primaire avec Data Modeler*



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

Figure 1-10 *Tables dotées d'une clé primaire avec Data Modeler*



Pour créer une relation entre ces deux tables, vous devez tirer un lien de la table `Compagnie` vers la table `Pilote` en sélectionnant l'icône de clé étrangère. La boîte de dialogue suivante s'affiche (voir figure 1-11). Elle décrit les caractéristiques de la nouvelle contrainte référentielle (voir le chapitre 3). Assurez-vous que la table source corresponde à la table de référence (« père »).

Vous remarquerez que la table « fils » se voit dotée d'une nouvelle colonne (la clé étrangère que l'outil nomme par défaut *`nomtablecible_nomcleprimaire`*). Dans cet exemple, la clé étrangère se nomme `Compagnie_comp` que vous devrez renommer `compa` pour être en phase avec le modèle relationnel de notre exemple. Il reste à rendre non obligatoire cette relation (double-cliquez au niveau du lien) pour obtenir le modèle relationnel final avant de générer le script SQL (voir figure 1-12).

Figure 1-11 *Définition d'une clé étrangère avec Data Modeler*

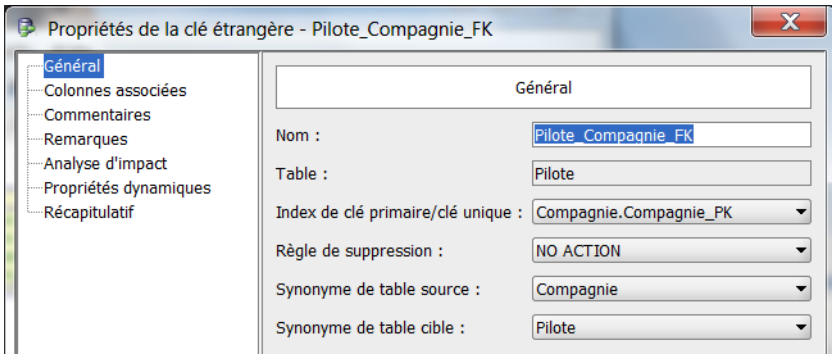
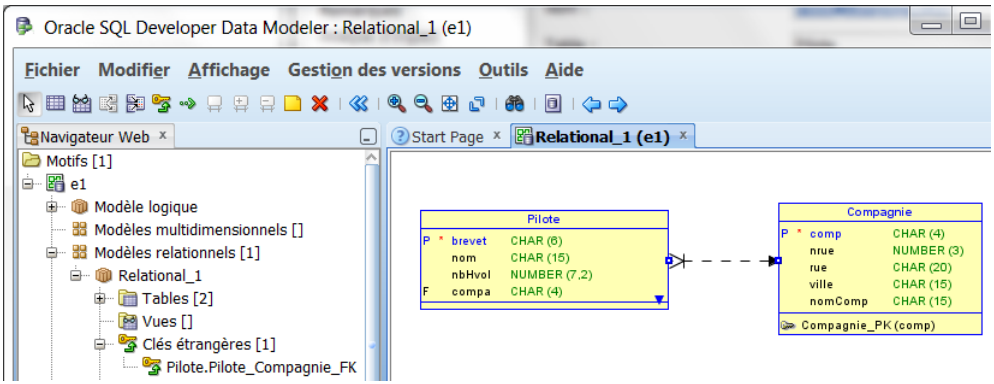
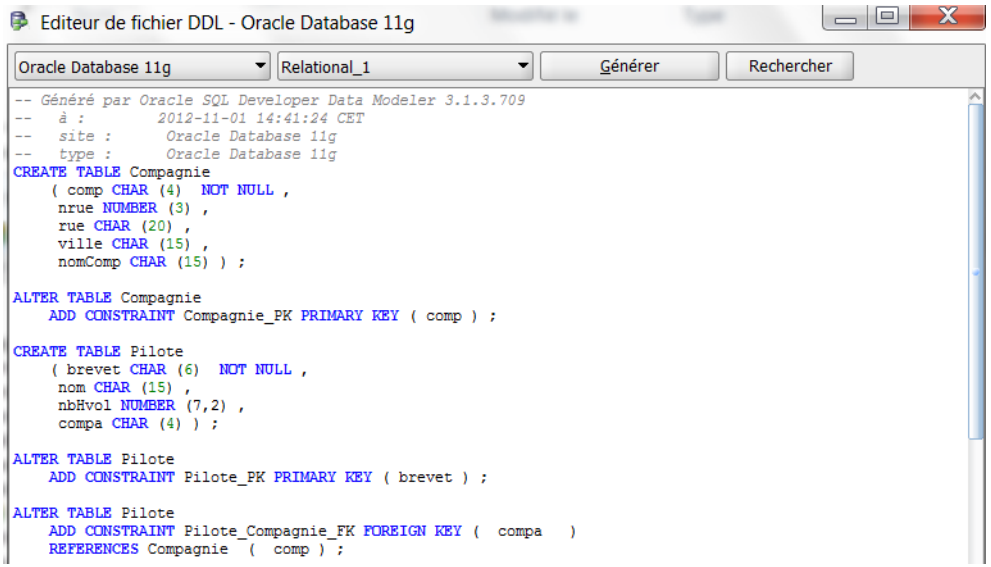


