

Référence

Optimisation des bases de données

Mise en œuvre sous Oracle

Laurent Navarro

Réseaux
et télécom

Programmation

Développement
web

Sécurité

Système
d'exploitation



PEARSON

Optimisation des bases de données

Mise en œuvre sous Oracle

Laurent Navarro

Avec la contribution technique
d'Emmanuel Lecoester

PEARSON

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4156-3

Copyright © 2010 Pearson Education France

Tous droits réservés

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

Table des matières

Préface	IX
À propos de l'auteur	XI
Introduction	1
1 Introduction aux SGBDR	5
1.1 Qu'est-ce qu'une base de données ?	5
1.1.1 Système de gestion des bases de données	6
1.2 Modèle de stockage des données	7
1.2.1 Organisation des données	7
1.2.2 Le RowID	10
1.2.3 Online Redo Log et Archived Redo Log	10
1.2.4 Organisation des tables	11
1.2.5 <i>Row Migration</i> et <i>Row Chaining</i>	12
1.2.6 Le cache mémoire	13
1.3 Intérêt des index dans les SGBDR	14
1.4 Analyse du comportement du SGBDR	15
1.4.1 Exécution d'une requête SQL	15
1.4.2 Optimiseur CBO (<i>Cost Based Optimizer</i>)	16

Axe 1

Étude et optimisation du modèle de données

2 Modèle relationnel	21
2.1 Présentation	21
2.2 Les bons réflexes sur le typage des données	25
2.2.1 Les types sous Oracle	28

2.2.2	Les types sous SQL Server.....	30
2.2.3	Les types sous MySQL.....	31
3	Normalisation, base du modèle relationnel	33
3.1	Normalisation	33
3.1.1	Première forme normale (1NF).....	34
3.1.2	Deuxième forme normale (2NF).....	35
3.1.3	Troisième forme normale (3NF)	36
3.1.4	Forme normale de Boyce Codd (BCNF)	37
3.1.5	Autres formes normales	38
3.2	Dénormalisation et ses cas de mise en œuvre.....	38
3.2.1	La dénormalisation pour historisation	38
3.2.2	La dénormalisation pour performance et simplification en environnement OLTP	40
3.2.3	La dénormalisation pour performance en environnement OLAP	42
3.3	Notre base de test.....	43

Axe 2

Étude et optimisation des requêtes

4	Méthodes et outils de diagnostic	49
4.1	Approche pour optimiser	49
4.1.1	Mesurer.....	50
4.1.2	Comprendre le plan d'exécution	55
4.1.3	Identifier les requêtes qui posent des problèmes.....	61
4.2	Outils complémentaires	64
4.2.1	Compteurs de performance Windows.....	64
4.2.2	SQL Tuning Advisor (Oracle)	65
4.2.3	SQL Access Advisor (Oracle).....	66
4.2.4	<i>SQL Trace</i> (Oracle).....	67
4.2.5	Outils SQL Server.....	72
4.2.6	Outils MySQL	73

5	Techniques d'optimisation standard au niveau base de données	77
5.1	Statistiques sur les données	77
5.1.1	Ancienne méthode de collecte	78
5.1.2	Nouvelle méthode de collecte	78
5.1.3	Sélectivité, cardinalité, densité	80
5.2	Utilisation des index	85
5.2.1	Index B*Tree	86
5.2.2	Index sur fonction	99
5.2.3	Reverse Index	100
5.2.4	Index bitmap	101
5.2.5	Bitmap Join Index	108
5.2.6	Full Text Index	111
5.3	Travail autour des tables	118
5.3.1	Paramètres de table	118
5.3.2	Index Organized Table	122
5.3.3	Cluster	132
5.3.4	Partitionnement des données	135
5.3.5	Les vues matérialisées	146
5.3.6	Reconstruction des index et des tables	148
6	Techniques d'optimisation standard des requêtes	153
6.1	Réécriture des requêtes	153
6.1.1	Transformation de requêtes	154
6.1.2	IN versus jointure	155
6.1.3	Sous-requêtes versus anti-jointures	156
6.1.4	Exists versus <i>Count</i>	158
6.1.5	<i>Exists</i> versus <i>IN</i>	159
6.1.6	Clause <i>Exists</i> * versus constante	160
6.1.7	Expressions sous requêtes	161
6.1.8	Agrégats : <i>Having</i> versus <i>Where</i>	162
6.2	Bonnes et mauvaises pratiques	163
6.2.1	Mélange des types	163
6.2.2	Fonctions et expressions sur index	163
6.2.3	Impact de l'opérateur <> sur les index	164
6.2.4	Réutilisation de vue	164

6.2.5	Utilisation de tables temporaires.....	165
6.2.6	Utilisation abusive de <i>SELECT *</i>	165
6.2.7	Suppression des tris inutiles.....	166
6.2.8	Utilisation raisonnée des opérations ensemblistes	166
6.2.9	Union versus <i>Union ALL</i>	169
6.2.10	<i>Count(*)</i> versus <i>count(colonne)</i>	170
6.2.11	Réduction du nombre de parcours des données	171
6.2.12	LMD et clés étrangères.....	172
6.2.13	Suppression temporaire des index et des CIR	173
6.2.14	<i>Truncate</i> versus <i>Delete</i>	173
6.2.15	Impacts des verrous et des transactions.....	173
6.2.16	Optimisation du <i>COMMIT</i>	175
6.2.17	DBLink et vues.....	176
7	Techniques d'optimisation des requêtes avancées	177
7.1	Utilisation des hints sous Oracle	177
7.1.1	Syntaxe générale	179
7.1.2	Les hints <i>Optimizer Goal</i>	179
7.1.3	Les hints <i>Access Path</i>	180
7.1.4	Les hints <i>Query Transformation</i>	181
7.1.5	Les hints de jointure.....	182
7.1.6	Autres hints.....	183
7.2	Exécution parallèle	183
7.2.1	Les hints de parallélisme.....	185
7.3	Utilisation du SQL avancé.....	186
7.3.1	Les <i>Grouping Sets</i>	186
7.3.2	<i>Rollup Group By</i>	188
7.3.3	<i>Cube Group By</i>	189
7.3.4	Utilisation de <i>WITH</i>	191
7.3.5	Les fonctions de classement (<i>ranking</i>).....	192
7.3.6	Autres fonctions analytiques.....	194
7.3.7	L'instruction <i>MERGE</i>	195
7.3.8	Optimisation des updates multitables	196
7.3.9	Insertion en mode <i>Direct Path</i>	198

7.4	PL/SQL.....	198
7.4.1	Impact des triggers	198
7.4.2	Optimisation des curseurs (<i>BULK COLLECT</i>)	200
7.4.3	Optimisation du LMD (<i>FORALL</i>).....	203
7.4.4	SQL dynamique et <i>BULK</i>	208
7.4.5	Traitement des exceptions avec <i>FORALL</i>	209
7.4.6	Utilisation de cache de données	210
7.4.7	Utilisation du profiling	212
7.4.8	Compilation du code PL/SQL	212

Axe 3

Autres pistes d'optimisation

8	Optimisation applicative (hors SQL)	217
8.1	Impact du réseau sur le modèle client/serveur.....	217
8.2	Regroupement de certaines requêtes	217
8.3	Utilisation du binding.....	219
8.4	Utilisation de cache local à l'application.....	221
8.5	Utilisation du SQL procédural.....	221
8.6	Gare aux excès de modularité.....	221
9	Optimisation de l'infrastructure	223
9.1	Optimisation de l'exécution du SGBDR.....	223
9.1.1	Ajustement de la mémoire utilisable.....	223
9.1.2	Répartition des fichiers	224
9.2	Optimisation matérielle.....	224
9.2.1	Le CPU	224
9.2.2	La mémoire vive (RAM).....	225
9.2.3	Le sous-système disque.....	225
9.2.4	Le réseau	226
	Conclusion	227

Annexes

A	Gestion interne des enregistrements.....	231
A.1	Le RowID.....	231
A.2	Row Migration et Row Chaining	232
B	Statistiques sur les données plus en détail.....	235
B.1	Statistiques selon l'ancienne méthode de collecte	235
B.2	Statistiques selon la nouvelle méthode de collecte	236
B.3	Histogrammes.....	240
B.4	Facteur de foisonnement (<i>Clustering Factor</i>)	241
C	Scripts de création des tables de test BIGEMP et BIGDEPT.....	245
D	Glossaire.....	249
	Index.....	251

Préface

L'optimisation des applications est un sujet bien vaste, souvent objet des débats d'experts et générateur de quiproquos notamment sur les causes générant les effets constatés (lenteur d'affichage, fort trafic réseau, saturation du serveur de données).

L'optimisation se présente sous de multiples facettes :

- Tantôt, elle est purement applicative et touche à la méthode de programmation.
- Tantôt, les dérives proviennent d'un manque au niveau de la modélisation des données, plus généralement d'un problème d'index ou de paramétrage du serveur de données.
- Parfois même, l'optimisation est tout simplement liée à une limite de l'architecture physique du système.

Dans chacun de ces trois cas de figure majeurs, il est important de mesurer ce manque d'optimisation et de choisir les métriques les plus discriminantes afin de pouvoir mesurer les bénéfices exacts de l'optimisation apportée.

Dans cet ouvrage, Laurent Navarro apporte un lot de réponses concrètes et détaillées au développeur d'application, dans une vision résolument ciblée sur l'accès aux données. J'insiste bien sur ce point : ce livre n'est pas un catalogue des techniques d'optimisation propres à chaque serveur de données destiné aux administrateurs ou autres experts mais bien un premier pas vers une sensibilisation des développeurs de tous bords aux contacts avec un serveur de données.

Ce livre parcourt les principales techniques d'optimisation disponibles pour le développeur d'application : modèle de données, techniques standard d'accès aux données jusqu'à des techniques très avancées permettant d'exploiter au mieux les possibilités offertes par les principaux éditeurs de bases de données du marché.

Emmanuel Lecoester

*Responsable des rubriques SGBD & WinDev
de developpez.com*

À propos de l'auteur

Adolescent, je m'amusais avec des bases de données DBase II & III, puis j'ai débuté ma carrière dans l'industrie avec des bases en fichiers partagés Paradox. La nature des applications a évolué, et en 1994 j'ai commencé à travailler sur des bases Oracle (Version 7). À l'époque, cela nécessitait un serveur Unix et un DBA bardé de certifications. Au fil du temps, les SGBDR client/serveur se sont démocratisées. Des alternatives au leader sont apparues, contribuant pour une grande part à cette démocratisation. C'est ainsi qu'en 2000 j'ai fait la connaissance de MySQL (3.2 à l'époque) pour ma première application web développée en PHP. Puis j'ai eu l'occasion d'utiliser d'autres bases, telles que SQL Server de Microsoft, Firebird la version open-source d'Interbase ou encore PostgreSQL, autre alternative open-source qui ne rencontre malheureusement pas le succès qu'elle mériterait.

Parallèlement, les clients aussi se sont diversifiés, et je constatais que le périmètre d'utilisation des bases de données s'étendait. Au début réservé à l'informatique de gestion, les bases de données sont petit à petit apparues dans les secteurs de l'informatique industrielle et de l'informatique mobile. Ces secteurs n'étant pas forcément coutumiers de ces technologies, ils ont fait appel à des gens comme moi pour les conseiller dans leur évolution.

Aimant varier les plaisirs, outre les bases de données, je développe en C/C++, parfois en environnement contraint du point de vue des performances. C'est probablement de là que me vient cette curiosité qui me pousse à comprendre comment fonctionnent les choses et qui a pour suite naturelle l'optimisation. Ajoutez à cela le plaisir de partager ses connaissances et vous comprendrez comment est né ce livre, qui j'espère vous aidera dans votre travail.

Laurent Navarro, développeur d'applications, est basé à Toulouse et travaille depuis quinze ans avec des bases de données Oracle mais aussi quelques autres (SQL Server, MySQL, etc.).

*Il anime des formations SQL, PL/SQL, Pro*C et Tuning SQL depuis une dizaine d'années.*

Introduction

Pourquoi ce livre, à qui s'adresse-t-il ?

L'optimisation de bases de données est un problème à la fois simple et compliqué.

J'interviens souvent auprès de structures qui n'ont aucune notion d'optimisation et, dans ces cas-là, les choses sont plutôt simples car l'application de principes de base d'optimisation améliore rapidement la situation. En revanche, lorsque ces principes de base ont déjà été appliqués, la tâche peut être plus ardue...

L'objectif de cet ouvrage est de fournir les bases de l'optimisation. Il explique ce qui se passe dans la base de données afin que vous puissiez comprendre ce qui se passe sous le capot de votre SGBDR et, ainsi, vous permettre de choisir des solutions d'optimisation en ayant conscience des impacts de vos choix.

Il existe déjà des livres couvrant ce domaine, mais la plupart sont en anglais et s'adressent à des DBA (administrateurs de base de données). Ces ouvrages sont soit tellement pointus qu'un non-expert peut s'y noyer, soit tellement légers qu'on ne comprend pas forcément pourquoi les choses sont censées s'améliorer. Cela rend délicate la transposition à votre environnement. J'espère avoir trouvé un juste milieu avec cet ouvrage.

Pour moi, il est primordial d'aborder le problème de l'optimisation dès la conception d'un logiciel. Il est donc nécessaire que les développeurs s'approprient ces compétences plutôt que de demander aux DBA de trouver des solutions après la mise en production.

Ce livre s'adresse donc en premier lieu aux équipes de développement – développeurs, chefs de projet, architectes – mais aussi, bien sûr, aux DBA qui seront toujours les interlocuteurs naturels des équipes de développement dès qu'une base de données sera utilisée.

Quels sont les prérequis ?

Pour lire cet ouvrage, il vous suffit de savoir ce qu'est une base de données relationnelle.

Une connaissance du SQL sera un plus.

Quel est le périmètre de ce livre ?

Ce livre couvre les cas les plus fréquents, c'est-à-dire des applications ayant au plus quelques gigaoctets de données sur des serveurs classiques. Il ne couvre donc pas des sujets comme l'utilisation de GRID, de RAC ou des bases OLAP, même si de nombreux thèmes abordés ici sont présents dans ce type d'application aussi.

Cet ouvrage se focalise sur l'optimisation autour du développement. Les optimisations au niveau de l'infrastructure de la base de données ne seront que très peu abordées. D'autres livres plus orientés sur l'exploitation traitent déjà ce sujet bien mieux que je ne saurais le faire.

Organisation du livre

Cet ouvrage s'articule autour de neuf chapitres, regroupés dans trois parties :

- Le **Chapitre 1** est une introduction visant à présenter brièvement quelques mécanismes internes des SGBDR qui seront nécessaires à la compréhension des chapitres suivants.

La première partie concerne le premier axe d'optimisation : celle du modèle de données.

- Le **Chapitre 2** rappelle ce qu'est le modèle relationnel et traite les problématiques liées au typage des données.
- Le **Chapitre 3** traite de l'intérêt de la normalisation des bases de données et de l'utilité que peut présenter la dénormalisation appliquée à bon escient.

La deuxième partie concerne le deuxième axe d'optimisation : celle des requêtes.

- Le **Chapitre 4** présente les méthodes et l'outillage. Il traite des différents outils à votre disposition pour analyser les situations et explique comment comprendre les résultats.

- Le **Chapitre 5** décrit l'ensemble des objets d'optimisation et le cadre de leur utilisation. Il décrit les différents types d'index et d'organisation des tables ainsi que leurs cas d'usage.
- Le **Chapitre 6** présente les impacts des variantes d'écriture et décrit quelques bonnes et mauvaises pratiques.
- Le **Chapitre 7** introduit les hints qu'il faut utiliser avec parcimonie, un ensemble de techniques SQL avancées ainsi que des optimisations du PL/SQL.

La troisième partie contient d'autres axes d'optimisations.

- Le **Chapitre 8** traite des optimisations applicatives autres que l'optimisation des requêtes elles-mêmes.
- Le **Chapitre 9** aborde brièvement quelques optimisations envisageables au niveau de l'infrastructure.

Ressources en ligne

Afin d'illustrer les concepts présentés tout au long de ce livre, nous allons les appliquer à une base de test.

Cette base (structure et données) est disponible pour les SGBDR Oracle, SQL Server et MySQL sur le site de l'auteur, à l'adresse <http://www.altidev.com/livres.php>.

Remerciements

Merci à ma femme Bénédicte et mon fils Thomas pour leur soutien et leur patience durant l'écriture de ce livre. Merci à ma mère Ginette pour ses relectures. Merci à mes relecteurs, Emmanuel et Nicole. Merci à mon éditeur, Pearson France, et particulièrement à Patricia pour m'avoir fait confiance et à Amandine pour ses précieux conseils. Merci à l'équipe d'Iris Technologies pour m'avoir donné l'idée de ce livre en me permettant d'animer des formations sur ce sujet.

Introduction aux SGBDR

Pour optimiser une base Oracle, il est important d'avoir une idée de la manière dont elle fonctionne. La connaissance des éléments sous-jacents à son fonctionnement permet de mieux comprendre ses comportements. C'est pourquoi nous commencerons, à ce chapitre, par vous présenter ou vous rappeler brièvement ce qu'est un SGBDR et quelques-uns des éléments qui le composent. Ces notions vous seront utiles tout au long du livre. Le but ici n'est pas de faire de vous un spécialiste du fonctionnement interne d'Oracle, mais de vous donner quelques explications sur des concepts que nous référencerons ultérieurement.

Si vous êtes DBA, vous pouvez vous rendre directement au Chapitre 2.

1.1 Qu'est-ce qu'une base de données ?

Une base de données est un ensemble d'informations structurées. Elle peut être de nature :

- hiérarchique ;
- relationnelle ;
- objet ;
- documentaire ;
- ...

Actuellement, le marché est principalement composé de bases de données relationnelles avec, parfois, une extension objet. Cet ouvrage traite exclusivement de ce type de bases.

Dans les bases de données relationnelles, les données sont organisées dans des tables à deux dimensions, conformément au modèle relationnel que nous étudierons au Chapitre 2.

1.1.1 Système de gestion des bases de données

Un SGBDR (système de gestion de base de données relationnelle) est un système (logiciel) qui permet de gérer une base de données relationnelle. Les SGBDR peuvent être soit de type client/serveur (Oracle, SQL Server, MySQL, PostgreSQL, etc.), soit de type fichiers partagés (Access, SQL Server CE, Paradox, dBase, etc.). Dans le milieu professionnel, on retrouve principalement des SGBDR client/serveur même si les solutions en fichiers partagés ont eu leur heure de gloire et sont encore utilisées dans certaines applications. L'apparition de SGBDR client/serveur gratuits a fortement contribué à populariser ce modèle ces dernières années. Le modèle client/serveur nécessite généralement la présence d'un serveur, qui traite les requêtes transmises par le client et lui retourne le résultat. Le principal intérêt d'un SGBDR est qu'il va fournir les services suivants (tous les SGBDR ne proposent pas tous ces services) :

- implémentation du langage d'interrogation des données SQL (*Structured Query Language*) ;
- gestion des structures des données et de leur modification ;
- gestion de l'intégrité des données ;
- gestion des transactions ;
- gestion de la sécurité, contrôle d'accès ;
- abstraction de la plateforme matérielle et du système d'exploitation sous-jacent ;
- du point de vue logique, abstraction de l'organisation du stockage sous-jacent ;
- tolérance aux pannes et administration du système ;
- répartition de la charge.

1.2 Modèle de stockage des données

1.2.1 Organisation des données

Les tables sont les objets logiques de base du modèle relationnel. Or, un système d'exploitation ne connaît que la notion de fichiers. Le SGBDR permet de faire le lien entre la représentation logique et le stockage physique dans les fichiers. Nous allons voir comment en étudiant le SGBDR Oracle.

Les objets logiques (tables, index, etc.) sont stockés dans des espaces logiques appelés "tablespaces". Une base de données est constituée de plusieurs tablespaces, lesquels sont composés d'un ou de plusieurs fichiers appelés "datafiles". Ces fichiers peuvent se trouver sur des disques différents.

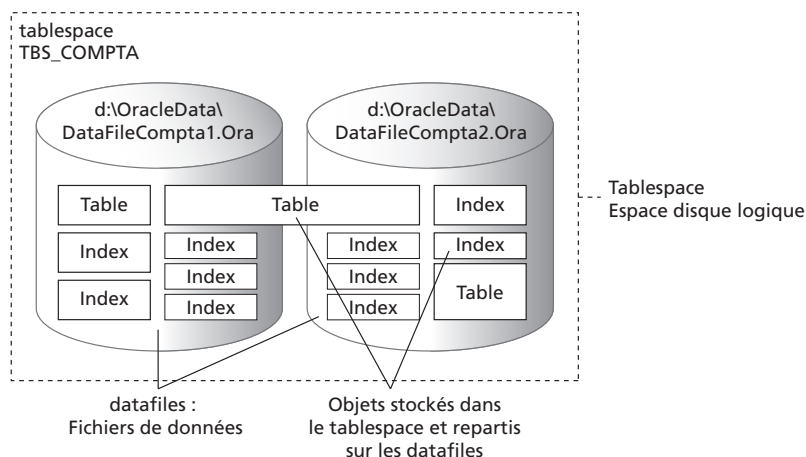


Figure 1.1

Organisation des objets dans un tablespace composé de deux datafiles.

Les objets sont attachés à un seul tablespace mais ils peuvent, par contre, être répartis sur plusieurs datafiles (voir Figure 1.1). Il n'y a aucun moyen d'influer sur la localisation des objets au niveau des datafiles, il est seulement possible de définir le tablespace associé à un objet. (Les objets partitionnés peuvent, eux, être attachés à plusieurs tablespaces. Nous les étudierons au Chapitre 5, section 5.3.4, "Partitionnement des données".)

Les datafiles, et donc les tablespaces, sont constitués de blocs de données (*data block*), dont la taille est configurée à la création du tablespace. Ce paramètre varie généralement entre 2 Ko et 16 Ko et, par défaut, est de 4 ou 8 Ko suivant la plateforme

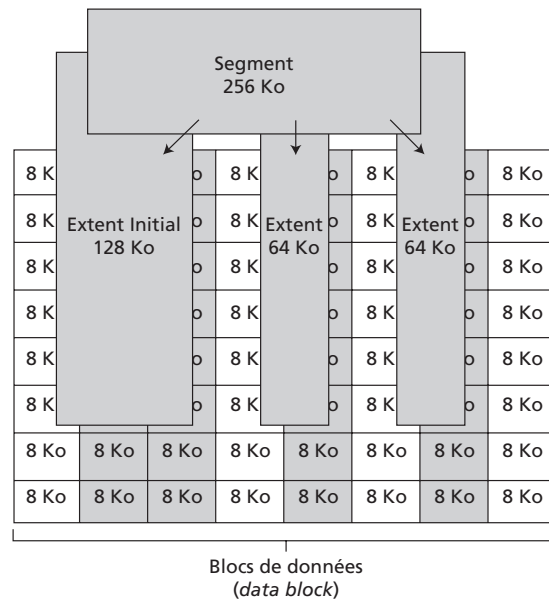
(au long de l'ouvrage, nous retiendrons une taille de 8 Ko qui est une valeur assez communément utilisée).

Chaque objet ou constituant d'objet ayant besoin de stockage s'appelle un "segment". Par exemple, pour une table classique, il y a un segment pour la table elle-même, un segment pour chacun de ses index et un segment pour chacun de ses champs LOB (voir Chapitre 2, section 2.2.1, "Les types sous Oracle").

Quand un segment a besoin d'espace de stockage, il alloue un *extent*, c'est-à-dire un ensemble de blocs de données contigus. Un segment est donc composé d'un ensemble d'extents qui n'ont pas forcément tous la même taille. (Les clauses *INITIAL* et *NEXT* spécifiées dans l'instruction de création de l'objet associé au segment permettent d'influer sur ces paramètres.) La Figure 1.2 présente la décomposition hiérarchique d'un segment en blocs de données. Il est à noter que, si un segment n'a plus besoin de l'espace qu'il a précédemment alloué, il ne le libère pas nécessairement.

Figure 1.2

*Décomposition
d'un segment en extents
puis en blocs.*



MS SQL Server

Sous SQL Server, la logique est relativement proche, sauf qu'un niveau supplémentaire est intégré, le niveau base de données (*database*). À la différence d'Oracle, une instance SQL Server contient plusieurs bases de données, qui ont chacune leurs espaces logiques nommés *data file group* (équivalents des tablespaces) eux-mêmes composés de *data file*.

MySQL

Sous MySQL, l'organisation du stockage est déléguée au moteur de stockage. Plusieurs moteurs sont supportés, MyISAM et InnoDB le plus fréquemment.

Le moteur MyISAM attribue plusieurs fichiers à chaque table.

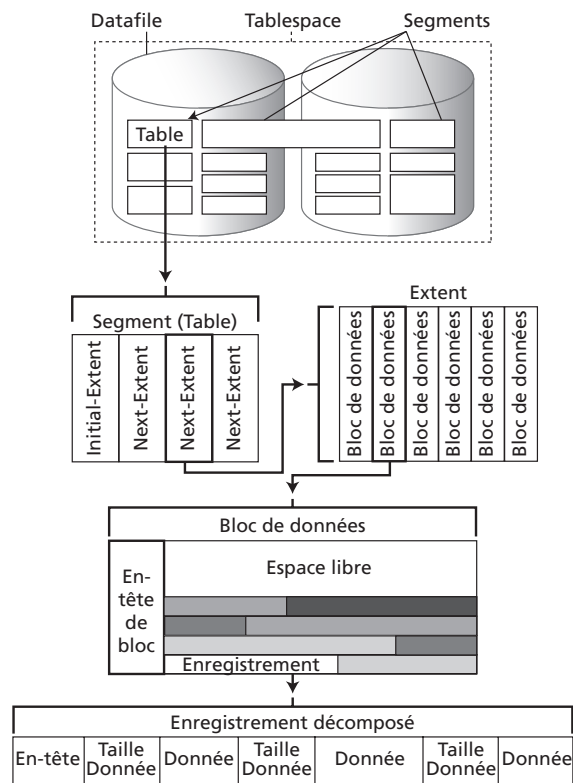
- Un fichier `.frm` pour décrire la structure de la table. Ce fichier est géré par MySQL et non pas par le moteur de stockage. Un fichier `.myd` qui contient les données.
- Un fichier `.myi` qui contient les index.

Le moteur InnoDB implémente le concept de tablespace qui contient tous les éléments de la base de données (tables, index, etc.). Les fichiers `.frm` gérés par MySQL sont présents en plus du tablespace.

Les blocs de données contiennent les enregistrements (lignes ou *row*) et des structures de contrôle. La Figure 1.3 montre un schéma complet de l'organisation du stockage des données.

Figure 1.3

Schéma illustrant le stockage des données dans une base Oracle.



1.2.2 Le RowID

Le RowID (identifiant de ligne) est une information permettant d'adresser directement un enregistrement, sans avoir à parcourir aucune liste. C'est une sorte de pointeur. Il est composé de :

- l'identifiant de l'objet ;
- l'identifiant du datafile ;
- l'identifiant du bloc dans le datafile ;
- l'identifiant de la ligne dans le bloc.

Cette information sera rarement manipulée directement par vos applications. Elle est surtout destinée à un usage interne au SGBDR. Cependant, dans certains cas, il peut être intéressant d'y recourir depuis une application, par exemple pour désigner un enregistrement n'ayant pas de clé primaire ou pour adresser plus rapidement un enregistrement pour lequel vous avez récupéré le RowID. Le RowID d'un enregistrement s'obtient en interrogeant la pseudo-colonne `rowid`.

```
SQL> select nocmd, noclient, datecommande, rowid from cmd;
      NOCMD      NOCLIENT DATECOMMANDE ROWID
-----
      14524      106141 16/04/2004 AAARnUAAGAAASIfAAG
      14525      131869 16/04/2004 AAARnUAAGAAASIfAAH
      14526       83068 16/04/2004 AAARnUAAGAAASIfAAI
      14527     120877 16/04/2004 AAARnUAAGAAASIfAAJ
      14528       34288 16/04/2004 AAARnUAAGAAASIfAAK
      14529     103897 16/04/2004 AAARnUAAGAAASIfAAL
6 rows selected
```

Voir l'Annexe A, section A.1, pour plus d'informations sur les RowID.

1.2.3 Online Redo Log et Archived Redo Log

Les fichiers Online Redo Log sont des fichiers binaires qui contiennent toutes les modifications appliquées aux datafiles récemment. Ils permettent notamment de réparer les datafiles en cas d'arrêt brutal de la base.

Les fichiers Archived Redo Log sont des fichiers Redo Log archivés. Ils sont créés uniquement si le mode *archivelog* est actif. Ils permettent de réparer les datafiles, de compléter une restauration, de maintenir une base de secours en attente (*stand by database*) et sont aussi nécessaires pour certaines fonctions Oracle Warehouse.

Leur gestion a un coût en termes d'espace disque utilisé, donc si vous n'en avez pas besoin, désactivez cette fonction.

MS SQL Server

Les fichiers LDF, qui sont les journaux de transactions, jouent un rôle analogue aux Online Redo Log.

MySQL

Le moteur InnoDB intègre un mécanisme analogue aux Online Redo Log. Le moteur MyISAM n'a pas d'équivalent.

1.2.4 Organisation des tables

Les données sont organisées dans des tables à deux dimensions : les *colonnes*, ou champs, qui ne sont pas supposées évoluer différemment du modèle de données, et les *lignes*, enregistrements qui varient au fil du temps quand des données sont ajoutées ou modifiées. Par défaut, dans de nombreux SGBDR, la structure de stockage adoptée est une structure dite de "tas" (*heap*).

Ce type de structure stocke les enregistrements en vrac, sans ordre particulier, dans une zone de données. Quand un enregistrement est détruit, il libère de l'espace qu'un autre enregistrement pourra éventuellement réutiliser.

Sous Oracle, en mode MSSM (*Manual Segment Space Management*), les blocs contenant de l'espace libre, c'est-à-dire dont le pourcentage d'espace libre est supérieur au paramètre PCTFREE de la table, sont listés dans une zone de la table nommée FREELIST. Les blocs dans lesquels de l'espace a été libéré à la suite de suppressions d'enregistrements et dont le pourcentage d'espace utilisé repasse en dessous de la valeur du paramètre PCTUSED sont, eux aussi, répertoriés dans la FREELIST.

L'organisation des tables sous forme d'index est assez commune, les enregistrements sont stockés dans l'ordre de la clé primaire. On dit alors que la table est organisée en index (IOT, *Index Organized Table*). Nous étudierons ce type de table au Chapitre 5, section 5.3.2, "Index Organized Table".

MS SQL Server

Les tables sans index clustered sont organisées en tas, celles ayant un index clustered ont une organisation de type IOT.

MySQL

Sous MySQL, les tables utilisant le moteur MyISAM sont organisées en tas, celles utilisant le moteur InnoDB ont une organisation de type IOT.

1.2.5 Row Migration et Row Chaining

La migration d'enregistrement (*Row Migration*) est le mécanisme qui déplace un enregistrement qui ne tient plus dans son bloc, après une mise à jour de ses données ayant entraîné un accroissement de sa taille. Lors de la migration, le SGBDR insère un pointeur à l'emplacement initial de l'enregistrement qui désigne le nouvel emplacement que le SGBDR alloue pour y placer les données. Ainsi, on pourra toujours accéder à l'enregistrement en utilisant le RowID initial qui reste valide. Cependant, il faut noter que l'utilisation de ce pointeur sera source d'E/S (entrées/sorties) supplémentaires. En effet, l'accès à un enregistrement au moyen de son RowID se fait habituellement en *une* lecture de bloc, alors que celui à un enregistrement ayant migré nécessite de lire *deux* blocs (celui qui est pointé par le RowID plus celui qui est désigné par le pointeur). Afin de limiter l'apparition de ce phénomène, Oracle n'insère des lignes dans un bloc que si un certain pourcentage d'espace libre demeure à l'issue de cette opération (paramètre PCTFREE de la table ; par défaut, il vaut 10 %).

Le chaînage d'enregistrement (*Row Chaining*) est le mécanisme qui gère le fait qu'un enregistrement ne peut pas tenir dans un unique bloc de données (8 Ko par défaut) car sa taille est supérieure. Le processus va scinder les données et les répartir dans plusieurs blocs en plaçant dans chaque bloc un pointeur vers les données situées dans le bloc suivant. Comme lors de la migration d'enregistrement, ce mécanisme va causer des E/S supplémentaires.

Voir l'Annexe A, section A.2, pour plus d'informations sur ces mécanismes.

1.2.6 Le cache mémoire

Les accès disque sont très lents comparés aux accès en mémoire vive (RAM). Approximativement et en simplifiant : un disque dur a un temps d'accès qui se compte en millisecondes, alors que celui de la mémoire se compte en nanosecondes. La mémoire vive est donc un million de fois plus rapide que les disques durs.

Par ailleurs, la quantité de mémoire vive sur les serveurs étant de plus en plus importante, une optimisation assez évidente consiste à implémenter un système de cache mémoire dans les SGBDR.

Le but du cache mémoire est de conserver en mémoire une copie des informations les plus utilisées afin de réduire les accès au disque dur. Sous Oracle, ces caches mémoire se trouvent dans SGA (*System Global Area*), une zone mémoire principale, qui englobe diverses zones, parmi lesquelles :

- **Database Buffer Cache.** Contient les blocs manipulés le plus récemment.
- **Shared Pool.** Contient les requêtes récentes ainsi que le plan d'exécution qui leur est associé. Cette zone contient aussi un cache du dictionnaire des données.

Le buffer cache contient une copie mémoire des blocs les plus manipulés afin de réduire le nombre d'accès disque. Cela aura pour effet d'améliorer notablement les performances. Dans de nombreux cas, la quantité de mémoire disponible pour le buffer cache sera plus faible que celle des données manipulées. De fait, il faudra choisir quelles sont les données à conserver dans le cache et quelles sont celles qui doivent céder leur place à de nouvelles données. Ce choix sera fait par un algorithme de type MRU (*Most Recently Used*) qui privilégiera le maintien en mémoire des blocs les plus utilisés récemment. Afin d'éviter que le parcours d'une grosse table entraîne un renouvellement complet du cache, l'accès à ce type de tables est pondéré défavorablement au profit des petites tables, pondérées, elles, positivement. Ces dernières resteront plus longtemps dans le cache, alors que, généralement, les grosses tables n'y seront pas chargées entièrement.

MS SQL Server

Sous SQL Server, on retrouve un mécanisme tout à fait similaire.

MySQL

Sous MySQL, les choses sont un peu différentes. On retrouve bien un mécanisme de cache, mais d'une nature différente. Le cache de requête permet de garder en cache le résultat des requêtes précédemment exécutées.

Le cache mémoire correspondant au buffer cache d'Oracle est, lui, de la responsabilité du moteur de stockage :

- Le moteur MyISAM n'en a pas. Il part du principe que le système d'exploitation en possède un et qu'il sera tout aussi efficace d'utiliser celui-là. Cette solution est un peu moins intéressante mais elle a l'avantage de simplifier le moteur de stockage.
- Le moteur InnoDB intègre, lui, un cache mémoire similaire, sur le principe au buffer cache d'Oracle.

1.3 Intérêt des index dans les SGBDR

Nous avons vu précédemment que, dans une table organisée en tas, les données n'ont pas d'ordre particulier. Lorsque le SGBDR cherche une information suivant un critère particulier (on parle de clé de recherche), il n'a pas d'autre choix que de parcourir l'ensemble des enregistrements pour trouver ceux qui répondent au critère demandé. Vous imaginez aisément que, lorsqu'il y a beaucoup d'enregistrements, cette méthode n'est pas très intéressante. C'est pour répondre plus efficacement à ce genre de besoin que la notion d'index a été introduite dans les SGBDR. Un index de base de données ressemble un peu à l'index de ce livre. Il contient une liste ordonnée de mots et une référence vers la page qui les contient. De plus, l'index est bien plus petit que le livre.

Si nous retranscrivons cela en termes de base de données, un index contient la clé de recherche et une référence vers l'enregistrement. Son principal avantage est qu'il est ordonné. Cela permet d'y appliquer des techniques de recherche par dichotomie, beaucoup plus efficaces que les recherches linéaires dès que le nombre d'enregistrements s'élève.

Les index sont intéressants avec les tables organisées en tas. Cependant, le besoin est le même avec les tables organisées suivant des index (IOT) qui sont déjà triées. En effet, si l'ordre de la table n'est pas celui de la clé de recherche, le problème reste entier. Par exemple, si vous gérez une liste de personnes triée par numéro de Sécurité sociale, lorsque vous faites une recherche sur le nom, vous n'êtes pas plus avancé que la liste soit triée par numéro de Sécurité sociale ou pas triée du tout.

L'intérêt des index est donc indépendant du type d'organisation de table. Nous étudierons en détail au Chapitre 5, section 5.2.1, "Index B*Tree", les index B*Tree, les plus communs. Par la suite, nous aborderons les principaux types d'index disponibles.

1.4 Analyse du comportement du SGBDR

1.4.1 Exécution d'une requête SQL

Toute requête SQL envoyée au SGBDR suit le même parcours afin d'être exécutée. Nous allons étudier ici ce cheminement afin de comprendre comment le SGBDR passe d'une requête SQL à un ensemble de données résultat.

Il faut bien avoir en tête que le SQL, contrairement à la plupart des langages informatiques, ne décrit pas la façon de déterminer le résultat mais qu'il permet d'exprimer un résultat attendu. Ainsi, il n'explique pas comment manipuler les tables, mais les liens qu'il y a entre elles et les prédicats à appliquer sur les données. Le SGBDR doit trouver le meilleur moyen de répondre à une requête SQL.

Étape 1 : Parsing et traduction en langage interne

Cette étape consiste à interpréter le texte de la requête (*parsing*) et à effectuer des contrôles syntaxiques. Le SGBDR vérifie si la requête respecte la grammaire d'une requête SQL et si les mots clés sont bien placés. Cette étape inclut aussi les contrôles sémantiques, c'est-à-dire que le SGBDR vérifie si la requête manipule bien des tables et des colonnes qui existent et si l'utilisateur possède les droits permettant de faire ce qui est demandé dans la requête.

Étape 2 : Établissement d'un plan d'exécution

Le plan d'exécution est la façon dont le SGBDR parcourt les différents ensembles de données afin de répondre à la requête. Nous étudierons au Chapitre 4, section 4.1.2, "Comprendre le plan d'exécution", les éléments primaires constituant un plan d'exécution.

Lors de cette étape, le SGBDR établit plusieurs plans d'exécution possibles. En effet, tout comme il y a plusieurs algorithmes possibles pour écrire un programme, il y a plusieurs plans d'exécution possibles pour répondre à une requête.

Ensuite, l'optimiseur (voir section 1.4.2) choisi le plan d'exécution qu'il estime être le meilleur pour exécuter la requête, parmi les différents plans d'exécution possibles.

Étape 3 : Exécution de la requête

Cette étape consiste à exécuter le plan d'exécution défini à l'étape précédente, c'est-à-dire à aller chercher les informations dans les tables en passant par les chemins définis par le plan d'exécution et à en extraire le résultat demandé.

INFO

La technique du *binding* – que nous étudierons au Chapitre 8, section 8.3, "Utilisation du binding" – permet, lorsqu'une même requête est exécutée plusieurs fois avec des paramètres différents, de n'exécuter qu'une seule fois les étapes 1 et 2 et ainsi de passer directement à l'étape 3 lors des exécutions suivantes.

1.4.2 Optimiseur CBO (*Cost Based Optimizer*)

Le rôle de l'optimiseur est de trouver le plan d'exécution le plus performant pour exécuter une requête. Oracle est depuis la version 7 (début des années 1990) pourvu de deux optimiseurs de requêtes :

- celui qui est basé sur des règles (RBO, Rule Based Optimizer, introduit en V6) ;
- celui qui est basé sur les coûts (CBO, Cost Based Optimizer, introduit en V7).

Sur les versions récentes d'Oracle, l'optimiseur basé sur les règles n'existe plus vraiment. Les mots clés permettant la manipulation du RBO sont seulement présents à des fins de rétrocompatibilité (depuis la version 10g, le RBO n'est plus supporté). Oracle recommande, depuis plusieurs versions, de ne plus utiliser le RBO et concentre tous ses efforts sur le CBO, c'est donc celui-là que nous allons étudier.

MS SQL Server et MySQL

SQL Server et MySQL étant eux aussi munis d'un optimiseur de type CBO, les grands principes étudiés ici seront communs.

Le CBO est un optimiseur qui est basé sur l'estimation des coûts d'exécution des opérations des plans d'exécution. Pour une requête donnée, le SGBDR établit plusieurs plans d'exécution possibles, et le CBO estime pour chacun d'eux le coût d'exécution et choisit le moins élevé.

Comment estimer le coût d'un plan d'exécution ? Le SGBDR évalue le coût en ressources utilisées pour exécuter ce plan. Ces ressources sont :

- le temps CPU ;
- le nombre d'E/S (entrées/sorties) disque dur (I/O en anglais) ;
- la quantité de mémoire vive (RAM) nécessaire.

Le coût sera une synthèse entre l'utilisation du CPU et le coût des E/S, selon qu'il s'agit d'accès séquentiels (lecture de plusieurs blocs contigus) ou d'accès aléatoires (lecture monobloc).

Cependant, vous imaginerez aisément que ce coût dépend non seulement de la requête elle-même, mais aussi des données sur lesquelles elle porte. En effet, calculer le salaire moyen d'une table contenant 10 personnes sera moins coûteux que de calculer la même moyenne sur une table d'un million de personnes alors que la requête sera la même.

Pour estimer le coût d'une requête, le SGBDR a besoin de déterminer, pour chacune des étapes du plan d'exécution, le nombre d'enregistrements concernés, c'est-à-dire la cardinalité de l'opération. Cette cardinalité dépend des données elles-mêmes mais aussi de l'impact de chacune des conditions. Par exemple, une condition qui filtre sur un numéro de Sécurité sociale n'aura pas le même impact sur la cardinalité qu'une condition portant sur un département ou une année de naissance. Ce principe se nomme la sélectivité, nous l'étudierons au Chapitre 5, section 5.1.3, "Sélectivité, cardinalité, densité".

L'optimiseur détermine le coût de chaque plan d'action d'exécution. Pour cela, sans exécuter aucun d'eux, ni parcourir les données, il définit les cardinalités de chaque opération à partir de statistiques sur les données (voir Chapitre 5, section 5.1, "Statistiques sur les données").

L'optimiseur cherche à déterminer le meilleur plan ; cependant, la notion de "meilleur" peut, dans certains cas, différer en fonction de l'objectif, l'*Optimizer Goal*, qui peut être de deux types :

- `First_Rows` (premières lignes) ;
- `All_Rows` (toutes les lignes).

`First_Rows` privilégie le temps de réponse pour retourner les premières lignes. Cela peut être intéressant dans le cadre d'applications interagissant avec des utilisateurs.

Pourtant, la meilleure solution choisie pour cet objectif peut se révéler moins intéressante si, finalement, on ramène toutes les lignes.

All_Rows, lui, privilégie l'utilisation optimale des ressources. Il considère le temps de réponse pour retourner l'ensemble des lignes.

Il existe aussi des variantes à l'objectif First_Rows, ce sont les objectifs First_Rows_n, où n est une des valeurs suivantes : 1, 10, 100, 1000. Dans ce cas, l'optimiseur privilégie le temps de réponse permettant de retourner les n premières lignes.

L'objectif de l'optimiseur (*Optimizer Goal*) peut se définir à différents niveaux :

- celui de la configuration de l'instance, à travers le paramètre d'initialisation OPTIMIZER_MODE.
- celui de la session utilisateur, en agissant sur le même paramètre :

```
ALTER SESSION SET optimizer_mode = first_rows_10;
```

- celui d'une requête, en utilisant les hints que nous étudierons plus tard :

```
Select /*+FIRST_ROWS(10) */ * from commandes
```

Axe 1

Étude et optimisation du modèle de données

Cette partie aborde l'aspect conception des bases de données en remplaçant les bases de données relationnelles dans le contexte du modèle relationnel.

Elle a pour but de vous rappeler quelques règles, mais ne prétend nullement se substituer à des ouvrages traitant de la conception de bases de données tels que :

- *Création de bases de données* de Nicolas Larrousse (Pearson, 2006) ;
- *SQL* de Frédéric Brouard, Rudi Bruchez, Christian Soutou (Pearson, 2008, 2010) (les deux premiers chapitres traitent de la conception de bases de données).

L'application de méthodes de conception, telles que Merise, permettra d'avoir une démarche de conception structurée et contribuera à obtenir un modèle de données pertinent et sans écueils de performances majeurs.

Modèle relationnel

2.1 Présentation

Les bases de données relationnelles que nous étudions sont fondées sur le modèle relationnel inventé par Edgar Frank Codd en 1970 (décrit dans la publication *A Relational Model of Data for Large Shared Data Banks*) et largement adopté depuis.

Ce modèle s'appuie sur l'organisation des données dans des relations (aussi appelées "entités"), qui sont des tables à deux dimensions :

- **Les colonnes.** Ce sont les attributs qui caractérisent la relation.
- **Les lignes.** Aussi nommées "tuples", elles contiennent les données.

L'ordre des tuples n'a aucune importance ni signification.

Tableau 2.1 : Exemple d'une relation contenant des employés

<i>Nom</i>	<i>Prénom</i>	<i>Service</i>	<i>Localisation</i>
DUPOND	Marcel	Comptabilité	Toulouse
DURAND	Jacques	Production	Agen
LEGRAND	Denis	Ventes	Toulouse
MEUNIER	Paul	Production	Agen
MEUNIER	Paul	Achats	Agen
NEUVILLE	Henry	Ventes	Toulouse

Le modèle relationnel établit qu'il doit être possible d'identifier un tuple de façon unique à partir d'un attribut ou d'un ensemble d'attributs, lesquels constituent la clé de la relation. Une clé est dite "naturelle" si un attribut ou un ensemble d'attributs la constituent. Dans l'exemple précédent, on constate qu'il est délicat d'en trouver une car se pose le problème des homonymes qui, dans le pire des cas, pourraient travailler dans le même service. Lorsqu'une clé naturelle n'existe pas, on introduit un nouvel attribut identifiant qui en fera office. Il s'agit, dans ce cas, d'une clé *technique*. Cet identifiant peut être un numéro affecté séquentiellement. On recourt aussi à l'utilisation de clé technique quand la clé naturelle est trop grande ou pour des raisons techniques, par exemple des problèmes de changement de valeur de la clé lorsqu'il y a des relations maître/détails et que le SGBDR ne gère pas la mise à jour en cascade.

L'absence de clé primaire dans une relation doit être exceptionnelle (moins de 1 % des tables). Lorsque rien ne s'y oppose, il faut privilégier l'utilisation de clé naturelle.

On peut parfois avoir plusieurs clés pour identifier un tuple (c'est systématique si vous ajoutez une clé technique alors qu'il y a une clé naturelle). Toutes ces clés sont dites "candidates" et celle qui a été retenue pour identifier la relation est dite "primaire".

Tableau 2.2 : Exemple d'une relation contenant des employés avec un identifiant

<i>Identifiant</i>	<i>Nom</i>	<i>Prénom</i>	<i>Service</i>	<i>Localisation</i>
5625	DUPOND	Marcel	Comptabilité	Toulouse
8541	DURAND	Jacques	Production	Agen
4521	LEGRAND	Denis	Ventes	Toulouse
8562	MEUNIER	Paul	Production	Agen
7852	MEUNIER	Paul	Achats	Agen
6214	NEUVILLE	Henry	Ventes	Toulouse

Un des grands intérêts du modèle relationnel est qu'il répartit l'ensemble des données dans différentes relations et établit des associations entre ces relations au moyen de leur clé primaire. Ce principe permet d'éviter la duplication d'information et donc d'avoir des données plus consistantes (voir Tableaux 2.3 et 2.4).

Tableau 2.3 : Exemple d'une relation "Employé", contenant des employés, avec une association à un service

<i>IdEmployé</i>	<i>Nom</i>	<i>Prénom</i>	<i>IdService</i>
5625	DUPOND	Marcel	100
8541	DURAND	Jacques	120
4521	LEGRAND	Denis	150
8562	MEUNIER	Paul	120
7852	MEUNIER	Paul	180
6214	NEUVILLE	Henry	150

Tableau 2.4 : Exemple d'une relation "Service"

<i>IdService</i>	<i>NomService</i>	<i>Localisation</i>
100	Comptabilité	Toulouse
120	Production	Agen
150	Ventes	Toulouse
180	Achats	Agen

Le modèle relationnel a été conçu pour être utilisé avec une algèbre relationnelle qui permet d'effectuer des opérations sur les entités. Les principales sont :

- **La jointure.** Relie deux entités par leur association.
- **La sélection.** Filtre les tuples dont les attributs répondent à un prédicat.
- **La projection.** Sélectionner seulement certains attributs d'une entité.

Cette algèbre a été implémentée puis étendue dans les SGBDR au moyen du langage SQL. Nous n'allons pas nous étendre sur la théorie de l'algèbre relationnelle qui, même si elle en constitue les bases, est par moments assez éloignée de ce qu'on peut faire avec du SQL.

En termes de génie logiciel, il est plutôt question de modèle conceptuel des données (MCD). Lorsque celui-ci est basé sur le modèle relationnel, on parle alors de modèle entité/association (et non pas de modèle entité/relation, qui est une traduction erronée du terme anglais *entity-relationship model*). La méthode MERISE, qui a bercé

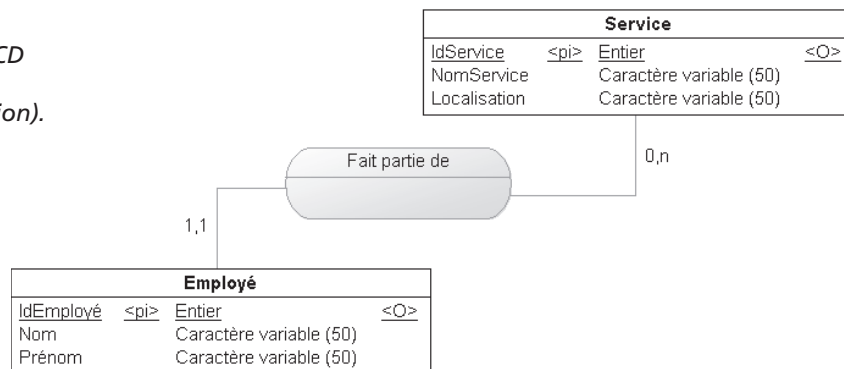
des générations d'informaticiens, utilise généralement ce modèle pour représenter le MCD. Ainsi, en France, on désigne souvent par MCD le modèle entité/association. La méthode MERISE parle aussi de modèle logique des données (MLD) qui est très proche de l'implémentation finale en base de données. Le MLD est une version indépendante du SGBDR du MPD (modèle physique des données). Le MLD et le MPD sont souvent confondus, d'autant plus quand la base représentée n'est prévue que pour être implémentée sur un SGBDR.

Le MCD et le MPD sont deux modèles à la fois redondants et complémentaires. Ils décrivent tous deux une base de données, mais avec des niveaux de détail et d'abstraction différents. Le MCD se veut plus conceptuel, par exemple :

- Il désigne les entités par leur nom logique et non pas par le nom de la table.
- Les associations sont nommées par un verbe.
- Il peut contenir des associations entre les entités qui ne seront pas implémentées par la suite sous forme de clé étrangère dans la base de données.
- Il ne répète pas les clés nécessaires aux associations et ne fait pas apparaître le fait qu'une table est nécessaire pour implémenter une relation M-N.

Figure 2.1

*Exemple de MCD
(diagramme
entité-association).*



Le MCD se veut indépendant des bases de données, il utilise donc des types de données "universels". La plupart des outils de modélisation peuvent convertir automatiquement des MCD en MPD mais, généralement, le résultat nécessite d'être retouché. Si vous convertissez sans aucune précaution un MCD en MPD, vous risquez d'avoir des tables qui s'appellent "compose", par exemple, ou un autre

verbe qui a toute sa place dans un MCD mais qui est moins indiqué comme nom de table.

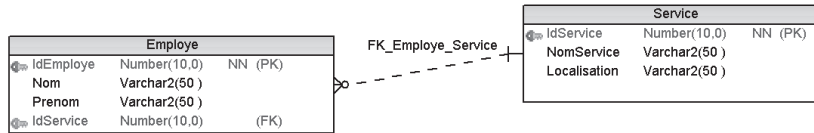


Figure 2.2

Exemple de MPD (modèle physique des données).

Le MPD est propre à une base de données et permet de générer automatiquement les scripts de création des tables. Il contient les noms des tables, l'ensemble des champs (qui s'appelaient "attributs" dans le MCD) avec leurs types et précisions, les contraintes d'intégrité référentielle, les index, etc. Si votre base doit être importante, prévoyez de passer un peu de temps pour affiner le MPD, à l'aide, entre autres, de ce que nous allons étudier au cours de cet ouvrage.

Les partisans du MCD disent que c'est une phase indispensable, alors que ses détracteurs affirment qu'il ne sert à rien. Personnellement, je pense que la vérité doit être quelque part entre les deux. Cela dépend des habitudes de chaque organisation, de la taille de la base, des outils dont vous disposez et du niveau d'abstraction souhaité.

2.2 Les bons réflexes sur le typage des données

Le typage des données paraît être un sujet anodin mais il ne l'est pas, et je suis surpris parfois de voir des bases de données avec des types incorrectement choisis. La conversion automatique du MCD en MPD peut être une source de mauvais typage, car les types "universels" ne proposent pas forcément la finesse disponible avec le SGBDR.

Il y a principalement quatre familles de types de champs :

- numériques ;
- textes ;
- dates et heures ;
- binaires.

Pour les données alphanumériques, il n'y aura pas trop d'ambiguïté. Le nom d'une personne devra être mis dans un champ de type texte. Il existe généralement trois variantes de types textes qui peuvent avoir des incidences sur le stockage et la manipulation des champs :

- les types textes de longueur fixe (généralement désignés par `char`) ;
- les types textes de longueur variable (généralement désignés par `varchar`) ;
- les types textes de grande capacité (désignés par `text`, `bigtext` ou `CLOB`).

Les champs de type texte de longueur fixe utilisent systématiquement l'espace correspondant à la longueur pour laquelle ils sont définis (hormis sur certains SGBDR où le fait d'être `NULL` n'occupe pas d'espace). Si la chaîne est plus petite que la longueur de la colonne, elle sera complétée par des espaces qui seront éventuellement supprimés par le SGBDR ou par la couche d'accès aux données. Ces champs sont assez peu utilisés mais ils peuvent présenter un intérêt sur des colonnes qui contiennent des valeurs de taille fixe. L'avantage de ce type est qu'il évite le surcoût lié à la gestion d'un champ de longueur variable. Par contre, sa manipulation provoque parfois des comportements déroutants pour les développeurs habitués aux types `varchar`.

Le type texte de longueur variable (`varchar`) est le plus utilisé. Il n'occupe que l'espace nécessaire à la donnée plus quelques octets pour définir la longueur de celle-ci.

Les types textes de grande capacité sont à réserver aux champs qui en ont vraiment besoin, c'est-à-dire qui ont besoin de contenir plus de 4 000 caractères (cela dépend des bases). Il en découle parfois une gestion du stockage très spécifique qui peut peser sur les performances et l'espace utilisé. De plus, certaines opérations sont parfois interdites sur ces types.

Tous ces types existent généralement aussi en version Unicode. Cependant, avec la percée d'UNICODE, certaines bases arrivent à présent à gérer le jeu de caractères UNICODE comme n'importe quel autre jeu de caractères, rendant l'intérêt des variantes UNICODE des types caractères moins évidents.

Concernant les données numériques, les SGBDR donnent plus ou moins de choix. Il sera judicieux de se poser quelques questions sur la nature et l'étendue des valeurs à manipuler dans ces champs afin de choisir le type le plus approprié. Les principaux types numériques sont :

- les types entiers signés ou non signés, sur 8, 16, 32 ou 64 bits ;

- les types décimaux à virgule flottante 32 ou 64 bits ;
- les types décimaux à virgule fixe.

Chacun d'eux occupe plus ou moins d'espace. Il faut penser que les types à virgule flottante font des approximations qui peuvent introduire des décimales supplémentaires ou en supprimer. Veillez à garder de la cohérence avec le type des variables qui manipuleront les données dans votre application (si votre application gère un float, il ne faut pas que la base définisse un entier 64 bits et *vice versa*).

Concernant les identifiants numériques, les types numériques ne sont pas toujours le bon choix. Pour les identifiants numériques séquentiels, un type entier est un choix adéquat (prendre un type décimal ne serait pas pertinent). Par contre, pour des identifiants tels que des numéros de Sécurité sociale (1551064654123), ce n'est pas forcément le meilleur choix, car :

- C'est une valeur trop grande pour tenir sur un entier 32 bits.
- Les types flottants risquent de l'approximer.
- Vous n'avez *a priori* aucune raison d'en faire la somme ou la moyenne.
- L'ordre alphabétique des identifiants est le même que l'ordre numérique car il y a toujours 13 chiffres.

Pour un tel identifiant, je conseille plutôt l'usage d'un type texte de longueur fixe. La présence de zéros en tête d'identifiant peut être un autre problème : ils seront effacés si vous utilisez un type numérique. Par exemple, la valeur 000241 sera stockée, et donc restituée, sous la forme 241 si vous utilisez un type numérique au lieu d'un type texte.

Concernant les données date et heure, il existe, suivant les SGBDR, des types permettant de stocker une date seule ou une date et une heure avec une précision plus ou moins grande sur la résolution, allant de quelques secondes à des fractions de seconde. J'ai vu assez régulièrement des développeurs ne pas utiliser ces types pour stocker des dates mais préférer des types textes. L'argument avancé est que les types date et heure sont parfois pénibles à manier aussi bien en SQL que dans les environnements de développement. Je leur concède que c'est souvent vrai. En effet, une date est plus difficile à manipuler qu'un entier, mais ce n'est cependant pas insurmontable. Il faut juste s'y pencher une bonne fois pour toutes, se faire une feuille récapitulative avec des exemples et, généralement, après tout va bien. Reste néanmoins le problème des fonctions de transformations en SQL qui ne sont pas

portables d'un SGBDR à l'autre. Si votre application doit fonctionner sur différents SGBDR, il faudra faire une petite couche d'abstraction.

Cela montre que le type date complique parfois un peu les opérations, mais qu'apporte-t-il par rapport à un type texte ? Si vous stockez dans la base avec une notation ISO (AAAAMMJJ), vous conservez la faculté de trier chronologiquement et de travailler sur des intervalles. Par contre, si vous utilisez un format plus français (ex : JJ/MM/AAAA), le tri et le filtrage par plage seront beaucoup plus compliqués. Bien évidemment, le type date vous permettra d'effectuer ce genre d'opération et aussi des choses qu'aucun de ces modes de stockage textuel ne permet en SQL, comme soustraire des dates entre elles, ajouter ou soustraire des intervalles de temps. De plus, en termes d'espace requis pour le stockage des données, les types dates seront systématiquement plus avantageux que leurs équivalents en format texte. Dernier avantage, avec les types dates, vous avez la garantie de n'avoir que des dates valides dans la base. Ce point est parfois considéré comme un inconvénient : il peut arriver que des applications récupèrent des chaînes contenant des dates erronées, ce qui provoque des erreurs lors du chargement de la base de données. Par expérience, je sais qu'avoir des dates invalides dans les tables pose des problèmes ; il vaut donc mieux, le plus en amont possible, avoir des données valides.

L'utilisation d'un entier pour stocker une date au format julien (nombre de jours depuis le 1^{er} janvier 4713 av. J.-C.) allie la plupart des atouts du format date (ordre, intervalles, taille) mais présente l'inconvénient de ne pas être très lisible (2524594 = 1/1/2000). Il n'est pas supporté de façon native par tous les SGBDR et ne permet pas toujours de gérer l'heure. L'utilisation d'un timestamp type unix présente les mêmes caractéristiques.

Les SGBDR gèrent généralement d'autres types de données, tels que les types binaires permettant de manipuler des images, des sons ou des fichiers, des types XML permettant de stocker du XML et de faire des requêtes dessus, et quelques autres types plus spécifiques à chaque SGBDR.

2.2.1 Les types sous Oracle

Pour gérer les valeurs numériques sous Oracle, on utilise généralement le type NUMBER avec des précisions variables suivant que l'on souhaite un entier ou une grande valeur décimale. Sous Oracle, la taille du stockage dépend de la valeur et non pas de la définition du champ. Ainsi, stocker la valeur 1 prend moins de place que stocker la valeur 123 456 789,123. Le type NUMBER peut prendre deux paramètres :

- **precision**, qui correspond au nombre de chiffres significatifs maximal que les valeurs du champ pourront avoir. Il peut prendre une valeur entre 1 et 38, ce qui correspond aux chiffres de part et d'autre du séparateur décimal. Le symbole * est équivalent à la valeur 38.
- **scale**, qui correspond au nombre de chiffres significatifs maximal après la virgule. Il vaut 0 s'il n'est pas spécifié : c'est alors un moyen de définir un type entier.

Le type **FLOAT** est un sous-type du type **NUMBER** et ne présente pas un intérêt particulier. Les types **BINARY_FLOAT** et **BINARY_DOUBLE** sont, eux, calqués sur les types flottants "C" et occupent respectivement 4 et 8 octets. Les types **ANSI** (**INT**, **INTEGER**, **SMALLINT**) sont en fait des synonymes de **NUMBER(38)** qui représente un entier. **DECIMAL** est un synonyme de **NUMBER**.

Les types caractères disponibles sont :

- **CHAR** et **NCHAR** pour les textes de tailles fixes limité à 2 000 caractères (**NCHAR** est la version Unicode).
- **VARCHAR2** et **NVARCHAR** pour les textes de tailles variables limités à 4 000 caractères. **VARCHAR** est un synonyme de **VARCHAR2**.
- **CLOB** et **NCLOB** pour les textes de grande taille, limités à 4 Go. Ces types sont traités comme un **LOB** (*Large Object*). Les données sont stockées dans un segment séparé, mais s'il y en a peu pour un enregistrement et que le stockage en ligne est actif (voir Chapitre 5, section 5.3.1, "Gestion des LOB"), la donnée pourra être stockée comme un **VARCHAR**. Le type **LONG**, ancêtre des **CLOB**, est obsolète et ne doit plus être utilisé.

Les types dates disponibles sont :

- **DATE**, qui permet de stocker date et heure avec une résolution à la seconde, occupe un espace de 7 octets. Il n'existe pas de type date seule, on utilise donc le type date sans spécifier d'heure, l'absence d'heure est traduite par minuit.
- **TIMESTAMP**, qui permet de stocker date et heure avec une résolution en fractions de seconde jusqu'à une nanoseconde.
- **TIMESTAMP WITH TIME ZONE** est analogue à **TIMESTAMP** mais stocke en plus la zone horaire.

Les types binaires disponibles sont :

- **RAW**, pour les binaires de moins de 2 000 octets.

- BLOB, pour les binaires de taille variable, limités à 4 Go. Concernant le stockage, il fonctionne comme les CLOB.
- LONG RAW, ancêtre de BLOB, ne doit plus être utilisé.
- BFILE est analogue à BLOB si ce n'est que les données sont stockées dans le système de fichier du serveur géré par le système d'exploitation et non pas dans un tablespace.

2.2.2 Les types sous SQL Server

Les types numériques disponibles sont :

- BIT, pour les booléens, stocké sur 1 bit ;
- TINYINT, pour les entiers non signés de 8 bits ;
- SMALLINT, INT, BIGINT, pour les entiers signés de respectivement 16, 32 et 64 bits ;
- DECIMAL et NUMERIC, pour les décimaux à virgule fixe, stockés sur 5 à 17 octets suivant la précision ;
- FLOAT, décimaux à virgule flottante, stocké sur 4 à 8 octets suivant la précision, REAL = FLOAT(24).

Les types caractères disponibles sont :

- CHAR et NCHAR, pour les textes de taille fixe limités à 8 000 caractères ;
- VARCHAR et NVARCHAR, pour les textes de taille variable limités à 8 000 caractères ;
- TEXT, NTEXT, VARCHAR(max) et NVARCHAR(max), pour les textes de grande taille limités à 2 Go.

Les types dates disponibles sont :

- DATE, qui permet de stocker une date seule (depuis SQL Server 2008) ;
- SMALLDATETIME, qui permet de stocker date et heure avec une résolution d'une minute, stocké sur 4 octets ;
- DATETIME, qui permet de stocker date et heure avec une résolution de trois à quatre secondes, stocké sur 8 octets ;

- DATETIME2, qui permet de stocker date et heure avec une résolution en fractions de seconde jusqu'à 100 nanosecondes, stocké sur 6 à 8 octets suivant la résolution (depuis SQL Server 2008) ;
- TIME, qui permet de stocker une heure seule (depuis SQL Server 2008).

Les types binaires disponibles sont :

- BINARY, pour les binaires de taille fixe de moins de 8 000 octets ;
- VARBINARY, pour les binaires de taille variable de moins de 8 000 octets ;
- IMAGE, pour les binaires jusqu'à 2 Go, remplacé par VARBINARY(max) (depuis SQL Server 2005).

2.2.3 Les types sous MySQL

Les types numériques disponibles sont :

- BOOL, pour les booléens, stocké sur 8 bits.
- TINYINT, pour les entiers non signés, stocké sur 8 bits.
- SMALLINT, MEDIUMINT, INT, BIGINT, pour les entiers signés de respectivement 16, 24, 32 et 64 bits. Il est possible d'ajouter le mot clé UNSIGNED pour manipuler des entiers non signés. INTEGER est un synonyme de INT.
- DECIMAL et DEC, décimaux à virgule fixe ayant une précision jusqu'à 65 chiffres significatifs.
- FLOAT, DOUBLE, décimaux à virgule flottante, stocké sur 4 et 8 octets.

Les types caractères disponibles sont :

- CHAR, pour les textes de taille fixe limités à 255 caractères.
- VARCHAR, pour les textes de taille variable limités à 65 535 caractères.
- TINYTEXT, TEXT, MEDIUMTEXT et LONGTEXT, pour les textes de taille variable limités respectivement à 255 octets, 64 Ko, 16 Mo, 4 Go. Le choix du type permet de fixer la taille du champ interne qui définit la taille de la donnée.

Les types dates disponibles sont :

- DATE, qui permet de stocker une date seule ;

- DATETIME, qui permet de stocker date et heure avec une résolution d'une seconde, stocké sur 8 octets ;
- TIMESTAMP, qui permet de stocker date et heure avec une résolution d'une seconde sur une plage de 1970 à 2038, stocké sur 4 octets ;
- TIME, qui permet de stocker une heure seule.

Les types binaires disponibles sont :

- TINYBLOB, BLOB, MEDIUMBLOB et LONGBLOB, pour les binaires de taille variable limités respectivement à 255 octets, 64 Ko, 16 Mo, 4 Go ;
- BINARY, pour les binaires de taille fixe de moins de 255 Octets ;
- VARBINARY, pour les binaires de taille variable de moins de 64 Ko.


Normalisation, base du modèle relationnel

3.1 Normalisation

Le but de la normalisation est d'avoir un "bon" modèle de données dans lequel les données seront bien organisées et consistantes. Un des grands principes est que chaque donnée ne doit être présente qu'une seule fois dans la base, sinon un risque d'incohérence entre les instances de la donnée apparaît.

Figure 3.1

Modèle physique des données (MPD) de la base de données initiale gérée par une feuille de tableur.

CommandesLivres		
	NoCommande	NN (PK)
	NomPrenom	
	Adresse	
	Livre1	
	Titre1	
	Quantité1	
	Prix1	
	MontantLivre1	
	Livre2	
	Titre2	
	Quantité2	
	Prix2	
	MontantLivre2	
	Livre3	
	Titre3	
	Quantité3	
	Prix3	
	MontantLivre3	
	MontantCommande	

La normalisation d'un modèle relationnel s'appuie sur des règles définies dans les différentes formes normales que nous allons étudier ci-après.

Nous allons mettre en pratique la normalisation en partant d'une petite base de données qui a pour but de gérer une librairie. Celle-ci, au fil du temps, est devenue une grande librairie et traite des centaines de commandes par jour. La "base de données" initiale était gérée sous un tableur et n'est donc pas du tout normalisée (voir Figure 3.1).

3.1.1 Première forme normale (1NF)

Pour qu'une entité soit en première forme normale (1NF), elle doit :

- avoir une clé primaire ;
- être constituée de valeurs atomiques ;
- ne pas contenir d'attributs ou d'ensembles d'attributs qui soient des collections de valeurs.

Un attribut qui a des valeurs atomiques n'est pas décomposable en sous-attributs. Par exemple, un attribut NomPrenom n'est pas considéré comme atomique, puisqu'il peut être décomposé en deux attributs : Nom et Prénom.

Ne pas contenir d'attributs ou d'ensembles d'attributs qui soient des collections de valeurs signifie que :

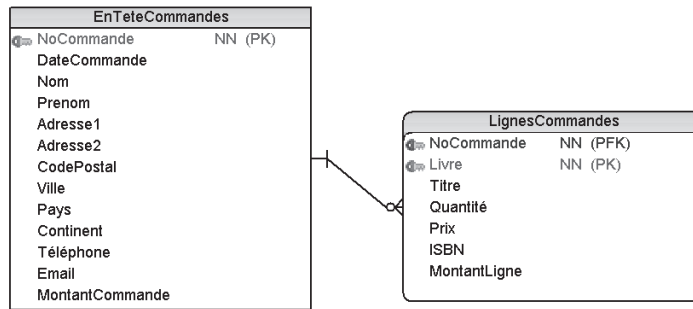
- Il ne doit pas y avoir d'attributs qui contiennent, en fait, plusieurs valeurs séparées par des caractères comme des espaces ou des virgules tels que "Dupond, Durand, Leclerc".
- Il ne doit pas y avoir d'ensembles d'attributs qui soient en fait des listes de valeurs, tels que les attributs Article1, Article2, Article3, etc., de notre exemple.

De tels attributs doivent être mis dans une entité séparée qui aura une association de type 1-N avec celle qui contenait ces attributs.

Mettons en pratique la première forme normale sur notre base exemple. Nous allons créer une entité des lignes de commandes pour résoudre le problème des collections sur les attributs (Livre, Titre, Quantité, Prix) et atomiser les attributs NomPrenom et Adresse. Cela donnera le résultat suivant qui peut, raisonnablement, être considéré comme 1NF. Nous profitons de cette étape pour enrichir un peu notre modèle en ajoutant l'attribut ISBN.

Figure 3.2

Modèle physique des données (MPD) en première forme normale (1NF).



INFO

Il existe plusieurs notations graphiques des modèles relationnels. Chaque outil choisit une des notations en prenant parfois des idées d'une autre notation.

Les diagrammes présentés à ce chapitre sont des MPD sans affichage des types, réalisés avec l'outil Toad Data Modeler, avec la notation IE qui utilise les conventions suivantes :

- Ligne association pleine : association identifiant dépendante ;
- Ligne association discontinue : association non-identifiant dépendante ;
- L'extrémité rond plus triangle désigne le côté fille de la relation ;
- Rectangle à angles droits : table ;
- Rectangle à angles arrondis : table fille d'une association identifiant dépendante.

3.1.2 Deuxième forme normale (2NF)

Pour qu'une entité soit en deuxième forme normale (2NF), il faut :

- qu'elle soit en première forme normale (1NF) ;
- que tous les attributs ne faisant pas partie de ses clés dépendent des clés candidates complètes et non pas seulement d'une partie d'entre elles.

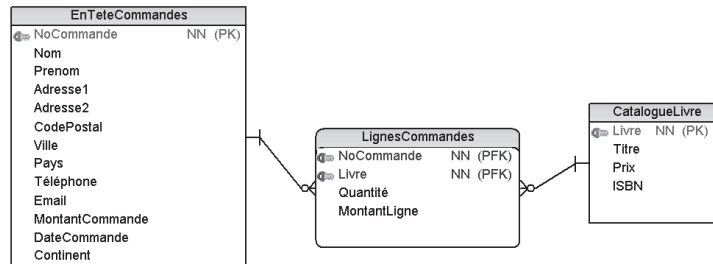
Cette forme ne peut poser de difficultés qu'aux entités ayant des clés composites (composées de plusieurs attributs). Si toutes les clés candidates sont simples et que l'entité est 1NF, alors l'entité est 2NF. Attention, la règle s'applique à toutes les clés candidates et pas seulement à la clé primaire.

À la Figure 3.2, l'entité LIGNESCOMMANDES n'est pas 2NF car la clé est NoCommande plus Livre. Or, les attributs Titre, Prix et ISBN ne dépendent que de l'attribut Livre, donc, seulement d'une partie de la clé. Pour transformer ce modèle en deuxième forme normale, nous allons créer une nouvelle entité qui contiendra les

informations relatives aux livres. L'application de cette forme normale permet de supprimer les inconsistances possibles entre les titres d'un même livre qui aurait été commandé plusieurs fois, car le modèle précédent permettait d'avoir des valeurs de Titre différentes pour une même valeur de Livre.

Figure 3.3

Modèle physique des données (MPD) en deuxième forme normale (2NF).



3.1.3 Troisième forme normale (3NF)

Pour qu'une entité soit en troisième forme normale (3NF), il faut :

- qu'elle soit en deuxième forme normale (2NF) ;
- que tous les attributs ne faisant pas partie de ses clés dépendent directement des clés candidates.

À la Figure 3.3, l'entité ENTETECommandes n'est pas en 3NF car toutes les informations relatives au client ne sont pas dépendantes de la commande. Il faut introduire une nouvelle entité, qui contiendra les informations relatives aux clients.

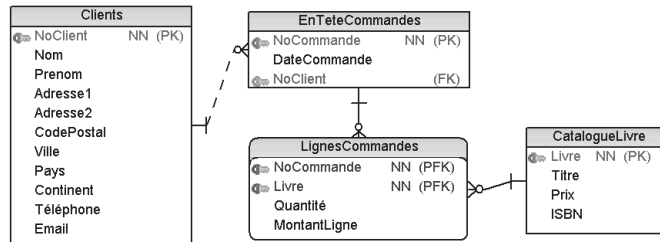
Il devrait donc rester dans l'entité ENTETECommandes les attributs DateCommande et MontantCommande. Mais est-ce que l'attribut MontantCommande dépend directement de la clé ? Apparemment oui, si on considère seulement l'entité ENTETECommandes. Par contre, si on considère aussi ses associations, alors on s'aperçoit que cet attribut est en fait la somme des MontantLigne et ne dépend donc pas seulement de la clé mais de l'entité LIGNESCommandes. C'est ce qu'on appelle un "attribut dérivé", c'est une redondance d'information. Pour être en 3NF, il ne doit pas y avoir d'attributs dérivés, il faut donc supprimer l'attribut MontantCommande. Les attributs calculés à partir des colonnes de la même entité sont eux aussi des attributs dérivés et ne sont pas admis dans la troisième forme normale.

On pourrait faire la même remarque pour l'attribut MontantLigne de l'entité LIGNESCommandes. Cependant, il y a un argument qui l'autorise à persister, c'est que l'attribut MontantLigne est égal à Quantité × Prix du livre au moment de la commande,

or, l'attribut Prix de l'entité CATALOGUELIVRE est le prix actuel du livre. Ce prix est susceptible de changer dans le temps, l'information de prix intégrée dans l'attribut MontantLigne n'est donc pas la même information que le Prix de l'entité CATALOGUELIVRE. Ainsi, l'attribut MontantLigne n'est pas un attribut dérivé de l'attribut Prix de l'entité CATALOGUELIVRE (voir Section 3.2.1).

Figure 3.4

Modèle physique des données (MPD) en troisième forme normale (3NF).



3.1.4 Forme normale de Boyce Codd (BCNF)

La forme normale de Boyce Codd est une version plus contraignante de la troisième forme normale.

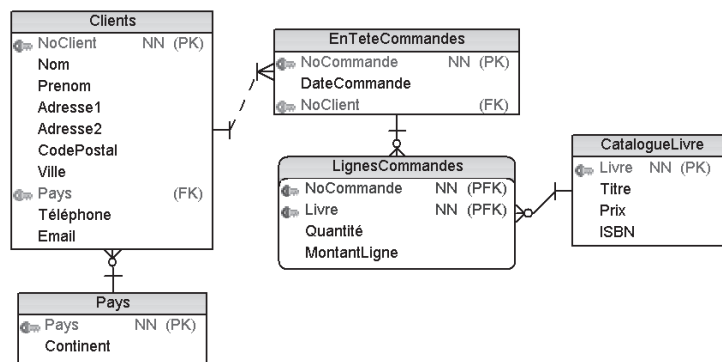
Pour qu'une entité soit en forme normale de Boyce Codd (BCNF), il faut :

- qu'elle soit en troisième forme normale (3NF) ;
- que tous les attributs ne faisant pas partie de ses clés dépendent exclusivement des clés candidates.

À la Figure 3.4, l'entité CLIENTS n'est pas en BCNF car l'attribut Continent dépend certes du client, mais il dépend aussi de l'attribut Pays. Pour être conformes, nous devons créer une nouvelle entité permettant d'associer chaque Pays à son Continent.

Figure 3.5

Modèle physique des données (MPD) en forme normale de Boyce Codd (BCNF).



3.1.5 Autres formes normales

Il existe d'autres formes normales (quatrième, cinquième et sixième formes normales). Cependant, elles sont un peu plus spécifiques. Tendre vers une troisième forme normale ou une forme normale de Boyce Codd est déjà un excellent objectif.

3.2 Dénormalisation et ses cas de mise en œuvre

La normalisation est une bonne intention qu'il est plus que souhaitable d'appliquer. Cependant, tout comme l'enfer est pavé de bonnes intentions, il faut savoir prendre un peu de recul par rapport aux règles afin de ne pas tomber dans certains excès. Si la normalisation fournit de bons guides, je pense qu'il faut savoir les remettre en cause s'il y a de bonnes raisons de le faire. Le plus important est de se poser les bonnes questions plutôt que d'appliquer aveuglément des règles. L'objet de cette section est de donner quelques règles pour ne pas respecter celles de la normalisation.

ATTENTION

Nous abordons ici la notion de dénormalisation. C'est-à-dire que nous allons étudier comment enfreindre certaines règles sur un modèle qui les applique.

Avant de dénormaliser un modèle, il faut d'abord le normaliser.

3.2.1 La dénormalisation pour historisation

Cette première forme de dénormalisation n'en est pas vraiment une. Nous l'avons déjà abordée, lors de l'étude de la troisième forme normale avec l'attribut Montant-Ligne de l'entité LIGNESCOMMANDES.

Les formes normales ont pour objectif d'éviter la redondance de l'information. Si on conçoit le modèle avec une vision statique, on risque de passer à côté du fait que certaines valeurs vont évoluer dans le temps et qu'il ne sera plus possible de retrouver après coup la valeur historique. Le prix courant d'un article et le prix de ce même article il y a six mois ne sont pas forcément les mêmes. Comme nous l'avons déjà vu, il s'agit de données distinctes. Même si, à certains instants, les valeurs sont identiques, elles n'ont pas le même cycle de vie.

La méthode la plus commune pour gérer l'historique d'une valeur consiste à recopier l'information dans l'entité qui l'utilise. Une autre manière possible, et plus

conforme à la logique des formes normales, pour gérer ce besoin est de conserver l'historique de la donnée référencée dans une entité séparée et horodatée plutôt que de répéter la donnée à chaque instance la référençant.

Si nous reprenons l'exemple du prix qui varie dans le temps, la première méthode possible consiste à recopier le prix dans chaque ligne de commande (voir diagramme de gauche à la Figure 3.6). On peut aussi créer une entité contenant l'historique du prix du livre. Ainsi, la donnée prix n'est pas répétée et on pourra à tout moment la retrouver à partir de la date de la commande (voir diagramme de droite).

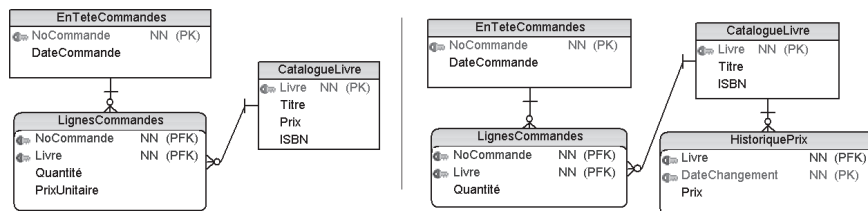


Figure 3.6

Deux solutions possibles pour implémenter une gestion des commandes avec un historique des prix.

Dans le cas présent, mon cœur balancerait plutôt pour la première solution (à gauche) car je sais qu'en termes d'implémentation la seconde solution (à droite) sera plus compliquée. En effet, quand vous aurez besoin de connaître le prix d'une ligne de commande, vous devrez rechercher l'enregistrement d'historique le plus récent qui a une date antérieure à la date de la commande. Cette requête-là n'est déjà pas très simple. Si on l'ajoute au fait qu'on souhaite calculer le montant total des commandes d'un client pour une année, ça devient alors complexe et peu performant avec la seconde solution. La première solution permet de répondre à cette requête de façon assez intuitive et efficace. Dans d'autres contextes, la seconde solution pourrait présenter plus d'avantages que la première. Par exemple, si vous avez de nombreux attributs à garder en historique, qu'ils soient volumineux (grands textes descriptifs, photos, etc.) et évoluent peu au regard des commandes. La seconde solution permet aussi de connaître les variations de prix d'un produit même si celui-ci n'a pas été commandé entre les deux variations.

Comme toujours, il faut peser le pour et le contre de chaque solution plutôt que chercher forcément la solution mathématiquement la plus juste.

Certains disent qu'il ne faut pas dénormaliser pour arranger le développeur et qu'il faut toujours choisir la solution la plus propre sous peine de se traîner des "casse-roles" pendant des années. C'est généralement vrai, cependant, comme le montre l'exemple précédent, l'application trop stricte de certaines règles peut conduire à complexifier une application et avoir des impacts substantiels sur les performances. L'important est de se poser les bonnes questions et de faire vos choix en toute honnêteté intellectuelle.

3.2.2 La dénormalisation pour performance et simplification en environnement OLTP

La normalisation a pour but, entre autres, d'éviter la duplication de l'information afin de renforcer la consistance des données. Cela peut avoir pour effet de nuire aux performances. Comme nous l'avons vu, le modèle de droite de la Figure 3.6 n'est pas adapté pour retrouver aisément l'ensemble des prix des lignes d'une commande. Ainsi, lorsque vous avez besoin d'afficher sur une facture le montant total de la commande, il faut faire une requête compliquée.

Imaginons un instant qu'il y ait une politique, variable pour chaque article, de remises de prix en fonction de la quantité commandée. L'ajout d'un champ `MontantRemise` pourrait être considéré comme un non-respect des formes normales car il dérive des attributs `Quantité` et `CodeArticle`. Cependant, je pense que, lorsqu'un attribut est une valeur qui est dérivée de façon complexe, on peut raisonnablement ne pas respecter à la lettre la troisième forme normale.

Abordons à présent le cas des attributs dérivés simples mais qui rendent service tels que `MontantLigne` qui est égal à $\text{Quantité} \times \text{Prix} - \text{MontantRemise}$. On ne peut pas dire que cela soit très compliqué à calculer, du coup ajouter cet attribut n'est qu'à moitié raisonnable. Une solution possible, et à mon avis fidèle à l'esprit des formes normales, est d'ajouter une colonne virtuelle (ou calculée suivant le SGBDR), laquelle contient le calcul et est maintenue par le SGBDR. Ainsi, il est sûr que cette information est consistante avec les informations dont elle dérive. La limite de ce type de champ est qu'il ne peut faire des calculs que sur les champs d'une même table.

Regardons maintenant un autre cas d'attributs dérivés simples, des dérivés par agrégation, tels que l'attribut `MontantCommande` qui est la somme des `MontantLignes` à la Figure 3.3. Sa présence ne respecte pas la troisième forme normale,

mais elle peut être très pratique pour calculer des indicateurs de chiffres d'affaires (CA) en temps réel. En plus d'être pratique, cela sera plus performant que de faire la somme des lignes de chaque commande. Cependant, il faudra maintenir cette valeur, donc faire régulièrement une requête qui recalcule la somme de Montant-Lignes pour mettre à jour l'attribut MontantCommande. Cela peut être acceptable pour le cas de MontantCommande, mais il ne faut pas trop dériver, sinon il y a un risque de devoir calculer trop de données agrégées à chaque modification d'une donnée.

En effet, afin de garantir la consistance de ces données, chaque fois que vous modifiez un enregistrement de LIGNESCOMMANDES, vous devez recalculer les valeurs qui en dérivent. Par exemple, si vous n'avez pas été raisonnable, vous pourriez devoir calculer toutes les valeurs suivantes : CA par jour, CA par mois, CA par jour par Livre, CA par mois par Livre, CA par client par an, CA par mois par pays, CA par mois par continent, etc. Vous risquez donc, si vous abusez de la dénormalisation, de plomber sérieusement vos performances lors des mises à jour des données de votre base, alors que votre objectif était d'améliorer les performances de votre système. L'utilisation de vues matérialisées que nous étudierons au Chapitre 5, section 5.3.5, "Les vues matérialisées", permet de maintenir ce genre de données de façon consistante et relativement performante car les calculs sur ces objets sont faits par variations et non pas par le recalcul complet des totaux (solution qui pourrait être extrêmement coûteuse).

INFO

Sur la plupart des SGBDR, vous pouvez implémenter des champs calculés à l'aide de déclencheurs (*triggers*). Certains SGBDR proposent aussi le concept de colonnes calculées qui sont, contrairement aux déclencheurs, limitées à des calculs portant sur la table courante.

Sous SQL Server, vous disposez des colonnes calculées persistantes (stockées) ou non persistantes (recalculées à la volée) en utilisant la syntaxe suivante dans l'instruction CREATE TABLE.

```
ColCalculee as ColonneA+ColonneB PERSISTED
```

Sous Oracle, depuis la version 11g, vous pouvez utiliser les colonnes virtuelles, qui ne sont pas stockées, à l'aide de la syntaxe suivante :

```
ColCalculee as (ColonneA+ColonneB)
```

Comme nous le verrons au Chapitre 7, section 7.4.1, "Impact des triggers", l'utilisation des triggers, même simples, peut entraîner une chute des performances s'il y a beaucoup de LMD.

3.2.3 La dénormalisation pour performance en environnement OLAP

On retrouve souvent les environnements OLAP (*OnLine Analytical Processing*) sous les noms d'"entrepôt de données" (datawarehouse) ou de "système d'aide à la décision" (DSS, *Decision Support System*). En environnement OLAP, la dénormalisation est la règle, car les formes normales ont été pensées pour un environnement OLTP. Lorsqu'on exécute des requêtes sur des centaines de milliers d'enregistrements, faire des jointures à tout-va est assez coûteux. Quand l'essence même de votre base de données est de faire de l'analyse des données, c'est du gâchis de passer son temps à refaire les mêmes jointures. Dans ce type de base de données, on duplique plutôt l'information afin de réduire le nombre de jointures et ainsi d'avoir de meilleures performances.



La modélisation de bases OLAP – qu'il n'est pas prévu d'étudier ici – est une science à part entière et dépend un peu des outils mis à disposition par votre moteur OLAP. Cependant, les techniques d'optimisation que nous allons aborder au cours de cet ouvrage sont applicables aux bases OLAP.

Dans les bases OLAP, vous entendrez couramment parler de modèle en étoile et en flocons, de dimensions, de faits. SQL Server et Oracle intègrent des outils pour vous aider à manipuler ces concepts.

Je vous invite à consulter dans la documentation de votre SGBDR les parties relatives aux bases OLAP.

Figure 3.7

Exemple de table dénormalisée pour un usage OLAP.

OLAPLignesCommandes		
	NoCommande	NN (PK)
	Livre	NN (PK)
	ISBN	
	Titre	
	Quantité	
	Prix	
	MontantLigne	
	DateCommande	
	Mois	
	Annee	
	Pays	
	Continent	

En environnement OLAP, vous aurez tendance à dupliquer dans les tables de faits les données qui vous servent d'axe d'analyse. Le modèle de données d'une base

OLAP sera construit en fonction des principaux types d'analyses que vous prévoyez de faire.

Dans l'exemple de la Figure 3.7, nous voyons que nous allons pouvoir faire les analyses suivantes :

- par livre ;
- par commande ;
- géographique (Pays, Continent) ;
- temporelle (Date de commande, Mois, Année).

3.3 Notre base de test

Afin d'illustrer les concepts que nous allons présenter tout au long de ce livre, nous allons les appliquer à une base de test inspirée des modèles que nous venons d'étudier.

Cette base (structure et données) est disponible pour les SGBDR Oracle, SQL Server et MySQL sur le site de l'auteur à l'adresse <http://www.altidev.com/livres.php>.

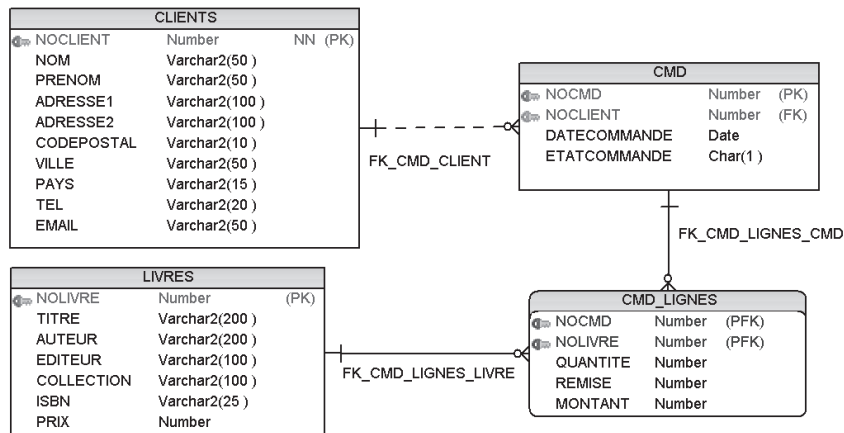


Figure 3.8

Modèle physique des données de notre base de test.

Tableau 3.1 : Volumétrie de notre base de test

Nom table	Nombre d'enregistrements
CLIENTS	45 000
CMD	1 000 000
CMD_LIGNES	2 606 477
LIVRES	2 973

Nous illustrerons quelques mises en œuvre avec deux autres bases de tests. La première, bien connue dans le milieu Oracle, est la base EMP/DEPT du schéma SCOTT, elle contient très peu d'enregistrements.

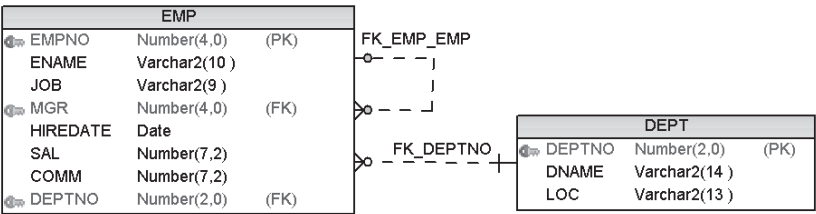
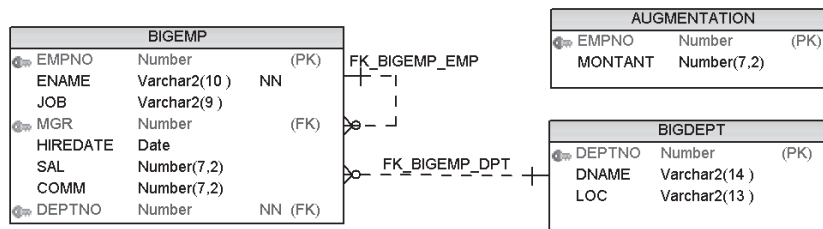


Figure 3.9
Modèle physique des données de la base de test EMP.

Tableau 3.2 : Volumétrie de la base de test EMP

Nom table	Nombre d'enregistrements
EMP	14
DEPT	4

La seconde base de test est une évolution de la précédente qui contient plus d'enregistrements (voir Annexe C pour plus de détails) et intègre une table contenant des augmentations à appliquer aux employés. Elle sera utile pour tester les requêtes modifiant les données.

**Figure 3.10**

Modèle physique des données de la base de test BIGEMP.

Tableau 3.3 : Volumétrie de la base de test BIGEMP

<i>Nom table</i>	<i>Nombre d'enregistrements</i>
BIGEMP	1 400 014
BIGDEPT	400 004
AUGMENTATION	1 400 014

Axe 2

Étude et optimisation des requêtes

Méthodes et outils de diagnostic

Avant de commencer à optimiser une base de données, il est important de bien comprendre ce qui se passe et comment agit le SGBDR. Cette phase est fondamentale. L'optimisation d'une base de données ne se résume pas à juste appliquer quelques recettes de cuisine sans rien comprendre à ce que l'on fait, ni à l'impact que chaque action peut avoir.

4.1 Approche pour optimiser

L'optimisation d'une base de données est une activité qui peut avoir lieu :

- de façon curative, c'est-à-dire lorsque les problèmes apparaissent lors de l'implémentation ou en production ;
- de façon plus préventive lors de la conception de la base.

Afin que des problèmes de performances ne surgissent pas durant la phase de production, ce qui nuirait aux utilisateurs, il est important de les anticiper dans les phases amont de la vie de l'application. Pour cela, il faut intégrer la problématique des performances dans la démarche de développement comme toute autre exigence. Cela signifie, entre autres, qu'il faudra se donner les moyens d'évaluer l'impact de la croissance du volume. Il n'est pas facile, lorsque la base de données ne contient que quelques centaines d'enregistrements, de mettre en évidence de futurs problèmes de performances. Il faut donc, en phase de développement et/ou de test, travailler sur une base qui se rapproche de ce que sera la réalité après quelques années de production. Cela peut se faire, en phase de maintenance, à partir d'une copie des données des bases de production et, en phase de création d'une nouvelle application, en produisant des données factices à l'aide de générateurs de données. Si un grand nombre

d'utilisateurs est prévu, il sera pertinent de faire, en complément, des tests de charge en simulant des utilisateurs simultanés.

En phase de conception, vous n'aurez pas de retours des utilisateurs pour signaler ce qui est lent et donc vous orienter vers les parties qui ont besoin d'être optimisées. Faut-il pour autant se creuser la tête sur toutes les tables de la base de données ? La réponse est oui, pour ce qui est de la normalisation du modèle de données, et non, pour ce qui est des optimisations au niveau du SGBDR (index, etc.). Généralement, au moins 80 % des données vont se concentrer dans moins de 20 % des tables, c'est donc sur ces dernières qu'il va falloir concentrer vos efforts. De façon générale, une table contenant moins d'un millier d'enregistrements n'a pas besoin d'être optimisée (sauf cas particuliers).

Malgré vos efforts durant la phase de conception, il est possible que certaines requêtes souffrent de problèmes de performances. Dans ce cas, il faudra appliquer la méthode suivante :

1. Évaluer la situation au moyen de mesures et de l'analyse des schémas d'exécution.
2. Appliquer une ou plusieurs solutions techniques.
3. Évaluer l'amélioration.

Ce chapitre va vous aider à traiter les points relatifs à l'évaluation. Les prochains chapitres traiteront des solutions techniques possibles évoquées au deuxième point de cette liste.

4.1.1 Mesurer


Comme l'a dit William Deming, "On ne peut améliorer que ce que l'on mesure !" En effet, si vous essayez différentes solutions pour améliorer une requête, comment saurez-vous quelle est la meilleure solution si vous ne faites pas de mesures pour les comparer ? Il est nécessaire de trouver des critères tangibles à évaluer afin de mesurer la progression apportée par les optimisations. Les principaux critères sont :

- le temps de réponse ;
- la consommation mémoire ;
- la consommation CPU ;
- le nombre E/S (entrées/sorties disque).

La mesure synthétique la plus pratique est le temps d'exécution. C'est un indicateur assez simple d'utilisation et c'est, de plus, celui qui est ressenti par l'utilisateur. Il faut cependant exécuter la requête plusieurs fois pour valider cette mesure, car le niveau de charge global du système et d'autres éléments influent sur ce paramètre. La plupart des environnements graphiques de développement affichent systématiquement le temps d'exécution de la dernière requête.

Sous Oracle SQL*Plus, il s'active avec la commande suivante :

```
set timing on;
```

Une autre solution, pour mesurer la consommation de ressources, consiste à afficher les statistiques relatives à l'exécution des requêtes. À la suite de chaque exécution, il est possible de récupérer, dans les vues système, les statistiques de la dernière exécution ainsi que le plan d'exécution utilisé. Attention, le plan d'exécution utilisé affiche des coûts et des cardinalités estimés et non les coûts réels. Certains environnements intègrent directement cette fonctionnalité, vous l'activez sur Oracle SQL Developer, par exemple, à travers l'icône  (raccourci F10 – Enregistrer la trace automatique).

Dans l'environnement Oracle SQL*Plus, on peut utiliser le mode AUTOTRACE dont la syntaxe est la suivante :

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]  
set autotrace on;
```

L'option ON implique les options explain et statistics :

- explain affiche le plan d'exécution.
- statistics affiche les statistiques d'exécution.
- traceonly permet de ne pas afficher le résultat lui-même mais seulement les informations relatives à l'exécution de la requête.

L'activation du mode AUTOTRACE nécessite le rôle PLUSTRACE, défini dans le script serveur @\$ORACLE_HOME/sqlplus/admin/plustrce.sql :

```
grant PLUSTRACE to utilisateur;
```

Le mode AUTOTRACE nécessite la présence dans le schéma de l'utilisateur de la table PLAN_TABLE qui est créée lors de l'exécution du script serveur @\$ORACLE_HOME/rdbms/admin/utlxplan.sql.

Ce mode peut aussi requérir l'accès à certaines vues de la métabase :

```
grant select on V_$MYSTAT to utilisateur;
```

Ou de façon plus large :

```
grant select_catalog_role to utilisateur;
grant select any dictionary to utilisateur;
```

Listing 4.1 : Exemple de trace d'une exécution sous SQL*Plus

```
SQL> set autotrace trace;
SQL> select * from cmd where noclient>30000;
848099 ligne(s) sélectionnée(s).
```

Plan d'exécution

Plan hash value: 2368838273

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		885K	15M	894 (2)	00:00:11
* 1	TABLE ACCESS FULL	CMD	885K	15M	894 (2)	00:00:11

Predicate Information (identified by operation id):

1 - filter("NOCLIENT">30000)

Statistiques


```
1 recursive calls
0 db block gets
59557 consistent gets
3142 physical reads
72 redo size
19938918 bytes sent via SQL*Net to client
622345 bytes received via SQL*Net from client
56541 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
848099 rows processed
```

Étudions de plus près le résultat obtenu :

- En premier lieu, on obtient le nombre de lignes sélectionnées.
- Ensuite, on a le plan d'exécution contenant les valeurs estimées et non pas les effectives. Le plan est complété avec quelques statistiques réelles de l'exécution.
- Enfin, se trouvent les statistiques suivantes :

<i>Nom</i>	<i>Description</i>
consistent gets	Nombre de blocs lus en mémoire après accès éventuel au disque
physical reads	Nombre de lectures effectives sur le disque (exprimé en blocs)
redo size	Quantité de REDO nécessaire en octets
bytes sent <i>via</i> SQL*Net to client	Octets envoyés sur le réseau du serveur vers le client
bytes received <i>via</i> SQL*Net from client	Octets envoyés sur le réseau du client vers le serveur
SQL*Net roundtrips to/from client	Nombre d'échanges réseau entre le client et le serveur
sorts (memory)	Nombre de tris effectués complètement en mémoire
sorts (disk)	Nombre de tris ayant requis au moins un accès disque
rows processed	Nombre de lignes retournées

MS SQL Server

Dans l'environnement SQL Server Management Studio (SSMS), depuis une fenêtre requête, l'équivalent du mode trace automatique est disponible par l'icône basculante  (Inclure le plan d'exécution réel ou Ctrl+M). Ce mode affiche, lors de chaque exécution, le plan d'exécution utilisé avec quelques informations statistiques réelles en plus des informations estimées.

MySQL

Pour tracer des instructions avec des statistiques depuis la version 5.0.37, vous pouvez utiliser l'instruction `SHOW PROFILE`. Pour cela, vous devez commencer par activer le profiling avec l'instruction :

```
set profiling=1;
```

puis, vous pouvez consulter les informations avec les instructions suivantes :

```
mysql> show profiles\G
***** 1. row *****
Query_ID: 1
Duration: 0.02978425
  Query: select max(nom) from clients where noclient>0
***** 2. row *****
Query_ID: 2
Duration: 0.02709625
  Query: select max(nom) from clients where noclient<30000
mysql> show profile for query 2;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000102 |
| Opening tables  | 0.000012 |
| System lock     | 0.000004 |
| Table lock      | 0.000006 |
| init            | 0.000020 |
| optimizing       | 0.000010 |
| statistics      | 0.000034 |
| preparing       | 0.000029 |
| executing        | 0.000004 |
| Sending data    | 0.026828 |
| end             | 0.000006 |
| query end       | 0.000004 |
| freeing items   | 0.000033 |
| logging slow query | 0.000001 |
| cleaning up     | 0.000005 |
+-----+-----+
```

Une autre solution consiste à utiliser l'instruction `SHOW STATUS` qui affiche environ 300 statistiques sur la session en cours. Pour mesurer l'impact d'une requête, il suffit de purger les statistiques avant d'exécuter la requête à l'aide de l'instruction :

```
Flush status;
```

puis d'afficher les statistiques en filtrant un groupe de statistiques avec l'instruction suivante :

```
mysql> show session status like 'Select%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Select_full_join | 0 |
| Select_full_range_join | 0 |
| Select_range | 1 |
| Select_range_check | 0 |
| Select_scan | 0 |
+-----+-----+
```

Les groupes de statistiques les plus intéressants sont Select, Handler, Sort, Created, Key

4.1.2 Comprendre le plan d'exécution

Le plan d'exécution est l'expression par le SGBDR de la méthode d'accès aux données pour répondre à une requête. Comme nous l'avons vu au paragraphe précédent, il s'affiche à la suite de l'exécution d'une requête lorsque la trace automatique est activée. On peut aussi le consulter sans exécuter la requête en passant par la commande `explain plan for` puis en lançant le script `@?/rdbms/admin/utlxpls.sql`. La plupart des outils de développement pour Oracle, y compris l'outil gratuit Oracle SQL Developer, intègrent une interface permettant de le consulter.

Ci-après, la trace d'exécution de la requête suivante :

```
select nom ,titre
from clients c join cmd on c.noclient=cmd.noclient
join cmd_lignes cl on cl.nocmd=cmd.nocmd
join livres l on cl.nolivre=l.nolivre
where C.pays='France' and L.editeur='Pearson';
```

Listing 4.2 : Plan d'exécution avec SQL*Plus

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time
0	SELECT STATEMENT		572	47476	2975	(2)	00:00:36
* 1	HASH JOIN		572	47476	2975	(2)	00:00:36
* 2	TABLE ACCESS FULL	LIVRES	35	1470	13	(0)	00:00:01
* 3	HASH JOIN		48490	1941K	2961	(2)	00:00:36
* 4	HASH JOIN		18613	563K	1066	(2)	00:00:13
* 5	TABLE ACCESS FULL	CLIENTS	841	16820	171	(1)	00:00:03
6	TABLE ACCESS FULL	CMD	1000K	10M	890	(2)	00:00:11
7	INDEX FAST FULL SCAN	PK_CMD_LIGNES	2606K	24M	1883	(1)	00:00:23

Predicate Information (identified by operation id):

- 1 - access("CL"."NOLIVRE"="L"."NOLIVRE")
- 2 - filter("L"."EDITEUR"='Pearson')
- 3 - access("CL"."NOCMD"="CMD"."NOCMD")
- 4 - access("C"."NOCLIENT"="CMD"."NOCLIENT")
- 5 - filter("C"."PAYS"='France')

Le plan d'exécution présente la liste des opérations qui vont être effectuées lors de l'exécution, reste à comprendre ce que cela veut dire. Nous allons voir ci-après les opérations les plus fréquemment rencontrées.

Figure 4.1

*Plan d'exécution sous Oracle SQL*Developer. Il contient les mêmes informations que sous SQL*Plus mais présentées différemment.*

OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		2975	
HASH JOIN		2975	572
Prédicats d'accès			
CL.NOLIVRE=L.NOLIVRE			
TABLE ACCESS FULL	LIVRES	13	35
Prédicats de filtre			
L.EDITEUR='Pearson'			
HASH JOIN		2961	48490
Prédicats d'accès			
CL.NOCMD=CMD.NOCMD			
HASH JOIN		1066	18613
Prédicats d'accès			
C.NOCLIENT=CMD.NOCLIENT			
TABLE ACCESS FULL	CLIENTS	171	841
Prédicats de filtre			
C.PAYS='France'			
TABLE ACCESS FULL	CMD	890	1000000
INDEX FAST FULL SCAN	PK_CMD_LIGNES	1883	2606477

Il en existe d'autres, plus "exotiques", souvent liées à une fonction en particulier, consultez la documentation Oracle pour plus d'information. Mais décodons d'abord le plan d'exécution de la Figure 4.1 :

1. Parcours complet de la table CLIENTS en filtrant PAYS= ' France ' .
2. Jointure de type HASH JOIN par la colonne Noclient entre le résultat de l'opération précédente et la table CMD *via* un parcours complet de la table CMD.
3. Jointure de type HASH JOIN par la colonne Nocmd entre le résultat de l'opération précédente et la table CMD_LIGNES *via* l'index PK_CMD_LIGNES (la table CMD_LIGNES n'apparaît pas directement car seules les colonnes Nocmd et Nolivres présentes dans l'index sont nécessaires).
4. Parcours complet de la table LIVRES en filtrant EDITEUR= ' Pearson ' .
5. Jointure de type HASH JOIN par la colonne Nolivres entre le résultat de l'opération précédente et le résultat de l'opération 3.

Le plan d'exécution se lit du fond de l'arbre vers la racine. La colonne Cardinality contient l'estimation du nombre de lignes manipulées par l'opération.

Opérations d'accès aux tables

Full Table Scan. Cette opération est assez simple à comprendre : la table est parcourue (scannée) entièrement, de façon linéaire, dans l'ordre le plus simple, c'est-à-dire l'ordre des blocs dans le tablespace.

Partition. Opération effectuée sur des partitions et non sur la table entière, spécifique de la manipulation de tables partitionnées.

Table Access By RowID. Cette opération permet un accès direct à un enregistrement au sein d'un bloc de données grâce au RowID qui contient l'adresse du bloc et l'offset de la ligne dans le bloc. Cela a généralement lieu après l'accès à un index qui fournit le RowID. Cette opération est aussi effectuée sur une condition du type `rowid= 'AAARdSAAGAAAGg0AAA'`.

Opérations d'accès aux index

Unique Scan. Parcourt l'arborescence de l'index pour localiser une clé unique. Typiquement utilisé sur une condition du type `noclient=37569`.

Range Scan. Parcourt une partie d'index de façon ordonnée. Typiquement utilisé sur une condition du type `noclient between 50000 and 60000`.

Full Scan. Parcourt un index en entier en respectant l'ordre de l'index.

Fast Full Scan. Analogue à Full Scan mais ne respecte pas l'ordre des données. Typiquement utilisé lorsqu'une condition peut être appliquée sur un index sans que l'ordre de tri soit mis en jeu. Par exemple :