Figure 1-12 *Modèle relationnel final avec Data Modeler*



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

Figure 1-13 *Script de génération des tables avec Data Modeler*

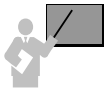


Suppression des tables

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

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

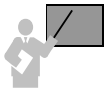
```
DROP TABLE [schéma.]nomTable [CASCADE CONSTRAINTS][PURGE];
```



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

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

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



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

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

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

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

Tableau 1-12 Scripts équivalents de destruction

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

`DROP TABLE Compagnie`
`  CASCADE CONSTRAINTS;`
`DROP TABLE Pilote;`
	`--schéma Compagnie`
	`DROP TABLE Pilote;`
	`DROP TABLE Compagnie;`

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

Exercices

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

Exercice 1.1 Présentation de la base de données

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

Les noms et types des colonnes sont les suivants :

Tableau 1-13 Caractéristiques des colonnes

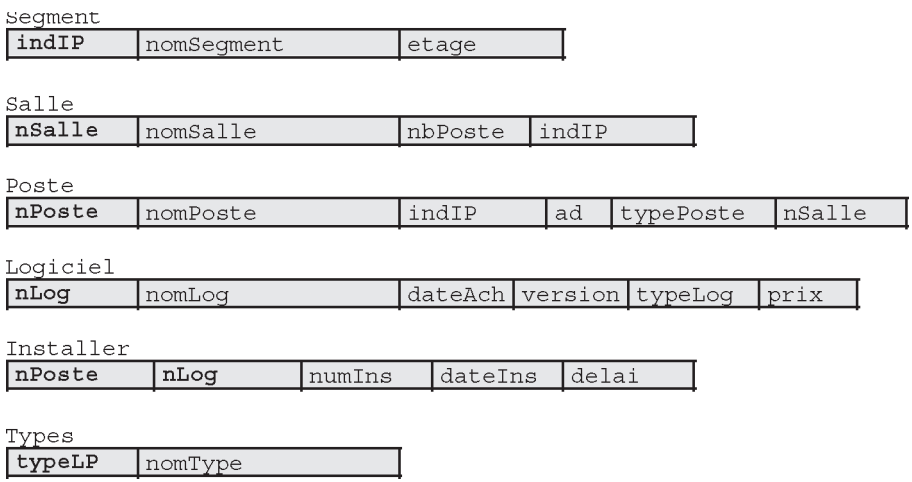
Colonne	Commentaires	Types
indIP	Trois premiers groupes IP (exemple : 130.120.80).	VARCHAR2(11)
nomSegment	Nom du segment.	VARCHAR2(20)
etage	Étage du segment.	NUMBER(2)
nSalle	Numéro de la salle.	VARCHAR2(7)
nomSalle	Nom de la salle.	VARCHAR2(20)
nbPoste	Nombre de postes de travail dans la salle.	NUMBER(2)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

Exercice 1.2 Création des tables

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

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

Figure 1-14 *Schéma des tables*



Exercice 1.3 Structure des tables

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

Exercice 1.4 Destruction des tables

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

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

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

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

Les colonnes à utiliser sont les suivantes :

Tableau 1-14 Caractéristiques des colonnes à ajouter

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

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

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

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

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

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

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

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)`

Clé étrangère

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

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

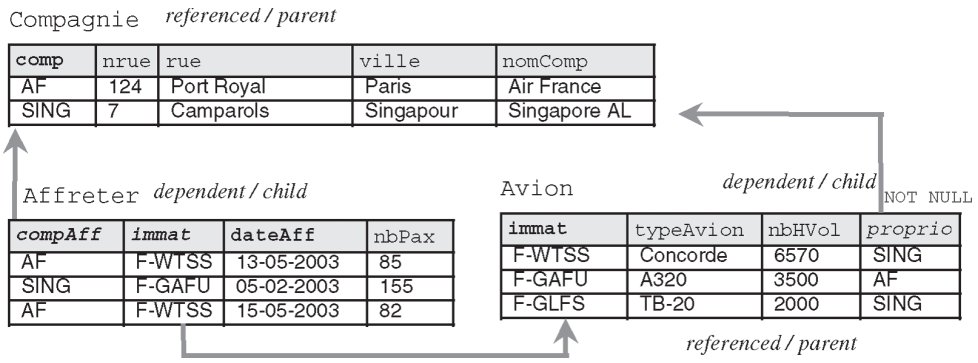
Clé primaire

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

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

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

Figure 3-5 *Après ajout de contraintes*



Suppression de contraintes

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

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

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

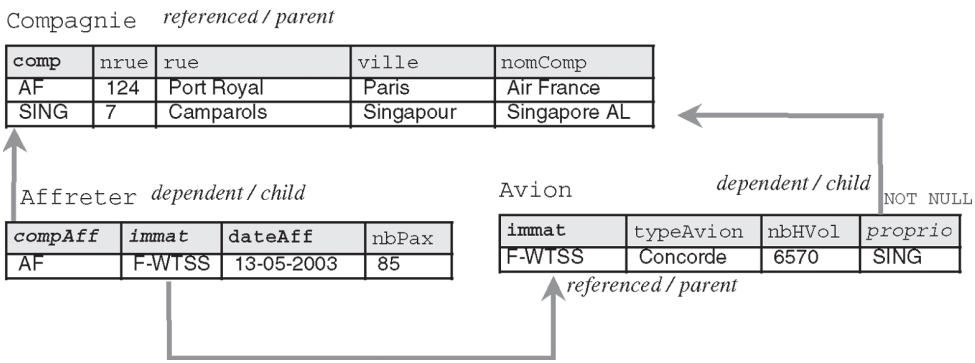
La syntaxe générale est la suivante :

```
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.

Figure 3-7 *Avant la désactivation de contraintes*



Contrainte de vérification

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

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

Clé étrangère

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

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

Clé primaire

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

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

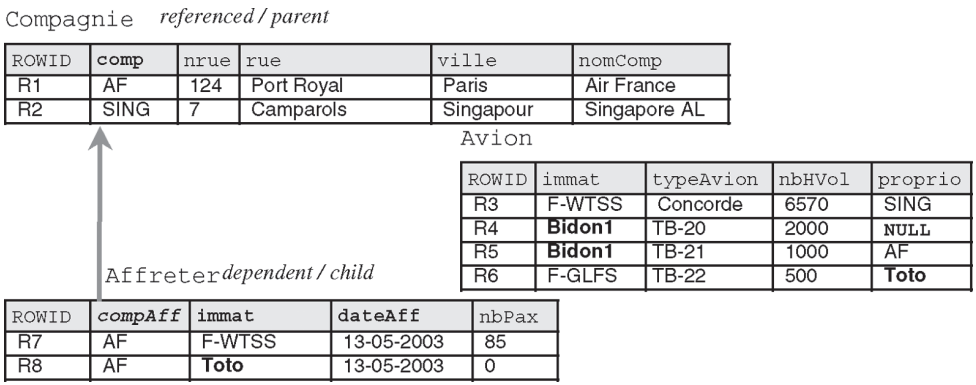
La désactivation de cette contrainte par `CASCADE` supprime aussi une des clés étrangères de la table `Affreter`. Insérons un affrètement qui référence un avion inexistant :

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

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

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

Figure 3-8 *Après désactivation de contraintes*



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

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**

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

Réactivation de contraintes

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

Syntaxe

La syntaxe générale est la suivante :

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

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

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

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

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

ALTER TABLE Avion ENABLE CONSTRAINT nn_proprio;
```

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.

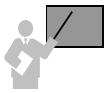
Le tableau suivant décrit deux tables permettant de stocker les enregistrements erronés après réactivation de contraintes.

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` :