```
select count(*) from clients where mod(noclient, 1000) = 150.
```

Skip Scan. Utilise les index multicolonne sans tenir compte des premières colonnes. Dans ce cas, l'index est considéré comme un ensemble de sous-index.

Bitmap. Utilise un index de type bitmap.

Partition. Utilise un index partitionné.

Opérations de jointure

Nested Loop (boucles imbriquées). Parcourt les sous-ensembles pour chaque valeur de l'ensemble de données pilotes.

Illustrons ce comportement avec la requête suivante :

```
select * from cmd
where noclient in (select noclient from clients
                  where pays='Cameroun' )
```

La sous-requête va être l'ensemble pilote en ramenant une liste de 30 clients. Pour chacun de ces Noclient, le SGBDR va exécuter la requête principale. Cela équivaut au pseudo-code suivant :

```
Foreach NoDuClientPilote in (select noclient from clients
                             where pays='Cameroun')
  select * from cmd where noclient = NoDuClientPilote
```

Merge Join. Rapproche deux ensembles de données triés. Le principe est que deux curseurs parcourent linéairement les ensembles triés. Cette solution est performante pour rapprocher deux ensembles déjà triés sur les clés de jointure (voir Figure 4.2).

Figure 4.2

*Illustration
d'un Merge Join par
les champs N° CMD.*

N° CMD	N° CLIENT	DATECOMMANDE	N° CMD	N° LIVRE	QUANTITE
14524	106141	16/04/2004	14524	6187	1
14525	131869	16/04/2004	14524	7789	1
14526	83068	16/04/2004	14525	4234	3
14527	120877	16/04/2004	14525	5554	3
14528	34288	16/04/2004	14526	5080	3
14529	103897	16/04/2004	14526	1579	3
14530	63544	16/04/2004	14527	4447	1
14531	92173	16/04/2004	14527	4396	4
14532	18994	16/04/2004	14528	4237	8
14533	22006	16/04/2004	14528	5380	5
14534	46777	16/04/2004	14528	2257	6
14535	103180	16/04/2004	14528	4339	7
14536	56563	16/04/2004	14528	4660	7
14537	107179	16/04/2004	14529	4651	1
14538	102964	16/04/2004	14529	5419	1
14539	77719	16/04/2004	14530	4177	1
14540	121630	16/04/2004	14530	7426	4
14541	44011	16/04/2004	14531	7423	2
14542	81100	16/04/2004	14531	8308	3

Hash Join. Construit une table de hachage permettant d'accéder plus rapidement aux clés (voir Figure 4.3).

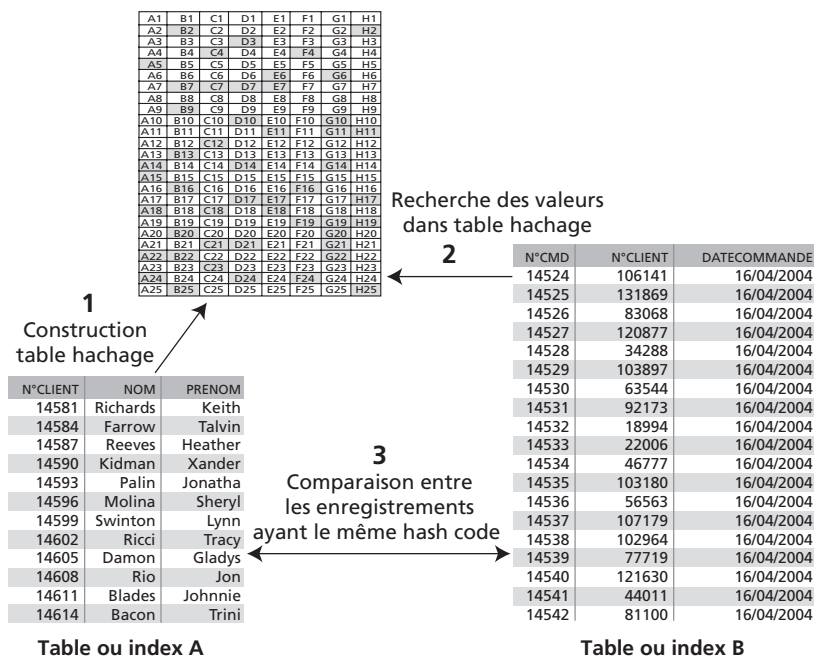


Figure 4.3

Exemple Hash Join de la forme `clients.noclient=cmd.noclient`.

1. On parcourt la table A (CLIENTS) car c'est celle qui a le moins d'enregistrements. Avec une fonction de hachage, on calcule le hashcode de chaque clé de la jointure pour construire une table de hachage (*hash table*).
2. On parcourt la table B (CMD) et, pour chaque valeur de la clé de jointure, on cherche si son hashcode existe dans la table de hachage (s'il n'existe pas, c'est que cette ligne n'a pas de correspondance dans l'autre table).
3. Si on a trouvé une correspondance, on teste pour chaque enregistrement de la table A correspondant au hashcode si la clé est égale à la clé de la table B.

Une fonction de hachage (qui donne le hashcode) est une fonction injective qui donnera toujours le même résultat pour une valeur donnée, mais plusieurs valeurs d'entrées pourront donner la même valeur de sortie. Le but de la table de hachage est de créer une liste de valeurs ordonnée et continue, qui permettra de regrouper des petits ensembles de données dans chacune de ses cases. Il suffira de comparer chaque clé avec seulement le sous-ensemble des clés qui a le même hashcode et non pas avec toutes les clés.

Idéalement, petit ensemble signifiera inférieur ou égal à un enregistrement (parfois 0, souvent 1, parfois plus). Ce souhait impliquera d'avoir une table de hachage la plus grande possible.

Le but de cette technique est de ne comparer chaque enregistrement de la table B qu'à peu d'enregistrements de la table A. Cette dernière sera forcément la plus petite afin de respecter l'objectif d'avoir le moins d'enregistrements correspondants à une case de la table de hachage.

Outer Join (jointure externe). Ce sont des variantes des jointures précédentes. Elles suivent le même principe, si ce n'est qu'on sélectionne aussi les lignes qui n'ont pas de correspondance.

Autres opérations

Union, Intersection, Minus. Effectuent des opérations ensemblistes. C'est généralement explicite dans la requête.

Sort. Effectue un tri.

Sort Aggregate. Effectue une opération d'agrégation (SUM, AVG, COUNT, etc.).

Filter. Filtre des données suivant un prédicat (synonyme de condition).

MS SQL Server


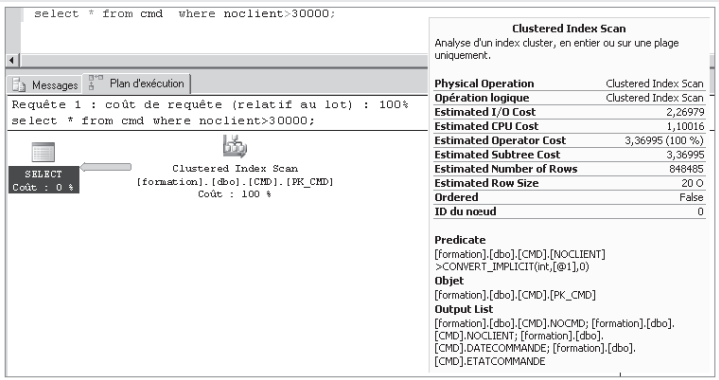
Dans l'environnement SQL Server Management Studio (SSMS), depuis une fenêtre requête, vous pouvez utiliser l'icône  (Afficher le plan d'exécution estimé ou Ctrl+L) pour afficher le plan d'exécution (voir Figure 4.4). La zone d'information jaune détaillant l'opération apparaît si l'on passe le curseur sur une des opérations.

Figure 4.4
Plan d'exécution
SQL Server.

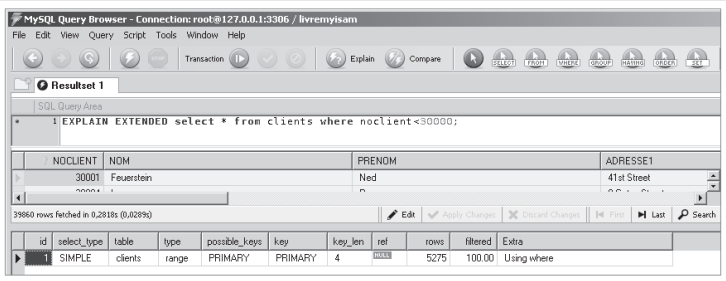


Les opérations ne sont pas les mêmes que sous Oracle, mais on retrouve les mêmes principes.

MySQL

Dans l'outil MySQL Query Browser, vous pouvez cliquer sur l'icône Explain pour afficher le plan d'exécution (voir Figure 4.5).

Figure 4.5
Plan d'exécution
MySQL.



Le plan s'affiche dans la zone du bas. Si vous travaillez en ligne de commande, vous pouvez utiliser EXPLAIN ou EXPLAIN EXTENDED comme illustré ci-après. L'utilisation de \G permet d'avoir une présentation verticale plus pratique à lire.

```
mysql> EXPLAIN EXTENDED select * from clients where noclient<30000 \G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: clients
         type: range
possible_keys: PRIMARY
         key: PRIMARY
        key_len: 4
         ref: NULL
        rows: 5275
    filtered: 100.00
      Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

4.1.3 Identifier les requêtes qui posent des problèmes

Avant de travailler à résoudre les problèmes, il faut correctement les identifier. En premier lieu, il faut identifier les requêtes qui posent le plus de problèmes et se concentrer sur celles-ci.

Différentes méthodes existent pour trouver les requêtes à optimiser.

Utilisation de V\$SQL

Une première manière, qui est la plus basique, consiste à utiliser la vue dynamique V\$SQL qui contient les requêtes exécutées récemment ainsi que des informations statistiques sur leur exécution. Je vous conseille de mettre des limites pour ne cibler que votre schéma et exclure les requêtes internes d'Oracle (Dans l'exemple ci-dessous, la valeur SCOTT doit être remplacée par votre nom de schéma).

```
SELECT * from V$SQL t
where LAST_active_TIME is not null
and t.PARSING_SCHEMA_NAME = 'SCOTT'
and upper(t.SQL_TEXT) not like '%SYS.%' and upper(t.SQL_TEXT) not like '%V$%'
and upper(t.SQL_TEXT) not like 'EXPLAIN %'
order by t.LAST_active_TIME desc
```

On pourra ajouter des conditions sur des noms de tables mises en jeu, comme dans la condition ci-après, pour réduire encore le volume des requêtes retournées.

```
and upper(t.SQL_TEXT) like '%CMD%'
```

Les colonnes de cette requête les plus intéressantes sont les suivantes :

- **Fetches, Executions.** Elles aident à repérer les requêtes ramenant beaucoup de lignes et celles exécutées fréquemment. Les deux sont importantes : en effet, une petite requête ayant un mauvais temps de réponse peut devenir pénalisante si elle est exécutée des milliers de fois.
- **Disk read, Buffer gets.** Elles aident à trouver les requêtes qui nécessitent de manipuler beaucoup de données dans la base, même si elles n'en ramènent que très peu.
- **Sql_text, Full text.** Elles contiennent le texte des requêtes qui ont été sélectionnées grâce aux champs précédemment décrits.

L'accès à la vue v\$SQL peut nécessiter les privilèges suivants :

```
grant select_catalog_role to utilisateur;  
grant select any dictionary to utilisateur;
```

Statistiques au niveau de l'instance

Une autre méthode s'appuie sur les outils Statspack ou AWR (*Automatic Workload Repository*) qui se servent des statistiques au niveau de l'instance. Cela dit, sur un serveur très actif qui fait tourner de nombreuses applications, vous ne trouverez peut-être pas d'informations pour vous aider en rapport avec une application particulière.

Le Statspack est un outil en fin de vie qui a pour but de collecter des statistiques au niveau de l'instance et, par la suite, d'en extraire des rapports.

Mode d'emploi rapide :

1. Si ce n'est déjà fait, installez-le en exécutant sur le serveur le script `rdbms/admin/spcreate.sql`.
2. Prenez des snapshots en exécutant :

```
exec statspack.snap;
```
3. Dans Oracle SQL*Plus, exécutez le script `spreport.sql` qui va générer des fichiers résultats dans le répertoire de travail de SQL*Plus.

AWR est le remplaçant du Statspack, il donne des informations très proches de celles du Statspack, en plus complet. Parfaitement intégré à la console d'administration OEM (*Oracle Enterprise Manager*), il est installé automatiquement sur une installation standard. Par défaut, il prend un cliché (*snapshot*) toutes les heures.

Les fonctions relatives à AWR sont disponibles dans la console d'administration web, dans la partie relative à l'instance, onglet Serveur, cadre Gestion des statistiques, lien Référentiel de charge globale automatique. Sur cet écran, vous pouvez visualiser les clichés disponibles et exécuter un rapport AWR entre deux clichés de votre choix.

Les rapports permettent de visualiser des statistiques sur le niveau de charge ainsi que les événements les plus significatifs, parmi lesquels les instructions SQL les plus consommatrices de CPU, d'E/S ou les plus fréquentes. Suite à l'exécution d'un rapport AWR, vous pouvez exécuter ADDM (*Automatic Database Diagnostic Monitor*) qui fera des propositions d'actions fondées sur les informations les plus pertinentes du rapport AWR. Dans certains cas, il peut proposer d'exécuter SQL Tuning Advisor (voir section 4.2.2 de ce même chapitre) afin de faire des propositions d'amélioration du SQL.

Suivi des performances de la console web

Une troisième méthode consiste à utiliser l'onglet performance de la base de données. Cet écran permet de visualiser les taux d'activité les plus élevés en temps réel ainsi que d'identifier les sessions et les instructions ayant le taux d'activité le plus élevé au moyen du lien "Taux d'activité les plus élevés".

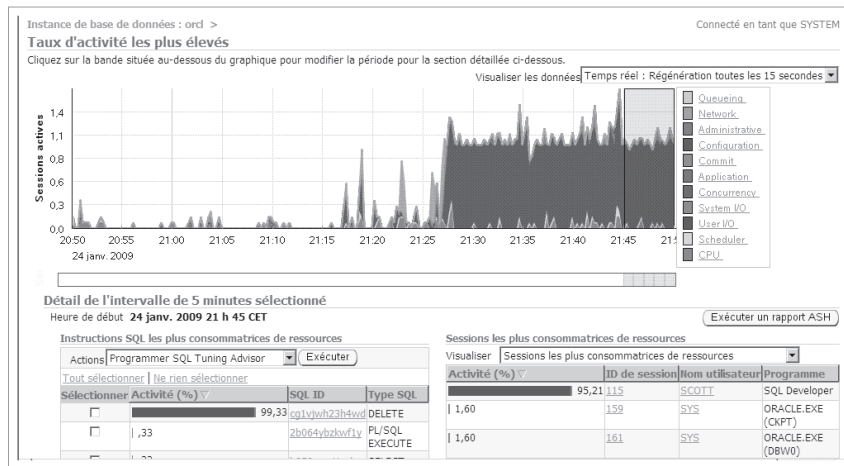


Figure 4.6

Écran de la console d'administration montrant les taux d'activité les plus élevés.

Dans la console, de nombreux écrans peuvent vous aider à trouver les requêtes, applications ou utilisateurs, qui consomment le plus de ressources. Ces écrans méritent d'être étudiés attentivement.

Trace des requêtes

Une quatrième manière consiste à tracer l'exécution de toutes les requêtes :

- Soit, si cela est possible, en traçant les exécutions côté application cliente à l'aide des outils de monitoring côté client afin d'étudier écran par écran le trafic généré et le temps de réponse de chacune des requêtes. Cette approche est conditionnée à la disponibilité de tels outils pour votre environnement de développement.
- Soit, en traçant des requêtes côté serveur, en utilisant l'outil SQL Trace, détaillé à la section 4.2.4 de ce chapitre.

4.2 Outils complémentaires

4.2.1 Compteurs de performance Windows

Sous Windows, il est facile d'accéder à un ensemble de paramètres permettant d'aider au diagnostic lors d'une chute de performance. Ces données sont consultables au moyen des compteurs de performance visualisables au travers du moniteur de performance (il existe des choses plus ou moins analogues sur les autres plates-formes).

Cet outil vous permettra de surveiller l'usage des ressources matérielles de votre serveur. Les compteurs les plus importants sont :

- **% processeur.** Affiche le taux d'usage du CPU. Si l'ensemble des CPU tend vers 100 %, cela signifie que les processeurs ont atteint leur limite de traitement.
- **Pages/s.** Indique le niveau d'échange entre la mémoire virtuelle et la mémoire physique. Si ce compteur présente en permanence des valeurs élevées, cela signifie que le système manque de mémoire physique ou que les applications en demandent trop.
- **Octets lus et écrits/seconde.** Montre le niveau de sollicitation du sous-système disque.

- **Mémoire disponible.** Permet de vous assurer que le serveur dispose encore d'un peu d'air. Généralement, si la mémoire vive passe sous la barre des 100 Mo, cela signifie qu'elle est saturée.

Cet outil permet de travailler au niveau serveur, il ne vous aidera pas dans le diagnostic d'une requête mais pour mettre en évidence une saturation au niveau du matériel. Cette saturation peut indiquer que le matériel atteint ses limites, mais les limites elles-mêmes peuvent être atteintes parce qu'il y a un problème en amont (base mal configurée, requêtes peu performantes).

MS SQL Server
De très nombreux compteurs de performances SQL Server sont disponibles. Ils permettent de se faire une bonne idée de l'activité de chaque instance de SQL Server.

4.2.2 SQL Tuning Advisor (Oracle)

Ce module, qu'on pourrait traduire par "Assistant d'optimisation SQL", permet d'analyser une requête et d'en dégager des propositions d'amélioration. Il est accessible depuis la console web, soit depuis la feuille de calcul SQL, soit depuis les écrans de suivi de la performance. Si nous soumettons la requête ci-après, nous obtenons les recommandations de la Figure 4.7 :

```
select * from clients where codepostal='40321'
```

Sélectionner les recommandations

Plan d'exécution d'origine (annoté)

Implémenter

Sélectionner	Résultats de la recherche	Recommandations	Raisonnement	Nouveaux détails du gain plan (%) d'exécution	Comparer les plans d'exécution
	Index: Le plan d'exécution de cette instruction peut être amélioré en créant un ou plusieurs index.	Envisagez d'exécuter Access Advisor pour améliorer la conception du schéma physique ou de créer l'index recommandé, SCOTT.CLIENTS ("CODEPOSTAL")	La création des index recommandés améliore de façon considérable le plan d'exécution de cette instruction. Il pourrait cependant être préférable d'exécuter "Access Advisor" en utilisant une charge globale SQL représentative contrairement à une seule instruction. Ceci permettra d'obtenir des recommandations d'index complètes prenant en compte le coût de maintenance des index et de la consommation d'espace supplémentaire.	98.82	

Retour

Figure 4.7
Recommandations sur une requête.

SQL Tuning Advisor propose de visualiser le plan d'exécution que cela donnerait en appliquant les recommandations au moyen des icônes de visualisation présentes dans les colonnes Nouveaux détails du plan d'exécution et Comparer les plans d'exécution (voir Figure 4.8). Le bouton Implémenter permet de créer directement l'index recommandé.

Cette fonction peut être utile, mais elle reste cependant relativement limitée et ne propose qu'une partie des optimisations possibles.

Comparer les plans d'exécution

Plan d'exécution d'origine (annoté)

Valeur de hachage de plan 4036073249

Tout développer | Tout réduire

Opération	ID de ligne	Objet	Type de l'objet	Ordre	Lignes	Octets	Coût Heure	Coût de l'UC	E/S - Coût
SELECT STATEMENT	0			2		0,096	171.3	17 915 372	170
TABLE ACCESS FULL	1	SCOTT.CLIENTS	TABLE	1		0,096	171.3	17 915 372	170

Nouveau plan d'exécution avec index

Valeur de hachage de plan 2559191466

Tout développer | Tout réduire

Opération	ID de ligne	Objet	Type de l'objet	Ordre	Lignes	Octets	Coût Heure	Coût de l'UC	E/S - Coût
SELECT STATEMENT	0			3		0,096	2.1	16 153	2
TABLE ACCESS BY INDEX ROWID	1	SCOTT.CLIENTS	TABLE	2		0,096	2.1	16 153	2
INDEX RANGE SCAN	2	IDX\$\$_01B40001	INDEX	1			1.1	8 371	1

Figure 4.8
Écran de comparaison des plans d'exécution.

4.2.3 SQL Access Advisor (Oracle)

Cette fonction de conseil présente des similitudes avec SQL Tuning Advisor. La principale différence est qu'elle travaille sur un ensemble de requêtes pour proposer ses conseils, alors que SQL Tuning Advisor travaille sur une seule requête à la fois. SQL Access Advisor prend en compte le nombre d'exécutions de chaque requête.

Vous accédez à cette fonction par le Centre de Conseil, en choisissant Fonctions de conseil SQL puis SQL Access Advisor.

INFO

Il est possible que vous rencontriez quelques difficultés à exécuter une tâche SQL Access Advisor. Cela peut venir du fait que vous utilisez un navigateur sur un poste de travail en français sur une base qui est configurée en anglais (voir avec l'instruction ci-après).

```
show parameter nls_language
```

Si c'est le cas, un workaround référencé consiste à effectuer l'opération suivante dans votre navigateur (ici expliqué avec Internet Explorer). Dans le menu Outils > Options Internet, cliquez sur Langues, ajoutez la langue "Anglais (états unis)" et passez-la en première position. Cela aura pour effet de passer les écrans de la console en langue anglaise mais vous permettra de mener à terme l'exécution de SQL Access Advisor.

Dans les conseils obtenus de SQL Access Advisor, il se peut que vous vous demandiez à quoi correspond l'action RETAIN INDEX qui suggère le code SQL suivant :

```
/* RETAIN INDEX "UnOwner"."UnIndex" */
```

Sachez que c'est juste une confirmation de la part de SQL Access Advisor qui estime que votre index est utile et que vous devriez le garder. Ce code SQL est seulement un commentaire, et rien de plus.

4.2.4 SQL Trace (Oracle)

SQL Trace est la méthode historique permettant de tracer l'activité d'une base de données. Ce n'est plus la plus recommandée par Oracle qui, à présent, conseille plutôt d'utiliser la console web. Conséquence, cette fonction a très peu évolué depuis plusieurs versions d'Oracle et n'est de ce fait, pas très pratique à mettre en œuvre.

Générer les traces

Il est possible d'activer la fonction de trace du SQL avec un filtrage aux niveaux suivants :

- instance ;
- session utilisateur ;
- identifiant client ;
- service, module, action.

Les traces sont placées dans des fichiers stockés sur le serveur dans le répertoire désigné par le paramètre UDUMP, dont vous pouvez consulter la valeur à l'aide de l'instruction suivante :

```
SHOW PARAMETERS user_dump_dest;
```

Vous pouvez modifier ce paramètre à l'aide de l'instruction :

```
ALTER SYSTEM SET user_dump_dest = '/nouveau_repertoire_user_dump' SCOPE=both;
```


Pour avoir des informations plus complètes dans les fichiers de trace, il faut s'assurer que le paramètre `timed_statistics` est à `true` et, le cas échéant, le modifier (par défaut, il est à `false`) à l'aide des instructions suivantes :

```
SHOW PARAMETERS timed_statistics;  
ALTER SYSTEM SET timed_statistics = true SCOPE=memory;
```

Pour activer la trace au niveau instance, vous pouvez exécuter une des instructions suivantes :

```
ALTER SYSTEM SET sql_trace = true SCOPE=MEMORY;  
EXECUTE DBMS_MONITOR.DATABASE_TRACE_ENABLE(waits => TRUE, binds => FALSE,  
instance_name => 'orcl');
```

Pour désactiver la trace au niveau instance, vous pouvez exécuter une des instructions suivantes :

```
ALTER SYSTEM SET sql_trace = false SCOPE=MEMORY;  
EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE(instance_name => 'orcl');
```

Pour les autres niveaux de filtrage (session, identifiant client, service, module, action), vous aurez besoin d'informations complémentaires que vous pourrez consulter dans la vue `V$SESSIONS` à l'aide de la requête suivante :

```
select sid,serial#,username,module,action,service_name,client_identifier  
from V$SESSION
```

Vous pouvez influencer sur la valeur `client_identifier` depuis l'application en utilisant la fonction PL/SQL suivante :

```
dbms_session.set_identifier('VotreIdentifiant');
```

Pour influencer sur les valeurs de module et action depuis votre application, vous pouvez utiliser les procédures suivantes du package `DBMS_APPLICATION_INFO` :

- `DBMS_APPLICATION_INFO.SET_MODULE (module_name, action_name);`
- `DBMS_APPLICATION_INFO.SET_ACTION (action_name);`

Pour activer la trace sur la session courante, vous pouvez utiliser les instructions suivantes qui activent la trace et, optionnellement, y affectent un identifiant permettant de retrouver le fichier de trace plus facilement :

```
ALTER SESSION SET sql_trace = true;  
ALTER SESSION SET tracefile_identifier = matrace25;
```

Pour stopper la trace :

```
ALTER SESSION SET sql_trace = false;
```

Attention, cette instruction requiert le privilège `ALTER SESSION` qui n'est pas donné par défaut aux utilisateurs. Oracle recommande d'ailleurs de ne pas l'accorder systématiquement et de façon permanente.

Vous pouvez activer la trace d'une autre session à partir d'une des instructions suivantes en utilisant les informations de la vue `V$SESSIONS` en paramètres :

```
execute dbms_system.set_sql_trace_in_session(142, 1270, true);
execute dbms_monitor.session_trace_enable(session_id => 142, serial_num
=> 1270, waits => true, binds => false);
```

Pour stopper la trace, exécutez une de ces instructions :

```
execute dbms_system.set_sql_trace_in_session(142, 1270, false);
execute dbms_monitor.session_trace_disable(session_id => 142, serial_num
=> 1270);
```

Pour activer et désactiver les traces SQL avec d'autres niveaux de filtrage, choisissez parmi les procédures suivantes du package `DBMS_MONITOR` :

- `CLIENT_ID_TRACE_ENABLE(client_id, waits, binds, plan_stat);`
- `CLIENT_ID_TRACE_DISABLE(client_id);`
- `SERV_MOD_ACT_TRACE_ENABLE(service_name, module_name, action_name, waits, binds ,instance_name, plan_stat);`
- `SERV_MOD_ACT_TRACE_DISABLE(service_name, module_name, action_name, instance_name);`

Analyser les traces avec tkprof

Nous venons de voir comment capturer des données dans des fichiers de trace. À présent, nous allons voir les moyens permettant de les consulter.

La première chose à faire sera d'identifier votre fichier de trace dans le répertoire contenant toutes les traces, à l'aide de l'heure par exemple. L'instruction `ALTER SESSION tracefile_identifier` étudiée précédemment pourra vous y aider.

Ensuite, l'utilitaire `tkprof` vous permettra de convertir ce fichier de trace binaire (.trc) en fichier texte (.txt) lisible. L'instruction ci-après traite cette conversion :

```
tkprof orcl_ora_2052_MATRACE25.trc orcl_ora_2052_MATRACE25.txt
```

Dans le fichier texte généré, vous trouverez les informations suivantes pour chaque requête :

```
SQL ID : 5x7w56mw30q9x
```

```
select count(*) from cmd_lignes where nolivre=6262 and remise=7
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.06	6.00	8286	8291	0	1
total	4	0.06	6.00	8286	8291	0	1

```
Misses in library cache during parse: 0
```

```
Optimizer mode: ALL_ROWS
```

```
Parsing user id: 81
```

```

Rows      Row Source Operation
-----
      1  SORT AGGREGATE (cr=8291 pr=8286 pw=8286 time=0 us)
     281  TABLE ACCESS FULL CMD_LIGNES (cr=8291 pr=8286 pw=8286 time=50914 us
cost=2317 size=1470 card=210)
*****
```

Par défaut, si une requête est exécutée plusieurs fois, elle sera consolidée et n'apparaîtra qu'une seule fois dans le fichier résultat. Le premier tableau de statistiques indiquera combien de fois chaque requête a été exécutée. Si vous avez activé les statistiques, vous aurez le détail de chaque opération du plan avec les statistiques (cr = Consistent Read Mode, pr = Buffer Gets).

L'option `tkprof explain` permet de réinterroger la base pour obtenir le plan d'exécution de votre requête. Attention, ce n'est pas forcément celui qui était utilisé au moment de la trace.

L'option `tkprof sys=no` permet de ne pas traiter les requêtes récursives effectuées par le SGBDR.

L'utilitaire `trcsess` permet d'agréger dans un même fichier le résultat de plusieurs fichiers de trace en fonction du numéro de session, client id, service, module, action en vue de les traiter avec `tkprof`.

Analyser les traces avec Trace Analyzer

Trace Analyzer, disponible sur Oracle MetaLink dans la note 224270.1, est un utilitaire analogue à tkprof qui donne un rapport au format HTML plus pratique à exploiter. Il fournit une synthèse des instructions les plus consommatrices et, pour chaque instruction, des informations dans une mise en forme agréable. Reportez-vous à sa documentation pour l'installer.

Pour l'utiliser depuis SQL*Plus connecté avec le compte qui a servi à faire la trace, exécutez le script trcanlzt.sql avec votre fichier de trace en paramètre.

```
@D:\Ora\product\11.1.0\db_1\trca\run\trcanlzt.sql orcl_ora_2052_MATRACE25.trc
```

Ce script génère un fichier zip qui contient le rapport en format HTML (voir Figure 4.9).

Consultez le répertoire Doc de Trace Analyzer pour avoir plus d'informations et des explications qui vous aideront à analyser les rapports.

Figure 4.9

Détail d'une requête dans le rapport généré par Trace Analyzer.

Call	Call Count	OS Buffer Gets (disk)	SG Consistent Read Mode (query)	SG Current Mode (current)	Rows Processed or Returned	Library Cache Misses	Times Waited Non-Idle	Times Waited Idle
Parse:	1	0	0	0	0	0	0	0
Execute:	1	0	0	0	0	0	0	0
Fetch:	2	8286	8291	0	1	0	0	0
Total:	4	8286	8291	0	1	0	0	0

ID	PIB	Estim Card	Actual Rows	Row Source Operation	SG Consistent Read Mode (cr)	OS Buffer Gets (pg)	OS Write Calls (pgw)	Time (secs)	DB	Cost	Estim Size (bytes)
1	0	1	1	PORT AGGREGATE	8291	8286	8286	0.000	0		
2	1	210	291	TABLE ACCESS FULL CND_LIGNES	8291	8286	8286	0.061	71612	2317	1470

Back to Top

Explain Plan

ID	PIB	Estim Card	Actual Rows	Cost	Explain Plan Operation	Search Cols ¹	Indexed Cols	Predicates
0		1		2317	SELECT STATEMENT			
1	0	1	1		PORT AGGREGATE			
2	1	210	291	2317	TABLE ACCESS FULL CND_LIGNES			[1]

(1) N/Y: Where X is the number of searched columns from index, which has a total of Y columns.
(2) Actual rows returned by operation (average if there were more than 1 execution).

Tables and Indexes

[...]

Owner	Table Name	Source Plan	Explain Plan	Current Count(*)	Num Rows	Sample Size	Last Analyzed	Avy Row Len	Chain Len	Empty Blocks	Avy Space	Global State	Part	Temp
SCOTT	CND_LIGNES	Y	Y	10000004	2606477	2606477	28-DEC -09 10:40:01	17	0	9425	0	YES	NO	N

(1) CBO statistics.
(2) COUNT(*) up to threshold value of 1000000 (tool configuration parameter).
Back to Top

[...]

Owner	Index Name	Owner	Index Name	Source Plan	Explain Plan	Index Type	Uniqueness	Cols Count	Indexed Columns
SCOTT	CND_LIGNES	SCOTT	PK_CND_LIGNES	N	N	NORMAL	UNIQUE	2	NOCMD NULLYR

Back to Top

[...]

Owner	Table Name	Owner	Index Name	Num Rows	Sample Size	Last Analyzed	Distinct Keys	Level	Leaf Blocks	Avy Leaf Blocks	Avy Leaf Len	Cluster Factor	Global State	Part	Temp
SCOTT	CND_LIGNES	SCOTT	PK_CND_LIGNES	2600906	424885	28-DEC -09 10:40:04	2600906	2	6887	1	1	571767	YES	NO	N

(1) CBO statistics.
Back to Top

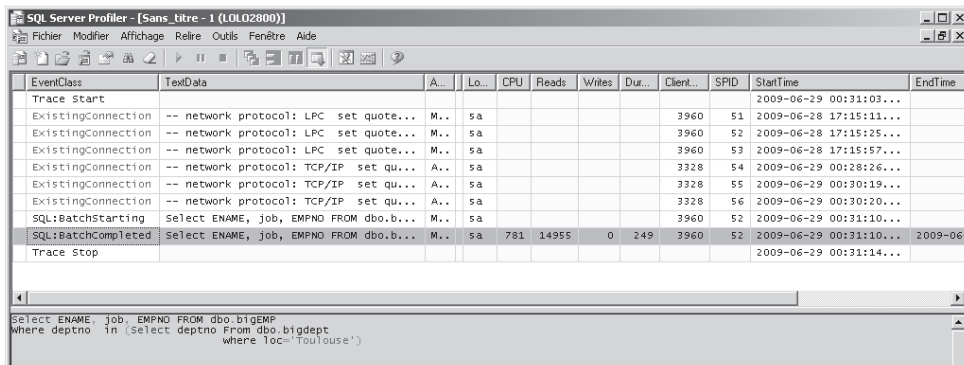
[...]

Owner	Index Name	Col Pos	Column Name	Ref/ Desc	Num Rows	Sample Size	Last Analyzed	Num Nulls	Num Distinct	Distincts	Num Buckets
SCOTT	PK_CND_LIGNES	1	NOCMD	ASC	2606477	2606477	28-DEC -09 10:39:55	0	1000112	9.9949e-07	1
SCOTT	PK_CND_LIGNES	2	NULLYR	ASC	2606477	2606477	28-DEC -09 10:39:55	0	2972	3.3647e-04	1

(1) CBO statistics

4.2.5 Outils SQL Server

SQL Server propose aussi des outils pour aider au diagnostic. L'équivalent de l'outil de trace est SQL Server Profiler qui permet de capturer le trafic SQL d'un serveur.



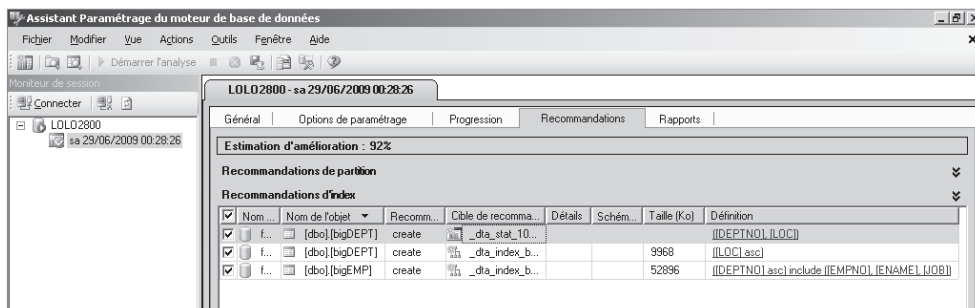
EventClass	TextData	A...	Lo...	CPU	Reads	Writes	Dur...	Client...	SPID	StartTime	EndTime
Trace Start										2009-06-29 00:31:03...	
ExistingConnection	-- network protocol: LPC set quote...	M..	sa					3960	51	2009-06-28 17:15:11...	
ExistingConnection	-- network protocol: LPC set quote...	M..	sa					3960	52	2009-06-28 17:15:25...	
ExistingConnection	-- network protocol: LPC set quote...	M..	sa					3960	53	2009-06-28 17:15:57...	
ExistingConnection	-- network protocol: TCP/IP set qu...	A..	sa					3328	54	2009-06-29 00:28:26...	
ExistingConnection	-- network protocol: TCP/IP set qu...	A..	sa					3328	55	2009-06-29 00:30:19...	
ExistingConnection	-- network protocol: TCP/IP set qu...	A..	sa					3328	56	2009-06-29 00:30:20...	
SQL:BatchStarting	Select ENAME, job, EMPNO FROM dbo.b...	M..	sa					3960	52	2009-06-29 00:31:10...	
SQL:BatchCompleted	Select ENAME, job, EMPNO FROM dbo.b...	M..	sa	781	14955	0	249	3960	52	2009-06-29 00:31:10...	2009-06-
Trace Stop										2009-06-29 00:31:14...	

Select ENAME, job, EMPNO FROM dbo.bigEMP
Where deptno in (select deptno From dbo.bigdept
where loc='Toulouse')

Figure 4.10

Écran de SQL Server Profiler.

L'équivalent de la fonction de conseil Oracle SQL Access Advisor est disponible via l'assistant de paramétrage du serveur (*Database Engine Tuning Advisor*).



Nom...	Nom de l'objet	Recomm...	Cible de recommandation	Détails	Schém...	Taille (Ko)	Définition
<input checked="" type="checkbox"/>	[dbo].[bigDEPT]	create	_dta_stat_10...				[[DEPTNO],[LOC]]
<input checked="" type="checkbox"/>	[dbo].[bigDEPT]	create	_dta_index_b...			9968	[[LOC] asc]
<input checked="" type="checkbox"/>	[dbo].[bigEMP]	create	_dta_index_b...			52896	[[DEPTNO] asc] include [[EMPNO],[ENAME],[JOB]]

Figure 4.11

Écran de l'assistant de paramétrage du serveur.

4.2.6 Outils MySQL

Sous MySQL, les outils disponibles sont un peu moins nombreux, du moins dans la version communautaire. La version Enterprise propose dans ses variantes les plus évoluées un outil de monitoring du serveur. Celui-ci, plutôt orienté production, intègre cependant Query Analyzer, un outil qui permet de suivre les requêtes occupant le plus de ressources.

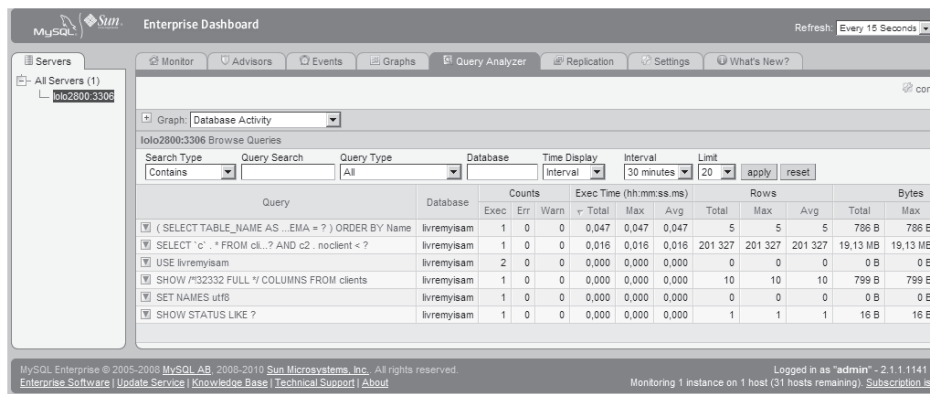


Figure 4.12

Écran du Query Analyzer.

En cliquant sur une requête, vous pourrez accéder à quelques statistiques et au plan d'exécution. *Attention*, le Query Analyzer passe par un proxy pour capturer les requêtes. Vous devez donc rediriger le flux de requête sur le port du proxy qui les enverra vers le serveur.

Si vous ne possédez pas une version Enterprise, il vous reste quand même quelques solutions pour savoir quelles requêtes optimiser. Vous pouvez activer la journalisation de toutes les requêtes avec le paramètre `log`. Ce mode de journalisation ne donne aucune indication relative au temps d'exécution, il permet juste de repérer les requêtes les plus exécutées.

Le paramètre `log_slow_queries` permet d'activer la journalisation des requêtes les plus lentes. Ce sont celles qui durent plus de `long_query_time` secondes ou qui n'utilisent pas d'index si `log-queries-not-using-indexes` est activé. `long_query_time` est exprimé en secondes et ne peut pas être inférieur à une seconde. Vous ne pourrez ainsi pas configurer de tracer les requêtes de plus de 0,2 seconde qu'il serait peut être pertinent d'analyser.

Le fichier journal des requêtes lentes est plus complet que le log général des requêtes, voici un exemple d'une entrée :

```
# Time: 100219 10:44:18
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 25.421875 Lock_time: 0.000000 Rows_sent: 4026540 Rows_
examed: 140
SET timestamp=1266572658;
select c.* from clients c,clients c2 where c.adresse1 like '%A%' and c2.
noclient<15000;
```

On y trouve le texte de la requête ainsi que le temps d'exécution.

Si vous souhaitez tracer plus de requêtes, vous pouvez vous inspirer de la logique du Query Analyzer en utilisant le serveur proxy MySQL disponible sur <http://dev.mysql.com/downloads/>.

Vous devrez le configurer pour qu'il trace les requêtes qui vous intéressent. Ce serveur de proxy peut exécuter du code LUA pour chaque requête qui transite à travers lui. Le répertoire `share\doc\mysql-proxy` contient des exemples de scripts qui pourront vous aider et plus particulièrement le fichier `analyze-query.lua`. En vous documentant un peu, vous pourrez apporter quelques modifications à ce script en remplaçant par exemple les affichages par une écriture dans un fichier à l'aide des instructions suivantes :

```
-- Déclaration au début du fichier
local logger = io.open("querylog.txt", "a+")
-- À la place de print(o)
logger:write(o)
logger:flush()
```

Adaptez les valeurs de `proxy.global.config.analyze_query` et, éventuellement, modifiez la condition qui précède la journalisation par `if o then` si vous ne souhaitez rien filtrer.

Dans le fichier de configuration du proxy, ajoutez cette instruction pour prendre en compte ce script :

```
proxy-lua-script          = share\\doc\\mysql-proxy\\VotreAnalyze-query.lua
```

Une fois que vous aurez adapté ces scripts, vous pourrez obtenir pour chaque requête les informations ci-après, qu'il ne vous restera plus qu'à analyser :

```
# 2010-02-19 10:33:31 [11] user: root, db: livremyisam
Query:          "select * from clients where adresse1 like '%A%'"
Norm_Query:     "SELECT * FROM `clients` WHERE `adresse1` LIKE ? "
query_time:     1004.00 us
```

```
global(avg_query_time): 1091.57 us
global(stddev_query_time): 2087.51 us
global(count): 7
query(avg_query_time): 1004.00 us
query(stddev_query_time): -1.#J us
query(count): 1
```

Le répertoire `share\doc\mysql-proxy` contient aussi le fichier `histogram.lua` qui vous permettra d'obtenir un histogramme des requêtes.

D'autres outils sont disponibles sur les sites suivants :

- <http://www.mysqlperformanceblog.com/tools/>
- <http://hackmysql.com/>

Techniques d'optimisation standard au niveau base de données

5.1 Statistiques sur les données

Les statistiques sont fondamentales pour le fonctionnement du CBO (*Cost Based Optimizer*). Des statistiques erronées ou non représentatives peuvent le conduire à utiliser un plan d'exécution inadapté qui pourrait donc entraîner des performances moins bonnes que ce qu'elles pourraient être.

Sous Oracle 10g, le job `GATHER_STATS_JOB` est préconfiguré pour collecter les statistiques. Sous Oracle 11g, c'est le package `DBMS_AUTO_TASK_ADMIN` qui gère cette activité. Cette tâche est paramétrée pour s'effectuer durant la plage de maintenance qui est, par défaut, configurée de 22 heures à 2 heures en semaine et de 6 heures à 2 heures le week-end. La base ne sera pas forcément entièrement analysée chaque fois. La collecte commencera par les objets qui n'ont pas de statistiques ou qui ont été le plus modifiés.

En cas de création d'une table ou si la nature des données change de façon significative, il faut forcer la collecte au lieu de simplement attendre la collecte automatique suivante. C'est particulièrement vrai en phase de développement alors que le développeur détruit et recrée des tables et charge des jeux de tests différents.

Ce chapitre présente la collecte des statistiques. Vous trouverez à l'Annexe B des informations plus détaillées sur leur nature.

5.1.1 Ancienne méthode de collecte

Historiquement, il fallait utiliser les commandes suivantes pour collecter les statistiques :

```
ANALYZE TABLE <NomTable> COMPUTE STATISTICS;  
ANALYZE INDEX <NomIndex> COMPUTE STATISTICS;
```

Oracle recommande de ne plus les utiliser pour la partie collecte des statistiques de l'optimiseur.

5.1.2 Nouvelle méthode de collecte

À présent, il est conseillé d'utiliser le package DBMS_STATS, qui a un niveau d'analyse plus poussé et fournit donc à l'optimiseur des informations plus détaillées que l'ancienne méthode.

Ce package permet de gérer les statistiques avec les opérations suivantes :

- collecte des statistiques (Gathering Optimizer Statistics) :
 - d'une table (procédure GATHER_TABLE_STATS),
 - d'un index (procédure GATHER_INDEX_STATS),
 - d'un schéma (procédure GATHER_SCHEMA_STATS),
 - de toute la base (procédure GATHER_DATABASE_STATS) ;
- effacement de statistiques ;
- transfert de statistiques (export/import) ;
- restauration d'anciennes statistiques ;
- gestion des statistiques des types utilisateurs (user-defined) ;
- gestion des statistiques étendues (Oracle 11g).

Parmi toutes ces opérations, la collecte des statistiques est l'opération la plus utilisée. Par défaut, elle parcourt toutes les données des tables à analyser. Cette opération peut donc prendre un temps significatif sur de grosses bases de données. Dans ce cas, il peut être intéressant de collecter les statistiques par échantillonnage à l'aide du paramètre ESTIMATE_PERCENT, présent dans toutes les procédures GATHER_*.

La collecte des statistiques d'une table implique la collecte pour la table, les colonnes et les index associés à cette table.

Exemple de collecte des statistiques du schéma SCOTT :

```
execute dbms_stats.GATHER_SCHEMA_STATS('SCOTT');
```

Les opérations de transfert des statistiques peuvent servir à mettre au point des requêtes sur une base de test moins remplie, à partir des statistiques de la base de production. L'optimiseur utilisant les statistiques pour faire ses choix, il se comportera comme si la base de test contenait les données de production.

Concernant les statistiques des types utilisateurs (*user-defined*), elles sont utiles uniquement si vous utilisez des types utilisateurs dans vos tables et les index de domaine. Vous n'aurez donc normalement jamais besoin des opérations qui y sont associées.

Les statistiques sont stockées dans la métabase et accessibles via des requêtes SQL (voir Annexe B). On peut constater qu'elles contiennent de nombreuses informations sur les tables, les index et chaque colonne.

Les principales informations collectées sur les tables sont :

- le nombre d'enregistrements ;
- l'espace occupé en bloc ;
- la taille moyenne d'un enregistrement.

Les principales informations collectées sur les index sont :

- le nombre d'enregistrements ;
- le nombre de blocs feuilles ;
- des informations relatives au foisonnement (voir Annexe B, section B.4).

Les principales informations collectées sur les colonnes sont :

- le nombre de valeurs distinctes et nulles ;
- les valeurs hautes et basses ;
- la taille moyenne des données de la colonne ;
- la densité de la colonne (voir ci-après).

Au-delà de leur utilisation par l'optimiseur, certaines de ces statistiques peuvent être utiles pour avoir des informations sur la base de données. En effet, à l'aide de ces statistiques, le développeur ou le DBA peut repérer facilement les plus grosses tables, les colonnes qui ne sont jamais remplies, etc.

5.1.3 Sélectivité, cardinalité, densité

La sélectivité est l'estimation de la fraction de lignes sélectionnées lors d'une opération de filtrage. Elle permet de calculer la cardinalité d'une opération qui est égale à :

(Nombre de lignes – Nombre de Valeurs NULL) × Sélectivité.

On dira, par exemple, que la condition `NoCmd=123456` sur la table des commandes contenant un million d'enregistrements a une forte sélectivité puisqu'elle réduit la sélection à une ligne sur un million et que la condition `DateCommande>To_Date('1/1/2009','dd/mm/yyyy')` a une faible sélectivité car elle filtre seulement 37 % des valeurs (374 311 lignes sur un million de lignes).

On voit que la sélectivité dépend de la nature des données et de la nature des conditions. Ainsi, lorsqu'une condition est une égalité, la sélectivité est égale à la densité de la colonne (colonne Density des statistiques).

La densité est définie par $\frac{1}{\text{NombreValeurDistincte}}$. On voit que, plus il y a un nombre de valeurs distinctes important, meilleure est la sélectivité.

Lorsqu'une condition est un intervalle (opérateurs BETWEEN, <, >), la sélectivité vaut :

$\frac{\text{taille de l'étendue de l'intervalle}}{\text{taille de l'étendue de la colonne}}$, la taille de l'étendue de la colonne étant déterminée

par les colonnes `Low_Value` et `High_Value` (les chaînes de caractères et les dates sont converties en une valeur numérique).

Si l'intervalle est ouvert (par exemple, `NoCmd > 1000`), on considère qu'il s'arrête à la borne du côté ouvert.

Notez que, par défaut, le SGBDR considère que les données sont linéairement réparties. Nous verrons plus tard comment les histogrammes permettent d'intégrer le fait que ce n'est pas toujours le cas.

Dans certains cas, le SGBDR utilise des valeurs fixes pour la sélectivité car il n'a pas d'éléments permettant de calculer la sélectivité.

L'utilisation d'une fonction implique potentiellement une densité du résultat de la fonction différente de celle de la colonne passée en paramètre. Concernant l'utilisation d'un LIKE commençant par un caractère % ou _, le SGBDR ne possède aucune statistique sur le fait qu'une chaîne contienne telle ou telle sous-chaîne, il ne peut donc estimer aucune sélectivité fondée sur les statistiques. Un LIKE ne commençant

pas par un % est considéré comme un intervalle, ainsi `Nom like 'I%'` sera converti en `Nom >='I'` and `Nom < 'J'` par l'optimiseur.

<i>Condition</i>	<i>Sélectivité</i>
<i>Fonction(Nom)='XX'</i>	1 %
<i>Fonction(Nom) <> 'XX'</i>	5 %
<i>Fonction(Nom) > 'XX'</i>	5 %
Utilisation du binding : <i>Nom > : B (voir Chapitre 8, section 8.3, "Utilisation du binding")</i>	5 %
Utilisation de <code>like</code> : <i>Nom like '%I%'</i>	5 %

Combinaison des conditions

Lorsqu'il y a plusieurs conditions, le SGBDR considère qu'elles sont indépendantes et multiplie donc les sélectivités entre elles. Cette logique marche globalement tant que les colonnes sont indépendantes, mais lorsqu'elles sont corrélées, les résultats sont erronés.

Prenons un exemple avec la requête suivante, sachant que la ville 'Berlin' est située dans le pays 'Germany' :

```
select * from clients where ville='Berlin' and pays='Germany'
```

Analysons le comportement du SGBDR :

Nombre d'enregistrements dans la table CLIENTS	45 000
Sélectivité de <i>pays='Germany'</i>	10,1 %
Estimation de <i>pays='Germany'</i>	4 546 enregistrements
Sélectivité de <i>ville='Berlin'</i>	0,1 %
Estimation de <i>ville='Berlin'</i>	34 enregistrements
Sélectivité combinée	0,01 % (10,1 % × 0,1 %)
Estimation de <i>pays='Germany' And ville='Berlin'</i>	4 enregistrements
Nombre réel d'enregistrements vérifiant <i>pays='Germany' And ville='Berlin'</i>	34 enregistrements

Nous constatons qu'il y a une erreur de facteur 8 (4 enregistrements au lieu de 34), ce qui risque de conduire l'optimiseur à choisir un chemin d'exécution inadapté dans certaines requêtes.

Ce problème peut être contourné avec l'utilisation de l'échantillonnage dynamique (*Dynamic Sampling*), activable à l'aide du hint `DYNAMIC_SAMPLING` (voir Chapitre 7, section 7.1, "Utilisation des hints sous Oracle").

```
/*+ dynamic_sampling(AliasDeTable Niveau) */
```

Le premier paramètre du hint est l'alias de la table sur laquelle il faut faire l'échantillonnage dynamique. Le second paramètre est le niveau d'échantillonnage. Celui-ci peut prendre les valeurs 1 à 10 : la valeur 1 échantillonne sur 32 blocs de données, les valeurs 2 à 9 échantillonnent de 2 à 256 blocs, la valeur 10 analyse toute la table.

```
select /*+ dynamic_sampling(c 1) */ * from clients c  
where ville='Berlin' and pays='Germany'
```

Cette nouvelle version de la requête utilisant le hint `DYNAMIC_SAMPLING` donne une cardinalité de l'opération de 32, ce qui est très proche de la réalité qui est de 34 enregistrements.

Données non réparties linéairement

Nous avons vu précédemment que, lors de la sélection d'intervalles, le SGBDR considère par défaut que les données sont réparties linéairement entre la borne basse et la borne haute. Cependant, ce n'est pas toujours le cas.

Cela arrive lorsque les données ne sont naturellement pas réparties linéairement. Par exemple, dans un annuaire national, il y aura plus de personnes à Paris qu'en Ardèche.

Cela arrive aussi lors de l'utilisation de valeurs spéciales comme on peut en retrouver dans certains modèles de données. Imaginez que vous ayez besoin de pouvoir affecter des employés à un département 'Non affecté'. Vous donnez à ce département un numéro spécial, par exemple 9999999, bien en dehors de la plage des numéros classiques, compris par exemple entre 0 et 10 000, afin de ne pas le confondre avec les autres numéros de départements (cela aurait aussi pu être 0 ou -9999999).

Vous vous retrouvez ainsi avec 99,99 % des valeurs entre 0 et 10 000 et quelques valeurs à 9999999.

Cependant, l'optimiseur, considérant que les données sont réparties linéairement, estime qu'entre 0 et 1 000 il y a 0,01 % des valeurs alors qu'il y en a plutôt 10 %.

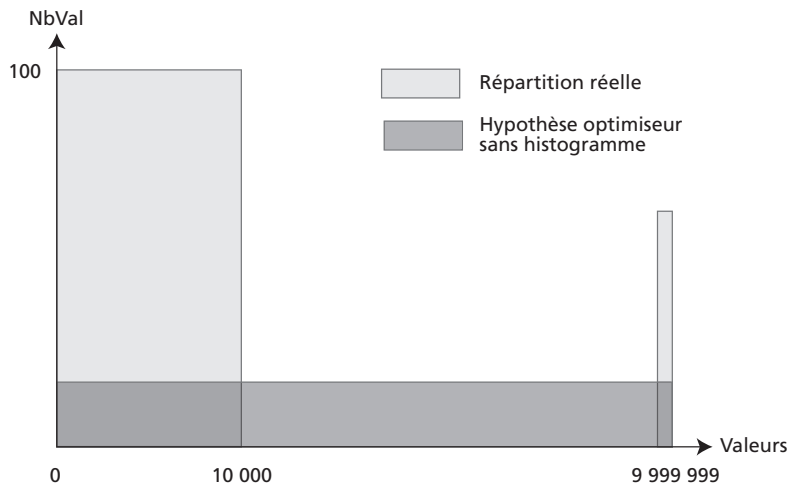


Figure 5.1

Comparaison de répartition réelle et linéaire.

Afin d'améliorer les performances de l'optimiseur sur des données non réparties linéairement, le SGBDR met les histogrammes à notre disposition. Ils seront un moyen de synthétiser la répartition des données et de résoudre le problème des valeurs extrêmes particulières et, dans certaines limites, celui des répartition non linéaires des données.

Il existe deux types d'histogrammes :

- de fréquence ;
- de distribution.

L'histogramme de fréquence calcule pour chaque valeur distincte le nombre d'occurrences. Il est utilisé pour les distributions ayant peu de valeurs distinctes (moins de ≈ 100 valeurs). Il est directement utilisé pour calculer la sélectivité lors des tests d'égalité.

L'histogramme de distribution coupe les données en n tranches égales et détermine les bornes de chaque tranche.

Normalement, le SGBDR détectera les colonnes sur lesquelles le calcul d'un histogramme est pertinent et déterminera le type d'histogramme en fonction du nombre de valeurs distinctes.

Vous trouverez des informations plus détaillées sur les histogrammes à l'Annexe B, section B.3.

Statistiques étendues (11g)

Oracle 11g introduit la notion de statistiques étendues qui permet de créer des statistiques (y compris des histogrammes) sur des ensembles de colonnes ou des expressions en utilisant la fonction `DBMS_STATS.CREATE_EXTENDED_STATS (ownname, tabname, extension)`. *ownname* et *tabname* permettent de désigner la table et *extension* contient soit une liste de colonnes entre parenthèses séparées par des virgules, soit une expression entre parenthèses. Cette fonction retourne le nom de la statistique et n'est donc pas utilisable directement avec l'instruction `EXEC` mais peut être appelée depuis une instruction `SELECT` :

```
Select dbms_stats.create_extended_stats(null,'CLIENTS','(PAYS,VILLE)')
AS NomStat from dual;
```

Cette instruction a pour effet de créer une statistique sur la combinaison de colonnes Pays, Ville de la table `CLIENTS` du schéma courant.

Nous forçons le calcul d'un histogramme sur cette colonne (opération qui ne sera généralement pas nécessaire) :

```
exec dbms_stats.gather_table_stats(null,'clients',method_opt => 'for columns
SYS_STUNUIEHWAFAS55IIEQVJF#W$ size 254');
```

`SYS_STUNUIEHWAFAS55IIEQVJF#W$` est la valeur retournée par l'instruction `SELECT` de création de la statistique étendue ou par la requête suivante qui permet de récupérer le nom d'une statistique étendue.

```
Select dbms_stats.show_extended_stats_name(null,'clients','(PAYS,VILLE)')
AS NomStat FROM dual;
```

Une fois l'histogramme créé et les statistiques actualisées, si on exécute à nouveau la requête suivante :

```
select * from clients where ville='Berlin' and pays='Germany'
```

nous pouvons constater que l'estimation par l'optimiseur de la requête est à présent de 41 enregistrements au lieu de 4 enregistrements initialement estimés ce qui est plus proche de la réalité qui est de 34 lignes. Cependant, cette estimation est moins bonne que celle faite par l'échantillonnage dynamique mais elle est moins coûteuse.

Nous avons vu précédemment que l'utilisation de fonctions dans des prédicats conduisait l'optimiseur à utiliser des valeurs prédéfinies. Il estime ainsi que la requête ci-après retourne 1 % des enregistrements de la table :

```
select * from clients where upper(adresse1)='887 PONCE STREET'
```

Si nous créons une statistique sur l'expression et si nous collectons les statistiques de la table `CLIENTS`, nous constatons que l'optimiseur estime à présent correctement qu'il n'y a qu'une seule ligne sélectionnée, ce qui est bien le cas :

```
select dbms_stats.create_extended_stats(null,'clients','(upper(adresse1))')
from dual;
exec dbms_stats.gather_table_stats(null,'clients');
```

MS SQL Server

Sous SQL Server, les statistiques ne sont pas systématiquement calculées pour chaque colonne, par contre, elles intègrent systématiquement un histogramme sur les clés d'index primaires et secondaires.

Les instructions `CREATE STATISTICS` et `UPDATE STATISTICS` permettent de créer et de mettre à jour des statistiques manuellement.

SQL Server donne, comme les statistiques étendues d'Oracle 11g, la possibilité de calculer des statistiques sur des combinaisons de colonnes afin de résoudre le problème de la mauvaise estimation des combinaisons de conditions.

MySQL

MySQL intègre aussi des statistiques et des histogrammes.

L'instruction `ANALYZE TABLE` permet de forcer la collecte des statistiques.

Certaines données des statistiques sont consultables en interrogeant la table `INFORMATION_SCHEMA.STATISTICS` mais il y a bien plus d'informations que celles présentées dans cette table indépendante des moteurs, car la gestion des statistiques est propre à chaque moteur.

5.2 Utilisation des index

Un premier type d'optimisation consiste à mettre en place des objets dédiés à l'optimisation des tables : les index. Sous Oracle, ils peuvent être soit de type B*Tree soit de type bitmap. Leur action est généralement assez efficace, sans pour autant que leur mise en place nécessite beaucoup de changements au niveau de l'application. Le tout sera de choisir le type d'index le plus adapté à la situation. Les index présentent l'avantage de pouvoir être également mis en œuvre au sein de logiciels réalisés par des tiers et pour lesquels vous n'avez pas accès au code source.

Les index, d'une manière générale, permettent d'améliorer – parfois considérablement – les performances en lecture, mais ils peuvent légèrement dégrader les performances en écriture car le SGBDR doit les maintenir en plus des tables lors

des modifications des données indexées (insertion, modification de la clé d'index, suppression de l'enregistrement). Sur des tables qui ont des accès en écriture très concurrents, la mise à jour des index peut provoquer des contentions d'accès à ceux-ci et donc des latences d'écriture.

5.2.1 Index B*Tree

Utilisés historiquement par les bases de données relationnelles, les index B*Tree sont présents dans de nombreux SGBDR.

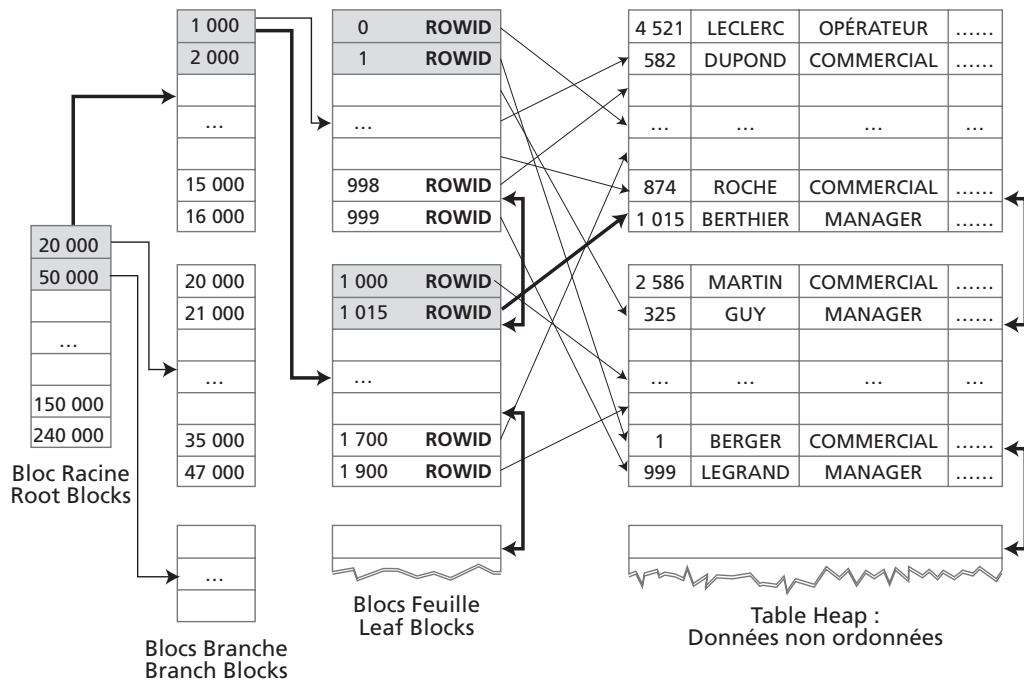
Leur principe est de contenir les valeurs des clés de l'index de façon ordonnée dans une structure arborescente. Ainsi, ils permettent de trouver très rapidement une valeur précise en parcourant l'arbre de la racine vers les feuilles et non pas en recherchant la valeur dans l'ensemble des enregistrements de la table. De plus, les feuilles sont chaînées entre elles (voir Figure 5.2). Aussi, lorsqu'il est nécessaire de parcourir l'index séquentiellement, on passe de feuille en feuille sans devoir remonter au niveau des branches.

Un index B*Tree porte sur une ou plusieurs colonnes qu'on appelle "clés de l'index". Suivant le type de l'index, les clés doivent être uniques (dans le cas d'index uniques) ou peuvent contenir des doublons. Elles peuvent être des champs de type numérique, date ou chaîne de caractères mais pas de type BLOB ou CLOB. Lors de la création de l'index, on spécifie pour chaque champ de la clé son ordre de tri qui, par défaut, sera ascendant (ASC) mais peut être décroissant (DESC) comme dans la clause `ORDER BY` de l'instruction `SELECT`.

Pour comprendre le fonctionnement, prenons un exemple. Pour obtenir l'enregistrement ayant pour clé la valeur 1015 dans l'index B*Tree représenté à la Figure 5.2, le SGBDR va procéder de la manière suivante :

1. Il parcourt le bloc racine jusqu'à ce qu'il rencontre une valeur supérieure.
2. Il va sur le bloc pointé par la valeur qui précède, ce qui le conduit au premier bloc branche.
3. Dans le bloc branche, il répète la même opération, ce qui le conduit à un bloc feuille.
4. Dans le bloc feuille, il recherche la valeur. S'il ne la trouve pas, il s'arrête à la première valeur supérieure rencontrée et renvoie l'information signalant que la valeur recherchée n'existe pas dans la table. S'il la trouve, l'entrée d'index pointé permet d'obtenir le RowID de l'enregistrement correspondant.

5. Le RowID permet de récupérer l'enregistrement dans le tas (*heap*).



Le RowID permet de pointer sur les lignes stockées dans le Tas

Figure 5.2

Exemple de parcours d'un index B*Tree.

On constate que nous avons besoin de lire seulement trois blocs de données d'index plus un bloc de données dans la partie tas de la table pour obtenir l'enregistrement, alors qu'un parcours complet de la table aurait nécessité d'accéder à tous ses blocs.

La hauteur de l'arbre est variable mais excède rarement trois ou quatre niveaux. Si on considère qu'un bloc non feuille peut pointer sur 100 à 500 blocs (cela dépend de la taille de la clé), l'exemple précédent (d'une hauteur de trois) pourrait avoir 10 000 à 250 000 feuilles. La hauteur de l'arbre a une progression logarithmique par rapport au nombre d'enregistrements dans la table. Puisque le nombre d'accès nécessaires pour accéder à un élément dépend de la hauteur de l'arbre, il est donc lui aussi logarithmique.

Sans index, sur une table de n lignes, il faut en moyenne parcourir $n/2$ lignes pour trouver une ligne unique et n lignes pour trouver toutes les occurrences d'une valeur

dans une colonne non unique alors que, dans un index, il faut en moyenne parcourir une proportion de $\log(n)$ lignes pour trouver une ligne. Par exemple, pour une table d'un million d'enregistrements, sans index il faudra parcourir 1 000 000 d'enregistrements alors qu'avec un index il suffira de parcourir environ 30 enregistrements.

Étant donné la progression logarithmique, accéder à un enregistrement en passant par une clé indexée sur une table de 10 millions d'enregistrements ne nécessitera pas beaucoup plus d'accès que sur une table de 10 000 enregistrements.

Par défaut, toutes les clés primaires ont un index B*Tree unique qui est créé automatiquement lors de la déclaration de la clé primaire.

Nous venons de voir que, plus il y a d'enregistrements dans une table, plus le gain est grand. Le corollaire est vrai aussi : moins il y a d'enregistrements dans une table, moins l'intérêt d'un index est grand. En effet, sur des petites tables – moins de 1 000 lignes – et ayant des lignes de petite taille, il n'est pas pertinent d'ajouter des index. Lorsque des index sont présents mais inutiles, ils ne servent pas car l'optimiseur n'y voit aucun intérêt. Ils sont par contre maintenus et occupent de l'espace.

Nous avons vu que la dernière étape consiste à récupérer l'enregistrement à partir du RowID obtenu dans le B*Tree. Cette opération est plutôt performante puisque le RowID contient l'adresse du bloc contenant l'enregistrement. Cependant, lorsque plusieurs valeurs sont recherchées par le biais de l'index, le SGBDR fait des va-et-vient entre l'index et le tas qui peuvent, finalement, être assez coûteux. Donc, si l'optimiseur estime qu'il va récupérer à travers l'index plus de 5 % des données de la table, il choisira de ne pas l'utiliser. En effet, au-delà de ce seuil (approximatif), les performances risquent de se dégrader et il sera moins performant d'utiliser l'index que de ne pas l'utiliser.

Le nombre de lignes récupérées dans l'index dépend soit de la nature des données, soit de la nature des conditions. Ainsi, il ne sera pas pertinent de placer des index sur des colonnes qui ont beaucoup de valeurs dupliquées (c'est-à-dire peu de valeurs distinctes) puisque, quelle que soit la condition sur ce genre de colonne, l'optimiseur estimera que trop de lignes seront récupérées pour que cela soit intéressant. De la même façon, il n'est pas surprenant qu'un index ne soit pas utilisé sur des conditions peu sélectives qui auraient pour effet la récupération d'un grand nombre de lignes.

L'optimiseur décidera de l'usage d'un index en fonction de l'estimation qu'il aura faite du nombre de lignes qu'il va récupérer. Cette information dépend de l'image que l'optimiseur a des données et de sa capacité à estimer la sélectivité des conditions. On voit ici tout l'intérêt d'avoir des statistiques fidèles aux données.

Il existe cependant un cas où, même s'il y a un grand nombre d'enregistrements concernés, l'optimiseur choisira d'utiliser l'index à coup sûr : les requêtes où le parcours seul de l'index suffit à obtenir la réponse.

```
select count(*) from clients
where noclient between 10000 and 100000
```

Dans l'exemple précédent, le parcours de l'index de la clé primaire (Noclient) suffit pour répondre à la requête. Il est donc utilisé malgré le fait qu'il récupère 60 % des enregistrements. La requête suivante, elle, qui nécessite d'avoir accès au tas de la table pour tester le champ Nom, n'utilise pas l'index, car les va-et-vient entre l'index et le tas sont trop pénalisants sur un tel volume.

```
select count(*) from clients
where noclient between 10000 and 100000 and Nom is not null
```

Les index contenant les valeurs des clés de façon ordonnée, ils pourront aussi être utilisés pour les tris. Des tris peuvent être nécessaires dans différentes situations, telles que l'utilisation d'une clause ORDER BY, DISTINCT ou GROUP BY dans une requête ou le recours par l'optimiseur à des jointures de type Sort-Merge. Notez qu'un index classé de façon ascendante pourra être utilisé aussi bien pour des tris ascendants que descendants, la différence entre le sens du tri de l'index et celui du tri demandé n'est généralement pas significative.

ATTENTION

L'utilisation d'index a un impact lors des accès en écriture. Le coût de maintien des index est relativement faible mais, s'il y en a beaucoup et que la table subisse de nombreuses modifications, cela peut finalement devenir significatif.

Sur un autre plan, les index occupent de l'espace de stockage. Cet espace est, pour simplifier, celui qui est requis pour les feuilles et qui correspond à l'espace nécessaire pour les valeurs des clés plus 1 RowID par enregistrement. Il peut être significatif, voire dépasser l'espace de la table si toutes les colonnes sont indexées ou s'il y a de nombreux index.

Si vous avez un doute sur le fait qu'un index soit utilisé, vous pouvez le mettre sous surveillance à l'aide de l'instruction suivante :

```
alter index Nom_Index monitoring usage;
```

puis vérifier, quelques jours ou semaines plus tard, s'il a été utilisé depuis l'activation de la surveillance en consultant la colonne Used de la requête suivante :

```
SQL> select * from v$object_usage;
```

INDEX_NAME	TABLE_NAME	MONITORING USED	START_MONITORING	END_MONITORING
IS_CLIENTS_NOM	CLIENTS	YES	NO	01/22/2010 10:48:32

Combinaison des index B*Tree et index composites

De façon générale, un seul index B*Tree peut être utilisé par accès à une table. Si le plan d'exécution nécessite plusieurs accès à la même table (autojointure, par exemple), des index différents pourront être utilisés pour chaque accès. Cela signifie que si une requête contient deux conditions sur deux colonnes indexées séparément, alors seulement l'index le plus pertinent sera utilisé et le second ne servira à rien. Dans des cas comme celui-là, il peut être intéressant d'utiliser des index composites (aussi appelés index "multicolonnes") qui permettront d'indexer plusieurs colonnes.

L'ordre des colonnes a son importance, nous allons l'illustrer par quelques exemples :

Un index composite Pays,Ville sera très efficace dans les cas suivants :

```
where Pays='France' and ville='Toulouse'
where Pays='France' order by ville
where Pays='France' group by ville
order by pays,ville
```

Moyennement efficace dans les cas suivants :

```
Where ville='Toulouse'
```

Un index composite Ville, Pays sera très efficace dans les cas suivants :

```
where Pays='France' and ville='Toulouse'
Where ville='Toulouse'
```

Moyennement efficace dans les cas suivants :

```
where Pays='France' order by ville
```

Inefficace dans les cas suivants :

```
order by pays,ville
```

Un index composite sera le plus efficace dans les cas suivants :

- présence de conditions sur toutes les colonnes de l'index ;
- présence de conditions sur toutes les premières colonnes de l'index ;
- nécessité d'un tri suivant l'ordre de l'index ;
- présence d'une combinaison entre une condition sur les premières colonnes de l'index et un tri suivant l'ordre de colonnes restantes de l'index.

Un index composite sera peu efficace mais potentiellement intéressant malgré tout dans les cas suivants :

- S'il y a des conditions sur des colonnes d'un index qui ne soient pas consécutivement les premières de l'index. Dans ce cas, l'optimiseur pourra décider de faire un Skip Scan de l'index.
- Un tri sur les premières colonnes et des conditions sur les colonnes restantes de l'index car dans ce cas l'index devra être parcouru entièrement de façon séquentielle.

Un index composite sera inefficace dans de nombreux cas, mais plus particulièrement si la requête nécessite un tri sur les colonnes de l'index mais dans un ordre différent de celui de l'index.

Cas disqualifiant les index

Nous avons vu que l'optimiseur décidait d'utiliser les index quand il pensait que leur usage allait améliorer les performances. Il faut cependant avoir en tête qu'il existe des cas pour lesquels les index ne seront pas utilisés, car ils sont inutilisables.

L'utilisation de `like '%...'`, c'est-à-dire un LIKE dont le masque commence par une chaîne libre, ne permet pas d'utiliser un index en mode Unique Scan ou Range Scan.

L'utilisation de fonctions ou plus généralement d'expressions sur les colonnes empêche celle des index. Néanmoins, l'utilisation d'index sur fonction que nous étudierons plus tard permet d'apporter une solution à cette limitation si nécessaire.

Par exemple, les instructions suivantes empêchent l'utilisation des index sur les colonnes `Noclient` et `Ville`.

```
select * from clients where noclient*10=40000 ;  
select * from clients where upper(Ville)='PARIS' ;
```

INFO

Une condition entrant dans un des cas disqualifiants pourra être appliquée malgré tout sur l'index si l'optimiseur a déterminé pour d'autres raisons que l'index pouvait être intéressant.

Quelles colonnes indexer ?

Souvenez-vous, les clés primaires sont, par défaut, indexées, mais quelles autres colonnes est-il pertinent d'indexer ? On l'a vu, les index ont un coût, aussi bien en termes de ressources pour les maintenir à jour qu'en espace occupé. Il n'est donc pas intéressant de les maintenir s'ils ne servent quasiment jamais. Je vous conseille donc d'indexer :

- Les colonnes les plus utilisées dans les clauses WHERE sans qu'elles entrent dans les cas disqualifiants.
- Les colonnes les plus utilisées dans les jointures. Cela pourra favoriser l'usage de Sort Merge Join et rendra les Nested Loop plus performantes. Pour les Hash Join, l'intérêt sera de constituer la table de hachage (hash table) à partir de l'index plutôt que la table.
- Les colonnes qui ont une densité (nombre valeurs distinctes/nombre lignes) faible (inférieure à 5 %) en relation bien sûr avec le premier point. Inutile de créer des index qui ne servent à rien.
- Les colonnes filles de clé étrangère (Foreign Key) si la table mère subit des DELETE. En effet, chacun d'eux provoquera une requête sur ses tables filles pour contrôler le respect des contraintes d'intégrité référentielle (CIR).

Mise en œuvre

La syntaxe de base permettant de créer un index et un index unique est la suivante :

```
create index <Nom_Index> on <NomTable>(<col1>[,<col2>,...])
create unique index <Nom_Index> on <NomTable>(<col1>[,<col2>,...])
```

Nous avons déjà étudié l'impact des histogrammes sur les statistiques. Nous allons tester ici comment cela va se traduire par l'utilisation ou non d'un index.

Par exemple, si nous créons un index IS_CLIENTS_PAYS sur la colonne Pays de la table CLIENTS à l'aide de l'instruction suivante :

```
create index is_clients_pays on clients(pays);
```

et que nous testions la requête suivante avec et sans histogramme :

```
select * from clients where pays='Cameroun'
```

À la Figure 5.3a, la sélectivité du prédicat est estimée par la densité de la colonne à 2 % avec un fort foisonnement, et l'index ne suffit pas à répondre. L'optimiseur détermine donc qu'il sera moins coûteux de faire un Full Table Scan.

À la Figure 5.3b, l'histogramme de fréquence permet de déterminer une sélectivité plus forte du prédicat sur la colonne Pays $\approx 0.07\%$. L'optimiseur détermine donc qu'il est plus intéressant d'utiliser l'index `IS_CLIENTS_PAYS`. Nous remarquons que cela réduit considérablement les Consistent Gets en les divisant par 20.

Figure 5.3a

Sans histogramme.

0,125 secondes

OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		171	
TABLE ACCESS FULL	CLIENTS	171	692
Prédicats de filtre			
PAYS='Cameroun'			

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	8
db block gets	0
consistent gets	628
physical reads	0
redo size	0
bytes sent via SQL*Net to client	3454
bytes received via SQL*Net from client	603
SQL*Net roundtrips to/from client	2
sorts (memory)	1
sorts (disk)	0

Figure 5.3b

Avec histogramme.

0,109 secondes

OPERATION	OBJECT_NAME	COST	CARDINALITY	LAST_OUTPUT_ROWS
SELECT STATEMENT		6		
TABLE ACCESS BY INDEX ROWID	CLIENTS	6	17	30
INDEX RANGE SCAN	IS_CLIENTS_PAYS	1	17	30
Prédicats d'accès				
PAYS='Cameroun'				

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	8
db block gets	0
consistent gets	33
physical reads	0
redo size	0
bytes sent via SQL*Net to client	3454
bytes received via SQL*Net from client	603
SQL*Net roundtrips to/from client	2
sorts (memory)	1
sorts (disk)	0

Si nous exécutons la même requête avec `pays='USA'`, alors l'histogramme retourne une estimation de 40 % des enregistrements. En toute logique, l'optimiseur décide d'effectuer un Full Table Scan puisque l'utilisation de l'index sur un tel pourcentage de la table serait pénalisant. On constate bien que le plan d'exécution dépend de la nature des conditions mais aussi des données des conditions et de la table.

ASTUCE

Le SGBDR fabrique automatiquement l'histogramme sur la colonne Pays. Afin d'empêcher sa création pour les besoins de notre démonstration, nous exécutons :

```
execute dbms_stats.gather_table_stats (user,'clients',cascade  
=> true, method_opt => 'for columns pays size 1');
```

La création d'un histogramme sur un seul intervalle a pour effet d'empêcher la création de ce dernier.

Cependant, c'est parfois le cas inverse qui se produit. Ainsi, si Oracle ne crée pas d'histogramme sur une colonne alors que vous pensez que cela serait pertinent, vous pouvez forcer l'opération à l'aide de l'instruction suivante. Il est cependant probable que l'histogramme ne sera pas conservé ; lors du prochain calcul des statistiques, il faudra donc intégrer cette instruction à un job périodique :

```
execute dbms_stats.gather_table_stats (user,'clients',cascade  
=> true, method_opt => 'for columns pays size 254');
```

Compression d'index

La compression permet d'éviter la répétition des valeurs des premières colonnes de la clé dans un index composite et, ainsi, de réduire le nombre de blocs à manipuler. Cela améliore l'efficacité de l'index. La compression n'étant possible que sur les *n-1* premières colonnes d'un index, cette fonction est réservée aux index composites. Son objectif est de faire gagner de l'espace dans les feuilles afin d'en réduire le nombre. Compresser des colonnes avec peu de doublons pourra entraîner une perte d'efficacité.

Exemple d'un index compressé :

```
Create index IS_CLIENTS_PAYS_VILLE on Clients(Pays,ville) compress 1;
```

Cet index compressera les pays qui ne seront répétés qu'une seule fois par feuille.

MS SQL Server

SQL Server 2005 ne propose pas de mécanisme de ce genre. SQL Server 2008 introduit une compression analogue à celle des tables que nous étudierons à la section 5.3.1 de ce chapitre.

MySQL

Le moteur MyISAM offre la possibilité de compresser les index à l'aide de l'option `PACK_KEYS` configurée au niveau de la table.

Par défaut, si `PACK_KEYS` n'est pas mentionnée, les n premiers caractères communs des chaînes de caractères sont compressés. `PACK_KEYS=1` force en plus la compression des valeurs numériques. `PACK_KEYS=0` désactive cette fonction.

Le moteur InnoDB ne propose pas de compression d'index suivant le même principe, mais plutôt une compression des pages d'index et de table que nous étudierons à la section 5.3.1 de ce chapitre.

Comportement des index avec les valeurs nulles

Les SGBDR ont régulièrement des comportements déroutants dès qu'il s'agit de manipuler les valeurs nulles, et l'utilisation des index n'échappe pas à la règle.

Le SGBDR Oracle n'indexe pas les lignes contenant des `NULL` sur les premières colonnes pour lesquelles toutes les colonnes présentes dans l'index sont nullable (c'est-à-dire qui ne sont pas déclarées comme `NOT NULL`). Cela signifie qu'un index monocolonne ne gérera jamais les valeurs nulles, il ne sera donc jamais utilisé sur un prédicat `IS NULL`.

Si un index est composé de plusieurs colonnes et qu'au moins une de ces colonnes soit désignée `NOT NULL`, le prédicat `IS NULL` pourra l'utiliser. C'est un cas assez restrictif, donc retenez plutôt que, en général, les valeurs nulles ne fonctionnent pas avec les index.

La valeur nulle est souvent utilisée comme valeur spéciale. Par exemple, dans notre table des commandes, nous aurions pu avoir une colonne `DateLivraison` et rechercher les commandes non livrées avec le prédicat `DateLivraison IS NULL`. Nous aurions pu penser qu'en mettant un index sur cette colonne, la recherche aurait été performante, mais nous nous serions trompés. Dans un cas comme celui-ci, deux solutions s'offrent à nous : la première est d'utiliser un champ `EtatCommande`, la seconde consiste à ajouter un champ `NOT NULL` dans l'index `B*Tree`.

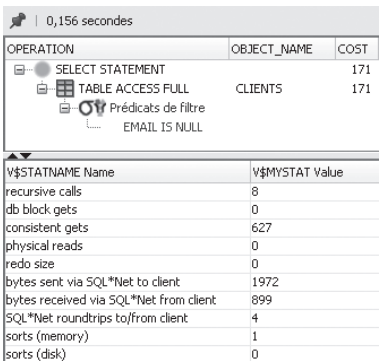
Il faudra donc, si c'est possible, éviter d'utiliser la valeur nulle comme valeur spéciale ou alors créer un index composite qui englobe en plus de cette colonne une colonne qui n'est pas nullable.

Afin de vérifier ce que nous venons d'expliquer, regardons ce qu'il se passe si nous créons un index sur la colonne Email et que nous recherchions les clients n'ayant pas d'adresse e-mail :

```
Create index IS_CLIENTS_EMAIL on Clients(email);
select * from clients where email is null;
```

Figure 5.4a

Plan d'exécution
d'une requête IS NULL
n'utilisant pas l'index.

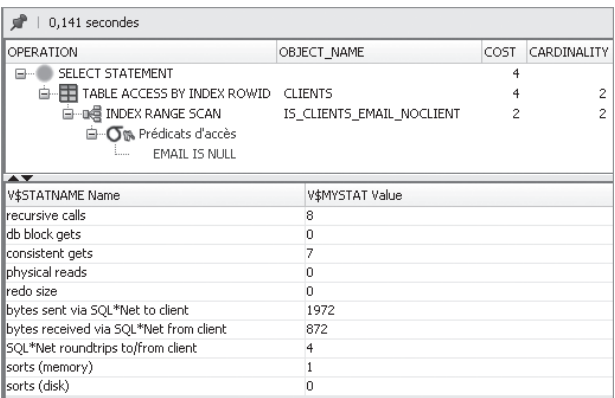


Nous constatons que l'index IS_CLIENTS_EMAIL n'est pas utilisé. Par contre, si nous créons un index composite sur le champ Email et sur le champ NoClient – qui est la clé primaire et qui n'est donc pas nullable –, l'index est bien utilisé avec le prédicat Email Is Null.

```
Create index IS_CLIENTS_EMAIL_NOCLIENT on Clients(email,NoClient);
select * from clients where email is null;
```

Figure 5.4b

Plan d'exécution
d'une requête IS NULL
utilisant un index.



MS SQL Server et MySQL

Aussi bien sous SQL Server que sous MySQL, le comportement des valeurs nulles lors de l'utilisation des index n'a rien de particulier. La valeur nulle est considérée comme une valeur comme une autre et les index sont bien utilisés sur les prédicats `IS NULL`.

Colonnes incluses

Nous avons vu que si l'optimiseur trouve toutes les données dans l'index, il peut décider de ne pas accéder à la table. Ce comportement peut être exploité pour améliorer les performances en incluant dans des index les données que vous souhaitez récupérer, on parle alors d'index "couvrant". Par exemple, si vous souhaitez souvent récupérer la liste des noms des clients d'un pays, vous pouvez créer un index sur ces colonnes, en mettant impérativement en premier celles qui servent de critère et en second celles qui sont les colonnes incluses.

```
create index IS_CLIENTS_PAYSNOM on CLIENTS (PAYS, NOM);
select nom from clients t where pays='Cameroun';
```

On constate dans le plan d'exécution de la Figure 5.4c que seul l'index est utilisé. Cela économise l'accès à la table qu'aurait nécessité un index sur la colonne Pays seule.

Figure 5.4c

*Plan d'exécution
d'une requête utilisant
un index incluant toutes
les colonnes.*

0,141 secondes		
OPERATION	OBJECT_NAME	COST
SELECT STATEMENT		2
INDEX RANGE SCAN	IS_CLIENTS_PAYSNOM	2
Prédicats d'accès		
PAYS='Cameroun'		
V\$STATNAME Name	V\$MYSTAT Value	
recursive calls	8	
db block gets	0	
consistent gets	5	
physical reads	0	
redo size	0	
bytes sent via SQL*Net to client	1679	
bytes received via SQL*Net from client	881	
SQL*Net roundtrips to/from client	4	
sorts (memory)	1	
sorts (disk)	0	

MS SQL Server

Sous SQL Server, cette approche est intégrée dans la syntaxe à l'aide du mot clé `INCLUDE` qui permet d'ajouter des champs à l'index sans qu'ils soient inclus dans l'arbre ni comme clé de tri mais seulement dans les feuilles.

```
create index IS_CLIENTS_PAYSNOM on CLIENTS (PAYS) INCLUDE(nom);
```

On constate la nuance par le fait que, sous Oracle, le résultat apparaît trié par nom puisqu'il s'agit d'un parcours de l'index qui est trié sur Pays et Nom alors qu'avec la syntaxe `INCLUDE` le résultat n'est pas trié par Nom bien que, là aussi, seul l'index soit parcouru. La différence est que la clé de tri de l'index sous SQL Server est seulement le champ Pays, le champ Nom n'est pas stocké dans un ordre particulier.

Le facteur de foisonnement

Le facteur de foisonnement (*Clustering Factor*) établit le nombre de liens qu'il y a entre l'ensemble des feuilles de l'index et les blocs dans le tas. Il permet d'estimer si le fait que plusieurs valeurs se trouvent dans une même feuille d'index nécessitera plutôt peu de lecture de blocs différents dans le tas ou plutôt beaucoup.

Les données relatives au facteur de foisonnement sont disponibles dans la métabase. On y accède par la requête suivante.

```
select t.INDEX_NAME,t.LEAF_BLOCKS,t.DISTINCT_KEYS,t.CLUSTERING_FACTOR
from sys.user_ind_statistics t where index_name like '%CLIENT%';
```

INDEX_NAME	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTOR
PK_CLIENTS	94	45000	759
IS_CLIENTS_VILLE	130	1096	43479

`Clustering_Factor/Leaf_Blocks` donne le nombre de liens qu'il y a en moyenne entre une feuille et des blocs de données dans le tas : un ratio inférieur ou égal à 3 est excellent.

Par contre, lorsque `Clustering_Factor` tend vers le nombre d'enregistrements, alors le `Clustering Factor` est considéré comme mauvais.

Prenons deux cas pour illustrer le facteur de foisonnement :

- **Premier cas.** L'index de clé primaire sur la colonne `Nocmd` de la table des commandes `CMD`. Si on suppose que les numéros de commandes sont créés séquentiellement, de façon ordonnée et jamais effacés, le tas sera globalement dans le même ordre que l'index. Si l'index retourne plusieurs `RowID` depuis un même bloc (ce qui est caractéristique d'un prédicat sur une partie de clé ou sur un inter-

valle), alors il est probable que ces RowID désignent le même bloc ou peu de blocs différents dans le tas (puisque, généralement, il y aura plus de blocs dans le tas que de blocs d'index). Dans ce cas, le facteur de foisonnement sera bon.

- **Second cas.** Un index sur la colonne Ville dans la table des clients. Si on suppose que les clients n'agissent pas de façon concertée ou dirigée, alors ils seront mis dans le tas dans l'ordre suivant lequel ils ont passé leur première commande mais ils seront normalement dans des villes différentes. Donc, le fait de rechercher tous les clients situés à "Toulouse" retournera, depuis une ou plusieurs pages consécutives de l'index, des RowID qui désigneront des blocs qui seront probablement sans aucun point commun. Dans ce cas, le facteur de foisonnement sera mauvais.

Un bon facteur de foisonnement permettra d'avoir de meilleures performances lors d'une opération Index Range Scan puisque, pour chaque feuille d'index, il y aura peu de blocs de données à charger depuis le tas. Pour les autres types d'accès, l'impact sera généralement faible. Un mauvais facteur de foisonnement aura pour effet de considérer un Index Range Scan moins intéressant (voir l'Annexe B, section B.4, pour un exemple d'impact). Il n'est pas trop possible d'influer sur le facteur de foisonnement.

5.2.2 Index sur fonction

Comme nous l'avons vu précédemment, l'exécution de fonctions sur des colonnes indexées a pour conséquence que les index ne sont pas utilisés sur ces colonnes, ce qui peut être pénalisant.

Par exemple, la requête suivante ignore l'index qui est présent sur la colonne Nom.

```
create index IS_CLIENTS_NOM on CLIENTS(NOM);  
Select * from clients Where Upper(Nom)='NEVILLE';
```

Pour pallier ce problème, il est possible de créer des index sur fonction. Il s'agit d'index B*Tree classiques sauf que leur clé n'est pas la valeur de la colonne indexée mais le résultat de la fonction sur la valeur de la clé. Dans notre exemple, l'index contiendra donc Upper(Nom) et non pas Nom.

```
create index IS_CLIENTS_UPPER_NOM on CLIENTS(Upper(NOM));  
Select * from clients Where Upper(Nom)='NEVILLE';
```

Malgré leur nom, les index sur fonction s'appliquent aussi à des expressions. Ainsi, il est possible de définir un index sur une expression quelconque, à condition qu'elle soit déterministe et répétable (il est interdit d'utiliser SYSDATE ou DBMS_RANDOM).

L'exemple ci-après crée un index sur une expression qui est utilisée par la requête suivante malgré le fait que les opérandes ont été permutés.

```
create index IS_CLIENTS_NOCLIENT10 on CLIENTS(NoClient+10);  
Select * from clients Where 10+NoClient=14813
```

Oracle 11g introduit la notion de colonne virtuelle. Ce sont des colonnes calculées dont le résultat n'est pas stocké. Elles peuvent être indexées et auront alors le même fonctionnement qu'un index sur fonction. Dans l'exemple suivant, nous créons une colonne virtuelle et nous l'indexons : qu'on l'interroge ou qu'on fasse une requête sur sa définition, dans les deux cas l'index est utilisé.

```
alter table clients add upper_nom varchar2(50) as (upper(nom));  
create index is_clients_upper_nom on clients(upper_nom);  
select * from clients t where upper(nom)='GORE';  
select * from clients t where upper_nom = 'GORE';
```

MySQL

MySQL ne propose ni la notion d'index sur fonction, ni la notion de colonne virtuelle.

MS SQL Server

SQL Server propose à la fois la notion d'index sur fonction et la notion de colonne virtuelle. Il est possible, pour cette dernière, de la définir "persistante", c'est-à-dire stockée dans la table et non pas calculée à la volée.

5.2.3 Reverse Index

Ce sont des index B*Tree qui indexent les valeurs miroirs (par exemple, le nom "Jones" devient "senoJ"). La syntaxe est identique, il faut juste spécifier le mot clé REVERSE à la fin de l'instruction :

```
Create index <Nom_Index> on <NomTable>(<col1>[,<col2>,...]) REVERSE
```

Le but de ce type d'index est d'éviter que deux valeurs qui se suivent dans leur ordre naturel se retrouvent dans la même feuille. L'objectif est ici de s'organiser pour avoir un mauvais facteur de foisonnement. Dans quel intérêt ?

Imaginez une table dont la clé est un numéro séquentiel et qu'il y ait de nombreuses insertions en parallèle. On se retrouve alors avec plusieurs requêtes qui vont vouloir mettre à jour la même feuille d'index. Or, cette opération ne peut pas être faite par deux requêtes simultanément : cela aurait pour effet de bloquer chacune d'elles le temps que la précédente ait terminé la mise à jour de la feuille d'index. Avec un

index reverse, elles vont mettre à jour des feuilles d'index différentes et ainsi elles ne se gêneront pas. Cela permet donc de réduire la contention d'accès en écriture des feuilles d'index, ce qui sera intéressant sur des requêtes parallèles, particulièrement en environnement Oracle RAC. Cependant, ce type d'index montre des limites car il n'est utilisable que sur des conditions de type égalité (=), ce qui signifie qu'il ne fonctionnera pas pour des opérations Index Range Scan.

De plus, contrairement à une rumeur répandue, ce type d'index ne fonctionne **PAS** sur les requêtes de type `Nom like '%nes'`, c'est-à-dire avec un masque de LIKE commençant par % et ayant une fin fixe.

5.2.4 Index bitmap

Les index bitmap sont adaptés à la manipulation de colonnes avec peu de valeurs distinctes. Ils ne sont utilisables que pour des prédicats d'égalité, mais Oracle arrive, dans certains cas, à convertir des intervalles par un ensemble d'égalité, rendant ainsi ces index opérants sur des intervalles et même sur des jointures.

Les index bitmap sont des masques de bits pour les valeurs distinctes des colonnes indexées : des ET et OU binaires permettent de faire des tests d'égalité. Nous l'illustrons ci-après avec la table EMP du schéma SCOTT des bases exemples d'Oracle. On voit que la colonne Job peut prendre cinq valeurs possibles (voir Figure 5.5a). L'index bitmap code ces valeurs par cinq masques de bits (voir Figure 5.5b). Chaque bit spécifie par un 1 que la ligne contient la valeur ou par un 0 qu'elle ne la contient pas. Si nous cherchons à sélectionner les personnes ayant un job de 'PRESIDENT' ou de 'COMPTABLE', il suffit de faire un OU binaire entre les deux masques pour connaître les lignes qui ont une de ces valeurs (voir Figure 5.5b). L'intérêt le plus significatif des index bitmap est qu'on peut en combiner plusieurs dans un même accès à une table. Pour cela, il suffira d'appliquer des opérations logiques sur les différents masques de chaque index comme s'il s'agissait du même index bitmap. Les index B*Tree ne sont pas conçus pour être combinés ; ils ne peuvent donc pas l'être efficacement. En revanche, s'il y a plusieurs index bitmap couvrant les prédicats d'une requête, ils seront combinés.

En fait, les index bitmap ne sont pas stockés comme c'est présenté à la Figure 5.5b car ils sont compressés. La technique de compression utilisée est sensible au foisonnement. Ainsi, si les lignes qui se suivent ont souvent la même valeur, la compression sera bien meilleure que si chaque ligne a une valeur différente de la précédente. Le taux de compression aura un impact sur l'espace requis pour stocker l'index bitmap et donc sur le nombre de Consistent Gets. Il faut avoir en tête que, si les

valeurs distinctes sont distribuées sur l'ensemble de la table plutôt que regroupées par paquets, l'index occupe plus d'espace.

Figure 5.5a
Représentation logique
d'un index bitmap.

N° ligne	EMPNO	ENAME	JOB	COMPTABLE	VENDEUR	PRESIDENT	MANAGER	INGENIEUR
1	14347	DUPOND	COMPTABLE	1	0	0	0	0
2	14484	MICHEL	VENDEUR	0	1	0	0	0
3	14621	LEROY	VENDEUR	0	1	0	0	0
4	14758	DURAND	MANAGER	0	0	0	1	0
5	14895	MEUNIER	VENDEUR	0	1	0	0	0
6	15032	MARTIN	MANAGER	0	0	0	1	0
7	15169	DUPONT	MANAGER	0	0	0	1	0
8	15306	LECLERC	INGENIEUR	0	0	0	0	1
9	15443	NEUVILLE	PRESIDENT	0	0	1	0	0
10	15580	DUJARDIN	VENDEUR	0	1	0	0	0
11	15717	BERTHIER	COMPTABLE	1	0	0	0	0
12	15854	GUY	COMPTABLE	1	0	0	0	0
13	15991	RENAUD	INGENIEUR	0	0	0	0	1
14	16128	LAVAU	COMPTABLE	1	0	0	0	0

Figure 5.5b
Représentation interne
d'un index bitmap.

N° ligne	1	2	3	4	5	6	7	8	9	10	11	12	13	14
COMPTABLE	1	0	0	0	0	0	0	0	0	0	1	1	0	1
VENDEUR	0	1	1	0	1	0	0	0	0	1	0	0	0	0
PRESIDENT	0	0	0	0	0	0	0	0	1	0	0	0	0	0
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
INGENIEUR	0	0	0	0	0	0	0	1	0	0	0	0	1	0

COMPTABLE OU PRESIDENT	1	0	0	0	0	0	0	0	1	0	1	1	0	1
------------------------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Index bitmap et opérations LMD

Il n'est pas recommandé d'utiliser les index bitmap sur des tables très mouvementées par des sessions concurrentes. Cela peut poser des problèmes de verrouillage à cause de la structure de stockage interne des RowID, laquelle permet d'économiser de l'espace. Ce sont des index plutôt orientés datawarehouse mais qui restent cependant tout à fait utilisables dans des environnements OLTP modérés.

Index bitmap et valeurs nulles

Les index bitmap gèrent mieux les valeurs nulles que les index B*Tree puisque les valeurs NULL sont considérées comme des valeurs distinctes au même titre que les autres.

Mise en œuvre

La syntaxe de création d'un index bitmap est très proche de celle d'un index B*Tree, il faut juste ajouter le mot clé bitmap. Ici aussi, il est possible de créer des index sur fonction avec les index bitmap.

```
create bitmap index <Nom_Index> on <NomTable>(<col1>[,<col2>,...]);
```

Les index bitmap présentent l'intérêt d'être adaptés à l'indexation de colonnes ayant peu de valeurs distinctes. Cependant, un index bitmap employé sur une seule colonne de faible sélectivité ne sera pas beaucoup plus efficace qu'un index B*Tree. L'avantage majeur des index bitmap apparaît lors de la combinaison de conditions sur des colonnes ayant chacune une faible sélectivité mais qui, ensemble, auront une forte sélectivité. Sinon, on retrouve le problème des index B*Tree où le coût des allers-retours entre l'index et le tas dépasse le gain apporté par l'index.

Les index bitmap sont réputés pour être adaptés à des colonnes ayant relativement peu de valeurs distinctes. Cependant, les résultats des tests ci-après montrent que, même avec 20 millions de valeurs distinctes, ils ont de bonnes performances. Néanmoins, ils occupent, dans ce cas, plus d'espace que des index B*Tree, alors que c'est généralement le contraire. On constate que, s'il y a peu de valeurs distinctes, ils ont des tailles très faibles par rapport aux index B*Tree, ce qui rendra leur manipulation d'autant plus efficace. Par contre, lorsque le nombre de valeurs distinctes augmente (à partir de un million de valeurs), on s'aperçoit que la taille de l'index bitmap explose. La taille d'un index B*Tree est constante mais plus importante que celle des index bitmap dans de nombreux cas puisque chacune de ses entrées contient la clé plus le RowID, contrairement aux index bitmap qui ont une technique de stockage optimisée.

Ci-après, on lit quelques résultats de tests de performance montrant le bon comportement des index bitmap sur une table de 20 millions d'enregistrements avec des colonnes indexées ayant de 2 à 20 millions de valeurs distinctes. On y trouve le temps nécessaire pour effectuer le calcul d'une moyenne sur une colonne de la table en fonction d'un critère sur une colonne indexée ayant plus ou moins de valeurs distinctes. Le test est fait avec un index bitmap, un index B*Tree et sans aucun index. Ces tests ont été réalisés avec des données ayant un très bon facteur de foisonnement, ce qui améliore un peu les choses, spécialement sur les tests ayant peu de valeurs distinctes.

Figure 5.6a

Évolution de la taille des index bitmap et B*Tree en fonction du nombre de valeurs distinctes (échelle logarithmique).

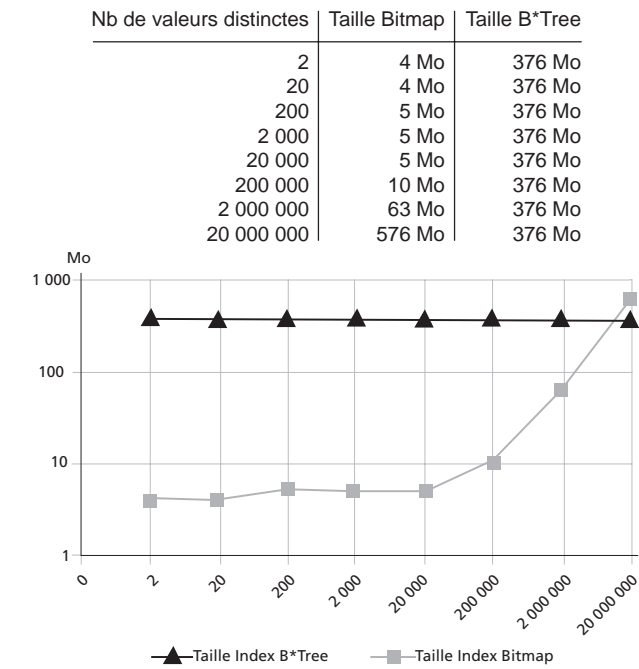


Figure 5.6b

Temps de réponse en fonction du nombre de valeurs distinctes.

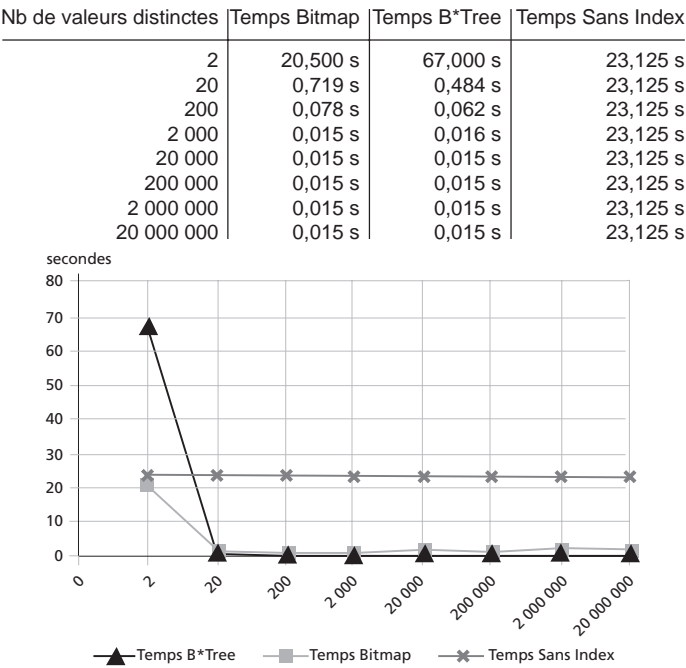
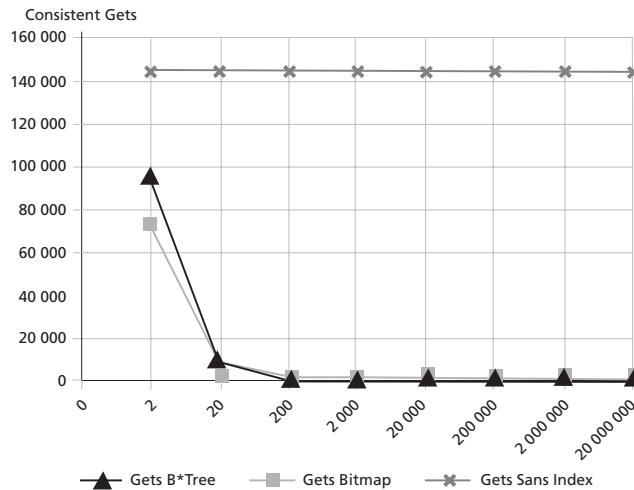


Figure 5.6c

Nombre de
Consistent Gets
en fonction du
nombre de valeurs
distinctes.

Nb de valeurs distinctes	Consistent Gets Bitmap	Consistent Gets B*Tree	Consistent Gets Sans Index
2	73 285	96 650	144 123
20	6 918	9 266	144 123
200	656	890	144 123
2 000	63	90	144 123
20 000	9	90	144 123
200 000	4	83	144 123
2 000 000	4	83	144 123
20 000 000	3	3	144 123



Évaluation

Évaluons les différentes solutions pour filtrer plusieurs colonnes de faible cardinalité, sur une table de 2 606 477 enregistrements avec des conditions sur la colonne Nolive ayant 2 972 valeurs distinctes et la colonne Remise ayant 3 valeurs distinctes.

Nous allons comparer les différentes solutions avec la requête suivante :

```
select * from cmd_lignes where nolivre=6262 and remise=7
```

Sans aucun index, il faut parcourir toute la table d'où de nombreux Consistent Gets (voir Figure 5.7).

Avec 2 index B*Tree, le SGBDR convertit les résultats des Range Scan en index bitmap afin de les combiner (voir Figure 5.8).

Figure 5.7
Sans aucun index.

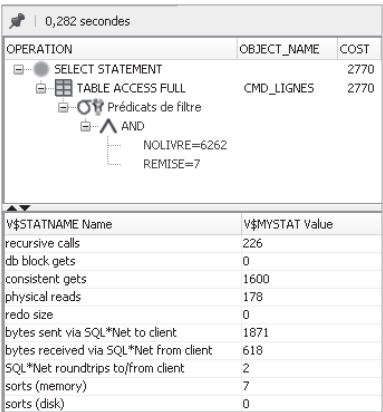
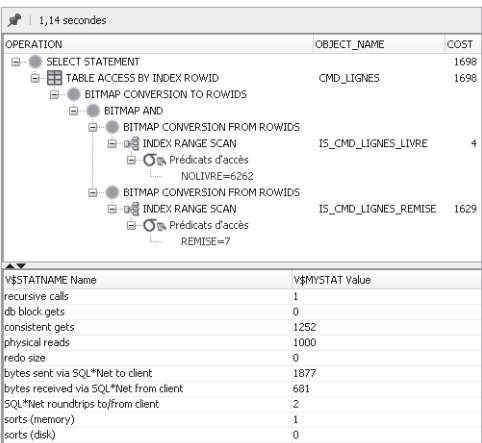


Figure 5.8
Avec deux index B*Tree.



On voit donc que, dans certains cas, plusieurs B*Tree peuvent être utilisés lors d'un même accès à une table, mais le coût lié à la conversion en index bitmap est élevé.

En fait, dans cet exemple en particulier, l'optimiseur avait choisi de n'utiliser que l'index sur Nolivres. Nous avons forcé ce comportement afin de vous le présenter.

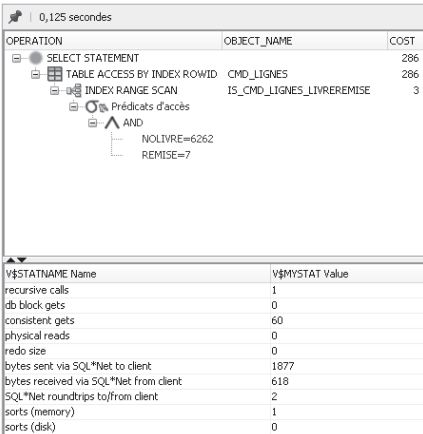
```
create index IS_cmd_lignes_Livre on cmd_lignes(nolivres);  
create index IS_cmd_lignes_Remise on cmd_lignes(Remise);
```

Si les index B*Tree ne sont pas conçus pour être combinés, les index composites gèrent parfaitement les conditions sur deux colonnes. Nous obtenons ainsi un bon résultat (voir Figure 5.9).