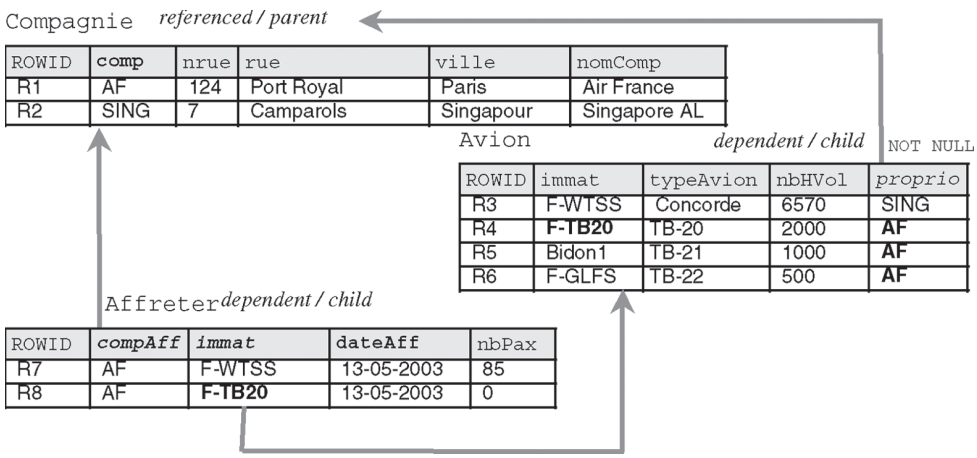
```
UPDATE Affreter SET immat = 'F-TB20' WHERE immat = 'Toto';
```

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.

```
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*



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`.

Exercice 3.5 Ajout de colonnes dans la base *Chantiers*

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

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

Vérifiez la structure de chaque table avec `DESC`.

Exercice 3.6 Mise à jour de la base *Chantiers*

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

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

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

Exceptions

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

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

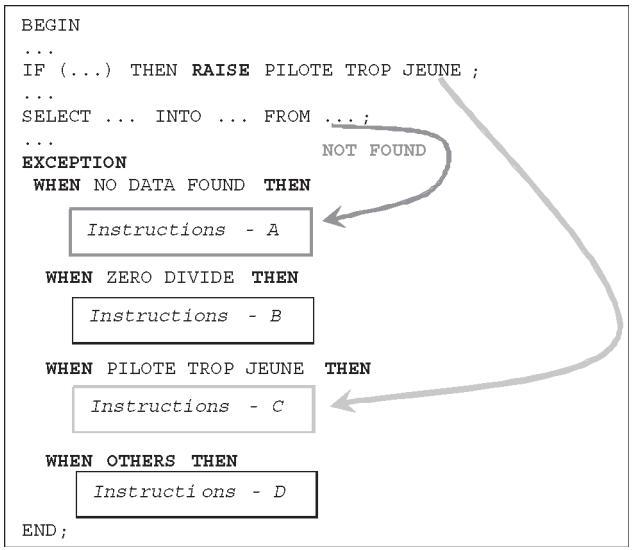
Généralités

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

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

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

Figure 7-6 *Principe général des exceptions*



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

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

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

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

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

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

Étudions à présent les trois types d'exceptions qui existent sous PL/SQL, en programmant des procédures simples interrogeant la table Pilote illustrée à la figure 7-3.

Exception interne prédéfinie

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

Tableau 7-22 Exceptions prédéfinies

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

Le code d'erreur (SQLCODE) qui peut être récupéré par un programme d'application (Java par exemple sous JDBC), est inclus dans le numéro interne de l'erreur (pour la deuxième exception, il s'agit de -6 592).

Concernant l'erreur NO_DATA_FOUND, rappelez-vous qu'elle n'est opérationnelle qu'avec l'instruction SELECT. Une mise à jour ou une suppression (UPDATE et DELETE) d'un enregistrement inexistant ne déclenche pas l'exception. Pour gérer ces cas d'erreurs, il faut utiliser un curseur implicite et une exception utilisateur (voir la section « Utilisation du curseur implicite »).

Si vous désirez programmer une erreur qui n'apparaît pas dans cette liste (exemple : erreur référentielle pour une suppression d'un enregistrement d'une table identifiée par une clé étrangère), il faudra programmer une exception non prédéfinie (voir la section suivante).

Plusieurs erreurs

Le tableau suivant décrit une procédure qui gère deux erreurs : aucun pilote n'est associé à la compagnie de code passé en paramètre (NO_DATA_FOUND) et plusieurs pilotes le sont (TOO_MANY_ROWS). Le programme se termine correctement si la requête retourne une seule ligne (cas de la compagnie de code 'CAST').

Tableau 7-23 Deux exceptions traitées

Code PL/SQL	Commentaires

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

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

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

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

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

```
La compagnie AF a plusieurs pilotes!
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE procException1('RIEN');
La compagnie RIEN n'a aucun pilote!
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE procException1('CAST');
Le pilote de la compagnie CAST est Thierry Millan
Procédure PL/SQL terminée avec succès.
```

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

Même erreur sur différentes instructions

Le tableau 7-21 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)
```

Tableau 7-25 Bloc d'exceptions imbriqué

Code PL/SQL	Commentaires

`CREATE PROCEDURE procException3`
`(p_brevet IN VARCHAR2, p_heures IN NUMBER) IS`
`var1 Pilote.nom%TYPE;`
`BEGIN`	

`BEGIN`
`SELECT nom INTO var1 FROM Pilote`
`WHERE brevet = p_brevet;`
`DBMS_OUTPUT.PUT_LINE('Le pilote de ' || p_brevet`
`|| ' est ' || var1);`
	Bloc imbriqué.

`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote de brevet : '`
`|| p_brevet);`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM || ' (' || SQLCODE || ')');`
`END;`
	Gestion des exceptions de la première requête.

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

`EXCEPTION`
`WHEN NO_DATA_FOUND THEN`
`DBMS_OUTPUT.PUT_LINE('Pas de pilote ayant ce nombre`
`d''heures de vol : ' || p_heures);`
`WHEN OTHERS THEN`
`DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' ||`
`SQLERRM || ' (' || SQLCODE || ')');`
`END;`
	Gestion des exceptions de la deuxième requête.

Exception utilisateur

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

Déclaration

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

```
nomException EXCEPTION;
```

Déclenchement

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

Dans notre exemple, programmons les deux exceptions suivantes :

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

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

Tableau 7-26 Exceptions utilisateur

Code PL/SQL	Commentaires

`CREATE PROCEDURE saisiePilote`
`(p_brevet IN VARCHAR2,p_nom IN VARCHAR2,`
`p_nbHVol IN NUMBER, p_comp IN VARCHAR2) IS`
`erreur_piloteTropJeune EXCEPTION;`
`erreur_piloteTropExpérimenté EXCEPTION;`
	Déclaration de l'exception.

`BEGIN`
`INSERT INTO Pilote (brevet,nom,nbHVol,comp)`
`VALUES (p_brevet,p_nom,p_nbHVol,p_comp);`
`IF p_nbHVol < 200 THEN RAISE erreur_piloteTropJeune;`
`END IF;`
`IF p_nbHVol > 20000 THEN`
`RAISE erreur_piloteTropExpérimenté;`
`END IF;`
`COMMIT;`
	Corps du traitement (validation).

`EXCEPTION`
`WHEN erreur_piloteTropJeune THEN`
`ROLLBACK;`
`DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote manque`
`d''expérience');`
	Gestion de l'exception.

`WHEN erreur_piloteTropExpérimenté THEN`
`ROLLBACK;`
`DBMS_OUTPUT.PUT_LINE ('Désolé, le pilote a`
`trop d''expérience');`
	Gestion des autres exceptions.

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

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

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

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

Utilisation du curseur implicite

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

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

Tableau 7-27 Utilisation du curseur implicite

Code PL/SQL	Commentaires

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

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

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

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

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

Exception interne non prédéfinie

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

Déclaration

Deux commandes sont nécessaires dans la section déclarative à la mise en œuvre de ce mécanisme : déclarer le nom de l'exception et associer cet identificateur à l'erreur Oracle.

```
nomException EXCEPTION;
PRAGMA EXCEPTION_INIT(nomException, numéroErreurOracle);
```

Pour connaître le numéro de l'erreur qui vous intéresse, consultez la liste des erreurs dans la documentation d'Oracle (*Error Messages* qui est classée par numéros croissants et non pas par fonctionnalités). Cherchez par exemple les entrées correspondant à *foreign key* dans le chapitre des erreurs `ORA-02100 to ORA-04099`.

Vous pouvez aussi écrire un bloc PL/SQL qui programme volontairement l'erreur pour voir sous SQL*Plus le numéro qu'Oracle renvoie.

Déclenchement

Une exception non prédéfinie sera levée de la même manière qu'une exception prédéfinie, à savoir suite à une instruction SQL pour laquelle le serveur aura renvoyé une erreur.

Considérons les deux tables suivantes. La colonne `comp` de la table `Pilote` est clé étrangère vers la table `Compagnie`. Programmons une procédure qui supprime une compagnie de code passé en paramètre.

Figure 7-7 *Deux tables*

`Compagnie`

comp	ville	nomComp
AF	Paris	Air France
SING	Singapour	Singapore AL
CAST	Blagnac	Castanet AL
EJET	**Dublin**	**Easy Jet**

à détruire

`Pilote`

brevet	nom	nbHVol	comp
PL-1	Gilles Laborde	2450	AF
PL-2	Frédéric D'Almeyda	900	AF
PL-3	Florence Périssel	1000	SING
PL-4	Thierry Millan	2450	CAST
PL-5	Christine Royo	200	AF
PL-6	Aurélia Ente	2450	SING

Le tableau suivant décrit la procédure `détruitCompagnie` qui intercepte l'erreur `ORA-02292: enregistrement fils existant`. Il s'agit de contrôler le programme si la compagnie à détruire possède encore des pilotes référencés dans la table `Pilote`.