```
create index IS_CMD_LIGNES_LIVREREMISE on CMD_LIGNES (nolivres, remise)
```

L'utilisation des index composites est très efficaces lorsque les conditions des requêtes s'appliquent exactement aux colonnes de l'index. Si les conditions peuvent varier sur un ensemble de colonnes (par exemple une requête avec des conditions sur Col1, Col2 et Col5, une deuxième requête sur Col2, Col3 et Col4, une troisième requête sur Col1, Col3 et Col6), il faudrait créer autant d'index composites que de combinaisons possibles entre les colonnes – ce qui risquerait d'occuper pas mal d'espace de stockage et d'impacter les mises à jour de la table.

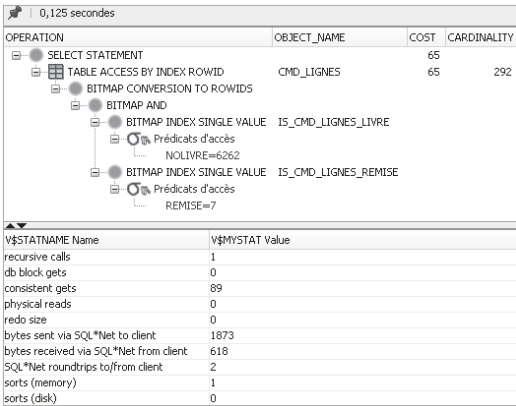
Figure 5.9
Avec index B*Tree composite.



Avec les deux index bitmap suivants:

```
create bitmap index IS_cmd_lignes_Livre on cmd_lignes(nolivre);  
create bitmap index IS_cmd_lignes_Remise on cmd_lignes(Remise);
```

Figure 5.10
Avec deux index bitmap.



On voit que les deux index bitmap sont naturellement combinés (voir Figure 5.10). Cette solution est légèrement moins bonne que l’index B*Tree composite de cet exemple mais, suivant les cas, la tendance peut s’inverser. Ces deux solutions ont souvent des performances comparables.

La solution avec les index bitmap allie performance, faible encombrement et flexibilité des conditions.

Au Tableau 5.1, nous voyons que les index bitmap sont plus petits.

Tableau 5.1 : Taille des différents index

Type index	Taille index NoLivre	Taille index Remise
Index bitmap	10 Mo	0.88 Mo
Index B*Tree	44 Mo	39 Mo
Index B*Tree composite	51 Mo	

5.2.5 Bitmap Join Index

Les Bitmap Join Index permettent de créer des index bitmap basés sur les valeurs d’une autre table avec laquelle on prévoit de faire une jointure. Dans l’exemple ci-après, nous créons un index dans la table CMD_LIGNES portant sur la colonne Collection de la table LIVRES. Ainsi, si nous effectuons une requête sur la table CMD_LIGNES avec comme condition la colonne Collection, le SGBDR n’aura pas besoin de faire la jointure avec la table LIVRES, ce qui apportera un gain significatif. Il faudra cependant continuer à écrire cette jointure dans la requête :

```
create bitmap index is_cmd_lignes_collect on cmd_lignes(l.collection)
from cmd_lignes cl,livres l where cl.nolivre=l.nolivre;
```

Voici un exemple de requête tirant parti du Bitmap Join Index :

```
select sum(quantite) from cmd_lignes cl,livres l
where cl.nolivre=l.nolivre and l.collection='Best-sellers';
```

Ce type d’index est particulièrement intéressant si vous avez besoin de faire des jointures uniquement pour appliquer des conditions par égalité. Dans ce cas, il n’y aura plus besoin de faire la jointure, ce qui apportera un gain significatif. Par contre, si la jointure reste nécessaire pour appliquer des conditions ou afficher les autres champs, l’index sera moins intéressant mais il pourra cependant toujours présenter un intérêt, car il réduira la taille de l’ensemble à joindre.

Il faut bien garder à l'esprit qu'un Bitmap Join Index est avant tout un index bitmap, c'est-à-dire qu'il est conçu pour les tests d'égalité, qu'il est surtout recommandé pour un faible nombre de valeurs distinctes et qu'il n'est pas adapté aux tables soumises à beaucoup de LMD concurrents.

Mise en œuvre

La syntaxe d'un Bitmap Join Index est la suivante :

```
create bitmap index <Nom_Index> on <NomTable>(<col1>[,<col2>,...])
from <NomTable>,<NomTableJointe>
Where <JointureEntreLesTables> ;
```

Évaluation

Nous allons évaluer les performances de cet objet avec la requête suivante :

```
select sum(quantite) from cmd_lignes cl,livres l
where cl.nolivres=l.nolivres and cl.remise=7 and l.collection='Programmeur'
```

Dans cette première trace d'exécution, il y a un index bitmap sur la colonne Remise mais il n'est pas utilisé car une opération table access full est requise pour la jointure.

Figure 5.11

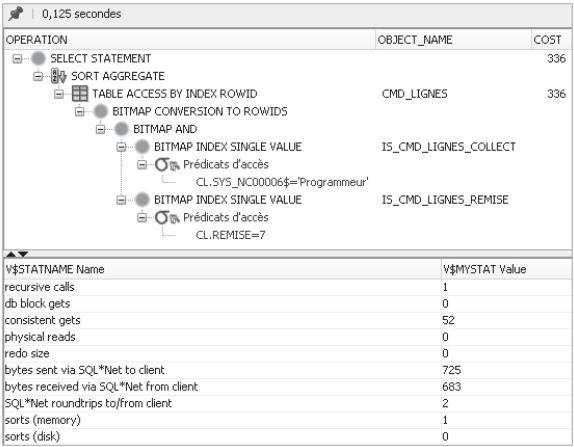
*Trace d'exécution sans
Bitmap Join Index.*

1,063 secondes			
OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		2787	
SORT AGGREGATE			1
HASH JOIN		2787	598
Prédicats d'accès			
CL.NOLIVRE=L.NOLIVRE			
TABLE ACCESS FULL	LIVRES	13	2
Prédicats de filtre			
L.COLLECTION='Programmeur'			
TABLE ACCESS FULL	CMD_LIGNES	2770	868826
Prédicats de filtre			
CL.REMISE=7			
▼			
V\$STATNAME Name	V\$MYSTAT Value		
recursive calls	1		
db block gets	0		
consistent gets	9411		
physical reads	0		
redo size	0		
bytes sent via SQL*Net to client	728		
bytes received via SQL*Net from client	683		
SQL*Net roundtrips to/from client	2		
sorts (memory)	1		
sorts (disk)	0		

Dans la deuxième trace d'exécution (voir Figure 5.12), nous avons placé un Bitmap Join Index sur le champ Collection (voir exemple précédent). Nous voyons que l'optimiseur n'estime plus nécessaire de faire une jointure avec la table LIVRE pour répondre à cette requête (alors que celle-ci est présente dans la requête). En conséquence, les deux index bitmap sont combinés ce qui est particulièrement efficace. Les Consistent Gets sont divisés par 180 et le temps d'exécution par 10.

Figure 5.12

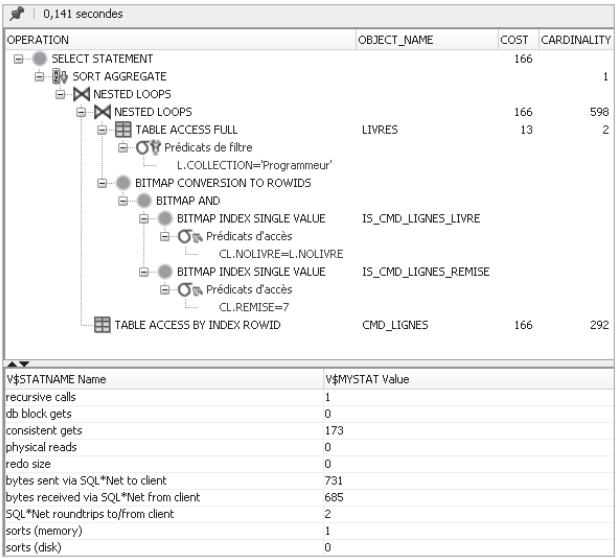
Trace d'exécution avec un Bitmap Join Index.



Dans la troisième trace d'exécution (voir Figure 5.13), nous avons placé des index bitmap sur Remise et Nolivre mais supprimé le Bitmap Join Index. Nous constatons une jointure par boucle imbriquée sur l'index bitmap de la colonne Nolivre qui est combinée au filtrage bitmap sur la colonne remise. Les performances sont moindres qu'avec le Bitmap Join Index mais cependant acceptables. Si l'index sur la colonne Nolivre avait été de type B*Tree, le plan aurait été quasiment le même car le résultat du Range Scan de l'index B*Tree aurait été converti en bitmap.

Figure 5.13

Trace d'exécution sans Bitmap Join Index mais avec des index bitmap sur NoLivre et Remise.



MS SQL Server et MySQL

Ni SQL Server, ni MySQL ne proposent d'index bitmap ou de Bitmap Join Index.

5.2.6 Full Text Index

En SQL de base, pour chercher un texte dans une chaîne de caractères, on écrit quelque chose comme ceci :

```
select count(*) from texteslivres where texte like '%voyage%';
```

C'est simple à comprendre mais ce n'est pas très efficace sur des textes un peu importants. De plus, si le texte est dans un champ de type CLOB, les performances s'écroulent à cause des accès au segment LOB qui est séparé du tas. Le principal problème de cette requête est qu'elle ne peut utiliser aucun index B*Tree car la présence de `like '%...'` est un cas disqualifiant pour l'usage des index.

Nous allons regarder comment optimiser ce type de requête, mais nous avons besoin d'abord d'une base de données avec un peu de texte. Pour cela, nous nous rendons sur le site <http://www.livrespourtous.com> et nous récupérons quelques livres de Victor Hugo et d'Émile Zola, nous découpons approximativement les pages et nous les chargeons dans la table ci-après. Afin d'augmenter un peu le volume, nous dupli- quons ces données en cinq exemplaires, ce qui nous permet d'avoir une table de 17 325 enregistrements pour une taille de 40 Mo. Cette base est disponible sur le site de l'auteur, à l'adresse <http://www.altidev.com/livres.php>.

```
create table TextesLivres(
  ID          NUMBER,
  Auteur      varchar2(100),
  Titre       varchar2(100),
  SousTitre   varchar2(100),
  TEXTE       CLOB,
  constraint PK_TextesLivres primary key (ID) );
insert into texteslivres select id+10000, auteur, titre, soustitre||' Bis',
texte from texteslivres where id <5000 order by 1;
insert into texteslivres select id+20000, auteur, titre, soustitre||' Ter',
texte from texteslivres where id <5000 order by 1;
insert into texteslivres select id+30000, auteur, titre, soustitre||'
Quater',    texte from texteslivres where id <5000 order by 1;
insert into texteslivres select id+40000, auteur, titre, soustitre||'
quinquies', texte from texteslivres where id <5000 order by 1;
```

La requête suivante a un temps d'exécution de 2,75 secondes et nécessite 5 144 Consistent Gets.

```
select count(*) from texteslivres where texte like '%voyage%';
```

La variante de cette requête utilisant la fonction UPPER a un temps d'exécution de 2,92 secondes et nécessite 22 999 Consistent Gets.

```
select count(*) from texteslivres where upper(texte) like '%VOYAGE%'
```

Si les textes sont supérieurs à $\approx 4\,000$ caractères, ils sont stockés dans un segment séparé (voir la section 5.3.1 pour plus de détails), ce qui va encore plus pénaliser ce genre de requête. Dans notre cas, seulement 0,4 % des enregistrements dépassent cette limite. Mais si nous modifions les paramètres de la table avec la clause `DISABLE STORAGE IN ROW`, qui permet de forcer le stockage systématique dans un segment séparé, alors notre requête a un temps d'exécution de 17 secondes et nécessite 111 632 Consistent Gets.

Maintenant que nous avons mis en évidence que l'opérateur `LIKE` n'est pas une solution très performante pour effectuer des recherches dans des champs texte, voyons les solutions possibles. Oracle met à notre disposition des outils qui sont adaptés à ces problématiques au moyen d'index spécifiques de domaine. Je vous conseille de consulter le document *Oracle Text Application Developer's Guide* dans la documentation Oracle pour avoir des informations détaillées. Nous étudierons ici les éléments clés qui permettent d'optimiser les requêtes de type Full Text Search de plus en plus prisées avec l'avènement du Web et de Google.

Oracle Text met à disposition des index adaptés aux recherches de textes. Ils sont capables de travailler sur des champs texte mais aussi sur des champs de type BLOB contenant des fichiers bureautiques. La phase d'indexation utilise un filtre qui convertit les documents (word, pdf, etc.) en texte et un analyseur lexical (*lexer*) qui fait une analyse lexicale du texte afin d'en extraire les mots (ainsi, les recherches se font au niveau des mots et non pas sur des morceaux de chaînes contrairement aux recherches à l'aide du prédicat `like '%chaîne%'`). L'analyseur lexical peut aussi convertir les caractères en majuscules (par défaut) et les signes diacritiques (caractères accentués, tréma, etc.) en leurs caractères de base afin de faire correspondre les recherches avec ou sans accents (désactivé par défaut).

Plusieurs types d'index sont à notre disposition, chacun orienté vers un usage particulier :

- **CONTEXT.** Adapté aux recherches de mots dans des grands champs texte ou des documents. Ce type d'index extrait tous les mots des documents, les compte et les associe aux enregistrements concernés. Les mots ne sont donc pas répétés, ce

qui permet d'avoir fréquemment un index plus petit que le texte original. Ce type d'index doit, par défaut, être synchronisé manuellement.

- **CTXCAT.** Adapté pour indexer de petits textes en les combinant avec des champs structurés afin de faire des recherches combinées plus efficaces. Il occupe pas mal de place et peut dépasser largement la taille du texte initial. Il n'est donc pas adapté à l'indexation de grands textes. Ce type d'index est synchronisé automatiquement.
- **CTXRULE.** Permet de classer les documents suivant leur contenu à partir de l'opérateur MATCHES.
- **CTXXPAT.** Adapté à l'indexation des documents XML.

Pour des recherches Full Text sur des champs volumineux comme dans notre exemple, il est donc recommandé d'utiliser un index de type CONTEXT.

Mise en œuvre d'un index CONTEXT

La syntaxe de base permettant de créer un index Full Text de type CONTEXT est la suivante :

```
create index <NomIndex> on <NomTable>(<colonnes>) indextype is CTXSYS.CONTEXT;
```

Ci-après, vous voyez un exemple de création d'un index où une conversion des signes diacritiques en caractères de base est spécifiée. La première étape consiste à créer des préférences pour le lexer. Elle est nécessaire une seule fois pour l'ensemble des index partageant les mêmes préférences de lexer. La seconde étape vise la création de l'index lui-même, en spécifiant le type CONTEXT et les éventuels paramètres.

```
begin
  ctx_ddl.create_preference('mylex', 'BASIC_LEXER');
  ctx_ddl.set_attribute ( 'mylex', 'base_letter', 'YES' );
end;
create index IS_FT_TEXTESLIVRES on TEXTESLIVRES(texte)
indextype is CTXSYS.CONTEXT parameters ( 'LEXER mylex' );
```

À la suite de la création de l'index, les tables suivantes sont créées : DR\$IS_FT_TEXTESLIVRES\$I, DR\$IS_FT_TEXTESLIVRES\$K, DR\$IS_FT_TEXTESLIVRES\$N, DR\$IS_FT_TEXTESLIVRES\$R. Dans l'exemple, elles occupent 20 Mo alors que notre table occupe 40 Mo. Si nous regardons de plus près la table DR\$IS_FT_TEXTESLIVRES\$I, nous constatons qu'elle contient chaque mot converti en majuscules et sans signes diacritiques, ainsi que le nombre d'occurrences de chaque mot.

Maintenant que les données sont indexées, nous pouvons faire des recherches à l'aide de l'opérateur CONTAINS (cet opérateur n'est opérationnel que si un index

textuel est présent). Comme nous le voyons dans l'exemple suivant, nous avons à notre disposition la fonction SCORE qui donne un score de pertinence de la réponse.

```
SELECT SCORE(1) score,id FROM texteslivres
WHERE CONTAINS(texte, 'voyage', 1) > 0
ORDER BY SCORE(1) DESC;
```

Cette requête ne retourne plus que 310 enregistrements, alors que la précédente, implémentée avec un prédicat LIKE, en annonçait 680. Cela vient juste du fait que la version avec like '%voyage%' considère aussi les valeurs suivantes : voyageaient, voyageait, voyageant, voyager, voyagera, voyages, voyageur, voyageurs, voyageuse, voyageuses alors que l'opérateur CONTAINS lui travaille sur les mots et fait donc la différence entre les mots voyage et voyager.

Pour chercher plusieurs mots, il ne faut pas utiliser un opérateur AND dans la clause WHERE mais la syntaxe & de l'opérateur CONTAINS comme dans l'exemple ci-après, qui recherche les textes contenant les mots "voyage" et "soleil".

```
SELECT count(*) FROM texteslivres
WHERE CONTAINS(texte, 'voyage & soleil', 1) > 0
```

L'opérateur CONTAINS gère à lui tout seul tout un langage de requête texte. Consultez la documentation de référence Oracle Text pour avoir plus de détails sur les possibilités de cet opérateur. On peut, par exemple, faire des recherches floues, des combinaisons, utiliser des thesaurus, des proximités de mots, etc. La requête ci-après trouvera les textes avec les mots proches de "voyage" tels que "voyageur" et exclura le mot "voyage" avec l'opérateur ~.

```
SELECT * FROM texteslivres
WHERE CONTAINS(texte, 'fuzzy(voyage) ~voyage', 1) > 0
```

Si on compare à la solution avec l'opérateur LIKE précédemment testée, la requête suivante prend 0,172 seconde et nécessite 1 165 Consistent Gets.

```
SELECT count(*) FROM texteslivres WHERE CONTAINS(texte, 'voyage', 1) > 0
```

Certes, c'est beaucoup pour un index mais cela reste en très net progrès puisque nous avons divisé le temps d'exécution par 16 par rapport à la solution initiale et même par 100 par rapport à la version qui utilisait un stockage dans un segment LOB séparé.

Bien que les index de type CONTEXT ne soient pas synchronisés automatiquement avec les données, leur mise à jour est incrémentale. Il faut utiliser la commande suivante pour exécuter une synchronisation de l'index avec les données :

```
EXEC CTX_DDL.SYNC_INDEX('IS_FT_TEXTESLIVRES');
```

Il est cependant possible de spécifier dans le CREATE INDEX des clauses de rafraîchissement automatique lors de la création de l'index :

```
SYNC(MANUAL |EVERY "interval-string" |ON COMMIT)
```

Au fur et à mesure que l'index grandit avec les synchronisations, il se fragmente, on doit donc régulièrement l'optimiser. Durant cette opération, dans sa version rapide (*Fast*), les entrées d'index associées à des enregistrements qui ont été supprimés ou qui ne sont plus dans l'enregistrement ne sont pas automatiquement effacées. Il faut, pour traiter ces cas-là, effectuer une optimisation complète afin de les purger (remplacer *FAST* par *FULL* ci-après).

```
EXEC CTX_DDL.OPTIMIZE_INDEX('IS_FT_TEXTESLIVRES','FAST');
```

Mise en œuvre d'un index CTXCAT

Les index CTXCAT sont adaptés à l'exécution de requêtes combinant une recherche Full Text avec des critères sur les autres colonnes de l'enregistrement. Ce type de requête est possible avec les index CONTEXT : il suffit de mettre les deux prédicats dans la requête. Cependant, cette solution nécessite de parcourir d'abord l'index CONTEXT puis d'aller vérifier les autres prédicats pour chaque enregistrement. Cela risque d'être peu efficace si les autres prédicats portant sur les champs de la table sont très sélectifs.

L'index CTXCAT, comme l'index CONTEXT isole les mots. En plus, pour chaque enregistrement, il stocke aussi les valeurs des autres champs qui sont mentionnés lors de sa création pour pouvoir faire des recherches combinées. Si le résultat est assez efficace, l'espace occupé est particulièrement important puisque chaque mot est stocké avec les valeurs des autres champs de l'index.

Comme nous l'avons dit précédemment, ces index ne sont pas adaptés à l'indexation de textes volumineux. Nous allons donc prendre comme exemple le champ Adresse de la table CLIENTS de notre base exemple et le combiner au champ Pays, puis comparer les performances sur le domaine des recherches combinées.

À cette fin, nous créons un index de type CTXCAT intégrant la colonne Pays de la table CLIENTS :

```
EXEC CTX_DDL.CREATE_INDEX_SET('ISET_IS_CLIENT_ADRESSE_PAYS');  
EXEC CTX_DDL.ADD_INDEX('ISET_IS_CLIENT_ADRESSE_PAYS','PAYS');  
CREATE INDEX IS_CLIENT_ADRESSE_PAYS ON CLIENTS(Adresse1)  
INDEXTYPE IS CTXSYS.CTXCAT PARAMETERS ('index set ISET_IS_CLIENT_ADRESSE_PAYS');
```

La requête suivante s'exécute en 0,015 seconde et nécessite 4 Consistent Gets :

```
select * from clients t where CATSEARCH(adresse1,'FOREST',  
'pays = ''Japan''')> 0;
```


alors que l'équivalent, avec un index de type CONTEXT et l'opérateur CONTAINS combiné à une clause WHERE, s'exécute en 0,016 seconde et nécessite 130 Consistent Gets. L'équivalent avec un prédicat LIKE s'exécute en 0,016 seconde et nécessite 628 Consistent Gets. Le temps d'exécution n'est pas très significatif étant donné le faible volume, mais le nombre de Consistent Gets est parlant : entre la solution la plus lente et la plus rapide, il y a un ratio de 150. Cependant, le prix à payer pour l'index CTXCAT est de 18 Mo alors que la table occupe seulement 5 Mo. Le coût de l'index CONTEXT est lui de 1,3 Mo.

Ci-dessous les instructions permettant d'effectuer le test utilisant un index CONTEXT :

```
CREATE INDEX IS_FT_CLIENTS_ADRESSE1 ON CLIENTS(ADRESSE1) INDEXTYPE IS CTXSYS.
CONTEXT parameters ( 'LEXER mylex' );
select * from clients
where CONTAINS(adresse1, 'FOREST', 1) > 0 and pays='Japan';
```

Ci-dessous les instructions permettant d'effectuer le test n'utilisant aucun index :

```
select * from clients
where upper(adresse1) like '%FOREST%' and pays='Japan'
```

MS SQL Server

Sous SQL Server, la fonctionnalité de recherche Full Text est déléguée à des services externes à la base de données (Msftesql.exe et Msftefd.exe), de ce fait, les index Full Text ne sont pas dans la base de données. La fonctionnalité doit être activée pour chaque base de données soit dans SQL Server Management Studio, dans l'onglet Fichier des propriétés de la base de données, soit avec l'instruction suivante :

```
EXEC sp_fulltext_database @action = 'enable'
```

Il faut ensuite créer un catalogue, qui pourra être utilisé pour plusieurs tables. Cependant, pour indexer de gros volumes, il sera pertinent de créer des catalogues indépendants pour chaque table. L'instruction Create Fulltext Catalog permet de créer le catalogue en spécifiant les options de gestion des caractères diacritiques ainsi que l'emplacement des fichiers d'index sur le disque dur :

```
CREATE FULLTEXT CATALOG MonCatalogueFT as default
```

Il ne sera possible de créer qu'un seul index par table qui contiendra l'ensemble des colonnes à indexer :

```
Create Fulltext Index on texteslivres (texte)
KEY INDEX PK_TextesLivres WITH CHANGE_TRACKING AUTO
Create Fulltext Index on <NomDeLaTable>(<Colonnes, ...>)
KEY INDEX <ClePrimaireOuAutreIndexUnique>
WITH CHANGE_TRACKING <OptionsDeMiseAJour>
```

Ensuite, il est possible de faire des recherches Full Text avec différents opérateurs. L'opérateur CONTAINS permet de faire des recherches de termes simples en les combinant éventuellement avec des opérateurs booléens (AND, AND NOT, OR). Il intègre aussi la notion de préfixe "voyage*", les notions d'inflexion FORMSOF (INFLECTIONAL , voyage) et de synonymes au moyen d'un thesaurus.

```
select count(*) from texteslivres where CONTAINS(texte, 'voyage')
```

La fonction CONTAINSTABLE est une variante de CONTAINS qui permet de retourner une table à deux colonnes contenant la clé de l'index (Key) et le classement par pertinence (Rank) :

```
select * from CONTAINSTABLE(texteslivres,texte, 'voyage')
```

L'opérateur FREETEXT permet de faire une recherche analogue à CONTAINS en incluant les inflexions et les synonymes mais avec une syntaxe plus simple puisqu'il suffit de spécifier la liste de mots recherchés :

```
select * from texteslivres where FREETEXT(texte, 'voyage train')
```

La fonction FREETEXTTABLE retourne une table comme CONTAINSTABLE mais avec la syntaxe de FREETEXT.

MySQL

MySQL intègre aussi un mécanisme de recherche Full Text mais seul le moteur MyISAM intègre un index spécialisé permettant d'avoir de meilleurs résultats. Ci-après, un exemple de création d'index et de recherche est présenté :

```
Create FULLTEXT index IS_FT_TEXTESLIVRES ON TEXTESLIVRES(texte);  
SELECT * FROM texteslivres WHERE match(texte) against ('voyage')
```

Les recherches sont faites, par défaut, en mode langage naturel mais il est possible de travailler en mode booléen. La requête suivante recherche les enregistrements avec le mot "voyage" mais sans le mot "voyageur". Le signe plus (+) exige la présence du mot, le signe moins (-) exige son absence, sinon la présence est optionnelle mais augmente la pertinence. L'opérateur * permet de faire des recherches sur le début d'un mot (voyag*), > et < donnent plus ou moins d'importance à certains mots, les guillemets servent à chercher des correspondances exactes sur des phrases.

```
SELECT * FROM texteslivres WHERE match(texte) against ('+voyage  
voyages -voyageur' in BOOLEAN mode)
```

Par défaut, le résultat est trié par pertinence. Tout comme sous Oracle, il est possible de créer un index sur plusieurs colonnes et d'effectuer une recherche simultanément sur plusieurs colonnes.

5.3 Travail autour des tables

5.3.1 Paramètres de table

PCTFree = 0

À travers la clause de stockage et le paramètre PCTFREE (valant 10 % s'il est omis), le SGBDR réserve lors des insertions de données PCTFree % d'espace dans chaque bloc pour agrandir les données sans devoir faire de migration de ligne (*row migration*) lors des UPDATE qui auront peut-être lieu dans le futur.

Certaines tables contiennent des données dont les enregistrements ne grossiront jamais, l'espace réservé par l'option PCTFREE est donc perdu. De plus, il entraînera l'usage de PCTFree % blocs supplémentaires pour stocker le même nombre d'enregistrements que si le paramètre était à 0 %. Cela augmentera donc le nombre de Consistent Gets et réduira les performances d'autant.

Sur les tables qui ont des lignes qui ne varient pas en taille, il est donc pertinent de spécifier le paramètre PCTFREE à 0 dans la clause de stockage.

```
Create table xxx ( . . . ) PCTFREE 0;
```

Répartition des tablespaces

Pour chaque objet, il est possible de définir un tablespace spécifique. Si cette clause est omise, c'est le tablespace par défaut de l'utilisateur qui est choisi. Il sera donc le même pour tous les objets qui n'ont pas spécifié de tablespace.

Si vous disposez de plusieurs disques (ou ensembles de disques), vous avez tout intérêt à définir des tablespaces propres à chacun des disques. Ainsi, vous pourrez répartir les objets qui travaillent simultanément – et qui causent des Physical Reads – sur des disques différents et ainsi cumuler les bandes passantes des différents disques.

Chaque tablespace est composé de datafiles. Ces fichiers peuvent être répartis sur différents disques, mais il faut penser qu'on n'a aucun moyen d'influer sur la répartition des objets dans les datafiles. Donc, si vous souhaitez maîtriser cette répartition, créez plutôt des tablespaces avec des datafiles sur un seul disque ou groupe de disques.

```
Create table xxx ( . . . ) TABLESPACE NomDuTablespace;
```

MS SQL Server

Sous SQL Server, la même logique est applicable, sauf qu'on parlera de *filegroup* à la place de *tablespace*. De la même façon, il est possible d'affecter chaque objet à un *filegroup*.

MySQL

Avec le moteur MyISAM, vous pouvez spécifier les options `DATA DIRECTORY` et `INDEX DIRECTORY` pour chaque table et ainsi répartir vos tables sous différents disques.

Avec le moteur InnoDB, par défaut, il n'y a qu'un seul tablespace qui peut être constitué de plusieurs fichiers répartis sur plusieurs disques mais, de la même façon que sous Oracle avec les datafiles, vous ne pouvez pas influencer sur la répartition des données dans les fichiers. L'option `INNODB_FILE_PER_TABLE` permet d'avoir un tablespace par table mais elle ne permet pas de les répartir sur plusieurs volumes.

Compression de table

Cette option permet de compresser les doublons présents dans les blocs. On devrait d'ailleurs parler plutôt de "déduplication" que de "compression", car ce terme fait penser, à tort, à l'usage d'algorithmes de compression, tels que Lempel-Ziv. Cependant, la déduplication a l'avantage d'être mieux adaptée à l'usage dans une base de données.

Le principe est de compresser les données au sein d'un bloc. Les données identiques du bloc sont répertoriées dans un dictionnaire au début du bloc. De ce fait, chaque occurrence des données contient seulement une référence à l'entrée dans le dictionnaire au lieu de la donnée. Le dictionnaire est uniquement constitué de valeurs entières : ainsi, les données "Navarro Laurent" et "Laurent Navarro" ne seront pas compressées car elles sont, certes, composées de parties identiques mais elles sont des valeurs différentes. De fait, la compression sera généralement inopérante pour compresser des champs de taille importante.

La compression s'active avec l'option `COMPRESS` dans la clause de stockage de la table et elle n'est disponible que sur les tables organisées en tas (pas sur les IOT que nous étudierons un peu plus loin). `COMPRESS` fait référence à la compression de base, qui a été introduite dans la version 9i sous le nom de *DSS table compression*. Ce type ne compresses que les données insérées en direct path (via `sql*loader` en mode direct path ou avec le hint `APPEND`. Voir Chapitre 7, section 7.3.9, "Insertion en mode Direct path").

La version 11g Release 1 a introduit `COMPRESS FOR ALL OPERATION`, qui a été remplacée dans la version 11g Release 2 par `COMPRESS FOR OLTP`. Cette option permet d'appliquer la compression à toutes les opérations ; elle est à présent recommandée pour un usage en environnement OLTP.

La compression de table introduit quelques restrictions, telles qu’un nombre de colonnes inférieur à 255, et surtout des limites sur la suppression de colonnes.

Sur notre table exemple, CMD_LIGNES, nous obtenons une réduction de l’espace de 25 %. En termes de performances sur une requête nécessitant un Full Table Scan d’une grosse table, ce qui demande donc des Physical Reads, nous constatons un gain de l’ordre de 30 %. En écriture, les performances sont assez bonnes aussi, car la surcharge CPU est généralement compensée par la réduction des entrées/sorties. Le seul cas qui semble être plus fréquemment défavorable à la compression est l’utilisation d’UPDATE. Cette technique ne sera donc pas très adaptée à des tables qui subissent de très nombreuses modifications d’enregistrements.

La version 11g Release 2 a revu complètement la syntaxe des options de compression. Ces changements sont résumés au Tableau 5.2.

```
Create table xxx ( . . . ) COMPRESS;
```

Tableau 5.2 : Options de compression des différentes versions d'Oracle

Syntaxe 11g Release 2	Syntaxe 11g Release 1	Syntaxe 10g et 9i
COMPRESS ou COMPRESS BASIC	COMPRESS ou COMPRESS FOR DIRECT_LOAD OPERATION	COMPRESS
COMPRESS FOR OLTP	COMPRESS FOR ALL OPERATION	Non disponible
COMPRESS FOR QUERY et COMPRESS FOR ARCHIVE	Nouvelles options 11g Release 2 nécessitant le moteur Exabyte	

MS SQL Server

La version 2008, dans ses éditions Entreprise et Développeur, a introduit deux notions de compression : la compression de ligne et la compression de page.

La compression de ligne est un mécanisme qui optimise l'espace requis pour le stockage des types de taille fixe.

La compression de page travaille au niveau de la page. Cette technique recherche les valeurs ayant le même début et les doublons afin de les stocker dans un dictionnaire. Cette méthode est comparable à celle d'Oracle, hormis le fait qu'elle travaille, en plus, sur les débuts des valeurs et pas seulement sur les valeurs entières.

La compression des données s'active à l'aide de l'option de table DATA_COMPRESSION avec les options suivantes :

DATA_COMPRESSION = { NONE | ROW | PAGE }

Gestion des LOB

Les LOB (*Large Object* correspondant aux types CLOB et BLOB) sont stockés dans un segment séparé de celui du tas. Cependant, par défaut, la clause `ENABLE STORAGE IN ROW` est activée. De ce fait, les LOB d'une taille inférieure à ≈ 4 Ko sont stockés avec le reste de l'enregistrement, alors que les LOB plus gros sont stockés dans le segment séparé.

Cela risque de ralentir les Table Scan ne nécessitant pas les données LOB car il y a plus de blocs à lire que si les données LOB étaient toutes dans un segment séparé. Toutefois, ce mode de stockage permet d'accélérer la récupération des enregistrements avec des petits LOB en évitant les accès supplémentaires au segment LOB. Selon le type d'accès dont vous avez besoin, il faudra choisir le mode le plus adapté. Si vous ne récupérez jamais les LOB dans les Table Scan mais seulement en accès monoenregistrement, il sera peut-être pertinent de les stocker séparément. Laisser le stockage des petits LOB activé en ligne a pour effet d'agrandir la zone tas de la table. Par contre, cela évite des va-et-vient entre le tas et le segment LOB pour les petites données contenues dans les champs LOB.

Pour spécifier cette option, il faut ajouter dans la clause `STORAGE` de la table :

```
lob (<LeChampLob>) store as <NomDuSegment> (disable storage in row CHUNK 8K);
```

`NomDuSegment` est le nom du segment qui contiendra idéalement le nom de la table et du champ, par exemple `SegLob_Matable_Monchamp`. Le paramètre `CHUNK` contient la taille utilisée pour les allocations d'espace LOB pour une donnée dans le segment LOB. Cette valeur est forcément un multiple de la taille du bloc (8 Ko par défaut). Pour des fichiers, la taille initiale de 8 Ko et le fait de désactiver le stockage en ligne sont adaptés, alors que pour des champs CLOB qui sont des champs texte potentiellement gros, ce choix n'est pas toujours judicieux. Cela entraînera la création de beaucoup de blocs de 8 Ko, pleins de vide, pour rien si ce sont des petites chaînes. Ayez bien conscience que si vous désactivez le stockage en ligne, un CLOB de 10 caractères occupera 8 Ko.

Compression des LOB

La version 11g Release 1 a introduit la notion de `SecureFiles` pour stocker les champs LOB. Une des options intéressantes du point de vue de l'optimisation est la compression des champs LOB. Complètement transparente, elle a seulement un impact sur les performances, surtout lors des opérations d'écriture. Elle n'est adaptée que pour des données qui se compressent bien. Il est donc sans intérêt d'activer

la compression d'un LOB destiné à stocker des formats déjà compressés (JPEG, MPEG, ZIP, Audio, etc.).

```
lob (<LeChampLob>) store as <NomDuSegment> (compress [ HIGH | MEDIUM | LOW ] );
```

Sur notre table exemple, contenant des pages de livres, nous obtenons un gain d'espace de 30 %. Le parcours séquentiel du champ LOB sur plusieurs enregistrements est en toute logique pénalisé, mais ce n'est généralement pas l'opération la plus fréquente sur ce type de champ. En revanche, la recherche *via* un index Full Text n'est, en toute logique, pas du tout affectée puisque seul l'index est utilisé. Dans ce cas-là, seule la manipulation des LOB est impactée.

SecureFiles met aussi à disposition une fonction de déduplication de données à l'aide de l'option DEDUPLICATE. Celle-ci entraîne, elle aussi, une surcharge significative mais seulement en écriture. Elle peut éventuellement être intéressante si vos données s'y prêtent, c'est-à-dire si plusieurs enregistrements contiennent exactement la même valeur. Sur la table exemple LIVRES, qui contient les données en quatre exemplaires, cela fonctionne bien, mais il semble que cette option ait pour effet de désactiver le stockage en ligne, ce qui, dans notre cas, est pénalisant.

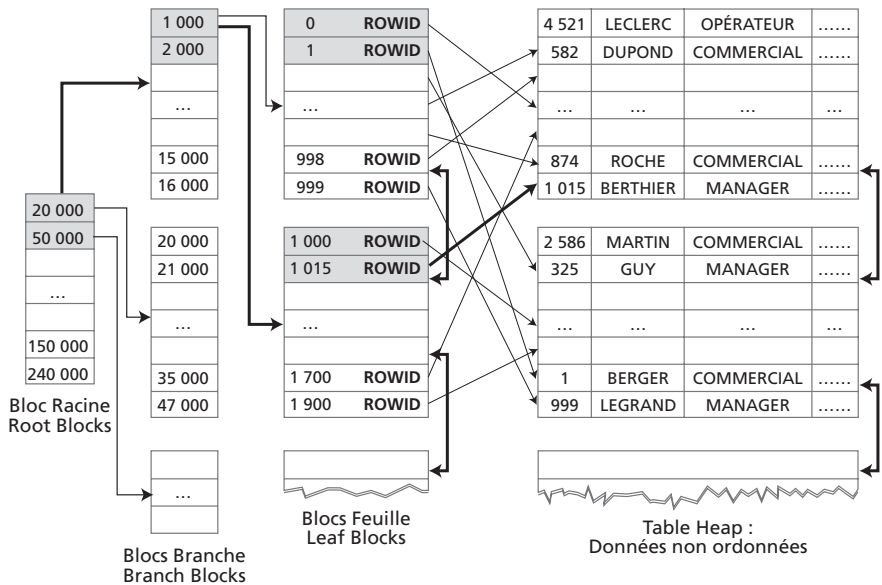
5.3.2 Index Organized Table

Pour rappel, la structure d'une table classique organisée en tas (*heap*) ou HOT (*Heap Organized Table*) utilisant des index B*Tree est la suivante (voir Figure 5.14a).

Les IOT (*Index Organized Table*) ont une organisation sensiblement différente puisqu'elles intègrent les données dans l'index de clé primaire (voir Figure 5.14b). Sur une table organisée en tas, le tas et la clé primaire sont deux objets distincts alors que sur une table de type IOT, ils sont confondus.

Cette organisation présente les avantages suivants :

- **Gain d'espace.** Il n'est plus nécessaire :
 - de dupliquer les données de la clé primaire entre le tas et l'index ;
 - de stocker les RowID permettant de faire le lien entre l'index B*Tree et le tas.
- **Gain en possibilité d'usage de l'index.** Au-delà de 5 % de la table à parcourir, on estime qu'il n'est plus intéressant d'utiliser un index classique à cause des va-et-vient entre lui et le tas. Cette règle ne s'applique pas de la même façon aux IOT, qui n'ont pas ces problèmes puisque l'index de clé primaire et les données sont confondus.



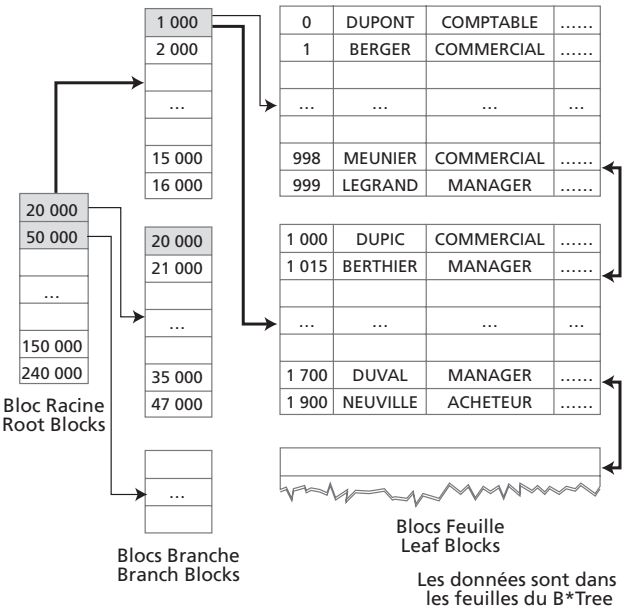
Le RowID permet de pointer sur les lignes stockées dans le Tas

Figure 5.14a

Table Heap et un index.

Figure 5.14b

Structure d'une table IOT



Les données sont dans les feuilles du B*Tree

Toute médaille ayant son revers, voilà les principaux inconvénients des IOT :

- La modification des valeurs de la clé primaire ou l'insertion non séquentielle provoquent des mouvements de données plus importants puisque cela peut conduire à déplacer des lignes et non pas juste des clés dans des index.
- Les index secondaires nécessitent plus d'espace (voir ci-après la section "index B*Tree et IOT").
- Les cas où le parcours seul de l'index de clé primaire répond à la demande seront moins performants car, à présent, parcourir l'index de clé primaire revient à parcourir la table, ce qui se traduira généralement par plus de données à manipuler.

Mise en œuvre

La déclaration d'une table organisée en index (IOT) se fait, dans le `CREATE TABLE`, par la spécification de l'organisation `INDEX` (au lieu de `HEAP` par défaut). Il n'est pas possible de convertir une table `HOT` en `IOT` et *vice versa*.

```
Create table xxx ( . . . ) ORGANIZATION INDEX;
```

Index B*Tree et IOT

Sur une table de type `IOT`, la clé primaire est intégrée, mais il est cependant possible de créer des index secondaires sur d'autres colonnes, comme sur les tables organisées en tas. Ces index seront d'une nature un peu différente.

Les index `B*Tree` contiennent habituellement, pour chaque entrée, la clé de l'index plus le `RowID` pointant sur une position dans le tas. Cependant, il existe une différence majeure entre les deux formats de table : sur celles organisées en tas, les lignes ne bougent pas dans le tas, alors que sur celles de type `IOT`, elles peuvent être déplacées car elles doivent respecter l'ordre de la clé primaire. Cela signifie que l'emploi d'un `RowID` pour désigner les enregistrements sur les index secondaires des tables de type `IOT` est délicat. On ne peut pas envisager de mettre à jour tous les index à chaque mouvement de ligne. Il faudrait pour cela retrouver et modifier les entrées correspondantes dans tous les index, ce qui serait particulièrement pénalisant.

Donc, les index `B*Tree` sur `IOT` ne contiennent pas de `RowID` pour retrouver la ligne pointée mais un `Logical RowID` plus la clé primaire. Le `Logical RowID` est le numéro du bloc dans lequel était la ligne au moment de l'insertion. C'est donc l'endroit où elle devrait toujours être, puisqu'on suppose dans un `IOT` que les données sont insérées séquentiellement. Cependant, au cas où l'enregistrement aurait bougé, on utiliserait les données de la clé primaire placées dans l'index pour retrouver l'enregistrement en parcourant le `B*Tree` de la table.

Les index B*Tree sur IOT contiennent la clé de l'index, Logical RowID à l'insertion et la clé primaire. Ils sont donc plus gros que les index sur les tables organisées en tas. C'est d'autant plus vrai que la clé primaire sera grosse. On veillera à ne pas avoir trop d'index secondaires sur les tables IOT qui n'ont pas une clé primaire de petite taille.

De plus, si la table est mouvementée et que cela conduise à des déplacements d'enregistrements dans d'autres blocs, le Logical RowID contenu dans l'index conduira souvent au mauvais bloc. Il faudra alors parcourir le B*Tree de la clé primaire pour trouver l'enregistrement demandé, ce qui provoquera inutilement quelques opérations supplémentaires. Ce type de problème ne fera qu'augmenter au fil du temps car il y aura de plus en plus de déplacements d'enregistrements. Une reconstruction de l'index permettra de refaire pointer les Logical RowID sur les blocs adéquats.

Index bitmap et IOT

Pour mettre en œuvre un index bitmap sur un IOT, il faut que la table possède une table de mapping, spécifiant les paramètres suivants lors de la création de la table :

```
ORGANIZATION INDEX MAPPING TABLE
```

Impact sur les opérations LMD

L'organisation IOT des tables est un peu moins performante que l'organisation en tas si on insère des données de façon non linéaire par rapport à la clé primaire. Cependant, même dans ce cas, ce n'est pas désastreux au moment de l'insertion car elle a lieu dans un B*Tree. Par contre, cela risque de provoquer des déplacements d'enregistrements et, du coup, l'effet sera peut-être plus notable sur les données existantes car les index secondaires auront des Logical RowID erronés, ce qui entraînera des accès supplémentaires lors des requêtes SELECT.

À la suite de nombreuses mises à jour sur les clés ou d'insertions non séquentielles, des "trous" peuvent apparaître dans la table. Il sera alors peut-être judicieux de la réorganiser afin de la compacter.

```
ALTER TABLE clients_iot MOVE ONLINE;
```

Tableau 5.3 : Comparaison des temps de mise à jour des clés de 30 % de la table

<i>Paramètre</i>	<i>Table HEAP</i>	<i>Table IOT</i>
Temps	0,89 s	6,047 s
Consistent Gets	322	21 918
CPU	39	201

```
update clients set NOCLIENT=NOCLIENT+1000000 where pays='USA'
```

Avant sa mise à jour, la table CLIENTS en organisation IOT occupait 6 144 Ko, après, elle occupe 8 192 Ko. Cela est lié au fait que les enregistrements ont été déplacés de leurs feuilles existantes pour aller dans de nouvelles feuilles. La plus grande valeur était 145 000 avant la mise à jour et celle-ci a déplacé les enregistrements modifiés dans des feuilles situées au-delà de cette valeur.

Tableau 5.4 : Comparaison des performances d'un index secondaire sur une table organisée en IOT avant et après déplacement des clés

Paramètre	Avant	Après
Consistent Gets	3 570	4 408
CPU	0	2

```
select count(nom) from clients where pays='Japan';
update clients set NOCLIENT=NOCLIENT+1000000 where pays='Japan';
select count(nom) from clients where pays='Japan';
```

Il y a une augmentation de 25 % des Consistent Gets liée au fait que les enregistrements ne sont plus là où ils devraient être, d'après les Logical RowID, ce qui provoque des parcours de l'index intégré de la clé primaire.

Évaluation de l'espace occupé

Concernant le gain d'espace, si nous testons sur notre base exemple avec la table CMD :

Taille du tas (26 Mo) + Index clé primaire (30 Mo) = 56 Mo.
Taille IOT = 28 Mo.

On constate un gain significatif. De façon générale, la taille de l'IOT sera légèrement plus grande que la taille du tas, l'écart étant constitué des branches du B*Tree. Il est à noter que plus l'index de clé primaire compte de colonnes et moins les enregistrements en ont, plus le ratio d'espace économisé est important par rapport à une table organisée en tas (ce qui est le cas dans notre exemple).

Par contre, si nous regardons l'évolution de la taille de l'index secondaire sur la colonne Noclient, nous voyons que, sur la table organisée en tas, il occupe 18 Mo alors que, sur la table de type IOT, il occupe 25 Mo soit presque 40 % de plus alors que la clé primaire est plutôt petite puisque c'est une valeur numérique.

Overflow segment

Comme nous l'avons vu précédemment, une table de type IOT a pour effet de pénaliser les opérations pouvant se faire uniquement sur l'index de la clé primaire, puisque l'index contient à présent l'ensemble des champs. Afin de réduire cet impact, il est

possible de mettre en œuvre un segment de débordement (*overflow segment*), qui est, en fait, une table organisées en tas contenant une partie des champs.

Cette solution fait chuter les performances quand il faut souvent récupérer les données qui sont dans ce segment – par exemple, dans le cas d'un Full Table Scan qui aurait besoin des données qui sont situées dans le segment de débordement. Par contre, dans les autres cas, les performances sont meilleures car la partie index de la table IOT est moins volumineuse.

La mise en œuvre se fait avec les mots clés **OVERFLOW** et **INCLUDING** qui précisent la dernière colonne à être dans la partie index, les colonnes suivantes seront donc dans le segment de débordement (ci-après : Jobhistory et Managercomments).

```
CREATE TABLE IOTEMPOverflow (
  EMPNO          NUMBER ,
  ENAME          VARCHAR2(10) NOT NULL ,
  JOB            VARCHAR2(9) ,
  JobHistory     VARCHAR2(4000),
  ManagerComments VARCHAR2(4000),
  constraint PK_IOTEMPOverflow primary key(empno))
Organization index including JOB overflow ;
```

MS SQL Server

Nous retrouvons sous SQL Server la même fonctionnalité sous le nom de *Clustered Index*. La création d'un index clustered (un seul possible par table) aura pour effet de transformer la table d'une organisation en tas en une organisation en index.

Par défaut, une clé primaire crée un index de type clustered. Donc, si le contraire n'est pas spécifié, toute table qui a une clé primaire est de type Index Organized Table.

Dans l'instruction **CREATE TABLE**, la clause **CONSTRAINT** permet de préciser le type d'index :

```
CONSTRAINT constraint_name
{ { PRIMARY KEY | UNIQUE } [ CLUSTERED | NONCLUSTERED ] }
```

Les index secondaires sont par défaut **NON CLUSTERED** :

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
ON <object> ( column [ ASC | DESC ] [ ,...n ] )
```

Concernant les index secondaires sur table IOT, contrairement à ce qui se passe avec Oracle, l'index ne contient pas de Logical RowID mais seulement la clé de la table. Cela a pour effet de moins pénaliser les index secondaires sur IOT de SQL Server que sur Oracle si les enregistrements changent de bloc mais ils seront un peu moins performants si la table a une croissance respectant l'ordre de la clé. L'index souffre par contre du même problème de taille si la clé primaire n'est pas petite, bien que cela soit un peu réduit par l'absence du Logical RowID.

MySQL

Le moteur InnoDB utilise un stockage de type IOT exclusivement.

Le moteur MyISAM ne propose pas d'option pour implémenter les IOT.

C'est donc le type de la table dans les instructions CREATE TABLE et ALTER TABLE qui décidera du type de l'organisation de la table.

```
CREATE TABLE tbl_name (create_definition,...)
    {ENGINE|TYPE} = {InnoDB| MYISAM}
```

Évaluation des performances

Effectuons un premier test sur une requête portant uniquement sur les index :

```
select count(*) from cmd_lignes
```

Figure 5.15
Trace d'exécution
sur une table Heap.

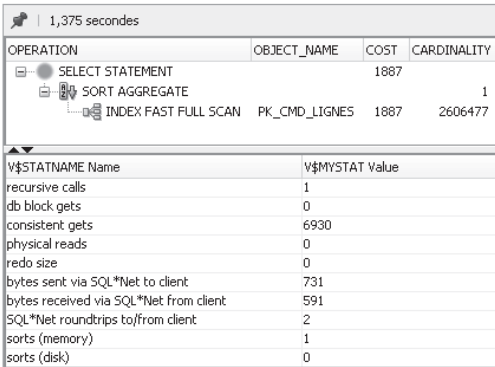
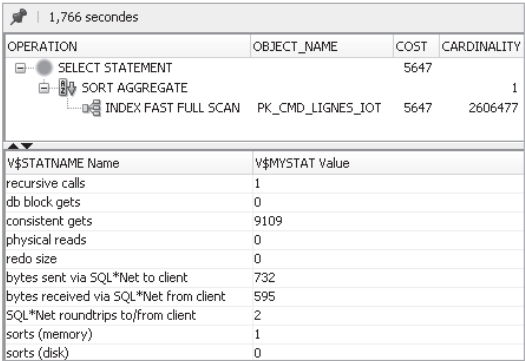


Figure 5.16
Trace d'exécution
sur une table IOT.