Tableau 7-28 Exception interne non prédéfinie

Code PL/SQL	Commentaires

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

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

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

`  WHEN OTHERS THEN`
`  DBMS_OUTPUT.PUT_LINE('Erreur d''Oracle ' || SQLERRM`
`  || '(' || SQLCODE || ')');`
	Gestion des autres exceptions.

`END;`	

La trace de l'exécution de cette procédure est la suivante. Notez que si on applique cette procédure à une compagnie inexistante, le programme se termine normalement sans passer dans la section des exceptions.

```
SQL> EXECUTE détruitCompagnie('AF');
Désolé, il reste encore un pilote à la compagnie AF
Procédure PL/SQL terminée avec succès.

SQL> EXECUTE détruitCompagnie('EJET');
Compagnie EJET détruite.
Procédure PL/SQL terminée avec succès.
```

Propagation d'une exception

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

Exercice 7.5 Transaction de la base *Chantiers*

Écrivez la procédure `finAnnee` permettant de rajouter à chaque véhicule les kilométrages faits lors des visites de l'année. Vous utiliserez un seul curseur pour parcourir tous les véhicules. Il faudra ensuite supprimer toutes les missions de l'année (visites et détails des trajets des employés transportés).

Exercice 7.6 Déclencheurs de la base *Chantiers*

Déclencheur ligne

Écrivez le déclencheur `TrigPassagerConducteur` sur la table `transporter` permettant de vérifier qu'à chaque nouveau transport, le passager déclaré n'est pas déjà enregistré en tant que conducteur le même jour.

Déclencheur composé

Écrivez le déclencheur composé `TrigcapaciteVehicule` sur la table `transporter` permettant de contrôler, qu'à chaque nouveau transport, la capacité du véhicule n'est pas dépassée.

Vous éviterez le problème des tables mutantes en :

- déclarant dans la zone de définition commune un tableau recensant le nombre de personnes transportées par visite ;
- déclarant dans cette même zone un curseur qui va parcourir toutes les visites ;
- chargeant le tableau dans la section `BEFORE STATEMENT` ;
- examinant le tableau dans la section `BEFORE EACH ROW` et en le comparant avec les données à insérer.

Les messages à afficher pour tracer et rendre plus lisible ce déclencheur sont :

- dans la section `BEFORE EACH ROW` : "Enregistrement du transport de *nom*" puis éventuellement " Premier trajet de la visite" ;
- dans la section `AFTER EACH ROW` : "Transport de *nom* bien enregistré" puis "Il ne reste plus que *x* place(s) disponible(s)" ;
- dans la section `AFTER STATEMENT` : "Nombre de trajet(s) traité(s) : *nombre*" ;

Les messages d'erreur à produire le cas échéant sont les suivants :

- "Capacité max atteinte *n* pour la visite *chantier* du *date*, pour le véhicule *v*" ;
- "BASE INCORRECTE : Capacité dépassée *n* pour la visite *chantier* du *date*, pour le véhicule *v*".

Le second tableau compare les outils en fonction d'autres paramètres d'utilisation.

Tableau 12-30 Autres facteurs influant les utilitaires

	Explain plan	Autotrace	Runstats	*SQL Trace*/10046
Affecté par la mise en cache	Non	Oui	Oui	Oui
Estimé ou réalisé	Estimé	Estimé	Réalisé	Réalisé
Fourni par Oracle	Oui	Oui	Non	Oui
Post-processing	Non	Non	Non	Oui (`tkprof`, `OraSRP`)
Facilité de comparaison	Difficile	Oui	Oui	Difficile

L'outil *SQL Trace/Event 10046* semble le plus polyvalent du fait de la production de plan réels. *Explain Plan* et *Autotrace* sont des solutions simples à mettre en œuvre mais toutefois moins fiables (plans prévisionnels). Les résultats de runstats peuvent fluctuer fortement (cache et version du SGBD utilisé).

Organisation des données

Cette section décrit les composants de la boîte à outils qui vous servira à optimiser vos applications. Plusieurs mécanismes peuvent être conjointement mis en œuvre : les contraintes, les index, la mise en *cluster*, le partitionnement, les vues matérialisées et la dénormalisation.

Des contraintes au plus près des données

Vous devez définir, sur vos colonnes, le maximum de contraintes d'intégrité afin de renseigner au mieux l'optimiseur. Bien que la contrainte `CHECK` ne soit pas encore utilisée par l'optimiseur, il est possible que dans le temps cette fonctionnalité soit présente.

Les colonnes NOT NULL

Le fait de déclarer des contraintes `NOT NULL` ne vous empêche pas de réaliser aussi des tests du côté de l'application. En effet, il peut être utile de vérifier qu'une valeur est présente dans un champ de saisie d'un formulaire plutôt que d'attendre d'envoyer un grand nombre d'octets au serveur qui renverra une erreur du fait d'un `NOT NULL`.

En supposant que la table `Sport` dispose de la colonne `federation` (dont les valeurs actuelles sont non nulles), le tableau suivant présente deux déclarations de contrainte `NOT NULL`.

Tableau 12-31 Déclaration de NOT NULL

Déclaration avec `CHECK`	Déclaration en ligne (*in line*)

`ALTER TABLE Sport`
`  ADD CONSTRAINT ck_federation`
`  CHECK (federation IS NOT NULL);`
	`ALTER TABLE Sport`
	`  MODIFY federation NOT NULL;`

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

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

Les colonnes UNIQUE

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

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

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

Tableau 12-32 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 multicolonnes (`nom+prenom+tel`) sera bénéfique pour les extractions dont un prédicat est basé sur le nom, le prénom et le numéro, et sur un accès aux trois colonnes simultané.

Indexation

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

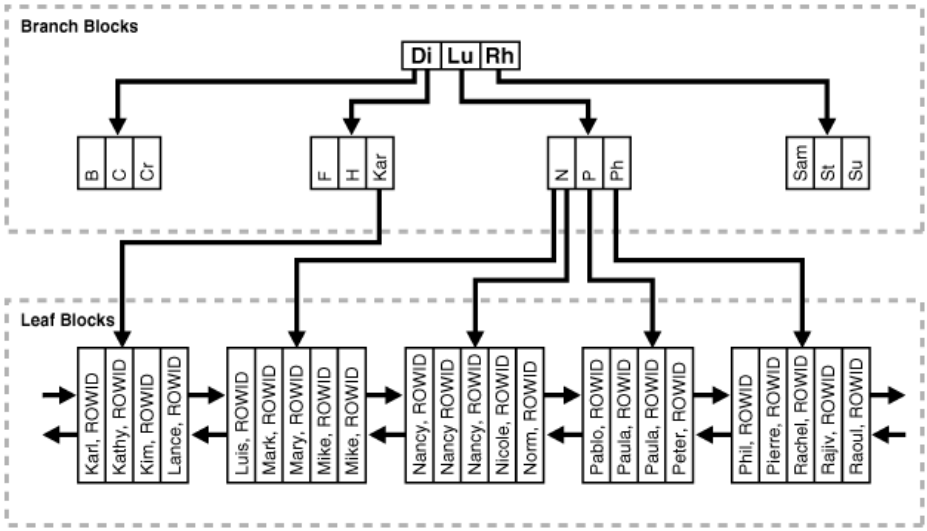
Étudions les cas d'utilisation des index d'Oracle de sorte à rendre une requête plus optimale.

Index B-tree

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

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

Figure 12-7 *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 :

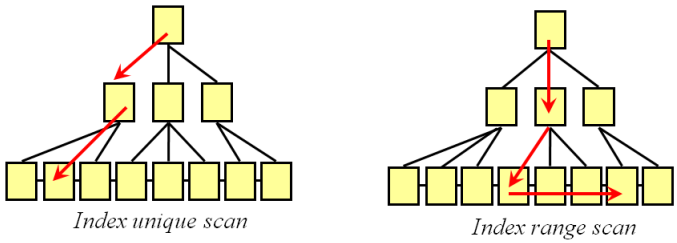
- Malgré les mises à jour de la table, ils restent équilibrés (les blocs feuilles sont au même niveau). En conséquence, quelle que soit la valeur cherchée, le temps de parcours est sensiblement identique. Les blocs intermédiaires sont remplis, en moyenne, aux trois-quarts de leur capacité.
- Les performances d'extraction, répondant à la majorité des prédicats des requêtes, sont excellentes, notamment les comparaisons d'égalité et d'intervalles.
- Les répercussions des mises à jour sont efficaces et ne se dégradent pas en fonction d'une forte augmentation de la taille des tables.