Les requêtes portant uniquement sur l'index de clé primaire sont plus rapides sur des table organisées en tas plus B*Tree car l'index B*Tree est plus petit (56 Mo) que la table IOT (75 Mo). Sur la table organisée en IOT, on constate un Index Fast Full Scan, ce qui est équivalent à un Full Table Scan sur une table organisée en tas. Sur les requêtes nécessitant uniquement l'utilisation d'index secondaires, l'avantage sera là aussi aux tables organisées en tas car, nous l'avons vu, les index secondaires sont plus petits sur les tables organisées en tas que sur les tables IOT, mais l'écart sera généralement moindre qu'avec l'usage de la clé primaire.

Nous allons faire un deuxième test portant sur un parcours d'intervalles de la clé primaire nécessitant d'accéder à des colonnes non indexées. La requête ci-après travaille sur une plage de 5 477 enregistrements

```
select sum(montant) from cmd_lignes
where nocmd between 100000 and 102000;
```

On constate que l'optimiseur décide d'effectuer un Range Scan sur l'index de clé primaire pour récupérer les enregistrements correspondants par leur RowID afin d'effectuer le calcul de la somme.

Figure 5.17

Trace d'exécution sur une table organisée en tas.

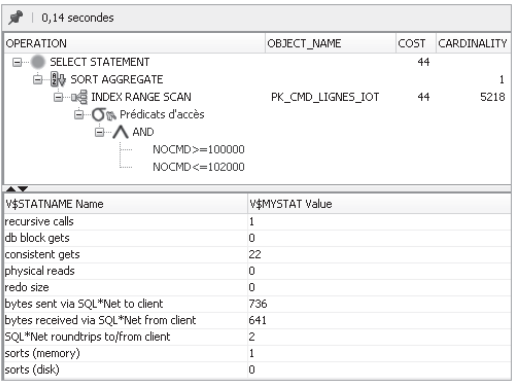
0,156 secondes			
OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		1148	
SORT AGGREGATE			1
TABLE ACCESS BY INDEX ROWID	CMD_LIGNES	1148	5218
INDEX RANGE SCAN	PK_CMD_LIGNES	16	5218
Prédicats d'accès			
AND			
NOCMD >= 100000			
NOCMD <= 102000			
▼			
V\$STATNAME Name	V\$MYSTAT Value		
recursive calls	1		
db block gets	0		
consistent gets	1251		
physical reads	0		
redo size	0		
bytes sent via SQL*Net to client	735		
bytes received via SQL*Net from client	637		
SQL*Net roundtrips to/from client	2		
sorts (memory)	1		
sorts (disk)	0		

À la Figure 5.18, on voit que le SGBDR scanne uniquement la partie de la table concernée et récupère les données dans la même passe. On a un meilleur temps de réponse et moins de Consistent Gets. C'est le cas de prédilection de l'IOT¹.

1. En fonction du facteur de foisonnement des données du test réalisé sur la table organisée en HEAP, le gain pourrait être encore plus important.

Figure 5.18

Trace d'exécution sur une table organisée en IOT.



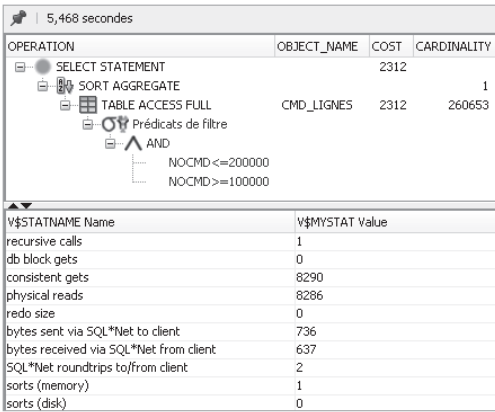
Nous allons réaliser un troisième test, analogue au précédent mais portant sur un intervalle plus important. La requête ci-après travaille sur un intervalle de 261 205 enregistrements, soit 10 % de la table.

```
select sum(montant) from cmd_lignes
where nocmd between 100000 and 200000
```

Sur la table organisée en tas, l’optimiseur choisit de faire un Full Table Scan sans laisser la table en cache (visible car chaque exécution contient des Physical Reads) [voir Figure 5.19].

Figure 5.19

Requête sur une table organisée en tas.



Sur la table organisée en index (IOT), l’optimiseur fait un Range Scan en suivant la clé primaire, ce qui est toujours le cas de prédilection des IOT. De plus, il laisse la table en cache, probablement à cause de son statut d’index (voir Figure 5.20).

Si on augmente encore l'intervalle pour couvrir 50 % de la table, on constate que l'optimiseur ne choisit plus de faire d'Index Range Scan sur l'IOT mais un Index Fast Full Scan. Cela vient probablement du fait qu'un Index Range Scan parcourt les blocs dans l'ordre de l'index – et effectue donc des lectures aléatoires – alors qu'un Index Fast Full Scan privilégie des lectures séquentielles en lisant les blocs dans l'ordre du segment. Or, selon la performance du disque ou selon que les données soient en cache ou pas, le surcoût d'une lecture aléatoire comparée à une lecture séquentielle peut se réduire de façon significative par rapport aux coûts estimés qui sont de 5 656 pour la version Full Scan et de 10 401 pour la version Range Scan. Si nous forçons un Index Range Scan à l'aide du hint `INDEX_RS`, nous constatons une réduction des Consistent Gets de 9 109 à 4 535 et une réduction du temps d'exécution de 1,43 seconde à 0,91 seconde, nous avons donc un résultat différent de l'estimation de l'optimiseur.

Figure 5.20

Requête sur une table organisée en IOT.

0,39 secondes		
OPERATION	OBJECT_NAME	COST
SELECT STATEMENT		2082
SORT AGGREGATE		
INDEX RANGE SCAN	PK_CMD_LIGNES_IOT	2082
Prédicats d'accès		
AND		
NOCMD >= 100000		
NOCMD <= 200000		
▼		
V\$STATNAME Name	V\$MYSTAT Value	
recursive calls	1	
db block gets	0	
consistent gets	912	
physical reads	0	
redo size	0	
bytes sent via SQL*Net to client	737	
bytes received via SQL*Net from client	641	
SQL*Net roundtrips to/from client	2	
sorts (memory)	1	
sorts (disk)	0	

Nous pourrions penser que cela est dû au fait que les données sont dans le cache, mais si nous exécutons l'instruction `alter system flush buffer_cache;` pour vider le cache et que nous réexécutons les requêtes, nous trouverons des quantités de Physical Reads et de Consistent Gets très proches. Malgré tout, nous observons la même tendance, la version Range Scan s'exécute en 3,78 secondes et la version Full Scan en 6,15 secondes. Les choses pourraient être différentes suivant la façon dont la table a été remplie, les performances des disques, etc. Donc, comme toujours, testez !

```
select /*+index_rs(t)*/ sum(montant) from cmd_lignes_iot t
where nocmd between 100000 and 600000
```


5.3.3 Cluster

Le cluster est un objet d'optimisation qui existe depuis longtemps dans les bases Oracle mais qui n'a jamais connu de franc succès, ce qui, de mon point de vue, est tout à fait justifié étant donné les problèmes qu'il pose. Ce chapitre a donc pour principal objet d'expliquer le fonctionnement des clusters et la raison pour laquelle ils ne vont probablement pas vous convenir. Je n'exclus cependant pas le fait que, dans quelques cas, ils puissent être intéressants.

Un cluster est un regroupement entre deux tables – pour simplifier, on pourrait dire qu'un cluster est une jointure forte. C'est un objet qui est orienté pour des tables ayant entre elles une relation maître/détails. L'idée est que le cluster va stocker dans une même zone ces deux tables au lieu de les stocker dans deux zones distinctes. Il pousse même la chose jusqu'à stocker les enregistrements maîtres et les enregistrements détails dans le même bloc de données. L'intérêt réside dans le fait que, lorsque vous faites une jointure entre ces deux tables, en lisant un seul bloc vous récupérez les données maîtres et détails et en plus il n'y a pas de choses compliquées à faire pour réaliser la jointure.

Pour l'instant, le cluster a l'air formidable, voyons donc où est le problème. Afin de pouvoir stocker les enregistrements détails avec l'enregistrement maître, le cluster va réserver un bloc (soit généralement 8 Ko) pour chaque clé, ce qui risque d'avoir pour effet de faire exploser l'espace nécessaire pour stocker les données et ainsi d'augmenter le nombre d'E/S, ce qui sera finalement assez pénalisant.

Par exemple, sur notre base de test, où nous allons nous limiter à 100 000 commandes au lieu d'un million :

Tas et clé primaire de CMD + CMD_LIGNES = 56 Mo + 130 Mo = 186 Mo pour un million de commandes soit environ 20 Mo pour 100 000 commandes.

Cluster CLUS_CMD plus les index = 811 Mo + 9 Mo + 4 Mo + 3 Mo = 827 Mo soit un peu plus de 40 fois plus.

Avec une taille de bloc de 2 Ko (la plus petite valeur possible) au lieu 8 Ko, l'espace occupé ne serait plus que de 200 Mo ce qui est encore 10 fois plus que sans cluster.

Exemple de création de tables dans un cluster :

```
create cluster CLUS_CMD (NoCmd number) size 8k;  
Create Index IDX_CLUS_CMD ON CLUSTER CLUS_CMD;  
Create Table CMD_CLUS (  
    NOCMD          NUMBER not null,
```

```

    NOCLIENT      NUMBER,
    DATECOMMANDE  DATE,
    ETATCOMMANDE  CHAR(1),
    Constraint PK_clus_CMD primary key (NoCmd))
cluster CLUS_CMD(NoCmd);

Create Table CMD_LIGNES_CLUS (
    NOCMD      NUMBER not null,
    NOLIVRE    NUMBER not null,
    QUANTITE    NUMBER,
    REMISE      NUMBER,
    MONTANT     NUMBER,
    constraint PK_CLUS_CMD_LIGNES primary key (NOCMD, NOLIVRE),
    constraint FK_CLUS_CMD_LIGNES_CMD foreign key (NOCMD) references CMD_
    CLUS(NOCMD) )
cluster CLUS_CMD(NOCMD);

insert into CMD_CLUS  select * from CMD where nocmd <114524;
commit;

insert into CMD_LIGNES_CLUS select * from CMD_LIGNES where nocmd <114524;
commit;

```

Évaluation

Nous allons vérifier avec quelques tests si cette mauvaise réputation est méritée ou pas. Nous allons exécuter la requête suivante qui porte sur quelques milliers d'enregistrements (10 000 commandes et 25 000 lignes de commandes) et utilise des données de la table maître CMD et de la table détails CMD_LIGNES en filtrant par la clé du cluster, à savoir le champ Nocmd (cas d'utilisation optimale du cluster). Cet exemple est un peu compliqué, le champ Montantremisemardi a seulement pour but d'empêcher l'optimiseur de simplifier la requête afin de forcer la jointure.

```

select avg(montantttotal) MontantMoyen,avg(NbrItem)NbrItemMoyen,
       sum(NbrItem) NbrTotalItem,count(distinct nocmd) NbrCmd,
       sum(montantttotal) MontantTotal,sum(MontantRemiseMardi) TotalRemiseMardi
from (
    select c.nocmd,c.datecommande,sum(montant) montantttotal,count(*) NbrItem
       ,sum(case when to_char(c.datecommande,'d')=2 then montant*0.03
          else 0 end) MontantRemiseMardi
    from cmd_clus c join cmd_lignes_clus cl on c.nocmd=cl.nocmd
    where c.nocmd between 70000 and 80000
    group by c.nocmd, c.datecommande
);

```

MONTANTMOYEN	NBRITEMMOYEN	NBRTOTALITEM	NBRCMD	MONTANTTOTAL	TOTALREISEMARDI
221,324	2,488	24884	10001	2213465,18	7977,66

Nous constatons un avantage significatif en temps d'exécution pour la solution cluster car pour notre test les blocs impliqués restent dans le cache. Si nous vidons le cache à l'aide de l'instruction `alter system flush buffer_cache;`, la solution cluster nécessite 10 038 Physical Reads et 11,7 secondes contre 5,7 secondes et 8 360 Physical Reads. La tendance se confirme de la même façon, en défaveur du cluster, si nous augmentons la taille de l'intervalle à parcourir qui aura pour effet d'augmenter le nombre de blocs et donc de réduire leur probabilité de présence en cache. Le coût estimé par l'optimiseur de chacune des solutions confirme d'ailleurs cette tendance défavorable pour le cluster.

La solution cluster est donc intéressante s'il y a assez d'enregistrements pour qu'une jointure soit coûteuse, mais pas trop, pour que la taille ne devienne pas pénalisante. Pour la plupart des autres cas, le cluster sera à éviter. Cela laisse une plage d'intérêt assez réduite. Si votre problématique s'y trouve, alors il peut y avoir des gains significatifs, mais si ce n'est pas clairement le cas, je vous conseille de rester à l'écart de cet objet qui peut vous faire payer assez cher le fait de l'avoir choisi à tort.

Figure 5.21
Trace d'exécution sans cluster.

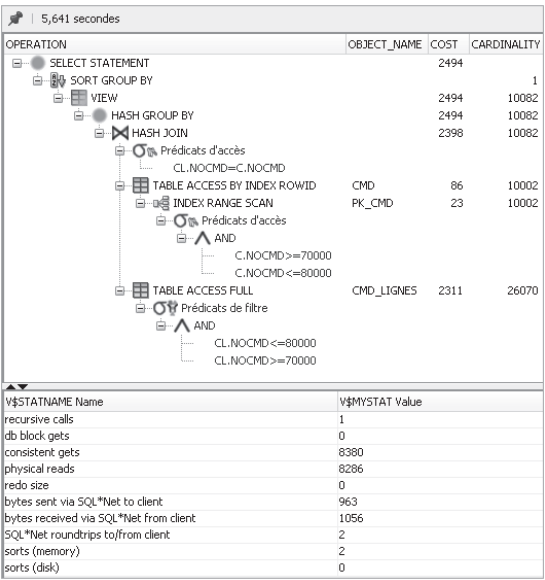


Figure 5.22

Trace d'exécution
avec cluster.

0,375 secondes			
OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		20153	1
SORT GROUP BY			
VIEW		20153	10852
HASH GROUP BY		20153	10852
NESTED LOOPS		20048	10852
TABLE ACCESS BY INDEX ROWID	CMD_CLUS	10042	10002
INDEX RANGE SCAN	PK_CLUS_CMD	36	10002
Prédicats d'accès			
AND			
C.INOCMD >= 70000			
C.INOCMD <= 80000			
TABLE ACCESS CLUSTER	CMD_LIGNES_CLUS	1	1
Prédicats de filtre			
AND			
CL.INOCMD >= 70000			
CL.INOCMD <= 80000			
CL.INOCMD = C.INOCMD			

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	1
db block gets	0
consistent gets	30040
physical reads	0
redo size	0
bytes sent via SQL*Net to client	960
bytes received via SQL*Net from client	1067
SQL*Net roundtrips to/from client	2
sorts (memory)	2
sorts (disk)	0

5.3.4 Partitionnement des données

Le partitionnement permet de diviser une grosse table en morceaux physiquement plus petits (les partitions), tout en gardant la vision logique d'une grosse table. Il sera possible d'effectuer des opérations de maintenance sur les partitions individuellement, ce qui permettra d'augmenter la disponibilité des données. Les partitions sont créées à partir des valeurs des colonnes qui seront définies comme les clés de partitionnement. Il peut y avoir jusqu'à 16 colonnes et elles doivent être spécifiées NOT NULL. Une table peut être divisée en 64 000 partitions au plus. Oracle 11g Release 1 introduit la possibilité de partitionner suivant une colonne calculée en autorisant le partitionnement sur des colonnes virtuelles.

Le partitionnement n'est pas disponible sur toutes les éditions d'Oracle (*idem* pour MySQL et MS SQL Server).

Du point de vue de l'optimisation, le principal intérêt des partitions est que lorsque l'optimiseur pourra déterminer les partitions nécessaires, les requêtes n'utiliseront que celles-ci et ignoreront les autres, ce qui améliorera les performances. Il faut donc, lorsque vous utilisez des tables partitionnées, essayer le plus possible de mettre des conditions qui permettent de sélectionner les partitions concernées. L'autre intérêt est de pouvoir spécifier les tablespaces associés à chaque partition, et ainsi de répartir les données sur différents volumes afin d'augmenter les débits et de pouvoir paralléliser au mieux. Il sera possible, *a posteriori*, de fusionner, diviser et déplacer chacune des partitions.

Le partitionnement occupe, presque à lui seul, un livre entier dans la documentation Oracle. Le but ici n'est pas d'être exhaustif, mais de donner quelques clés pour juger si sa mise en œuvre est pertinente pour vous, et si elle l'est, de vous donner les bases.

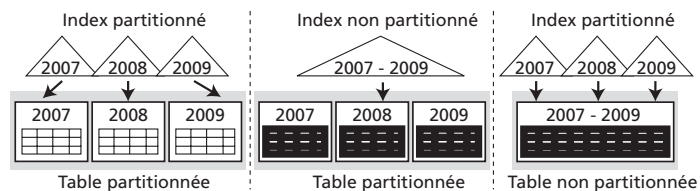
Mise en œuvre

Il est possible de partitionner les tables (HEAP ou IOT) et les index (B*Tree ou bitmap) et de les combiner de différentes façons (voir Figure 5.23) :

- table et index partitionnés suivant le même découpage (relativement fréquent) ;
- table partitionnée et index non partitionné (relativement fréquent) ;
- table non partitionnée et index partitionné (plutôt rare).

Figure 5.23

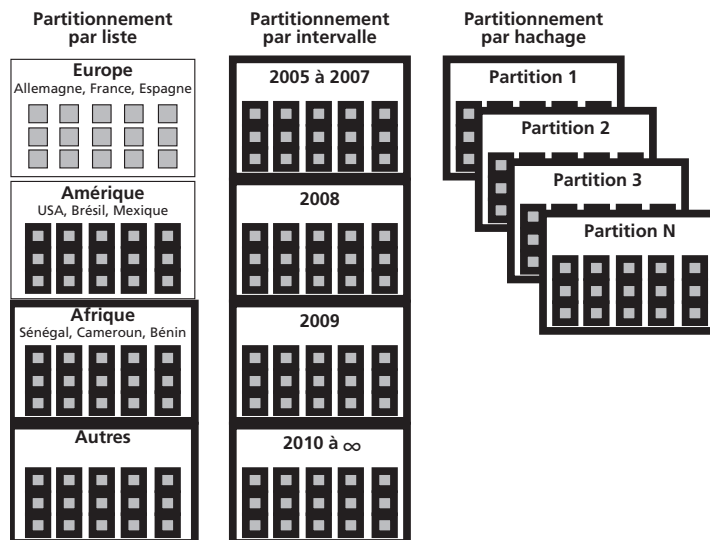
Combinaisons possibles entre tables partitionnées et index partitionnés.



Il existe principalement trois types de partitionnement :

Figure 5.24

Principaux types de partitionnement.



Partitionnement par liste

Il permet de définir pour chaque partition la liste des valeurs qui la constituent plus la valeur DEFAULT qui représente toutes les autres valeurs possibles. Ci-après un exemple du partitionnement de la table CLIENTS par la colonne Pays.

```
create table CLIENTS_PART (
  NOCLIENT    NUMBER not null,
  NOM          VARCHAR2(50),
  PRENOM       VARCHAR2(50),
  ADRESSE1     VARCHAR2(100),
  ADRESSE2     VARCHAR2(100),
  CODEPOSTAL   VARCHAR2(10),
  VILLE        VARCHAR2(50),
  PAYS         VARCHAR2(50),
  TEL          VARCHAR2(20),
  EMAIL        VARCHAR2(50),
  constraint PK_CLIENTS_PART primary key (NOCLIENT)
)
PARTITION BY LIST(PAYS)
(
  PARTITION Clients_Amerique VALUES ('USA','Brésil','Mexique')
    tablespace TableSpace1,
  PARTITION Clients_Europe VALUES ('Allemagne','France','Espagne')
    tablespace TableSpace2,
  PARTITION Clients_Afrique VALUES ('Sénégal','Cameroun','Bénin')
    tablespace TableSpace3,
  PARTITION Clients_Autres VALUES(DEFAULT) tablespace TableSpace2
);
```

Partitionnement par intervalle

Il permet de définir pour chaque partition l'intervalle de valeurs qui la constituent en spécifiant la borne supérieure. Si une valeur est supérieure à la plus grande borne, l'opération échoue. Pour être sûr de n'exclure aucune valeur, il suffit de spécifier la valeur MAXVALUE qui représente la plus grande valeur possible.

```
create table CMD_PART(
  NOCMD        NUMBER not null,
  NOCLIENT     NUMBER,
  DATECOMMANDE DATE,
  ETATCOMMANDE CHAR(1),
  constraint PK_CMD_PART primary key (NOCMD)
)
PARTITION BY RANGE(DATECOMMANDE)
(
  PARTITION CMD_2005_2007 VALUES LESS THAN(TO_DATE('01/01/2008','DD/MM/YYYY'))
    TABLESPACE TableSpace1,
  PARTITION CMD_2008 VALUES LESS THAN(TO_DATE('01/01/2009','DD/MM/YYYY'))
    TABLESPACE TableSpace2,
  PARTITION CMD_2009_MAX VALUES LESS THAN(MAXVALUE) TABLESPACE TableSpace3
);
```

Oracle 11g introduit une variante qui se nomme, en anglais, *interval partitioning*. Elle permet de créer automatiquement les partitions au-delà de la dernière limite définie, suivant une fréquence établie sur un champ numérique ou date.

```
create table CMD_PART(
  NOCMD          NUMBER not null,
  NOCLIENT       NUMBER,
  DATECOMMANDE  DATE,
  ETATCOMMANDE  CHAR(1),
  constraint PK_CMD_PART primary key (NOCMD)
)
PARTITION BY RANGE(DATECOMMANDE)
  INTERVAL (NUMTOYMINTERVAL(1,'YEAR'))
(
  PARTITION CMD_2005_2007 VALUES LESS THAN(TO_DATE('01/01/2008','DD/MM/YYYY'))
    TABLESPACE TableSpace1,
  PARTITION CMD_2008 VALUES LESS THAN(TO_DATE('01/01/2009','DD/MM/YYYY'))
    TABLESPACE TableSpace2,
  PARTITION CMD_2009 VALUES LESS THAN(TO_DATE('01/01/2010','DD/MM/YYYY'))
    TABLESPACE TableSpace3
);
```

Partitionnement par hachage

Dans ce type de partitionnement, on définit seulement la liste des partitions et la colonne qui sert de paramètre pour calculer la clé de hachage. Cette solution est la plus simple à mettre en œuvre, mais probablement la moins pratique pour déterminer la manière d'optimiser l'usage des partitions.

```
create table CMD_PART_HASH(
  NOCMD          NUMBER not null,
  NOCLIENT       NUMBER,
  DATECOMMANDE  DATE,
  ETATCOMMANDE  CHAR(1),
  constraint PK_CMD_PART_HASH primary key (NOCMD)
)
PARTITION BY HASH(NOCMD)
  PARTITIONS 4
  STORE IN (Tablespace1, Tablespace2, Tablespace3, Tablespace4);
```

Partitionnement système

Introduit en version 11g, il permet de ne donner aucune règle de répartition des données entre les partitions à la création de la table mais de gérer cette répartition dans les instructions INSERT à l'aide du paramètre PARTITION. De la même façon, il sera possible de spécifier au niveau des instructions UPDATE, DELETE et SELECT sur quelle partition porte la requête. La syntaxe dans l'instruction CREATE TABLE est la suivante :

```

PARTITION BY SYSTEM (
  PARTITION Partition1,
  PARTITION Partition2 );

```

Les requêtes d'insertion sur ces tables doivent spécifier la partition utilisée à l'aide du mot clé **PARTITION**.

```

insert into TablePartitionnee partition (Partition1) values ( . . . );

```

Pour les requêtes **UPDATE**, **DELETE** et **SELECT**, cette directive est optionnelle. Son absence entraîne un parcours de toutes les partitions. Il peut y avoir incohérence entre la partition désignée et les autres prédicats, puisque ce point relève de la responsabilité du développeur.

```

delete TablePartitionnee partition (Partition1) where . . . ;
select * from TablePartitionnee partition (Partition1) where . . . ;

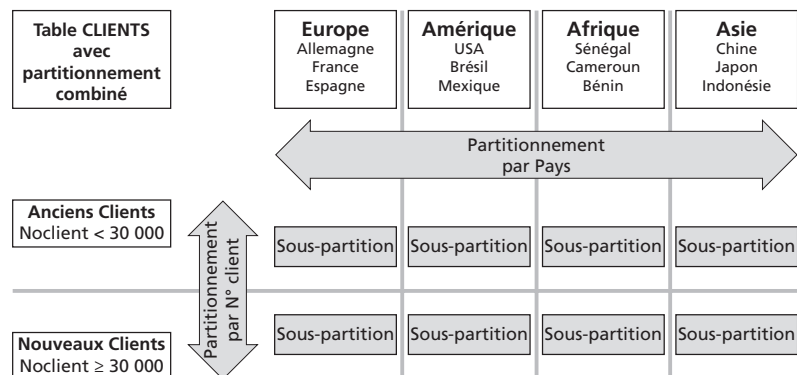
```

Partitionnement combiné

On peut combiner deux types de partitionnement afin de faire un partitionnement composite. Cela signifie que vous pourrez spécifier deux axes de partitionnement. Vous pouvez faire un sous-partitionnement régulier, c'est-à-dire découper toutes les partitions en sous-partitions identiques à l'aide de **SUBPARTITION TEMPLATE**, ou un sous-partitionnement libre, qui sera propre à chaque partition même si le champ permettant le sous-partitionnement est commun à toutes les sous-partitions. L'exemple suivant illustre les deux syntaxes possibles.

Jusqu'à la version 10g, les seules combinaisons possibles étaient partition par intervalle et sous-partition par Hash ou partition par intervalle et sous-partition par liste. La version 11g a introduit les possibilités suivantes : Intervalle/Intervalle, Liste/Intervalle, Liste/Hash, Liste/Liste.

Figure 5.25
Exemple de
partitionnement
combiné.



Exemple de création d'une table avec des sous-partitions régulières

```

create table CLIENTS_PART(
  NOCLIENT  NUMBER not null,
  NOM        VARCHAR2(50),
  PRENOM     VARCHAR2(50),
  ADRESSE1    VARCHAR2(100),
  ADRESSE2    VARCHAR2(100),
  CODEPOSTAL VARCHAR2(10),
  VILLE       VARCHAR2(50),
  PAYS        VARCHAR2(50),
  TEL         VARCHAR2(20),
  EMAIL       VARCHAR2(50),
  constraint PK_CLIENTS_PART primary key (NOCLIENT)
)
PARTITION BY LIST(PAYS)
  SUBPARTITION BY RANGE (NOCLIENT)
    SUBPARTITION TEMPLATE(
      SUBPARTITION Anciens VALUES LESS THAN (30000) TABLESPACE TableSpace1,
      SUBPARTITION Recents VALUES LESS THAN (MAXVALUE) TABLESPACE TableSpace2)
(
  PARTITION Clients_Amerique VALUES('USA', 'Canada','Brazil'),
  PARTITION Clients_Europe VALUES ('Germany', 'United Kingdom',
'France','Italy'),
  PARTITION Clients_Asie VALUES('Japan', 'China','India'),
  PARTITION Clients_Autres VALUES(DEFAULT)
);

```

Exemple de création d'une table avec des sous-partitions libres

```

create table CLIENTS_PART(
  NOCLIENT  NUMBER not null,
  NOM        VARCHAR2(50),
  PRENOM     VARCHAR2(50),
  ADRESSE1    VARCHAR2(100),
  ADRESSE2    VARCHAR2(100),
  CODEPOSTAL VARCHAR2(10),
  VILLE       VARCHAR2(50),
  PAYS        VARCHAR2(50),
  TEL         VARCHAR2(20),
  EMAIL       VARCHAR2(50),
  constraint PK_CLIENTS_PART primary key (NOCLIENT)
)
PARTITION BY LIST(PAYS)
  SUBPARTITION BY RANGE (NOCLIENT)
(
  PARTITION Clients_Amerique VALUES('USA', 'Canada','Brazil')
    (SUBPARTITION Clients_AM_Anciens VALUES LESS THAN (30000)
      TABLESPACE Tablespace1,
      SUBPARTITION Clients_AM_Recents VALUES LESS THAN (MAXVALUE)
      TABLESPACE Tablespace2),
  PARTITION Clients_Europe VALUES ('Germany', 'UK', 'France','Italy')
    (SUBPARTITION Clients_EU_Anciens VALUES LESS THAN (30000)
      TABLESPACE Tablespace3,

```

```

        SUBPARTITION Clients_EU_Recents VALUES LESS THAN (MAXVALUE)
        TABLESPACE Tablespace4),
PARTITION Clients_Asie VALUES('Japan', 'China','India')
(SUBPARTITION Clients_AS_Anciens VALUES LESS THAN (30000)
TABLESPACE Tablespace5,
SUBPARTITION Clients_AS_Recents VALUES LESS THAN (MAXVALUE)
TABLESPACE Tablespace6),
PARTITION Clients_Autres VALUES(DEFAULT)
(SUBPARTITION Clients_AU_Anciens VALUES LESS THAN (30000)
TABLESPACE Tablespace7,
SUBPARTITION Clients_AU_Recents VALUES LESS THAN (MAXVALUE)
TABLESPACE Tablespace8)
);

```

Partitionnement par référence

Oracle 11g Release 1 a introduit une nouvelle méthode assez intéressante. Elle permet de partitionner une table en fonction des valeurs contenues dans une autre table et ayant une relation de référence au moyen d'une clé étrangère (*Foreign Key*) avec elle. Ce type de partitionnement a été introduit pour gérer des relations de type maître/détails. Ci-après, un exemple entre la table des commandes et la table des lignes de commandes est proposé :

```

create table CMD_LIGNES_PART(
  NOCMD    NUMBER not null,
  NOLIVRE  NUMBER not null,
  QUANTITE NUMBER,
  REMISE   NUMBER,
  MONTANT  NUMBER,
  constraint PK_CMD_LIGNES_PART primary key (NOCMD, NOLIVRE),
  constraint FK_CMD_LIGNES_CMD_PART Foreign key (NOCMD) References CMD_
  PART(NOCMD)
)
PARTITION BY REFERENCE (FK_CMD_LIGNES_CMD_PART);

```

Partitionnement et index

Il existe trois types d'index travaillant avec les partitions :

- Les index normaux, qui permettent d'indexer des colonnes de façon indépendante des partitions de la table.
- Les index locaux préfixés, qui sont calqués sur les partitions et qui reprennent en premières colonnes les colonnes de la clé de partitionnement.
- Les index locaux non préfixés, qui sont calqués sur les partitions et qui ne reprennent pas les colonnes de la clé de partitionnement. Ces index-là ne peuvent pas être de type unique.

Les index normaux n'ont pas de spécificité de syntaxe, seuls les index locaux intègrent le mot clé LOCAL. Un index local sera dit "préfixé" si les premières colonnes sont celles de la clé de partitionnement. Voici un exemple d'index local préfixé :

```
create index IS_CLIENTS_PART_PAYSVILLE on CLIENTS_PART(pays,ville) local;
```

Les index locaux non préfixés prennent moins de place car ils ont l'avantage de ne pas répéter la clé de partitionnement, ce qui est intéressant pour des partitions monovaleurs. Sur les index locaux non préfixés, il faut parfois aider l'optimiseur avec un hint pour qu'il utilise l'index.

MS SQL Server

La gestion des partitions est proposée depuis SQL Server 2005, seul le partitionnement par intervalle est disponible. Il faut d'abord créer une fonction de partitionnement puis créer le schéma de partitionnement et enfin assigner le schéma de partitionnement à la table.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);

CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1
TO (test1fg, test2fg, test3fg, test4fg);

CREATE TABLE (. . . .) on myRangePS1(id);
```

MySQL

MySQL propose depuis la version 5.1 la gestion des partitionnements Hash, List et Range mais pas le partitionnement composite ou par référence.

Évaluation

Nous allons évaluer le partitionnement en utilisant la table CMD d'un million d'enregistrements partitionnée par intervalles sur la date de commande, comme montré précédemment. La requête sera la suivante :

```
select count(*) from cmd
where datecommande >=to_date('08/2008','MM/YYYY')
and datecommande <to_date('09/2008','MM/YYYY')
```

Figure 5.26

Trace d'exécution
sur table normale.

OPERATION	OBJECT_NAME	COST	CARDINALITY
SELECT STATEMENT		893	
SORT AGGREGATE			1
TABLE ACCESS FULL	CMD	893	15362
Prédicats de filtre			
AND			
DATECOMMANDE >= TO_DATE('2008-08-01 0			
DATECOMMANDE < TO_DATE('2008-09-01 00			

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	1
db block gets	0
consistent gets	3194
physical reads	0
redo size	0
bytes sent via SQL*Net to client	727
bytes received via SQL*Net from client	681
SQL*Net roundtrips to/from client	2
sorts (memory)	1
sorts (disk)	0

On constate ici que toute la table est parcourue.

Figure 5.27

Trace d'exécution
sur la table
partitionnée.

OPERATION	OBJECT_NAME	PARTITION_START	PARTITION_STOP	COST	CARDINALITY
SELECT STATEMENT				241	1
SORT AGGREGATE					1
PARTITION RANGE SINGLE		2	2	241	21050
TABLE ACCESS FULL	CMD_PART	2	2	241	21050
Prédicats de filtre					
AND					
DATECOMMANDE >= TO_DATE('2008-08-01					
DATECOMMANDE < TO_DATE('2008-09-01 0					

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	1
db block gets	0
consistent gets	821
physical reads	0
redo size	0
bytes sent via SQL*Net to client	727
bytes received via SQL*Net from client	686
SQL*Net roundtrips to/from client	2
sorts (memory)	1
sorts (disk)	0

On constate ici que seule la partition 2 est parcourue, d'où une chute des Consistent Gets et du temps d'exécution.

Partitionnement économique

Comme nous l'avons dit, le partitionnement n'est pas présent dans toutes les éditions des produits. Pour avoir cette option, il faut souvent acquérir les versions haut de gamme et ce n'est pas forcément compatible avec votre budget. Je vais vous présenter une solution alternative ci-après, orientée base décisionnelle. On peut certainement l'adapter à un usage OLTP, à l'aide d'un trigger INSTEAD OF, et simuler complètement le partitionnement, mais je ne vous le conseille pas.

Une solution plus économique consiste à créer autant de tables qu'on aurait créé de partitions et à adapter les requêtes à cette nouvelle organisation des données. Cela demandera sans doute pas mal de travail.

Une façon de réduire ce travail est d'agréger les tables dans une vue qui se nommera comme se nommait la table avant que vous ne la coupiez en morceaux. De cette façon, les requêtes marcheront toujours. Cependant, si on ne fait rien de plus, le résultat sera peu intéressant, si ce n'est de pouvoir profiter d'un peu de parallélisme. En fait, pour que cela présente un intérêt, nous allons appliquer des contraintes de type CHECK définissant des intervalles sur les colonnes qui servent de pseudo-clés de partitionnement. Nous allons donc créer les objets suivants :

```
create table CMD_2005_2007 (
  NOCMD          NUMBER not null,
  NOCLIENT       NUMBER,
  DATECOMMANDE   DATE constraint CMD_2005_2007_CKPART
                  check (DateCommande<TO_DATE('01/01/2008','DD/MM/YYYY')),
  ETATCOMMANDE   CHAR(1),
  constraint PK_CMD_2005_2007 primary key (NOCMD)
);

create table CMD_2008 (
  NOCMD          NUMBER not null,
  NOCLIENT       NUMBER,
  DATECOMMANDE   DATE constraint CMD_2008_CKPART
                  check (DateCommande>=TO_DATE('01/01/2008','DD/MM/YYYY')
                        and DateCommande<TO_DATE('01/01/2009','DD/MM/YYYY')),
  ETATCOMMANDE   CHAR(1),
  constraint PK_CMD_2008 primary key (NOCMD)
);

create table CMD_2009_MAX (
  NOCMD          NUMBER not null,
  NOCLIENT       NUMBER,
  DATECOMMANDE   DATE constraint CMD_2009_MAX_CKPART
                  check (DateCommande>=TO_DATE('01/01/2009','DD/MM/YYYY')),
  ETATCOMMANDE   CHAR(1),
  constraint PK_CMD_2009_MAX primary key (NOCMD)
);

create view CMD_PART as
  select * from CMD_2005_2007
  union all
  select * from CMD_2008
  union all
  select * from CMD_2009_MAX;
```

Puis exécuter la requête suivante :

```
select count(*) from cmd
where datecommande >=to_date('08/2008','MM/YYYY')
and datecommande <to_date('09/2008','MM/YYYY')
```

Figure 5.28

Trace d'exécution
d'une requête
sur une vue
Union All
de table avec des
contraintes.

0,204 secondes					
OPERATION	OBJECT_NAME	COST	CARDINALITY	LAST_OUTPUT_ROWS	LAST_CR_BUFFER_GETS
SELECT STATEMENT		929			
SORT AGGREGATE			1	1	821
VIEW	CMD_PART	929	15362	22083	821
UNION-ALL				22083	821
FILTER				0	0
Prédicats de filtre					
NULL IS NOT NULL					
TABLE ACCESS FULL	CMD_2005_2007	345	233	0	0
Prédicats de filtre					
AND					
DATECOMMANDE>=TO_DATE('2008-08-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
DATECOMMANDE<TO_DATE('2008-09-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
TABLE ACCESS FULL	CMD_2008	241	21850	22083	821
Prédicats de filtre					
AND					
DATECOMMANDE>=TO_DATE('2008-08-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
DATECOMMANDE<TO_DATE('2008-09-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
FILTER				0	0
Prédicats de filtre					
NULL IS NOT NULL					
TABLE ACCESS FULL	CMD_2009_MAX	345	685	0	0
Prédicats de filtre					
AND					
DATECOMMANDE<TO_DATE('2008-09-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
DATECOMMANDE>=TO_DATE('2008-08-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS')					
V\$STATNAME Name					
V\$MYSTAT Value					
recursive calls		29			
db block gets		0			
consistent gets		829			
physical reads		0			
redo size		0			
bytes sent via SQL*Net to client		727			
bytes received via SQL*Net from client		686			
SQL*Net roundtrips to/from client		2			
sorts (memory)		1			
sorts (disk)		0			

Des coûts sont assignés à l'interrogation des tables CMD_2005_2007 et CMD_2009_MAX, mais nous voyons dans la colonne Last_Output_Rows qu'aucune ligne n'est récupérée. Cela vient de l'ajout dans le plan d'exécution de prédicats faux NULL IS NOT NULL qui ont pour effet de "couper" ces branches d'exécution. Nous retrouvons donc en interrogation les mêmes performances qu'avec le partitionnement classique, l'insertion des données, elle, n'est pas gérée.

MS SQL Server

Il est possible d'utiliser la même astuce sous SQL Server, c'était d'ailleurs la solution recommandée avant la version 2005, qui intègre le partitionnement.

MySQL

Pour cette opération, MySQL propose le moteur MERGE sur les tables MyISAM pour ce genre d'astuce mais l'absence de contrainte CHECK empêchera d'utiliser la même astuce. Le moteur MERGE permet de spécifier la table dans laquelle il faut faire les insertions.

5.3.5 Les vues matérialisées

Les vues matérialisées (*Materialized Views*) sont des tables qui contiennent des données agrégées ou jointées issues d'autres tables. Elles sont définies par des requêtes, comme les vues, mais stockent les données comme les tables.

Le mécanisme de journalisation `MATERIALIZED VIEW LOG` permet de capturer les changements sur les données sources afin de faire des rafraîchissements incrémentaux (*Fast Refresh*).

Les vues matérialisées peuvent être rafraîchies en temps réel, lors des `COMMIT` (obligatoirement `FAST REFRESH`), ou manuellement, à la demande. Nous étudions cet objet en tant qu'objet d'optimisation, nous nous focaliserons donc sur le mode `REFRESH ON COMMIT`. Cependant, dans le cas de mises à jour des tables par lot (*batch*), un rafraîchissement manuel pourrait être plus adapté.

Mesurez bien les conséquences en espace disque, en ressources CPU et en temps de réponse sur les tables sources lors de l'utilisation de `REFRESH ON COMMIT`. En effet, chaque transaction a pour effet d'actualiser de façon synchrone les données impactées des vues matérialisées.

L'utilisation des *Materialized View* et du *Query Rewriting* (voir ci-après) nécessite d'avoir les privilèges suivants (même si on est DBA) :

```
GRANT GLOBAL QUERY REWRITE TO scott;  
GRANT CREATE MATERIALIZED VIEW TO scott;
```

Un exemple de mise en œuvre d'une vue matérialisée maintenue de façon incrémentale sur `COMMIT` est le suivant : nous commençons par créer les `MATERIALIZED VIEW LOG`. Ils permettent de capturer tous les changements sur les données sources afin de les répercuter de façon incrémentale sur la vue matérialisée sans avoir à la recalculer entièrement (il faut n'y mettre que les champs impliqués dans la vue). Ensuite, nous créons la vue matérialisée elle-même. `REFRESH FAST ON COMMIT` permet de préciser qu'elle sera maintenue de façon incrémentale à chaque `COMMIT` et `ENABLE QUERY REWRITE` de signaler que cette vue peut être utilisée dans le cadre du *Query Rewriting*. Puisque notre requête effectue des agrégats de données, il y a quelques contraintes, entre autres, il doit impérativement y avoir une colonne `count(*)` et `count(...)` de chaque colonne concernée par une somme ou moyenne.

```
-- Création de la journalisation sur les données sources  
CREATE MATERIALIZED VIEW LOG ON cmd WITH SEQUENCE  
  ,ROWID(nocmd,noclient,datecommande,etatcommande) INCLUDING NEW VALUES;  
CREATE MATERIALIZED VIEW LOG ON cmd_lignes WITH SEQUENCE  
  ,ROWID (NoCmd,Montant) INCLUDING NEW VALUES;
```

```
-- Création de la vue
CREATE MATERIALIZED VIEW MV_CMD
  PCTFREE 0
  BUILD IMMEDIATE
  REFRESH FAST on commit
  with rowid
  ENABLE QUERY REWRITE
AS select c.nocmd,c.noclient,c.datecommande,c.etatcommande
        ,sum(cl.montant) MontantCMD, COUNT(*) NbrItem
        ,count(Montant) NbrMontant
  from cmd c join cmd_lignes cl on c.nocmd=cl.nocmd
  group by c.nocmd,c.noclient,c.datecommande,c.etatcommande;

-- Création d'un index sur la vue
create index is_MV_CMD_NoCmd on MV_CMD (NoCMD);
```

Une fois la vue matérialisée créée, nous pouvons l'utiliser plutôt que les données sources dans nos requêtes. Elle sera mise à jour automatiquement à chaque COMMIT.

Query Rewriting

Ce processus permet à l'optimiseur de réécrire de façon transparente des requêtes écrites sur les données sources d'une vue matérialisée pour utiliser celle-ci à la place, si cela est plus performant. Si la vue matérialisée est maintenue manuellement, toute modification sur une des tables sources invalidera le fonctionnement du Query Rewriting jusqu'à ce que la vue matérialisée soit rafraîchie. Par exemple, la requête suivante :

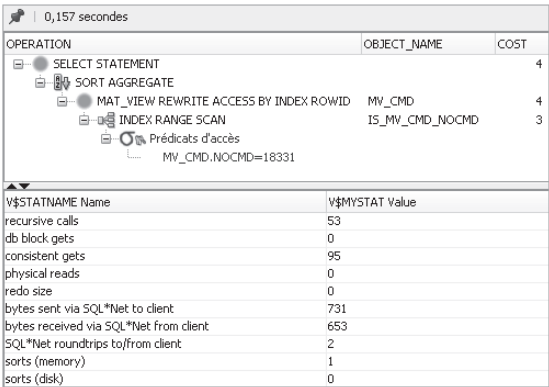
```
select sum(cl.montant)
  from cmd c join cmd_lignes cl on c.nocmd=cl.nocmd
 where c.NoCmd=18331
```

sera transformée de façon transparente par le Query Rewriting en :

```
select MontantCMD from MV_CMD where NoCmd=18331 ;
```

Comme on peut le voir sur ce plan d'exécution, l'optimiseur a décidé d'utiliser les données agrégées dans la vue matérialisée pour répondre à la requête qui portait sur les données non agrégées. On constate qu'il n'y a plus d'accès aux tables mentionnées dans la requête, l'utilisation de la vue matérialisée évite donc de faire la jointure. Si nous avions demandé `AVG(Montant)`, la requête aurait aussi été réécrite car `avg(Montant) = sum(Montant) / count(Montant)` qui utilise deux opérandes présents dans la vue matérialisée.

Figure 5.29
*Trace d'exécution d'une
requête réécrite par
l'optimiseur.*



MS SQL Server

Sous SQL Server, la même fonctionnalité existe sous le nom de "vue indexée". Les noms des tables doivent impérativement être préfixés par le schéma qui les contient. Les agrégats ne doivent pas manipuler de valeurs nulles (d'où l'utilisation de ISNULL) et doivent contenir un COUNT_BIG(*). Il faut ensuite indexer cette vue pour la matérialiser.

```
create view MV_CMD WITH SCHEMABINDING AS
select c.nocmd,c.noclient,c.datecommande,c.etatcommande
      ,sum(isnull(cl.montant,0)) MontantCMD
      ,COUNT_BIG(*) NbrItem
from dbo.cmd c join dbo.cmd_lignes cl on c.nocmd=cl.nocmd
group by c.nocmd,c.noclient,c.datecommande,c.etatcommande;
CREATE UNIQUE CLUSTERED INDEX IDX_MV_CMD ON dbo.MV_CMD(NoCmd);
```

MySQL

Il n'y a pas, sous MySQL, de fonctionnalité équivalente.

5.3.6 Reconstruction des index et des tables

Les mouvements liés au LMD (insert, update, delete) peuvent entraîner une fragmentation des segments et ainsi aboutir à des tables et des index "gruyères". Ce phénomène aura pour effet d'augmenter inutilement le nombre de blocs à manipuler et donc de réduire les performances. Pour les tables qui subissent beaucoup d'ajouts et de suppressions, ces opérations peuvent être intéressantes. Pour les autres, elles

seront généralement inutiles car les mécanismes de base de réutilisation des espaces libérés seront suffisamment efficaces.

Les instructions de reconstruction permettent, avec certaines options (MOVE et REBUILD), de spécifier de nouvelles clauses de stockage et même de changer de tablespace. Les reconstructions sont utiles aussi pour supprimer les effets des migrations de lignes (voir Chapitre 1, section 1.2.5, "Row Migration et Row Chaining").

Pour réorganiser les tables organisées en tas, il faut avant tout autoriser les mouvements de lignes :

```
Alter table Clients Enable Row Movement;
```

Le déplacement d'une table permet de la compacter en faisant une copie. Cette option est assez efficace mais, pendant l'opération de copie, l'espace requis est le double de l'espace initial. Cette option permet aussi de spécifier de nouveaux paramètres de stockage :

```
Alter table <NomTable> move;
```

ATTENTION

Cette instruction a pour effet particulièrement significatif d'invalider tous les index. Il faudra donc les reconstruire, ce qui peut avoir un coût certain. Le script PL/SQL ci-après permet de reconstruire tous les index invalidés :

```
begin
for r in (SELECT 'alter index '||index_name||' rebuild' cmd
          from user_indexes where status='UNUSABLE')
loop
  execute immediate r.cmd;
end loop;
end;
```

Une variante est l'instruction SHRINK TABLE qui ne marche que sur les segments situés dans des tablespaces en mode ASM (*Automatic Storage Management*) qui n'utilisent pas d'index sur fonction :

```
Alter table clients shrink space;
```

Pour réorganiser les index, deux options sont disponibles :

- **REBUILD.** C'est la reconstruction, opération efficace mais particulièrement coûteuse :

```
Alter index <NomIndex> rebuild ;
```

- **COALESCE.** Solution plus légère, elle fusionne les feuilles d'index consécutives, qui ont plus de 25 % d'espace libre sans libérer l'espace des feuilles :

```
Alter index <NomIndex> coalesce ;
```

Impact sur la volumétrie et les performances

Afin d'évaluer l'impact des reconstructions, nous allons effectuer quelques opérations sur la table **CLIENTS**.

Taille initiale. Table : 5 120 Ko ; index clé primaire : 768 Ko ; **IS_CLIENTS_PAYS** : 1 024 Ko.

1. Nous dupliquons la table sur elle-même afin de doubler sa taille.

Taille après doublement. Table : 10 240 Ko ; index clé primaire : 2 048 Ko ; **IS_CLIENTS_PAYS** : 3 072 Ko.

2. Nous effaçons la moitié des enregistrements.

Taille après effacement. Table : 10 240 Ko ; index clé primaire : 2 048 Ko ; **IS_CLIENTS_PAYS** : 3 072 Ko.

Comme on le voit, l'espace n'est pas libéré.

3. Nous effectuons un **SHRINK** sur la table et un **REBUILD** des index :

```
alter table CLIENTS shrink space;  
alter index PK_CLIENTS rebuild;  
alter index IS_CLIENTS_PAYS rebuild;
```

Taille après SHRINK et REBUILD. Table : 5 120 Ko ; index clé primaire : 832 Ko ; **IS_CLIENTS_PAYS** : 1 024 Ko.

Nous revenons à la taille initiale.

En termes d'espace occupé, l'impact est significatif. Ces opérations de maintenance peuvent être importantes si des opérations volumineuses ont eu lieu sur les tables (ce qui n'arrive normalement pas tous les jours dans la plupart des applications).

Voyons à présent l'impact de la défragmentation sur les performances.

Table full scan.

Avant : Consistent Gets : 26 302 ; timing : 2,05 s

Après : Consistent Gets : 8 608 ; timing : 1,33 s

Index full scan.

Avant : Consistent Gets : 532 ; timing : 0,033 s

Après : Consistent Gets : 227 ; timing : 0,032 s

Index direct par recherche d'une valeur unique.

Avant : Consistent Gets : 2 ; timing : 0,005 s

Après : Consistent Gets : 2 ; timing : 0,002 s

L'impact le plus significatif est sur le Table Full Scan qui lit tous les blocs de la table. On remarque que l'index, lui, s'accommode plutôt bien de l'effet gruyère.

Techniques d'optimisation standard des requêtes

L'optimisation du SQL est un point très délicat car elle nécessite de pouvoir modifier l'applicatif en veillant à ne pas introduire de bogues.

6.1 Réécriture des requêtes

L'utilisation d'objets d'optimisation est un élément qui, de manière générale, améliore les performances très significativement. Cependant, la façon d'écrire votre requête peut, elle aussi, dans certains cas, avoir un impact très significatif.

L'étape de transformation de requête (voir ci-après) est de plus en plus performante sur Oracle et SQL Server. Elle donne de très bons résultats, ce qui rend inutiles certaines recommandations dans le cas général. Cependant, sur MySQL ou sur des requêtes compliquées, où on peut considérer que le SGBDR n'arriverait pas à faire certaines transformations tout seul, l'utilisation des écritures les plus performantes permettra d'améliorer les résultats.

Nous allons faire quelques comparaisons d'écritures qui intégreront parfois un hint (voir Chapitre 7, section 7.1, "Utilisation des hints sous Oracle") afin d'empêcher certaines transformations.

De façon générale, il faut privilégier les jointures. Le principe du modèle relationnel étant que, lors de l'interrogation, des jointures seront faites, on peut supposer que les éditeurs de SGBDR ont concentré leurs efforts là-dessus. Cependant, sous MySQL avec le moteur MyISAM, les sous-requêtes sont souvent plus performantes.

6.1.1 Transformation de requêtes

L'étape de transformation de requêtes (*Query Transformation*) est partie intégrante du processus de traitement des requêtes. C'est une des fonctions de l'optimiseur CBO.

Cette étape consiste à transformer la requête soumise en une requête équivalente afin de la rendre plus performante. Cela va, au-delà, de choisir le meilleur chemin d'exécution. La transformation de requêtes peut, par exemple, décider de transformer une sous-requête en jointure. Vous pouvez influencer sur cette étape au moyen des hints, mais c'est rarement nécessaire.

Les principales transformations effectuées par le CBO sont :

- **Subquery unesting.** Cette transformation consiste à transformer des sous-requêtes en jointures.
- **Suppression d'éléments inutiles** (certaines jointures, des colonnes sélectionnées dans les sous-requêtes).
- **Predicate push.** Cette transformation consiste à dupliquer certains prédicats dans les sous-vues et les sous-requêtes.
- **View merging.** Intègre l'exécution des vues à la requête principale.
- **Query Rewriting.** Utilisation des vues matérialisées (voir Chapitre 5, section 5.3.5, "Les vues matérialisées").
- **Or Expansion.** Transforme des conditions OR en plusieurs requêtes fusionnées par une sorte d'"Union ALL".

Il n'y a rien de particulier à faire pour bénéficier du Query Rewriting. Connaître l'existence de ce mécanisme vous aidera à comprendre pourquoi il arrive parfois qu'un plan d'exécution ne ressemble vraiment pas à votre requête. Cette fonction est plus évoluée sous Oracle et SQL Server que sous MySQL.

INFO

Au cours de ce chapitre, nous présentons des variations de notation qui peuvent avoir un effet sur les performances. L'optimiseur étant assez performant, il transforme automatiquement les requêtes soumises en la version la plus performante. Nous recourons à des hints pour empêcher ces transformations afin de comparer les notations. L'utilisation de hint n'est nullement une recommandation, nous le mentionnons à titre indicatif de façon que, si vous exécutez ces requêtes, vous puissiez reproduire les mêmes résultats que ceux présentés ici.

Comme cela sera expliqué au Chapitre 7, section 7.1, "Utilisation des hints sous Oracle", il faut les utiliser en dernier recours et seulement en toute connaissance de cause.

6.1.2 IN versus jointure

Selon que l'écriture d'une requête se fasse avec une sous-requête et l'opérateur IN ou avec une jointure, l'impact sera différent. Pour forcer le parcours des tables, et non seulement celui des index, nous utilisons dans la requête des champs non indexés (Quantité et Editeur).

Listing 6.1 : Requête utilisant une jointure

```
select sum(CL.quantite) from cmd_lignes CL, livres L
where CL.nolivre = L.nolivre and L.editeur='Pearson'
```

Listing 6.2 : Requête utilisant une sous-requête

```
select sum(quantite) from cmd_lignes
where nolivre in (select nolivre from livres where editeur='Pearson')
```

Nous allons tester ces requêtes avec et sans index sur la colonne Nolivre de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

	<i>Sans index</i>		<i>Avec index</i>	
<i>MyISAM</i>	<i>Jointure</i>	<i>Sous-requête</i>	<i>Jointure</i>	<i>Sous-requête</i>
Temps	12,89 s	13,53 s	0,01 s	13,53 s
Key_read_requests	5 242 448	5 243 442	158	5 243 442
<i>InnoDB</i>			<i>Jointure</i>	<i>Sous-requête</i>
Temps			3,48 s	13,51 s
Reads			3 700	2 606 480

InnoDB créant automatiquement un index dans la table fille de la *Foreign Key*, il y a forcément un index sur la colonne Nolivre de la table CMD_LIGNES. La création de l'index est donc sans effets.

	<i>Sans index</i>		<i>Avec index</i>	
<i>Oracle</i>	<i>Jointure</i>	<i>Sous-requête</i>	<i>Jointure</i>	<i>Sous-requête</i>
Temps	6,15 s	8,75 s	0,01 s	0,60 s
Consistent Gets	8 336	4 891 735	807	12 175

Pour éviter la transformation de la version utilisant une sous-requête en version jointure par le mécanisme de transformation de requête, nous devons utiliser le hint `/*+no_unnest*/` et pour forcer l’usage de l’index, le hint `/*+ index (CL IS_cmd_lignes)*/`. Si nous ne mettons aucun hint, l’optimiseur choisit systématiquement de faire une jointure sans utiliser l’index.

	Sans index	Avec index
MS SQL Server	Jointure	Sous-requête
Temps	0,625 s	0,046 s
Estimated Cost	35,27	60,44

Afin de comparer les résultats entre les bases, nous désactivons le parallélisme sous SQL Server au moyen du code option `(MAXDOP 1)`.

Nous n’avons pas trouvé de combinaisons de hints permettant de forcer l’utilisation d’une sous-requête tant qu’il n’y a pas d’index. Une fois l’index placé, nous avons dû contraindre son utilisation avec le hint `WITH (INDEX(IS_CMD_LIGNES))`. En revanche, nous n’avons pas réussi à forcer une jointure pertinente utilisant cet index.

Conclusion : Oracle et SQL Server, grâce à leur capacité de transformation de requête évoluée, exécutent les requêtes de la même façon quelle que soit la notation employée, si aucun hint n’est utilisé.

Lorsque la transformation n’est pas effectuée, on constate que la solution à base de jointure est généralement plus performante.

Au cas où le SGBDR n’arriverait pas à faire la transformation, il est préférable de prendre l’habitude d’écrire des jointures plutôt que des sous-requêtes.

6.1.3 Sous-requêtes versus anti-jointures

Une anti-jointure est une jointure ouverte pour laquelle vous ne sélectionnez que les valeurs non jointes.

Les requêtes ci-dessous, permettent de sélectionner les livres qui n’ont jamais été commandés.

Listing 6.3 : Version Exists

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Listing 6.4 : Version anti-jointure

```
select L.* from livres L
left outer join cmd_lignes CL on CL.NOLIVRE=L.NOLIVRE
where cl.nolivres is null
```

Nous allons tester ces requêtes avec et sans index sur la colonne Nolivres de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivres);
```

MyISAM	Sans index		Avec index	
	Anti-jointure	Sous-requête	Anti-jointure	Sous-requête
Temps	10,11 s	10,11 s	0,02 s	0,02 s
Key_read_requests	29 850 364	29 850 364	2 974	2 974

L'anti-jointure est transformée en sous-requête.

InnoDB	Anti-jointure	Sous-requête
Temps	24,31 s	25,34 s
Reads	2 976	5 948

InnoDB créant automatiquement un index dans la table fille de la *Foreign Key*, il y a forcément un index sur la colonne Nolivres de la table CMD_LIGNES. La création de l'index est donc sans effets.

Oracle	Sans index		Avec index	
	Anti-jointure	Sous-requête	Anti-jointure	Sous-requête
Temps	0,340 s	2,282 s	0,375 s	0,031 s
Consistent Gets	6 976	113 419	5 522	8 909

Nous utilisons le hint `/*+NO_QUERY_TRANSFORMATION*/` pour empêcher la transformation de la sous-requête en anti-jointure.

MS SQL Server	Sans index	Avec index
	Anti-jointure	Anti-jointure
Temps	2,437 s	0,079 s
Estimated Cost	47,49	0,54

Quelle que soit la notation utilisée, la requête est transformée en anti-jointure.

Les résultats sont mitigés sous Oracle. Malheureusement, l'étape de transformation de requête n'opte pas toujours pour la meilleure option. L'optimiseur choisit systématiquement l'anti-jointure qui est, en effet, généralement la meilleure option. Ici, la table LIVRE est bien plus petite que la table CMD_LIGNES, dans ce cas, si un index est présent sur la table fille de la *Nested Loop*, la solution utilisant une sous-requête est la meilleure. Il faut donc utiliser un hint pour forcer ce choix. Sur les autres SGBDR, l'étape de transformation ne laisse pas le choix, les deux écritures se valent.

6.1.4 *Exists* versus *Count*

Les requêtes effectuant un test d'existence sont équivalentes à un comptage égal à 0.

Listing 6.5 : Version *Exists*

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Listing 6.6 : Version *0=count(*)*

```
select * from livres L
where 0 = (select count(*) from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Nous allons tester ces requêtes avec un index sur la colonne Nolive de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolive);
```

<i>MyISAM</i>	<i>Exists</i>	<i>Count</i>
Temps	0,020 s	1,110 s
Key_read_requests	11 915	166 782
<i>InnoDB</i>	<i>Exists</i>	<i>Count</i>
Temps	22,940 s	29,590 s
Reads	8 900	2 606 477
<i>Oracle</i>	<i>Exists</i>	<i>Count</i>
Temps	0,031 s	0,031 s
Consistent Gets	8 909	8 909

<i>MS SQL Server</i>	<i>Exists</i>	<i>Count</i>
Temps	0,062 s	0,062 s

Oracle et SQL Server considèrent ces deux écritures équivalentes, alors que MySQL les voit bien différentes. On suppose que dans la version utilisant `COUNT(*)`, il dénombre toutes les occurrences pour tester l'égalité à 0, alors que la version utilisant `EXISTS` s'arrête à la première occurrence trouvée.

6.1.5 *Exists versus IN*

Une sous-requête utilisant l'opérateur `IN` peut facilement être convertie en sous-requête utilisant l'opérateur `EXISTS`.

Nous allons tester ces requêtes avec un index sur la colonne `Nolivre` de la table `CMD_LIGNES`.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

Listing 6.7 : Version sous-requête *IN*

```
select sum(quantite) from cmd_lignes
where nolivre in (select nolivre from livres where editeur='Pearson')
```

Listing 6.8 : Version sous-requête *Exists*

```
select sum(quantite) from cmd_lignes cl
where exists (select nolivre from livres L
             where editeur='Pearson' and CL.nolivre=L.nolivre )
```

<i>MyISAM</i>	<i>In</i>	<i>Exists</i>	
Temps	14,010 s	15,340 s	
Key_read_requests	5 243 442	5 242 448	
<i>InnoDB</i>	<i>In</i>	<i>Exists</i>	
Temps	7,420 s	8,200 s	
Reads	2 606 480	5 212 684	
<i>Oracle</i>	<i>In</i>	<i>Exists</i>	<i>Forçage index</i>
Temps	8,594 s	8,594 s	4,234 s
Consistent Gets	4 891 735	4 891 735	2 462 067

L’optimiseur Oracle décide de ne pas utiliser l’index. Si nous forçons son usage à l’aide du hint `/*+ index(c1 is_cmd_lignes)*`, nous notons une amélioration des performances.

Nous utilisons le hint `/*+NO_QUERY_TRANSFORMATION*` pour empêcher la transformation des sous-requêtes en anti-jointures qui ont un temps de réponse de l’ordre 0,34 seconde.

<i>MS SQL Server</i>	<i>In ou Exists</i>	<i>Forçage index</i>
Temps	0,625 s	0,046 s

Comme sous Oracle, nous devons forcer l’utilisation de l’index avec le hint `WITH (INDEX(IS_CMD_LIGNES))`.

Oracle et SQL Server considèrent ces deux écritures équivalentes, alors que MySQL les voit différentes, l’avantage étant à l’opérateur `IN` sur les deux moteurs de MySQL.

6.1.6 Clause *Exists* * *versus* constante

On enseigne, depuis des années, que lorsqu’on emploie des sous-requêtes corrélées utilisant l’opérateur `EXISTS`, il faut retourner une constante (numérique ou texte) dans la clause `SELECT` de la sous-requête. Est-ce seulement pour des raisons de clarté ou aussi de performances ?

Listing 6.9 : Version *Exists* constante

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE);
```

Listing 6.10 : Version *Exists* *

```
select * from livres L
where not exists (select * from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE);
```

Nous ne documentons pas le détail des résultats. On constate que les colonnes retournées dans la sous-requête n’ont aucun impact sur les performances, alors qu’elles en avaient sur d’anciennes versions d’Oracle.

Cette pratique de mettre une constante garde cependant tout son sens du point de vue de la compréhension. Parfois, certains développeurs mettent ici un champ particulier laissant ainsi croire qu’il y aurait une sorte de lien possible avec ce champ.

6.1.7 Expressions sous requêtes

Listing 6.11 : Requête utilisant une jointure

```
select c.nocmd,datecommande ,sum(montant) Total
from cmd c,cmd_lignes cl where cl.nocmd=c.nocmd and datecommande<to_
date('2004-04-18','yyyy-mm-dd')
group by c.nocmd, datecommande
```

Listing 6.12 : Requête utilisant une expression sous-requête

```
select nocmd,datecommande
,(select sum(montant) from cmd_lignes where nocmd=c.nocmd) Total
from cmd c where datecommande<to_date('2004-04-18','yyyy-mm-dd')
```

Nous choisissons ici une requête qui empêche d'appliquer le prédicat de la table CMD directement à la table CMD_LIGNES et qui ne retourne que 173 lignes. En changeant la date, nous testons une autre version qui ramène beaucoup plus de lignes (132 141).

<i>Petite requête</i>			<i>Grosse requête</i>	
<i>MyISAM</i>	<i>Jointure</i>	<i>Expression sous-requête</i>	<i>Jointure</i>	<i>Expression sous-requête</i>
Temps	5,890 s	0,160 s	6,810 s	1,950 s
Reads	4 873 397	736	4 873 397	564 513
<i>Petite requête</i>			<i>Grosse requête</i>	
<i>InnoDB</i>	<i>Jointure</i>	<i>Expression sous-requête</i>	<i>Jointure</i>	<i>Expression sous-requête</i>
Temps	0,670 s	0,670 s	3,580 s	4,030 s
Reads	1 001 449	1 000 175	2 264 285	1 632 143
<i>Petite requête</i>			<i>Grosse requête</i>	
<i>Oracle</i>	<i>Jointure</i>	<i>Expression sous-requête</i>	<i>Jointure</i>	<i>Expression sous-requête</i>
Temps	0,390 s	0,060 s	1,48 s	1,51 s
Consistent Gets	11 490	3 316	11 490	117 464
<i>Petite requête</i>			<i>Grosse requête</i>	
<i>MS SQL Server</i>	<i>Jointure</i>	<i>Expression sous-requête</i>	<i>Jointure</i>	<i>Expression sous-requête</i>
Temps	0,125 s	0,109 s	0,812 s	0,953 s

Avec des requêtes ramenant peu de lignes, l'avantage est aux expressions sous-requêtes. Avec des requêtes ramenant beaucoup de lignes, l'avantage passe à la version jointure, sauf pour le moteur MyISAM qui a l'air plus à l'aise avec l'utilisation systématique d'expressions sous-requêtes. Seul l'optimiseur de SQL Server adapte automatiquement son plan d'exécution à la solution la plus efficace, indépendamment de l'écriture. Nous avons donc utilisé les hints `loop join` et `merge join` pour effectuer ces tests.

6.1.8 Agrégats : *Having* versus *Where*

Quand c'est possible, les conditions doivent être placées dans la clause `WHERE` plutôt que dans la clause `HAVING` qui est évaluée après les opérations d'agrégation. Cela permet de profiter d'éventuels index et réduit le volume à traiter durant l'opération d'agrégation.

Listing 6.13 : Requête utilisant *Having* (incorrecte)

```
select pays,ville,count(*) from clients t
group by pays,ville
having pays='France'
```

Listing 6.14 : Requête utilisant *Where*

```
select pays,ville,count(*) from clients t
where t.pays='France'
group by pays,ville
```

<i>MyISAM</i>	<i>Having</i>	<i>Where</i>
Temps	0,080 s	0,030 s
Reads	91 109	45 950
<i>InnoDB</i>	<i>Having</i>	<i>Where</i>
Temps	0,090 s	0,050 s
Reads	91 132	45 960
<i>Oracle</i>	<i>Having</i>	<i>Where</i>
Temps	0,047 s	0,015 s
Consistent Gets	624	624
<i>MS SQL Server</i>	<i>Having</i>	<i>Where</i>
Temps	0,046 s	0,046 s

Dans le plan d'exécution de SQL Server, on voit qu'il déplace le prédicat de la clause `HAVING` à la clause `WHERE`, il n'y a donc pas d'impact lié à l'écriture. Par contre, sur tous les autres SGBDR, l'impact est notable et d'autant plus que la sélectivité du prédicat est forte.

6.2 Bonnes et mauvaises pratiques

6.2.1 Mélange des types

Le mélange des types peut causer des grosses chutes de performances en disqualifiant des index à cause de conversions (*cast*) nécessaires.

Dans la pratique, c'est rarement une source de problème de performances mais plutôt une source de problème d'exécution (message d'erreur) car le SGBDR fait les conversions adéquates. Cependant, une bonne pratique est de veiller à faire explicitement les conversions adaptées.

6.2.2 Fonctions et expressions sur index

L'utilisation de fonctions (y compris de conversion) ou d'expressions sur des colonnes indexées a pour effet de disqualifier les index.

```
select * from cmd_lignes  
where 'C' || nocmd='C114826'
```

Cette requête entraîne un parcours complet de la table alors qu'il serait pertinent d'utiliser l'index sur la colonne `Nocmd`. S'il n'est pas possible de se passer de l'expression `'C' + nocmd`, les index sur fonction (voir Chapitre 5, section 5.2.2, "Index sur fonction") peuvent apporter une solution à ce type de problème. Ci-après, les deux options possibles qui permettent à cette requête de tirer profit de l'index sur le champ `Nocmd` :

```
select * from cmd_lignes where nocmd=114826
```

ou :

```
Create Index IS_CMD_LIGNE_CNOCMD on CMD_LIGNES('C' || nocmd);
```


6.2.3 Impact de l'opérateur <> sur les index

L'opérateur <> et ses équivalents != et ^= ont un impact notable sur les index car il ne les utilise pas. Illustrons ce comportement sur la requête suivante qui recherche la date de la plus ancienne commande qui n'est pas clôturée (Etatcommande <> 'C') :

```
select min(datecommande) from cmd where etatcommande <> 'C'
```

Nous avons préalablement créé un index et forcé le calcul d'un histogramme sur la colonne Etatcommande :

```
create index is_cmd_etat on cmd(etatcommande);
execute dbms_stats.gather_table_stats(user,'cmd',cascade => true
,estimate_percent =>100,method_opt => 'for all columns size 100');
```

La distribution de la colonne Etatcommande est la suivante :

<i>Etatcommande</i>	<i>Nombre d'enregistrements</i>
A	240
C	994 700
I	5 060

Cette requête traite moins de 1 % des lignes et profiterait donc d'un index placé sur la colonne Etatcommande, mais la présence de l'opérateur <> l'empêche. Une solution permettant de contourner ce problème est de transformer cette requête. Cela se fait soit par des égalités :

```
select min(datecommande) from cmd where etatcommande in ('A','I')
```

soit par des intervalles excluant la valeur :

```
select min(datecommande) from cmd where etatcommande >'C' or etatcommande <'C'
```

Ces deux solutions utilisent l'index présent sur la colonne Etatcommande et les résultats sont sans appel. Le temps d'exécution passe de 125 ms à 15 ms et les Consistent Gets passent de 3 194 à 58.

6.2.4 Réutilisation de vue

Parfois, une vue existante peut être une bonne base pour faire une requête. Dans ce cas, il faut vérifier que cette vue ne fait pas trop de travail inutile pour le besoin de la requête que vous souhaitez écrire. Assurez-vous qu'il n'y a pas plus de tables en jeu que celles que vous auriez utilisées si vous n'aviez pas choisi la vue. Par exemple,

vous souhaitez calculer le chiffre d'affaires journalier et vous disposez de la vue suivante :

```
create view V_CA_JOURNALIER_PAYS as
Select cmd.datecommande,cli.pays, sum(cl.montant) CA
from cmd join cmd_lignes cl on cmd.nocmd=cl.nocmd
join clients cli on cmd.noclient=cli.noclient
group by cmd.datecommande,cli.pays
```

Vous trouvez donc que la solution la plus simple est de partir de cette vue en écrivant la requête suivante :

```
select datecommande,sum(CA)
from V_CA_JOURNALIER_PAYS
group by datecommande
```

Cette solution est certes pratique mais elle a l'inconvénient de faire une jointure avec la table CLIENTS qui n'est pas du tout nécessaire pour répondre à votre requête. Le coût de votre requête passe ainsi de 7 290 à 20 136.

6.2.5 Utilisation de tables temporaires

Pour certaines requêtes compliquées (mettant en œuvre de nombreuses tables), il peut être pertinent de travailler avec des tables temporaires. Ce choix est intéressant si le contenu de ces tables est interrogé plusieurs fois afin de compenser le coût de leur création. Ces tables pourront être indexées, de préférence après l'insertion des données.

Sous Oracle, vous pouvez spécifier de ne pas tracer les opérations dans ce genre de tables en utilisant la clause NOLOGGING lors de leur création.

```
Create table xxx ( . . . ) NOLOGGING;
```

6.2.6 Utilisation abusive de **SELECT ***

L'utilisation de l'opérateur * dans la clause SELECT ramène tous les champs. Certains développeurs trouvent cela particulièrement pratique mais ne mesurent pas forcément les conséquences que cela peut avoir.

Si vous avez besoin de ramener tous les champs de la table, il n'y a rien à dire, `Select * From LaTable` est tout à fait correct. Cependant, il y a de nombreux cas où * est utilisé alors que seuls certains champs sont nécessaires. Dans ce cas, on subit inutilement :

- une augmentation du volume transféré entre le client et le serveur ;

- une augmentation du volume mémoire nécessaire par le client et le serveur ;
- une augmentation des E/S disque du serveur s'il y a des champs stockés dans des segments séparés (LOB, Text, etc.).

Prenez l'habitude de ne ramener que les données dont vous avez besoin en listant les champs dans la clause `SELECT`. Il est fréquent de trouver dans des standards de codage l'interdiction de l'opérateur `*`. La raison est que, si au moment du codage on a besoin de tous les champs, ce ne sera peut-être plus le cas si un champ est ajouté en phase de maintenance.

6.2.7 Suppression des tris inutiles

Les tris sont coûteux, il faut donc s'assurer qu'ils sont nécessaires. Dans la requête ci-après, sous Oracle, l'ajout du tri par Nom fait passer le coût de 171 à 1 179 et double le temps d'exécution.

```
select * from clients order by nom
```

Si le mot clé `DISTINCT` est présent, les conséquences sont similaires car il nécessite un tri.

6.2.8 Utilisation raisonnée des opérations ensemblistes

Les opérations sur les ensembles (`UNION`, `MINUS`, `INTERSECT`) sont assez coûteuses et ne sont jamais transformées automatiquement par l'optimiseur. Il faut y recourir seulement quand c'est la solution la plus adaptée. Les opérations `INTERSECT` peuvent souvent être remplacées par une jointure et les opérations `MINUS`, par une anti-jointure ou un `NOT IN`. La suppression des opérations ensemblistes permettra d'éliminer des tris intermédiaires et d'avoir un plan d'exécution plus efficace. Les opérations de type `UNION` sont souvent plus délicates à supprimer. Les transformations de l'opération `INTERSECT` sont illustrées à travers la requête suivante qui recherche les clients qui ont acheté le livre 1306 et qui ont passé moins de 50 commandes.

Listing 6.15 : Version utilisant *INTERSECT*

```
select distinct c.noclient,c.nom,c.prenom,c.pays
from   cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
join   clients c on cmd.noclient=c.noclient
where  cl.nolivre=1306
intersect
```

```
select c.noclient,c.nom,c.prenom,c.pays
from cmd join clients c on cmd.noclient=c.noclient
group by c.noclient,c.nom,c.prenom,c.pays
having count(*)<50
```

Temps d'exécution : 0,703 s, Consistent Gets : 6 721, Cost : 10 312

Listing 6.16 : Version utilisant une jointure

```
select A.*
from (select distinct c.noclient,c.nom,c.prenom,c.pays
      from cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
      join clients c on cmd.noclient=c.noclient
      where cl.nolivre=1306 ) A
, (select c.noclient,c.nom,c.prenom,c.pays
    from cmd join clients c on cmd.noclient=c.noclient
    group by c.noclient,c.nom,c.prenom,c.pays
    having count(*)<50 ) B
where a.noclient=b.noclient
```

Temps d'exécution : 0,703 s, Consistent Gets : 6 721, Cost : 6 945

Suite à cette transformation, le gain n'est pas forcément significatif, mais le fait d'utiliser une jointure permet de factoriser les jointures avec la table CLIENTS présente dans chacune des sous-requêtes (voir Listing 6.17). Cette factorisation peut améliorer les performances surtout si les ensembles retournés par les sous-requêtes sont gros.

Listing 6.17 : Version factorisant la jointure de la table clients

```
select c.noclient,c.nom,c.prenom,c.pays
from (select distinct cmd.noclient
      from cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
      where cl.nolivre=1306 ) A
, (select noclient
    from cmd
    group by noclient
    having count(*)<50 ) B
, clients c
where a.noclient=b.noclient
and a.noclient=c.noclient
```

Temps d'exécution : 0,531 s, Consistent Gets : 7 050, Cost : 2 828

Illustrons une transformation de l'opération MINUS sur la requête suivante qui recherche les clients qui n'ont pas acheté le livre 3307 et qui ont passé plus de 60 commandes.

Listing 6.18 : Version utilisant MINUS

```

select c.noclient,c.nom,c.prenom,c.pays
from cmd join clients c on cmd.noclient=c.noclient
group by c.noclient,c.nom,c.prenom,c.pays
having count(*)>60
MINUS
select distinct c.noclient,c.nom,c.prenom,c.pays
from cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
join clients c on cmd.noclient=c.noclient
where cl.nolivre=3307

```

Temps d'exécution : 0,672 s, Consistent Gets : 7 572, Cost : 10 312

Listing 6.19 : Version utilisant une anti-jointure

```

select A.*
from (select c.noclient,c.nom,c.prenom,c.pays
      from cmd join clients c on cmd.noclient=c.noclient
      group by c.noclient,c.nom,c.prenom,c.pays
      having count(*)>60 ) A
left outer join
  (select distinct c.noclient,c.nom,c.prenom,c.pays
   from cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
   join clients c on cmd.noclient=c.noclient
   where cl.nolivre=3307 ) B
on a.noclient=b.noclient
where b.noclient is null

```

Temps d'exécution : 0,672 s, Consistent Gets : 7 572, Cost : 6 945

Tout comme avec INTERSECT, le fait d'utiliser une jointure nous permet de factoriser les jointures avec la table CLIENTS présente dans chacune des sous-requêtes.

Listing 6.20 : Version utilisant une anti-jointure factorisant la jointure de la table clients

```

select c.noclient,c.nom,c.prenom,c.pays
from (select noclient
      from cmd
      group by noclient
      having count(*)>60 ) A
left outer join
  (select noclient
   from cmd_lignes cl join cmd on cl.nocmd=cmd.nocmd
   where cl.nolivre=3307 ) B
on a.noclient=b.noclient
, clients c
where A.noclient=c.noclient

```

Temps d'exécution : 0,516 s, Consistent Gets : 7 899, Cost : 2 827

6.2.9 Union versus *Union ALL*

Les opérateurs UNION et UNION ALL sont très proches. La seule différence est que, si une donnée est présente dans les deux ensembles, elle n'apparaîtra qu'une fois dans le cas de l'opérateur UNION. Mais il y a un piège : si la donnée apparaît deux fois dans un des ensembles, elle n'apparaîtra aussi qu'une seule fois dans l'ensemble résultat. On peut donc en déduire que

UNION = DISTINCT UNION ALL

Comme nous l'avons vu au paragraphe précédent, ce DISTINCT a un coût car il provoque l'exécution d'un tri.

L'opérateur UNION est souvent utilisé à tort à la place d'UNION ALL, qui est moins connu.

Figure 6.1

Plan d'exécution utilisant UNION.

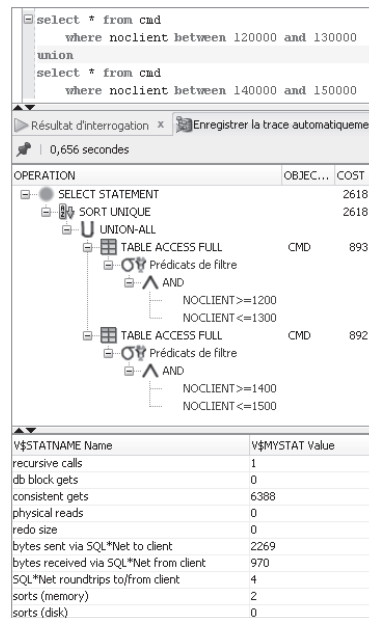


Figure 6.2

Plan d'exécution
utilisant UNION ALL.

OPERATION	OBJEC...	COST
SELECT STATEMENT		1785
UNION-ALL		
TABLE ACCESS FULL	CMD	893
Prédicats de filtre		
AND		
NOCLIENT >= 120000		
NOCLIENT <= 130000		
TABLE ACCESS FULL	CMD	892
Prédicats de filtre		
AND		
NOCLIENT >= 140000		
NOCLIENT <= 150000		

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	1
db block gets	0
consistent gets	7
physical reads	0
redo size	0
bytes sent via SQL*Net to client	2293
bytes received via SQL*Net from client	973
SQL*Net roundtrips to/from client	4
sorts (memory)	1
sorts (disk)	0

Nous constatons dans ces traces d'exécution que le coût, les Consistent Gets et le temps d'exécution augmentent de façon significative. On voit aussi qu'il y a un tri de plus. Plus l'ensemble résultat est gros, plus l'impact sera fort.

6.2.10 Count(*) versus count(colonne)

La fonction COUNT a pour objet de compter les valeurs non nulles d'une colonne, mais elle est souvent utilisée pour compter le nombre d'enregistrements. Dans ce cas, la notation COUNT(*) est préconisée. Cependant, par méconnaissance ou par habitude, certaines personnes continuent à utiliser COUNT(colonne).

L'utilisation de COUNT(*) permet à l'optimiseur de ne pas accéder à l'enregistrement entier s'il n'en a pas besoin pour répondre à la requête car il peut le faire avec des index.

La requête ci-après a un coût de 27 et n'accède qu'à l'index de la clé primaire :

```
select count(*) from clients
```

Alors que cette requête, qui retourne le même résultat si on considère que les noms sont remplis, provoque un parcours complet de la table et a un coût de 171 :

```
select count(nom) from clients
```

Cependant, si la colonne est notée obligatoire (NOT NULL), alors l'optimiseur convertit automatiquement COUNT(colonne) en COUNT(*).

6.2.11 Réduction du nombre de parcours des données

Plutôt que parcourir n fois un ensemble de données pour répondre à n questions, on essaiera de ne le parcourir qu'une fois en répondant simultanément aux n questions. L'instruction CASE sera utile à cette fin puisqu'elle permet d'avoir une syntaxe proche des clauses WHERE.

Listing 6.21 : Requête provoquant trois parcours des données

```
SELECT '<2000' categ,COUNT (*) FROM cmd_lignes WHERE montant < 2000
union all
SELECT '2000-4000' categ,COUNT (*) FROM cmd_lignes WHERE montant BETWEEN 2000
AND 4000
union all
SELECT '>4000' categ,COUNT (*) FROM cmd_lignes WHERE montant >4000 ;
```

CATEG	COUNT (*)
<2000	2604322
2000-4000	2146
>4000	9

Listing 6.22 : Requête ne parcourant qu'une seule fois les données

```
SELECT count (CASE WHEN montant < 2000 THEN 1 END) nbrinf2k,
       count (CASE WHEN montant BETWEEN 2000 AND 4000 THEN 1 END) nbr2k4k,
       count (CASE WHEN montant > 4000 THEN 1 END) nbrsup4k
FROM cmd_lignes;
```

NBRINF2K	NBR2K4K	NBRSUP4K
2604322	2146	9

Note : la fonction COUNT ne compte que les valeurs non nulles et l'absence de clause ELSE équivaut à ELSE NULL. Nous aurions obtenu le même résultat avec la fonction SUM, mais nous avons un meilleur temps de réponse en comptant les valeurs non nulles.

La seconde requête ne demande qu'un seul parcours de la table au lieu de trois pour la première. Elle sera ainsi généralement plus performante. Si la table est grande et nécessite des accès disques, l'écart sera d'autant plus net. Si la table peut tenir en mémoire, l'écart sera moindre, voire inversé.

Dans l'exemple que nous présentons ici, la table tient en mémoire. De ce fait, sous Oracle et SQL Server le gain n'est pas du tout significatif ; par contre, sous MySQL l'écart est significatif.

6.2.12 LMD et clés étrangères

Les contraintes d'intégrité référentielle implémentées par des clés étrangères (*Foreign Key*) sont contrôlées dans le sens fils → parent, lors de l'insertion de données dans les tables filles.

Lors des UPDATE de clé ou des DELETE dans la table référencée (table parente de la relation), le SGBDR contrôle dans les tables filles si des enregistrements existent, soit pour interdire l'opération, soit pour propager l'effacement ou la mise à jour si les options ON CASCADE DELETE ou ON CASCADE UPDATE sont activées. Dans les deux cas, pour chaque clé impactée, le SGBDR effectue une requête dans la table fille.

La table parente est, par conception, systématiquement indexée. Ainsi, le contrôle de la présence de la clé dans la table référencée lors de l'insertion d'une donnée dans la table fille sera performant, même si, sur de gros volumes, le coût peut devenir significatif par rapport à l'absence de contrainte de type clé étrangère.

Par contre, sur les tables filles, les champs sont moins souvent indexés, ce qui peut causer de gros problèmes de performances en cas d'opération portant sur des volumes importants.

Dans notre base exemple, imaginons que la suppression d'un client provoque l'effacement de ses commandes et des lignes de commandes associées. L'absence d'index sur les colonnes filles pourrait causer une énorme chute des performances en cas de suppression de centaines de clients. On note que, par conception, sur les relations dépendantes (comme ici entre CMD et CMD_LIGNES), la clé de la table parente se retrouve dans la clé de la table fille et est donc indexée.

Lors d'un test que nous avons effectué, nous avons constaté un ratio de 125 (5 secondes contre 650 secondes) ; par ailleurs, nous avons déjà eu l'occasion de voir chez un client un DELETE durer deux heures au lieu de quelques secondes sur des grandes tables. Attention, le coût des opérations impliquées par du LMD n'est pas affiché, ni évalué dans le plan d'exécution des instructions LMD.

Une bonne pratique consiste à indexer les colonnes filles des relations $1 \rightarrow N$, surtout si les parents sont mouvementés. Ces index sont fréquemment pertinents pour l'aspect interrogation des données.

6.2.13 Suppression temporaire des index et des CIR

Pour certaines opérations de chargement massif de données, il peut parfois être pertinent de supprimer les index et les CIR (principalement les *Foreign Key* et les clés primaires) et de les recréer une fois l'opération terminée. Cette solution permet d'économiser la réorganisation dynamique de l'index et d'améliorer le temps nécessaire au contrôle des contraintes. Cependant, elle n'est pas toujours intéressante et doit donc être évaluée au cas par cas (comme toute solution visant à améliorer les performances).

En cas de désactivation de contraintes, il faudra veiller à charger des données qui les respectent. Sinon, il ne sera pas possible de les réactiver par la suite, ce qui pourra poser des problèmes de qualité des données. Si les clés primaires ne peuvent pas être recréées à cause de la présence de doublons de clés, les index normalement associés aux clés ne seront pas créés non plus, ce qui aboutira à une probable chute des performances.

6.2.14 *Truncate versus Delete*

La commande `TRUNCATE` permet de vider entièrement le contenu d'une table. Elle est donc fonctionnellement équivalente à la commande `DELETE` sans clause `WHERE`.

Elle fait partie des commandes LDD (langage de définition des données) et travaille au niveau des structures de données plutôt qu'au niveau des enregistrements. Elle est ainsi bien plus performante car elle ne nécessite pas l'usage des fonctionnalités de journalisation et des transactions. Cependant, faire partie des commandes LDD lui impose quelques limites et induit quelques conséquences :

- Les mêmes privilèges que pour faire un `DROP` de la table sont requis (le privilège `DELETE` ne suffit pas).
- La table tronquée ne peut être référence d'aucune *Foreign Key*.
- Il est impossible de faire un `ROLLBACK` (cela provoque d'ailleurs un `COMMIT` implicite).

6.2.15 Impacts des verrous et des transactions

Le fait de pouvoir travailler à plusieurs personnes sur une même base gérant les notions de transactions isolées a des conséquences techniques énormes du point de

vue du SGBDR (Oracle, SQL Server, moteur InnoDB de MySQL). Nous pourrions consacrer un chapitre entier à ce sujet, mais cette section résume les points clés. Par exemple, lors de la modification d'un enregistrement :

- Si un ROLLBACK est exécuté, il faut remettre l'ancienne valeur.
- Tant qu'un COMMIT n'est pas exécuté :
 - les autres utilisateurs voient les anciennes données ;
 - si un autre utilisateur essaie de modifier les données, l'opération provoquera une erreur ou sera bloquée grâce à la gestion des verrous.

Tout cela est géré grâce à des mécanismes de verrouillage et de gestion des versions de lignes (MVCC, *Multi Version Concurrency Control*), qui sont très évolués, ce qui a donc un coût en termes de ressources.

La norme SQL propose quatre niveaux d'isolation qui ont des impacts différents sur les performances :

- **Read Uncommitted.** Ne gère pas les versions de lignes. On voit les dernières opérations même si elles ne sont pas validées, on parle alors de "lectures sales" (dirty read). Ce mode n'est pas recommandé, mais c'est le seul mode de travail connu des SGBDR ne gérant pas les transactions (moteur MyISAM de MySQL par exemple).
- **Read Committed.** On ne voit que ce qui est validé. C'est le mode de fonctionnement par défaut des SGBDR gérant les transactions.
- **Repeatable Read.** Si une même transaction réexécute une requête, elle verra toujours les mêmes données, même si des changements ont été validés ; cependant, de nouveaux enregistrements validés pourront apparaître.
- **Serializable.** Si une même transaction réexécute une requête, elle verra toujours les mêmes données.

Oracle gère seulement les modes Read Committed et Serializable au moyen des instructions suivantes :

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Oracle propose aussi une variante plus économe en ressource de Serializable en empêchant les modifications depuis la transaction courante à l'aide de l'instruction :

```
SET TRANSACTION ISOLATION LEVEL READ ONLY;
```

Sous MySQL, avec le moteur InnoDB les quatre niveaux sont disponibles, mais en fait Repeatable Read et Serializable sont équivalents.

SQL Server propose et implémente les quatre niveaux.

De façon générale, pour éviter des impacts négatifs sur les performances liées à ces mécanismes, appliquez les règles suivantes :

- Faites des transactions courtes.
- Si vous modifiez de gros volumes de données, essayez de décomposer l'opération en plusieurs transactions.
- Faites souvent des COMMIT.
- Ne verrouillez pas les données à outrance.

Par ailleurs, si vous ne craignez pas de mauvaise surprise avec les écritures non validées, vous pouvez utiliser, lorsque cela est disponible, Read Uncommitted, le niveau d'isolation minimal. Ce niveau peut afficher des incohérences dans les données si les transactions font des modifications dans plusieurs tables simultanément. Il est cependant adapté à de nombreuses requêtes dans de nombreux environnements. Il mérite d'être étudié s'il vous apporte des gains significatifs.

6.2.16 Optimisation du COMMIT

L'instruction COMMIT permet de terminer une transaction et dans de nombreux cas ne nécessite pas de précautions particulières. Néanmoins, dans des traitements par lot (*batch*) qui manipulent de gros volumes, quelques optimisations peuvent être apportées. En effet, l'appel de COMMIT valide la transaction et la rend ainsi durable (c'est le D d'ACID, voir encadré Info ci-après). Pour cela, la transaction est écrite dans le fichier journal de transactions provoquant une E/S.

Une première piste d'optimisation consiste à jouer sur la fréquence des COMMIT. En effet, lorsque vous modifiez un million d'enregistrements, vous pouvez décider de faire un COMMIT :

- à chaque ligne modifiée ;
- à la fin du traitement ;
- de façon périodique.

Si vous avez le choix (ce qui n'est pas toujours le cas), il est préférable de faire des COMMIT réguliers portant sur des volumes de quelques dizaines à quelques centaines

d'enregistrements. Un COMMIT à chaque ligne modifiée est coûteux de même qu'un COMMIT portant sur un gros volume de données.

Une seconde piste consiste à prendre quelques libertés avec l'aspect durable de la transaction, en laissant les écritures disque se faire de façon asynchrone au COMMIT et non pas de façon bloquante. C'est possible avec les options NOWAIT et BATCH de l'instruction COMMIT. Il ne faut pas utiliser ces options si la criticité des données ne permet pas de prendre de liberté avec l'aspect ACID de la transaction.

INFO

Une transaction doit être ACID, c'est-à-dire répondre aux caractéristiques suivantes :

- **Atomique.** La totalité ou aucune des opérations composant la transaction doit être validée.
 - **Consistante.** La base est dans un état cohérent à la fin de la transaction qu'elle soit validée ou annulée.
 - **Isolé.** La transaction n'est pas visible par les autres transactions tant qu'elle n'est pas validée (voir la section précédente).
 - **Durable.** Une fois la transaction validée, elle doit persister même en cas d'incident matériel.
-

6.2.17 DBLink et vues

Sous Oracle, l'exécution de requêtes ayant des jointures entre des tables adressées par un DBLink, comme illustré dans la requête ci-après, pose souvent des problèmes de performances. Une solution de contournement consiste à créer une vue effectuant la jointure dans la base distante et à interroger cette vue à travers le DBLink.

Listing 6.23 : Requête avec une jointure à travers un DBLink

```
select count(*)
from cmd@RemoteBase1 c,cmd_lignes@RemoteBase1 cl
where c.nocmd = cl.nocmd and noclient=41256
```

Listing 6.24 : Requête avec une jointure dans une vue et interrogation de la vue à travers un DBLink

```
-- Création d'une vue sur la base distante.
Create view Vue1 AS
select cl.*,c.noclient,c.datecommande,c.etatcommande
from cmd c,cmd_lignes cl
where c.nocmd = cl.nocmd ;
-- Requête locale faisant reference à la vue sur la base distante.
select count(*) from Vue1@RemoteBase1 where noclient=41256
```

Techniques d'optimisation des requêtes avancées

7.1 Utilisation des hints sous Oracle

Les hints (indications en anglais) sont des conseils que l'on donne à l'optimiseur pour l'orienter dans le choix d'un plan d'exécution. Ce dernier sera du coup différent de celui qu'il aurait choisi normalement en estimant que c'était le moins coûteux pour exécuter la requête demandée. L'utilisation des hints peut compenser une erreur d'estimation de l'optimiseur, mais un hint forcera un chemin d'accès, même s'il devient complètement aberrant. Il faut bien avoir en tête que l'optimiseur, bien qu'il ne soit pas parfait, s'adapte alors qu'un hint écrit dans une requête ne s'adaptera pas tant que vous n'aurez pas réécrit la requête.

L'optimiseur n'est pas obligé de suivre le conseil (ce qui est plutôt rare). S'il ne comprend pas un hint, il l'ignore sans afficher aucun message d'erreur (ce qui peut parfois donner le sentiment qu'il ne suit pas le conseil).

Les hints peuvent conduire à des gains significatifs et à des chutes de performances désastreuses. Ils ne sont généralement pas nécessaires, ils doivent être mis en œuvre avec précaution et parcimonie, vous ne devez les utiliser qu'en dernier recours et en parfaite connaissance de cause.

Avec les versions plus anciennes d'Oracle, les choses étaient différentes, l'optimiseur était moins performant et l'utilisation des hints était un passage incontournable. Aujourd'hui, ils peuvent certes encore permettre des optimisations, mais cela se restreint généralement au cas, plutôt rare, où l'optimiseur fait une mauvaise évaluation. Le reste du temps, les hints sont plutôt un risque.

Nous allons étudier rapidement les plus utiles, mais il ne faut surtout pas en mettre partout.

MS SQL Server

Tout comme sous Oracle, on retrouve le concept de hint sous SQL Server. Ils font, par contre, pleinement partie de la syntaxe du SQL et peuvent provoquer des erreurs s'ils sont syntaxiquement incorrects. Microsoft met en garde de la même façon qu'Oracle sur leur mise en œuvre qui n'est généralement pas nécessaire et qui doit être réservée à des administrateurs et des développeurs expérimentés.

Trois types de hints sont disponibles :

- **Les hints de jointures.** Placés dans les clauses de jointures, ils permettent de spécifier le type de jointure (LOOP, HASH, MERGE, REMOTE).

```
table1 inner hash join table2
table1 left outer merge join table2
```

- **Les hints de requêtes.** Ils sont placés à la fin de la requête dans la clause OPTION. De nombreuses options sont disponibles.

```
Select ... from ... OPTION (MAXDOP 4)
Select ... from ... OPTION (FORCE ORDER)
```

- **Les hints de tables.** Placés après le nom de la table dans la clause WITH, ils permettent, entre autres, d'agir sur le niveau d'isolation, les verrous ou les index utilisés.

```
Select ... from table1 WITH (SERIALIZABLE , INDEX(myindex) )
```

Consultez la documentation pour plus d'informations sur les hints. Soyez toujours très prudent lorsque vous mettez en œuvre un hint.

MySQL

On retrouve aussi sous MySQL le concept de hint mais le choix est plus modeste. Ils sont intégrés à la syntaxe et se décomposent en trois types :

- **Le hint de jointure.** Placé dans les clauses de jointures, il permet de spécifier l'ordre de jointure de type INNER à l'aide du mot clé STRAIGHT_JOIN.

```
table1 STRAIGHT_JOIN table2 on ...
```

- **Les hints de requêtes.** Ils sont placés juste après le mot clé SELECT. Les valeurs possibles sont : HIGH_PRIORITY, STRAIGHT_JOIN, SQL_SMALL_RESULT, SQL_BIG_RESULT, SQL_BUFFER_RESULT, SQL_CACHE, SQL_NO_CACHE, SQL_CALC_FOUND_ROWS

```
Select SQL_NO_CACHE * from table1
```

- **Les hints de tables.** Placés après le nom de la table, ils influent sur les index utilisés ou pas. La syntaxe est la suivante :

```
{USE | IGNORE | FORCE } {INDEX|KEY} [{FOR {JOIN|ORDER BY|GROUP  
BY}}] ([index_list])  
Select ... from table1 USE INDEX(myindex)
```

Consultez la documentation pour plus d'informations sur les hints. Soyez toujours très prudent lorsque vous mettez en œuvre un hint.

7.1.1 Syntaxe générale

Les hints sont placés dans un commentaire qui suit le mot clé de l'instruction (SELECT, INSERT, UPDATE, DELETE, MERGE).

Le commentaire est de la forme `/*+ monhint */` ou `--+ monhint` :

```
Select /*+ monhint */ nom from clients . . .
```

Attention, s'il y a un espace avant le +, le hint ne sera pas considéré comme tel. Si le hint est incorrect (syntaxiquement ou sémantiquement), il est simplement ignoré.

Les hints peuvent avoir fréquemment des paramètres qui sont des noms de table et d'index. Dans le cas des tables, il faut en fait mentionner l'alias associé à la table et non pas la table elle-même, puisqu'elle peut apparaître plusieurs fois dans la requête. Rappel : quand aucun alias n'est spécifié pour une table, l'alias par défaut est le nom de la table. S'il y a plusieurs paramètres dans le hint, ils sont séparés par un espace (pas de virgule).

```
Select /*+ monhint(t) */ nom from client t . . .
```

Il est possible de spécifier plusieurs hints en les écrivant les uns derrière les autres séparés par un espace.

```
select /*+index(cmd is_cmd_client) no_index(cli pk_clients) */  
count(datecommande)  
from clients cli join cmd on cli.noclient=cmd.noclient
```

7.1.2 Les hints *Optimizer Goal*

All_Rows. Spécifie que cette requête privilégiera la récupération de toutes les lignes (voir Chapitre 1, section 1.4.2, "L'optimiseur CBO (*Cost Based Optimizer*)").

First_Rows(*n*). Spécifie que cette requête privilégiera la récupération des *n* premières lignes. Ce hint encouragera l'utilisation des index même pour des gros ensembles à scanner. Dans la requête ci-après, l'index sur le champ Livre est utilisé si on spécifie le hint, sinon il ne l'est pas :

```
select /*+ First_Rows(10)*/ * from cmd_lignes t
where nolivre<102354
```

7.1.3 Les hints *Access Path*

Full (table_alias). Permet de forcer un parcours complet d'une table au lieu d'utiliser un index.

Index (table_alias Nom_index). Permet de forcer l'utilisation d'un index en particulier lors de l'accès à une table. Ce hint est le plus utilisé car il empêche l'optimiseur d'écarter, à tort, un index s'il y a des erreurs d'estimation des cardinalités.

```
select /*+index(cmd is_cmd_client) */ *
from cmd where noclient >10000
```

No_Index (table_alias Nom_index). Interdit l'utilisation d'un index particulier lors de l'accès à une table.

```
select /*+no_index(cmd is_cmd_client) */ *
from cmd where noclient >10000000
```

Index_FFS (table_alias Nom_index). Permet de forcer un Fast Full Scan sur un index.

No_Index_FFS (table_alias Nom_index). Interdit un Fast Full Scan sur un index.

Index_SS (table_alias Nom_index). Permet de forcer un Skip Scan sur un index.

No_Index_SS (table_alias Nom_index). Interdit un Skip Scan sur un index.

Index_RS (table_alias Nom_index). Permet de forcer un Range Scan sur un index (n'apparaît pas dans la documentation officielle).

INDEX_COMBINE(table_alias Nom_index1 Nom_index2 ...). Force une combinaison de type bitmap de plusieurs index B*Tree (voir Figure 5.8, à la section 5.2.4, "Index bitmap").

```
select /*+ INDEX_COMBINE(clients is_clients_pays is_clients_nom ) */ *
from clients where pays ='France' and nom = 'Roberts'
```

INDEX_JOIN (table_alias Nom_index1 Nom_index2 ...). Force une combinaison par jointure de plusieurs index.

7.1.4 Les hints *Query Transformation*

NO_QUERY_TRANSFORMATION. Interdit toutes les transformations de requêtes.

USE_CONCAT. Transforme des prédicats OR en concaténation (une sorte d'UNION).

```
select /*+ USE_CONCAT */ * from cmd_lignes
where nocmd=14827 or nolivre=3289
```

La requête précédente ressemble à celle-ci (à la gestion de doublons près) :

```
select * from cmd_lignes where nocmd=14827
union
select * from cmd_lignes where nolivre=3289
```

NO_EXPAND. Interdit la transformation des prédicats OR en concaténation. C'est l'opposé du hint USE_CONCAT.

REWRITE. Force l'utilisation des vues matérialisées s'il y en a de disponibles.

NO_REWRITE. Interdit l'utilisation des vues matérialisées.

MERGE (query_block). Force l'intégration d'une vue dans le schéma d'exécution principal.

```
select /*+MERGE(C)*/ cli.noclient,nom, C.Nbr
from clients cli,
     (select noclient,count(*) Nbr
      from cmd group by noclient) C
where cli.noclient=C.noclient;
```

L'impact sur le plan d'exécution de la requête précédente est de supprimer la création intermédiaire de la vue.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)
0	SELECT STATEMENT		45000	1713K		2390 (3)
* 1	HASH JOIN		45000	1713K	1104K	2390 (3)
2	TABLE ACCESS FULL	CLIENTS	45000	571K		171 (1)
3	VIEW		45156	1146K		2083 (3)
4	HASH GROUP BY		45156	220K	12M	2083 (3)
5	TABLE ACCESS FULL	CMD	1000K	4882K		890 (2)

Dans le plan d'exécution ci-après (avec le hint MERGE), on constate que la jointure est faite entre CMD et CLIENTS et ensuite seulement le GROUP BY est effectué.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)
0	SELECT STATEMENT		996K	40M		12663 (1)
1	HASH GROUP BY		996K	40M	99M	12663 (1)
* 2	HASH JOIN		996K	40M	2200K	1703 (1)
3	TABLE ACCESS FULL	CLIENTS	45000	1669K		171 (1)
4	INDEX FAST FULL SCAN	IS_CMD_CLIENT	1000K	4882K		611 (1)

NO_MERGE (query_block). Empêche l’intégration d’une vue dans le schéma d’exécution principal. C’est l’opposé du hint MERGE.

UNNEST. Permet de faire des transformations de type boucle imbriquée vers jointure.

NO_UNNEST. Empêche les transformations de type boucle imbriquée vers jointure qui sont, très souvent, effectuées lorsqu’il y a des sous-requêtes.

PUSH_PRED(query_block) / NO_PUSH_PRED(query_block). Force ou interdit l’application d’un prédicat dans une vue.

7.1.5 Les hints de jointure

LEADING. Permet de spécifier la table menante dans une jointure.

```
select /*+leading (cmd cli) */ cli.noclient,nom, cmd.datecommande
from clients cli,cmd
where cli.noclient=cmd.noclient
and cli.noclient between 100000 and 100020
```

ORDERED. Permet de spécifier que les jointures doivent être faites dans l’ordre d’apparition dans la clause FROM.

USE_NL,NO_USE_NL. Force ou interdit une jointure par boucles imbriquées (*Nested Loop*).

USE_NL_WITH_INDEX. Force une jointure par boucles imbriquées seulement si un index est présent sur la table jointe. Ce hint sera sans effet si l’index disparaît ou est invalidé. En effet, l’impossibilité d’utiliser un index pourrait avoir des conséquences désastreuses en termes de performances : une boucle imbriquée sur une grosse table non indexée a des performances catastrophiques car elle provoque un parcours complet de la table pour chaque valeur de la table menante.

USE_MERGE, NO_USE_MERGE. Force ou interdit une jointure par Sorted Merge.

USE_HASH, NO_USE_HASH. Force ou interdit une jointure par table de hachage.

7.1.6 Autres hints

Il existe une série de hints destinés au parallélisme que nous étudierons à la section 7.2.1 de ce chapitre.

CACHE (table_alias). Désigne la table comme très utilisée pour qu'elle reste longtemps dans le cache. C'est le comportement par défaut pour les petites tables.

NOCACHE (table_alias). Désigne la table comme peu utilisée pour qu'elle quitte rapidement le cache. C'est le comportement par défaut pour les grosses tables.

QB_NAME. Permet de spécifier un bloc pour y appliquer un hint.