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

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

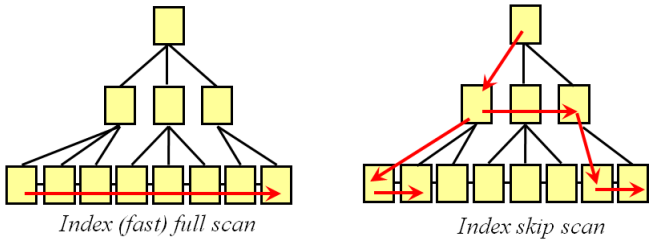
- *index unique scan* passe par la racine de l'arbre ; généralement toutes les colonnes de l'index sont concernées par une égalité dans le prédicat `WHERE`. Il s'agit en principe de la manière la plus optimale, mais qui n'est pas toujours utilisée par l'optimiseur au profit de *range scan*.
- *index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

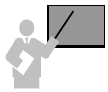
Figure 12-8 *Accès direct et par parcours par intervalles d'un index B-tree*



- *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 12-9 *Parcours séquentiel et par saut d'un index B-tree*





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

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

Tableau 12-33 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 12-34 Performances d'une extraction avec et sans index

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

Avec index		Sans index	
CIVILITE	PRENOM	NOM	TEL
Mme.	CELINE	LARRAZET	05-62-18-04-76
3 centièmes de secondes		36 secondes	
114 recursive calls		533 recursive calls	
72734 consistent gets		1442719 consistent gets	
0 sorts (memory)		48080 sorts (memory)	

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

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

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'espérez 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.

Tableau 12-35 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 Adherentbis (UPPER(nom)) TABLESPACE tbs_index;` `CREATE INDEX idx_ROUNDsolde ON Adherentbis (ROUND(solde,1)) TABLESPACE tbs_index;`	`WHERE nom='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'` *Table access full* (10229 *b* – 2795 *c*)	`WHERE UPPER(nom)='DUCLOS' AND civilite='Mr.' AND tel LIKE '+33%'` *Index range scan* (578 *b* – 107 *c*)
	`WHERE ROUND(solde,1)=9030.8` *Index range scan* (3 *b* – 3 *c*)	`WHERE solde=9030.75` *Table access full* (10229 *b* – 2795 *c*)

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

chaque type d'indexage, sont donnés la taille de l'index en Mo, le type de parcours de l'index, le nombre de blocs lus (*b*) et le coût (*c*).

Tableau 12-36 Utilisation d'index B-tree sur une colonne ayant des valeurs NULL

`SELECT COUNT(nom) FROM Adherent WHERE…`	**Condition sur la nullité**	`tel='06-81-94-44-31'`	`tel IS NOT NULL`
Sans index	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Table access full* (10228 *b* – 2796 *c*)	*Table access full* (10228 *b* – 2793 *c*)
Index *B-tree* (taille : 38 Mo) `CREATE INDEX idx_tel_btree ON Adherent (tel);`	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Index range scan* (4 *b* – 5 *c*)	*Index fast full scan* (4756 *b* – 1293 *c*)

Index fonction (taille : 0,68 Mo)
`CREATE FUNCTION f_tel_null (p_tel Adherentbis.tel%type) RETURN NUMBER DETERMINISTIC AS BEGIN IF p_tel IS NULL THEN RETURN 1; ELSE RETURN NULL; END IF; END f_tel_null;`
`CREATE INDEX idx_tel_btree ON Adherent (f_tel_null(tel));`
	`f_tel_null(tel)=1`
	Index fast full scan (28 *b* – 84 *c*)
		Table access full (10228 *b* – 2796 *c*)	Sans objet

Index composé (taille : 41 Mo) `CREATE INDEX idx_tel_btree ON Adherent (tel,0);`	`tel IS NULL` *Index range scan* (166 *b* – 76 *c*)	*Index range scan* (4 *b* – 5 *c*)	*Index fast full scan* (5150 *b* – 1399 *c*)
Index fonction NVL2 (taille : 0,62 Mo) `CREATE INDEX idx_tel_btree ON Adherent (NVL2(tel,NULL,0));`	`NVL2(tel,NULL,0)=0` *Index fast full scan* (74 *b* – 21 *c*)	*Table access full* (10228 *b* – 2796 *c*)	Sans objet
Index unique (taille : 37 Mo) `CREATE UNIQUE INDEX idx_tel_btree ON Adherentbis (tel);`	`tel IS NULL` *Table access full* (10228 *b* – 2793 *c*)	*Index unique scan* (4 *b* – 3 *c*)	*Index fast full scan* (4599 *b* – 1250 *c*)

Les index les plus performants sont :

- Pour répondre au prédicat `IS NULL`, ceux qui utilisent une fonction (déterministe pour l'un et `NVL2` pour l'autre).
- Pour répondre au prédicat `col=valeur`, l'index classique, unique ou composé, qui offre les mêmes résultats.
- Pour répondre au prédicat `IS NOT NULL`, l'index unique.