```
SELECT /*+ QB_NAME(qb) FULL(@qb c) */ *  
FROM clients c WHERE nom = 'SMITH';
```

DRIVING_SITE. Permet d'influer sur l'exécution de requêtes utilisant des DBLink et des bases distantes.

DYNAMIC_SAMPLING (table_alias niveau_echantillonnage). Permet de faire un échantillonnage dynamique des données pour compléter les informations fournies par les statistiques, afin de les ajuster avec les prédicats de la requête (voir Chapitre 5, section 5.1.3, "Sélectivité, cardinalité, densité").

7.2 Exécution parallèle

Il s'agit ici de faire exécuter une requête par plusieurs processeurs de façon parallèle. Nous n'allons pas entrer dans le détail de tout ce qu'il est possible de faire avec l'exécution parallèle, mais juste donner un aperçu.

Comme vous pouvez vous en douter, ce découpage en sous-tâches a un certain coût d'organisation. De plus, l'exécution parallèle a pour caractéristique de s'exécuter principalement en faisant des lectures disques plutôt qu'en utilisant le cache de données. Cela conforte l'idée que cette opération est plutôt destinée à travailler

sur de gros volumes. Ce type d'optimisation ne sera pertinent que dans les cas suivants :

- Il y a plusieurs processeurs disponibles.
- Il n'y a pas de goulet d'étranglement au niveau du disque dur.
- Il y a suffisamment de choses à faire pour que la mise en place du parallélisme soit compensée par le gain lié à ce dernier.

L'utilisation du parallélisme sur des cas non adaptés provoquera un ralentissement des opérations. L'exécution parallèle est particulièrement adaptée à des tables partitionnées, réparties sur des disques différents.

Les opérations concernées par la parallélisation sont :

- les requêtes interrogations de type SELECT ;
- les requêtes de manipulation des données (LMD : UPDATE, DELETE sur partitions différentes et INSERT sur requête) ;
- certaines opérations de LDD.

L'exécution parallèle peut être configurée au niveau de la session pour les trois types d'opérations à l'aide des paramètres suivants :

- PARRALEL QUERY
- PARRALEL DML
- PARRALEL DDL

Les paramètres de parallélismes peuvent avoir une des trois valeurs suivantes :

- ENABLE. Les fonctions de parallélisme seront utilisées si elles sont spécifiées par un hint d'une requête ou au niveau d'un objet manipulé.
- DISABLE. Les fonctions de parallélisme ne seront pas utilisées.
- FORCE. Permet de forcer un niveau de parallélisme.

Par défaut, le parallélisme est configuré ainsi :

- PARRALEL QUERY = ENABLE

■ `PARRALEL DML = ENABLE`

■ `PARRALEL DDL = DISABLE`

Si vous souhaitez spécifier des hints de parallélisme dans les requêtes DML, vous devrez d'abord exécuter l'instruction suivante :

```
alter session enable parallel dml;
```

L'exécution parallèle peut être configurée au niveau de chaque requête comme montré ci-après :

```
select /*+ PARALLEL(c , 4) */max(C.DATECOMMANDE) from cmd c;  
insert /*+ parallel (Cmd2009,4,1) */ into Cmd2009  
  select * from cmd Where datecommande<To_Date('2010','YYYY');  
alter index IS_CMD_CLIENT rebuild parallel 4;
```

L'exécution parallèle peut être configurée au niveau des objets. Ainsi, toute opération sur ces objets sera parallélisée sans que vous ayez à le spécifier au niveau de l'opération.

```
ALTER TABLE cmd PARALLEL (DEGREE 4);
```

Les opérations parallèles prennent généralement un paramètre, nommé "degré de parallélisme" (DOP, *degree of parallelism*), qui définit le nombre de sous-tâches concurrentes qui composeront l'opération. Il est important de choisir un DOP en adéquation avec le matériel. C'est inutile, voire inefficace, de spécifier un DOP supérieur au nombre d'unités de traitement (nombre de CPU × nombre de cœurs par CPU).

7.2.1 Les hints de parallélisme

PARALLEL(table_alias , degré //). Permet de spécifier un niveau de parallélisme pour les accès à une table.

NO_PARALLEL (table_alias). Permet de désactiver le parallélisme pour les accès à une table.

PARALLEL_index (table_alias , idx,degré). Permet de spécifier un niveau de parallélisme sur les Index Scan.

NO_PARALLEL_index (table_alias). Permet de désactiver le parallélisme sur les Index Scan.

PQ_DISTRIBUTE. Permet d'influer sur les modes de communication entre les sous-tâches pour effectuer les jointures parallèles.

MS SQL Server

SQL Server gère aussi le parallélisme de façon assez analogue à Oracle. Cependant, il est bien plus efficace sur la parallélisation des petites requêtes et active d'ailleurs par défaut ce comportement (sauf sur les éditions express).

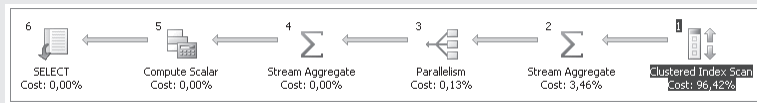
Le hint MAXDOP permet d'influer sur le niveau de parallélisme. La valeur 1 permet de le désactiver.

Exemple de plan d'exécution d'une requête avec un degré de parallélisme à 4 :

```
select sum(quantite) from cmd_lignes
option (maxdop 4)
```

Figure 7.1

Plan d'exécution d'une requête parallélisée sous MS SQL Server.



MySQL

MySQL ne propose pas de solution pour exploiter plusieurs processeurs dans une même requête.

7.3 Utilisation du SQL avancé

Certaines fonctions avancées permettent de faire des requêtes qu'il était quasiment impossible de faire en SQL conventionnel ou alors au prix d'une grosse chute des performances. La plupart ont été introduites en version 8i ou 9i d'Oracle, dans les dernières versions de SQL Server et ne sont pas disponibles sous MySQL.

7.3.1 Les Grouping Sets

Ils sont disponibles depuis Oracle 9i et SQL Server 2008.

Les *Grouping Sets* permettent d'utiliser les fonctions d'agrégats sur plusieurs groupes différents dans la même requête. La requête suivante :

```
Select JOB,ENAME, count(*) From BIGEMP
Group by Grouping Sets ((JOB), (ENAME))
```

équivalent sémantiquement à :

```
Select JOB,NULL ENAME, count(*) From BIGEMP
Group by JOB
union all
Select NULL,ENAME, count(*) From BIGEMP
Group by ENAME
```

et produit le résultat ci-après :

JOB	ENAME	COUNT (*)
-----	-----	-----
	ALLEN	100001
	JONES	100001
	FORD	100001
	CLARK	100001
	MILLER	100001
	SMITH	100001
	WARD	100001
	MARTIN	100001
	SCOTT	100001
	TURNER	100001
	ADAMS	100001
	BLAKE	100001
	KING	100001
	JAMES	100001
SALESMAN		400004
CLERK		400004
PRESIDENT		100001
MANAGER		300003
ANALYST		200002
19 rows selected		

En termes de performances, GROUPING SETS devrait normalement être deux fois plus rapide que deux requêtes GROUP BY successives puisqu'on ne parcourt qu'une fois l'ensemble de données. Dans les faits, l'estimation faite par l'optimiseur indique que le coût est deux fois moindre pour la requête utilisant GROUPING SETS que pour la requête utilisant les GROUP BY simples. Cependant, l'écart n'est plus que de 15 % pour les Consistent Gets, et la tendance s'inverse en temps d'exécution : la requête utilisant des GROUP BY simples est plus rapide sur Oracle. Sur SQL Server, la requête utilisant GROUPING SETS est plus rapide de 20 %.

Il conviendra, encore plus que d'habitude, d'évaluer cette solution dans votre contexte afin de juger de son réel intérêt.

7.3.2 Rollup Group By

Il est disponible sur SQL Server, MySQL 5.x et depuis Oracle 9i.

L'opérateur `ROLLUP GROUP BY` permet de générer les sous-totaux de `GROUP BY` multi-colonnes dans une requête unique, comme si on effectuait successivement les `GROUP BY` mentionnés en supprimant à chaque itération le niveau de groupe le plus à droite. La requête suivante :

```
Select Deptno, Job, count(*) From EMP
group by rollup(Deptno, Job)
```

équivalent sémantiquement à :

```
Select Deptno, Job, count(*) From EMP
Group by Deptno, Job
union all
Select Deptno,null, count(*) From EMP
Group by Deptno
union all
Select null,null, count(*) From EMP
Order by 1,2
```

et produit le résultat ci-après :

DEPTNO	JOB	COUNT(*)
10	CLERK	1
10	MANAGER	1
10	PRESIDENT	1
10		3
20	CLERK	2
20	ANALYST	2
20	MANAGER	1
20		5
30	CLERK	1
30	MANAGER	1
30	SALESMAN	4
30		6
		14

13 rows selected

En termes de performances, `GROUP BY ROLLUP` devrait être trois fois plus rapide que trois requêtes `GROUP BY` successives, puisque le SGBDR ne parcourt qu'une fois l'ensemble de données sources. Dans la pratique, le ratio est plutôt de 2,2, ce qui est déjà bien. Ce ratio se retrouve sur le temps d'exécution, le coût de l'optimiseur et les Consistent Gets.

MS SQL Server

Le gain de performances est analogue à celui d'Oracle.

Les versions antérieures à la version 2008 utilisent une notation alternative transformant ainsi la requête :

```
Select Deptno,JOB, count(*) From EMP  
group by Deptno,job with rollup
```

MySQL

MySQL supporte la notation ROLLUP. Cependant, les résultats en performances ne sont pas au rendez-vous, puisque la version ROLLUP est 20 % plus lente que celle utilisant trois GROUP BY. La notation est la même que l'ancienne notation de SQL Server, il faut donc écrire ainsi :

```
Select Deptno,JOB, count(*) From EMP  
group by Deptno,job with rollup
```

7.3.3 Cube Group By

Il est disponible sur SQL Server et depuis Oracle 9i.

L'opérateur CUBE GROUP BY permet de générer les sous-totaux de toutes les dimensions des GROUP BY multicolonne dans une requête unique, comme si on effectuait successivement les GROUP BY en faisant toutes les combinaisons possibles.

```
Select Deptno,JOB, count(*) From EMP  
group by cube(deptno,job)
```

équivalent sémantiquement à :

```
Select Deptno,JOB, count(*) From EMP  
Group by Deptno,JOB  
union all  
Select Deptno,null, count(*) From EMP  
Group by Deptno  
union all  
Select null,job, count(*) From EMP  
Group by job  
union all  
Select null,null, count(*) From EMP
```

et produit le résultat ci-après :

DEPTNO	JOB	COUNT (*)
		14
	CLERK	4
	ANALYST	2
	MANAGER	3
	SALESMAN	4
	PRESIDENT	1
10		3
10	CLERK	1
10	MANAGER	1
10	PRESIDENT	1
20		5
20	CLERK	2
20	ANALYST	2
20	MANAGER	1
30		6
30	CLERK	1
30	MANAGER	1
30	SALESMAN	4

18 rows selected

En termes de performances, `GROUP BY CUBE` devrait être quatre fois plus rapide que quatre requêtes `GROUP BY` successives, puisqu'il ne parcourt qu'une seule fois l'ensemble de données sources. Dans la pratique, le ratio est plutôt de 3. Ce ratio se retrouve sur le temps d'exécution, le coût de l'optimiseur et les Consistent Gets.

MS SQL Server

Le gain de performance est plutôt de l'ordre de 50 %.

Les versions antérieures à la version 2008 utilisent une notation alternative transformant ainsi la requête :

```
Select Deptno,JOB, count(*) From EMP
group by Deptno,job with cube
```

MySQL

Ne supporte pas la notation `CUBE`.

7.3.4 Utilisation de *WITH*

L'opérateur *WITH*, introduit avec Oracle 9i, permet de déclarer une sous-requête qui va être utilisée plusieurs fois dans la requête, ce qui évite la répétition d'une sous-requête à plusieurs endroits de la requête. Cela présente un intérêt en termes de clarté de code mais offre aussi l'avantage d'améliorer parfois les performances en favorisant la création de tables temporaires internes.

Étudions la requête suivante qui affiche les informations relatives à la plus grosse ligne de commande de livres de l'éditeur Pearson :

```
select cmd.nocmd,cmd.noclient,cmd.datecommande,
       cl.quantite,cl.montant,cl.nolivre
from cmd join cmd_lignes cl on cmd.nocmd=cl.nocmd
       join livres l on cl.nolivre=l.nolivre
where editeur='Pearson'
and quantite=(select max(cl.quantite) quantite
              from cmd_lignes cl join livres l on cl.nolivre=l.nolivre
              where editeur='Pearson' )
```

Nous remarquons que la requête principale et la sous-requête expriment toutes les deux une jointure entre les tables *LIVRES* et *CMD_LIGNES* et filtrent sur l'éditeur *Pearson*. Le plan d'exécution nous apprend que le SGBDR fait aussi le travail deux fois. La requête ci-après est sémantiquement identique mais ne répète pas les opérations, ni dans la requête, ni dans le plan d'exécution :

```
with PearsonBookLignes
as (select cmd.nocmd,cmd.noclient,cmd.datecommande
      ,cl.quantite,cl.montant,cl.nolivre
      from cmd join cmd_lignes cl on cmd.nocmd=cl.nocmd
      where nolivre in (select nolivre from livres where editeur='Pearson'))
select L.* from PearsonBookLignes L
where quantite=( select max(quantite) from PearsonBookLignes )
```

La version utilisant *WITH* est presque deux fois plus performante. Cependant, il faut prendre garde car si la requête nommée dans la clause *WITH* est trop volumineuse, la tendance peut s'inverser. Dans ce cas, l'utilisation du hint */*+ inline */* dans la requête située dans le *WITH* pourra être adaptée.

MS SQL Server

Cette notation est disponible depuis SQL Server 2005, mais nous n'avons pas noté de gains de performance significatifs.

7.3.5 Les fonctions de classement (*ranking*)

Le SQL n'étant pas un langage procédural, il n'est pas facile d'exprimer des choses telles que le 10^e salaire, le 50^e employé ou le numéro d'ordre d'un classement. Dans le passé, sous Oracle, on utilisait la pseudo-colonne Rownum et sous SQL Server et MySQL, la clause LIMIT, mais cela s'apparentait à du bricolage. Les versions 8.1.7 d'Oracle et 2005 de SQL Server ont introduit les fonctions de classement (*ranking*). Elles permettent d'obtenir le classement d'un enregistrement dans un ensemble quand on a spécifié les critères de classement.

Par exemple, la requête ci-après donne pour chaque employé le rang de son salaire, son quartile, son pourcentage dans la distribution et le numéro de la ligne :

```
Select ename,sal,RANK() over(order by sal desc) Rang
      ,DENSE_RANK() over(order by sal desc) RangDense
      ,ROW_NUMBER() over(order by sal desc ) NoLigne
      ,round(PERCENT_RANK() over(order by sal desc)*100) PctRang
      ,NTILE(4) over(order by sal desc ) Quartile
From EMP
```

ENAME	SAL	RANG	RANGDENSE	NOLIGNE	PCTRANG	QUARTILE
KING	5000	1	1	1	0	1
FORD	3000	2	2	2	8	1
SCOTT	3000	2	2	3	8	1
JONES	2975	4	3	4	23	1
.						
JAMES	950	13	11	13	92	4
SMITH	800	14	12	14	100	4

La différence entre la fonction RANK et la fonction DENSE_RANK se situe au niveau de la gestion des enregistrements ayant le même classement. Dans notre exemple, SCOTT et FORD ont le même salaire et sont tous les deux deuxièmes. La fonction RANK considère qu'il n'y a pas de troisième et JONES est donc quatrième, alors que la fonction DENSE_RANK considère que JONES est troisième. Le classement est établi en suivant la clause ORDER BY mentionnée dans la clause OVER() associée à la fonction. PERCENT_RANK donne la position en pourcentage. NTILE(N) donne le N-tile de la valeur, donc pour N=4 donne le quartile. ROW_NUMBER donne le numéro de la ligne suivant l'ordre spécifié sans doublon.

Ces fonctions permettent aussi d'effectuer des classements par partition, c'est-à-dire sur des sous-ensembles de données. Par exemple, la requête ci-après donne pour

chaque employé le rang de son salaire au sein de son département. On constate qu'à chaque changement de département le classement recommence à 1.

```
Select deptno,ename,sal,
RANK() over (PARTITION BY deptno order by sal desc) Rang
From EMP
```

DEPTNO	ENAME	SAL	RANG
10	KING	5000	1
10	CLARK	2450	2
10	MILLER	1300	3
20	SCOTT	3000	1
20	FORD	3000	1
20	JONES	2975	3
20	ADAMS	1100	4
20	SMITH	800	5
30	BLAKE	2850	1
.			

Les fonctions de classement permettent d'écrire des requêtes qui ramènent les n premiers enregistrements suivant un ordre (Top n). Pour cela, il suffit de mettre une condition sur le classement. Cependant, il n'est syntaxiquement pas possible de spécifier ces fonctions dans la clause WHERE. La requête suivante provoque l'erreur : ORA-30483: fonctions de fenêtrage interdites ici.

```
Select deptno,ename,sal,RANK() over (PARTITION BY deptno order by sal desc)
Rang From EMP
where RANK() over (PARTITION BY deptno order by sal desc) =1;
```

Nous devons utiliser une sous-requête :

```
Select deptno,ename,sal
From (Select deptno,ename,sal
      ,RANK() over (PARTITION BY deptno order by sal desc) Rang
      From EMP )
where Rang=1;
```

Pour obtenir le résultat suivant :

DEPTNO	ENAME	SAL
10	KING	5000,00
20	SCOTT	3000,00
20	FORD	3000,00
30	BLAKE	2850,00

L'utilisation des fonctions de *ranking* pour les requêtes de type Top n n'améliore pas forcément les performances par rapport à une solution simple avec une sous-requête

comme dans l'exemple ci-après (il n'est cependant pas toujours possible de transformer la requête de façon aussi simple) :

```
Select deptno,ename,sal From EMP
where (deptno,sal) in (
    Select deptno,max(sal)
    From EMP group by deptno )
```

7.3.6 Autres fonctions analytiques

Les fonctions analytiques ont la capacité de travailler sur un ensemble d'enregistrements de façon simple et souvent performante (les fonctions de classement en font partie).

Les fonctions d'agrégations classiques (MIN, MAX, SUM, AVG, COUNT) existent en versions analytiques. Dans ce mode, elles s'appliquent sur une fenêtre glissante autour de l'enregistrement. LEAD et LAG permettent de manipuler les enregistrements précédents et suivants.

L'exemple ci-après permet de faire des moyennes glissantes grâce à AVG sur une fenêtre couvrant les quatre enregistrements précédents et calcule l'écart de la colonne Sal avec les lignes suivantes et précédentes grâce aux fonctions LEAD et LAG :

```
SELECT ename, sal
      ,AVG(sal) OVER (ORDER BY sal
                     ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING) AS MoyenneGlissante
      ,sal-lag(sal,1) OVER (ORDER BY sal ) AS EcartPrec
      ,lead(sal,1) OVER (ORDER BY sal )-sal AS EcartSuivant
FROM emp
order by sal;
```

ENAME	SAL	MOYENNEGLISSANTE	ECARTPREC	ECARTSUIVANT
SMITH	800			150
JAMES	950	800	150	150
ADAMS	1100	875	150	150
WARD	1250	950	150	0
.				
SCOTT	3000	2275	25	0
FORD	3000	2575	0	2000
KING	5000	2855	2000	

L'exemple ci-après montre comment faire une recherche du percentile médian de chaque département. La fonction PERCENTILE_CONT recherche le percentile de façon continue par interpolation, alors que la fonction PERCENTILE_DISC retourne la valeur présente dans les données immédiatement inférieures ou égales à la version continue.

```
SELECT deptno ,avg(sal) as Moyenne
      ,PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY sal) as mediane
      ,PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY sal) as mediane_disc
FROM emp
group by deptno;
```

DEPTNO	MOYENNE	MEDIANE	MEDIANE_DISC
10	2916	2450	2450
20	2175	2975	2975
30	1566	1375	1250

Les fonctions REGR_xxx permettent d'effectuer des régressions linéaires sur les données. (Voir la documentation de référence SQL d'Oracle.)

Concernant l'utilisation de fonctions d'agrégations personnalisées, un besoin récurrent est de pouvoir lister, dans une colonne, les enregistrements liés par une relation maître/détails. Cette opération était possible dans le passé grâce à une fonction qui parcourait le sous-ensemble à l'aide d'un curseur. À la version 9, une autre manière plus élégante était apparue, à partir d'un type et d'une méthode d'agrégation, mais c'était compliqué à mettre en œuvre et toujours moins performant qu'une solution native. La version d'Oracle 11g Release 2 résout ce besoin en ajoutant la fonction d'agrégation native LISTAGG, qui permet de concaténer des chaînes de caractères avec un séparateur dans un champ, comme illustré dans cet exemple :

```
select deptno,LISTAGG(ename, ',' ) WITHIN GROUP (ORDER BY ename) as Liste
from emp group by deptno;
```

DEPTNO	LISTE
10	CLARK,KING,MILLER
20	ADAMS,FORD,JONES,SCOTT,SMITH
30	ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD

7.3.7 L'instruction **MERGE**

L'instruction MERGE, introduite avec Oracle 9i, permet de fusionner une table source dans une table destination. Tous les enregistrements de l'ensemble de données sources présents dans la table de destination seront mis à jour et ceux qui ne sont présents que dans la source seront insérés. Ceux qui ne sont présents que dans la destination ne seront pas modifiés.

Cette instruction revient à faire un Update et un Insert. Elle apporte un gain de performances pour les opérations de fusion de table. La syntaxe est la suivante :


```

MERGE INTO <TableDestination> USING <TableSource> on (<ClauseJointure>)
WHEN MATCHED THEN UPDATE SET <Champ>=<Expression>, ...
WHEN NOT MATCHED THEN INSERT (Champ, ...) values (Expression, ...)

```

MS SQL Server

L'instruction MERGE est aussi disponible. Elle peut intégrer également le fait de supprimer les enregistrements présents dans la destination et pas dans la source à l'aide de la clause WHEN NOT MATCHED BY SOURCE.

MySQL

MySQL ne propose pas de solution équivalente à MERGE.

7.3.8 Optimisation des updates multitables

Pour faire un update dans une table relativement à des données contenues dans une autre table, on écrit habituellement des sous-requêtes. Par exemple, la requête suivante augmente les salaires des employés listés dans la table AUGMENTATION :

```

update bigemp e
set sal=sal+(select Montant from augmentation where empno=e.empno)
where empno in (select empno from augmentation )

```

Une première solution possible pour améliorer les performances consiste à utiliser l'instruction MERGE amputée de la clause WHEN NOT MATCHED. Cette solution peut apporter un gain de performance allant jusqu'à 10 %.

```

MERGE into bigemp E using augmentation A on (a.empno=e.empno)
WHEN MATCHED THEN UPDATE set sal=sal+Montant

```

Une autre solution, particulièrement méconnue, est l'utilisation d'une vue dynamique dans l'instruction UPDATE :

```

update (select sal,Montant from augmentation a, bigemp e
        where a.empno=e.empno)
set sal=sal+Montant

```

Cette solution apporte un gain très significatif, puisque le temps d'exécution de la requête UPDATE passe de 33 secondes à 19 secondes. L'explication se trouve facilement dans les plans d'exécution ci-après. En effet, la dernière solution n'effectue qu'un accès à la table AUGMENTATION. Les Consistent Gets passent de 2 949 005 à 12 781.

Figure 7.2

Plan d'exécution de l'UPDATE avec sous-requête.

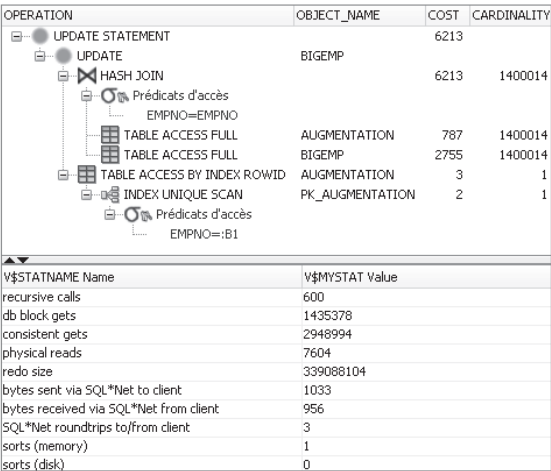
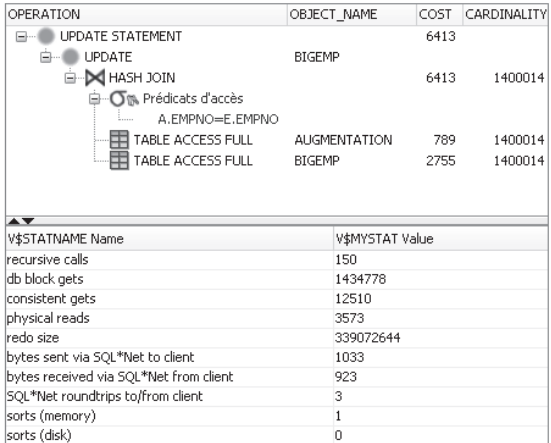


Figure 7.3

Plan d'exécution de l'UPDATE avec jointure.



MS SQL Server

La motivation nous poussant à remplacer une instruction UPDATE par un MERGE n'existe pas sous SQL Server, car l'instruction UPDATE est capable de faire un UPDATE en utilisant une jointure.

MySQL

MySQL ne propose pas de solution alternative pour améliorer les updates multitables.

7.3.9 Insertion en mode *Direct Path*

Le mode d'insertion *Direct Path* effectue les insertions dans de nouveaux blocs de données, sans passer par le buffer cache. Le gain potentiel réside dans l'économie de la recherche de place dans les blocs existants et l'économie de mémoire dans le cache. Les inconvénients de ce mode sont nombreux aussi :

- Il écrit directement sur les disques. Si ces derniers sont peu performants ou occupés, la tendance peut s'inverser.
- Il pose un verrou sur toute la table. Ce verrou peut être long à obtenir et pénalisant pour les autres transactions.
- Un COMMIT est obligatoire immédiatement après une insertion.

Pour activer les insertions en mode *Direct Path*, il suffit d'ajouter le hint APPEND à l'instruction INSERT. Ce hint est utilisable sur les requêtes INSERT utilisant une requête SELECT.

```
INSERT /*+ APPEND */ INTO ma_table SELECT * from autre_table
```

En PL/SQL, avec l'instruction FORALL que nous étudierons à la section 7.4.3 de ce chapitre, le mode *Direct Path* est activable à l'aide du hint APPEND_VALUES.

7.4 PL/SQL

7.4.1 Impact des triggers

Les triggers sont des outils très pratiques mais qui peuvent nuire très sérieusement aux performances sur des insertions massives de données. Il faudra particulièrement veiller à l'efficacité du code contenu dans les triggers des tables très mouvementées. Même si le code du trigger est optimal, l'appel même du trigger à un coût. Nous illustrons ce phénomène en transférant le contenu de la table CLIENT (45 000 enregistrements) dans la table CLIENTS2 en convertissant le champ Nom en majuscules, soit en appelant la fonction UPPER() dans une requête SQL, soit avec un trigger qui effectue cette opération.

Listing 7.1 : Première solution, complètement en SQL

```
insert into clients2
select noclient, upper(nom), prenom, adresse1, adresse2, codepostal, ville,
pays, tel, email from clients
```

Listing 7.2 : Deuxième solution, utilisant un trigger PL/SQL

```
create or replace trigger TRG_BEFORE_INS_CLIENTS2
  before insert on clients2 for each row
begin
  :new.nom:=upper(:new.nom);
end;
/
insert into clients2
select noclient, nom, prenom, adresse1, adresse2, codepostal, ville, pays,
tel, email from clients
```

Le résultat est sans appel : la première solution s'exécute en 94 ms, alors que la deuxième, avec le trigger, s'exécute en 485 ms. L'écart de temps correspond principalement à l'invocation du trigger PL/SQL.

Il faut essayer de réduire les exécutions des triggers et, pour cela, la clause méconnue *WHEN* peut être utile. Elle permet d'évaluer une condition afin de décider d'exécuter le trigger ou pas. Dans le cadre de notre exemple, il est inutile d'exécuter le trigger si la donnée est déjà en majuscules, nous pouvons donc mettre cela en condition préalable dans la clause *WHEN*.

Listing 7.3 : Troisième solution, utilisant un trigger PL/SQL avec une clause *WHEN*

```
CREATE OR REPLACE TRIGGER TRG_BEFORE_INS_CLIENTS2
  before insert on clients2 for each row
  when (new.nom!=upper(new.nom))
begin
  :new.nom:=upper(:new.nom);
end;
/
insert into clients2
select noclient, upper(nom), prenom, adresse1, adresse2, codepostal, ville,
pays, tel, email from clients
```

Nous constatons que si les données en entrée sont déjà en majuscules, le gain est très significatif, puisque le temps d'exécution passe à 109 ms, soit un surcoût de 15 ms. Ce gain provient du fait qu'il n'y a pas à invoquer le moteur PL/SQL pour tester la clause *WHEN*. Si les données ne sont pas en majuscules, le surcoût est là aussi de 15 ms (on passe de 485 ms à 500 ms).

Une autre solution pour éviter l'exécution inutile de trigger se trouve au niveau de la clause *OF* sur les triggers *UPDATE*. Cette clause permet de n'exécuter le trigger que si les colonnes qu'elle spécifie sont présentes dans la requête *UPDATE*. L'utilisation de cette clause avec un trigger en *UPDATE* qui effectue le même travail que le trigger *INSERT* est illustrée ci-après.

Listing 7.4 : Trigger utilisant une clause *UPDATE OF*

```
CREATE OR REPLACE TRIGGER TRG_BEFORE_UPD_CLIENTS2
  before update OF nom on clients2 for each row
begin
  :new.nom:=upper(:new.nom);
end;
/
update clients2 set prenom=prenom;
```

L'exécution de la requête UPDATE ne concernant pas la colonne Nom prend 469 ms avec la clause OF alors qu'elle prend sinon 1 125 ms à cause de l'exécution inutile du trigger et de la réécriture inutile de la valeur du champ Nom qu'elle nécessite.

7.4.2 Optimisation des curseurs (*BULK COLLECT*)

La manière classique en PL/SQL consiste à utiliser des curseurs dans des boucles FOR qui permettent de récupérer les lignes une par une. Cela fonctionne bien mais ce n'est pas très performant lorsque le volume des enregistrements devient important car il n'y a qu'un enregistrement récupéré (*fetch*) par itération. Afin d'améliorer les performances de la récupération des enregistrements, nous avons à notre disposition les instructions BULK COLLECT utilisables dans les instructions suivantes :

- Select .. Into
- Fetch .. Into
- Returning .. Into

La récupération s'effectue alors par blocs dans des tableaux et non pas dans des variables scalaires. Il suffit juste de faire précéder le mot clé INTO par BULK COLLECT.

Illustrons sa mise en œuvre par le remplacement d'un curseur sur un programme simple qui calcule la somme des salaires :

Listing 7.5 : Version en PL/SQL classique

```
declare
  Cursor c1 is select sal from bigemp;
  somme number;
begin
  somme:=0;
```

```
for rec in C1 loop
    somme:=somme+rec.sal;
end loop;
dbms_output.put_line(somme);
end;
```

Temps d'exécution : 1,26 s

Listing 7.6 : Version en PL/SQL utilisant *SELECT BULK COLLECT INTO*

```
declare
    somme number;
    TYPE Table_N IS table OF Number ;
    lstsal Table_N;
begin
    somme:=0;
    SELECT sal BULK COLLECT INTO lstsal FROM bigemp;
    for j in lstsal.first..lstsal.last loop
        somme:=somme+lstsal(j);
    end loop;
    dbms_output.put_line(somme);
end;
```

Temps d'exécution : 0,51 s (On constate un gain de performances significatif.)

On constate que la solution utilisant l'instruction `BULK COLLECT` dissocie la récupération des données de leur traitement en les stockant dans un tableau. Cette solution, en plus du gain de performances, offre si nécessaire plus de flexibilité au niveau du parcours des données issues de la requête. Le fait de stocker le résultat dans un tableau ne contraint pas d'avoir un parcours unidirectionnel comme l'impose un curseur classique.

Cependant, comme rien n'est jamais magique, cette seconde solution consomme plus de ressource mémoire. Il se peut que cette solution pose des problèmes liés à une saturation de la mémoire si le volume des données est trop important. Si c'est le cas, vous pouvez utiliser un curseur `BULK COLLECT`.

Il s'agit d'un curseur qui récupère non pas un seul enregistrement par *fetch* mais un ensemble d'enregistrements qui sont mis dans un tableau dont la taille est spécifiée à l'aide de la clause `LIMIT`. Cette technique est une solution intermédiaire entre le curseur classique et le `SELECT INTO BULK COLLECT`, qui permet de récupérer les données par paquets de taille intermédiaire. Cette solution nécessite généralement l'utilisation de deux boucles imbriquées (voir Listing 7.7).

Listing 7.7 : Version en PL/SQL utilisant *FETCH BULK COLLECT INTO*

```
declare
  Cursor c1 is select sal from bigemp e;
  somme number;
  TYPE Table_N IS table OF Number ;
  ltsal Table_N;
begin
  somme:=0;
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO ltsal LIMIT 1000;
    EXIT WHEN c1%NOTFOUND;
    for j in ltsal.first..ltsal.last loop
      somme:=somme+ltsal(j);
    end loop;
  end loop;
  dbms_output.put_line(somme);
end;
```

Le temps d'exécution de cette version est de 0,64 seconde. C'est toujours en progrès significatif par rapport à un curseur conventionnel mais en recul par rapport à la solution `SELECT INTO BULK COLLECT`. Cela s'explique par le fait que la requête ne pose pas de problème de volume, le coût des opérations supplémentaires impliquées par cette solution n'est compensé par aucun gain.

Afin de mettre en évidence d'éventuels problèmes de performances liés aux volumes, nous allons faire quelques tests en faisant fluctuer la clause `LIMIT` sur une variante qui ramène les enregistrements complets au lieu de ne ramener que la colonne `Sal`.

Listing 7.8 : Version en PL/SQL utilisant *FETCH BULK COLLECT* ramenant les enregistrements entiers

```
declare
  Cursor c1 is select e.* from bigemp e;
  somme number;
  TYPE Table_N IS table OF c1%rowtype ;
  ltsal Table_N;
begin
  somme:=0;
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO ltsal LIMIT 100;
    EXIT WHEN c1%NOTFOUND;
    for j in ltsal.first..ltsal.last loop
      somme:=somme+ltsal(j).sal;
    end loop;
  end loop;
  dbms_output.put_line(somme);
end;
```

Tableau 7.1 : Analyse des performances en fonction de la clause LIMIT

<i>Clause LIMIT</i>	<i>Temps d'exécution</i>
1	19,5 s
10	4,5 s
100	3,25 s
1 000	2,59 s
10 000	2,59 s
100 000	3,11 s
500 000	3,27 s
1 000 000	3,70 s

Nous constatons que la récupération (fetch) par blocs de quelques milliers d'enregistrements est optimale pour ce cas-là. La valeur 1 000 est une valeur adaptée à pas mal de cas, elle permet de réduire par 1 000 le nombre de fetches tout en utilisant une quantité de mémoire raisonnable pour le tableau. Si vous n'avez pas le temps de faire une évaluation plus approfondie, c'est une valeur assez polyvalente qui convient dans la plupart des cas.

7.4.3 Optimisation du LMD (*FORALL*)

La modification de données à partir de curseurs (classique ou BULK COLLECT) a pour effet de multiplier le nombre d'exécutions d'instructions de type LMD (INSERT, UPDATE, DELETE). Cette méthode peut avoir un impact significativement négatif sur les performances par rapport à du SQL classique.

Afin d'améliorer les performances de ce type d'opération, il existe un équivalent au BULK COLLECT pour le LMD au travers de l'instruction FORALL qui a la syntaxe suivante :

```
FORALL index IN lower_bound..upper_bound  
  Instruction_SQL_LMD;
```

FORALL permet de manipuler des instructions du LMD avec des tableaux, ce qui réduit le nombre d'exécutions de requêtes. Chaque FORALL n'exécutera l'instruction qu'une seule fois avec des tableaux comme paramètres au lieu de valeurs scalaires.

Listing 7.9 : Exemple d'utilisation de *FORALL*

```

DECLARE
  TYPE Table_N IS table OF Number ;
  lstepno Table_N ;
  lsmontant Table_N ;
begin
  SELECT empno, aug BULK COLLECT INTO lstepno,lsmontant FROM empbigaug;
  forall j in lstepno.first..lstepno.last
    UPDATE empbig set SAL=SAL+lsmontant(j) where empno=lstepno(j);
end;
```

Nous allons évaluer les performances de cette solution à travers un petit programme qui augmente les salaires de la localisation "DALLAS" de 10 et ceux de la localisation "Toulouse" de 31. Le programme est équivalent aux instructions SQL suivantes que nous utilisons comme référence :

Listing 7.10 : Version SQL *UPDATE*

```

update bigemp e
set sal=sal+(select decode(loc,'CHICAGO',10,'Toulouse',31,0)
             from bigdept where deptno=e.deptno)
where deptno in (select deptno from bigdept
                 where loc in ('CHICAGO','Toulouse') );
```

Temps d'exécution : 11,32 s

Listing 7.11 : Version SQL *MERGE*

```

merge into bigemp e using
  (select empno,sal+decode(loc,'CHICAGO',10,'Toulouse',31,0) NewSal
   from bigemp e,bigdept d
   where e.deptno=d.deptno and loc in ('CHICAGO','Toulouse')) A
on (E.empno=A.empno)
when matched then update Set sal=Newsal;
```

Temps d'exécution : 12,81 s

Listing 7.12 : Version SQL *UPDATE* optimisé avec jointure

```

update (SELECT sal ,DECODE (loc, 'CHICAGO', 10, 'Toulouse', 31, 0) bonus
       FROM bigdept,bigemp
       WHERE bigdept.deptno = bigemp.deptno
       and loc IN ('CHICAGO', 'Toulouse') )
set sal=sal+bonus;
```

Temps d'exécution : 9,21 s

Listing 7.13 : Version PL/SQL de base

```
declare
  Cursor c1 is select empno,sal,loc from bigemp e,bigdept d
                where e.deptno=d.deptno for update of e.sal;
begin
  for rec in C1 loop
    if rec.loc in ('CHICAGO','Toulouse') then
      update bigemp
        set sal=sal+decode(rec.loc,'CHICAGO',10,'Toulouse',31,0)
        where current of c1;
    end if;
  end loop;
end;
```

Temps d'exécution : 66,07 s

Cette version effectue le filtrage des lignes dans le PL/SQL et ramène des lignes inutilement. Le filtrage des enregistrements dans un bloc PL/SQL au lieu d'une clause WHERE est généralement à bannir.

Listing 7.14 : Version PL/SQL de base avec une clause WHERE dans le curseur

```
declare
  Cursor c1 is select empno,sal,loc from bigemp e,bigdept d
                where e.deptno=d.deptno and loc in ('CHICAGO','Toulouse')
                for update of e.sal;
begin
  for rec in C1 loop
    update bigemp
      set sal=sal+decode(rec.loc,'CHICAGO',10,'Toulouse',31,0) *
      where current of c1;
  end loop;
end;
```

Temps d'exécution : 28,51 s

Nous voyons que, généralement, les instructions SQL simples sont plus performantes que l'équivalent en PL/SQL.

Listing 7.15 : Version utilisant un curseur BULK COLLECT

```
declare
  Cursor c1 is select empno,sal,loc,e.rowid from bigemp e,bigdept d
                where e.deptno=d.deptno and loc in ('CHICAGO','Toulouse');
  TYPE C1RecTab IS TABLE OF c1%rowtype;
  Recs  C1RecTab;
```

```

begin
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO Recs LIMIT 1000;
    EXIT WHEN c1%NOTFOUND;
    for j in Recs.first..Recs.last loop
      update bigemp
        set sal=sal+decode(recs(j).loc,'CHICAGO',10,'Toulouse',31,0)
        where rowid=recs(j).rowid;
    end loop;
  END LOOP;
  CLOSE c1;
end;

```

Temps d'exécution : 20,8 s

On remarque l'utilisation du RowID permettant d'implémenter l'équivalent de la clause `CURRENT OF` de la version précédente.

Listing 7.16 : Version utilisant un curseur *BULK COLLECT* et un *FORALL* avec clause *WHERE* sur le champ *Empno*

```

declare
  Cursor c1 is select empno,sal,loc from bigemp e,bigdept d
                where e.deptno=d.deptno and loc in ('CHICAGO','Toulouse');
  TYPE C1RecTab IS TABLE OF c1%rowtype;
  Recs  C1RecTab;
  TYPE Table_N IS table OF Number ;
  lstepno Table_N:=Table_N();
  lstnewsalary Table_N:=Table_N() ;
  TblIdx integer;
begin
  lstepno.extend(1000);lstnewsalary.extend(1000);
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO Recs LIMIT 1000;
    EXIT WHEN c1%NOTFOUND;
    TblIdx:=1;
    for j in Recs.first..Recs.last loop
      lstepno(TblIdx):=Recs(j).empno;
      if recs(j).loc = 'CHICAGO' then
        lstnewsalary(TblIdx):=Recs(j).sal+10;
      else
        lstnewsalary(TblIdx):=Recs(j).sal+31;
      end if;
      TblIdx:=TblIdx+1;
    end loop;
  end loop;

```

```

        forall j in 1..(TblIdx-1)
            update bigemp set sal=lstnewsalary(j) where empno=lstempno(j);
        END LOOP;
    CLOSE c1;
end;

```

Temps d'exécution : 11,3 s

On remarque un gain de performances assez important avec l'introduction de FORALL. Cependant, si nous regardons le schéma d'exécution de l'instruction UPDATE, nous voyons que l'index PK_BIGEMP est en toute logique mis à contribution alors que l'habituelle clause CURRENT OF travaille directement sur les RowID. La version suivante montre le gain apporté par le travail direct avec les RowID.

Listing 7.17 : Version utilisant un curseur BULK COLLECT et un FORALL avec clause WHERE sur le RowID

```

declare
    Cursor c1 is select sal,loc,e.rowid from bigemp e,bigdept d
                where e.deptno=d.deptno and loc in ('CHICAGO','Toulouse');
    TYPE C1RecTab IS TABLE OF c1%rowtype;
    Recs C1RecTab;
    TYPE Table_N IS table OF Number ;
    TYPE Table_RowId IS table OF Rowid ;
    lstnewsalary Table_N:=Table_N() ;
    lstrowid Table_RowId:=Table_RowId() ;
    SampleSize integer := 1000;
    TblIdx integer;
begin
    lstnewsalary.extend(SampleSize);lstrowid.extend(SampleSize);
    OPEN c1;
    LOOP
        FETCH c1 BULK COLLECT INTO Recs LIMIT SampleSize;
        EXIT WHEN c1%NOTFOUND;
        TblIdx:=1;
        for j in Recs.first..Recs.last loop
            if recs(j).loc = 'CHICAGO' then
                lstnewsalary(TblIdx):=Recs(j).sal+10;
            else
                lstnewsalary(TblIdx):=Recs(j).sal+31;
            end if;
            lstrowid(TblIdx):=Recs(j).rowid;
            TblIdx:=TblIdx+1;
        end loop;
        forall j in 1..(TblIdx-1)
            update bigemp set sal=lstnewsalary(j) where rowid=lstrowid(j);
        END LOOP;
    CLOSE c1;
end ;

```

Temps d'exécution : 9,57 s

Tableau 7.2 : Résumé des versions et des temps d'exécution associés

Méthode	Temps d'exécution (en secondes)
SQL UPDATE	11,32
SQL MERGE	12,81
SQL UPDATE optimisé	9,21
PL/SQL curseur de base sans WHERE	66,07
PL/SQL curseur de base avec WHERE	28,51
PL/SQL select BULK COLLECT	20,80
PL/SQL select BULK COLLECT plus FORALL sur Empno	11,30
PL/SQL select BULK COLLECT plus FORALL sur RowID	9,57

Nous constatons une certaine diversité dans les performances, mais l'introduction des techniques BULK COLLECT et FORALL a un effet indéniable. On arrive même à avoir des performances proches du SQL optimisé en PL/SQL, ce qui n'est pas fréquent étant donné la nature même du PL/SQL qui s'appuie sur le SQL.

INFO

Ces techniques, dites BULK, sont aussi présentes sur certaines API clientes telles que les pré-compilateurs (Pro*C, etc.).

7.4.4 SQL dynamique et BULK

Les méthodes de SQL dynamiques peuvent aussi utiliser les fonctionnalités BULK au moyen des notations suivantes :

```
FORALL i IN <Intervale>
  EXECUTE IMMEDIATE '<Instruction LMD avec des bindings>'
  USING <tableau>(i);
```

Vous pouvez même y insérer une clause RETURNING :

```
FORALL i IN tabdept.first .. tabdept.last
  'DELETE FROM bigemp WHERE DeptNo = :1
  RETURNING Empno INTO :2'
  USING tabdept(i) RETURNING BULK COLLECT INTO tabempno;
```

Dans le cas de curseurs dynamiques, vous pouvez utiliser la syntaxe suivante :

```
OPEN c FOR <variable contenant un select>;
FETCH c BULK COLLECT INTO <tableau>;
CLOSE c;
```

Dans le cas d'un `SELECT INTO` dynamique de type `BULK COLLECT`, vous pouvez utiliser :

```
EXECUTE IMMEDIATE <variable contenant un select>
  BULK COLLECT INTO <tableau>;
```

7.4.5 Traitement des exceptions avec *FORALL*

Il est possible que, lors de l'exécution d'instructions LMD, des exceptions soient levées car des contraintes ne sont pas respectées. Le résultat avec `FORALL` sera le même que si on avait utilisé un curseur : les lignes qui précèdent l'exception sont modifiées mais pas celles qui suivent la ligne à l'origine de l'exception. Ce comportement est parfois acceptable, surtout si vous prévoyez de faire un `ROLLBACK`. Par contre, si votre objectif est de conserver tout ce qui peut se faire et de tracer ce qui pose problème, la solution standard qui interrompt le traitement n'est pas la bonne.

Oracle met à notre disposition le mot clé `SAVE EXCEPTIONS` dans l'instruction `FORALL`. Il a pour effet de stocker les exceptions dans un tableau et de lever l'exception "ORA 24381 : Error in Array DML". Cela conduit à la situation suivante :

- Les exceptions sont disponibles dans le tableau `SQL%BULK_EXCEPTIONS`.
- Le nombre d'exceptions est accessible par `SQL%BULK_EXCEPTIONS.count`.
- La ligne du tableau origine de l'exception est dans `SQL%BULK_EXCEPTIONS(i).error_index`.
- Le code de l'exception est dans `SQL%BULK_EXCEPTIONS(i).Error_code`.

Pour traiter les exceptions, il faut capturer l'exception `ORA-24381` et parcourir le tableau `SQL%BULK_EXCEPTIONS`.

Pour capturer une exception, il faut d'abord la déclarer dans la section `DECLARE` au moyen des instructions suivantes :

```
ex_erreur_dml EXCEPTION;
PRAGMA EXCEPTION_INIT(ex_erreur_dml, -24381);
```

Listing 7.18 : Exemple de traitement des exceptions

```
BEGIN
  forall j in 1..(TblIidx-1) SAVE EXCEPTIONS
    update bigemp set sal=lstnewsalary(j) where rowid=lstrowid(j);
EXCEPTION
  WHEN ex_erreur_dml THEN
    NbrErreur := SQL%BULK_EXCEPTIONS.count;
    DBMS_OUTPUT.put_line(NbrErreur || ' erreurs');
    FOR i IN 1 .. NbrErreur LOOP
      DBMS_OUTPUT.put_line('Erreur: ' || i ||
        ' Index tableau : ' || SQL%BULK_EXCEPTIONS(i).error_index ||
        ' Message: ' || SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
    END LOOP;
END;
```

7.4.6 Utilisation de cache de données

Si une procédure – comme celles qui utilisent des tables de références – risque de rechercher de nombreuses fois les mêmes données dans les tables, il peut être pertinent d'utiliser les techniques de cache, en stockant une copie de la table ou les données récemment utilisées dans une variable tableau de type TABLE.

Illustrons cette technique avec le programme ci-après qui compte le nombre de commandes de clients situés en France en PL/SQL, volontairement sans utiliser de jointure à des fins de démonstration.

Listing 7.19 : Version de base

```
declare
  NbrCmd Number :=0;
  Pays varchar(15);
begin
  for c in (select * from cmd where rownum<100000)
  loop
    select pays into Pays from clients where noclient=c.noclient;
    if Pays='France' then
      NbrCmd:=NbrCmd+1;
    end if;
  end loop;
  dbms_output.put_line('Nbr Cmd = ' || NbrCmd);
end;
```

Listing 7.20 : Version utilisant une variable de cache

```
declare
  NbrCmd Number :=0;
  Pays varchar(15);
  TYPE Table_A15 IS table OF varchar(15) index by BINARY_INTEGER;
  CacheClient Table_A15;
begin
  for c in (select * from cmd where rownum<100000)
  loop
    if CacheClient.exists(c.noclient) then
```

```

        Pays:=CacheClient(c.noclient);
    else
        select pays into Pays from clients where noclient=c.noclient;
        CacheClient(c.noclient):=Pays;
    end if;
    if Pays='France' then
        NbrCmd:=NbrCmd+1;
    end if;
end loop;
dbms_output.put_line('Nbr Cmd = '||NbrCmd);
end;

```

On peut même imaginer donner au cache une durée de vie supérieure à la procédure en cours, en utilisant des variables de package qui ont pour durée de vie la durée de la session. Dans ce cas, il faut particulièrement veiller au problème de validité du cache, car ses données risquent de devenir obsolètes. Cette technique peut avoir pour effet d'occuper beaucoup de mémoire, car chaque association de valeurs va rester en mémoire et, cela, pour chaque exécution de la fonction ou pour chaque session si une variable de package est utilisée.

Oracle 11g apporte une solution complémentaire avec le cache de résultat de fonction. Elle permet de mémoriser en cache le résultat d'une fonction de façon partagée entre les sessions et en limitant les ressources utilisées grâce à un algorithme de type LRU (*Least Recently Used*). Cette fonctionnalité permet aussi de gérer l'invalidation des données en cache à la suite de modifications de table à l'aide de la clause `RELIES_ON`. Celle-ci est optionnelle et Oracle semble pouvoir s'en passer lors de l'analyse du code. Les paramètres de la base `RESULT_CACHE_MAX_SIZE` et `RESULT_CACHE_MAX_RESULT` permettent de gérer la mémoire disponible pour le cache de chaque fonction. Par défaut, chaque fonction peut utiliser au plus 5 % de la mémoire de cache de résultat (`RESULT_CACHE_MAX_SIZE`) qui est fixé à quelques pourcents de la taille de la SGA. Si vous prévoyez d'utiliser intensément cette fonctionnalité, il faudra probablement ajuster les valeurs de ses paramètres.

Listing 7.21 : Version utilisant une fonction avec l'option `RESULT_CACHE`

```

create or replace function GetPaysClient(pNoClient number) return varchar2
RESULT_CACHE relies_on (clients) is
    Pays varchar2(15);
begin
    select pays into Pays from clients where noclient=pnoclient;
    return(Pays);
end GetPaysClient;
/
declare
    NbrCmd Number :=0;
begin
    for c in (select * from cmd where rownum<1000000 order by noclient)
    loop
        if GetPaysClient(c.noclient)='France' then
            NbrCmd:=NbrCmd+1;
        end if;
    end loop;
    dbms_output.put_line('Nbr Cmd = '||NbrCmd);
end;

```


7.4.7 Utilisation du profiling

Le package DBMS_PROFILER permet de tracer l'exécution de code PL/SQL et ainsi de connaître, pour chaque ligne de code, le nombre d'exécutions et le temps total d'exécution. Cette fonction permet de détecter les points critiques pour améliorer les performances d'une procédure. DBMS_PROFILER stocke les résultats dans une table interrogeable *via* SQL. L'idéal est d'utiliser des outils tiers intégrant le *profiling* qui mettent à disposition des rapports comme celui de la Figure 7.4.

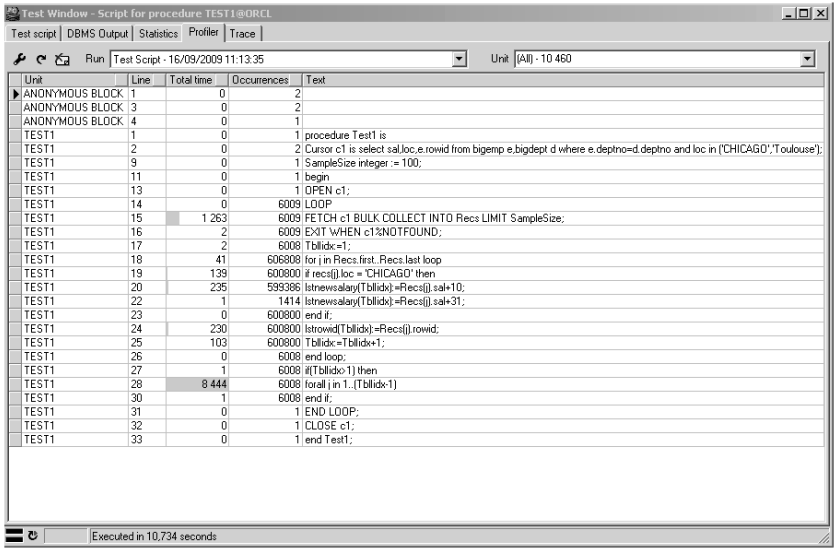


Figure 7.4
Rapport de profiling issu de l'outil PL/SQL Developer d'Allround Automations.

7.4.8 Compilation du code PL/SQL

Depuis de nombreuses versions d'Oracle, il est possible de compiler le code des procédures stockées que vous développez ainsi que celles qui sont livrées en standard. Cependant, cette procédure était assez complexe jusqu'à la version 11g qui a énormément simplifié les choses.

De façon standard, le code PL/SQL est interprété par un runtime du noyau, mais il est possible de le compiler afin d'avoir du code natif à la plateforme. Cette opération, si elle n'apporte pas forcément des gains significatifs, ne peut pas réduire les performances de façon significative. Il convient cependant de ne compiler que ce qui doit l'être. Tout le code exécutant du SQL ne subira aucune amélioration. Si votre procédure exécute principalement du SQL, il n'y a pas d'intérêt à la compiler.

Pour compiler en code natif à partir d'Oracle 11g, il suffit de recompiler la procédure en spécifiant le type de code NATIVE.

```
ALTER PROCEDURE <nomProcedure> COMPILE PLSQL_CODE_TYPE=NATIVE ;
```

Il faut bien veiller à ne pas laisser les informations de débogage (clause DEBUG) pour tirer parti des gains de performances. Le code compilé ne peut pas être profilé ou débogué. L'étape de compilation native ne devra donc avoir lieu qu'après la mise au point du code.

Listing 7.22 : Exemple de procédure effectuant des opérations non SQL et pouvant tirer pleinement profit de la compilation (tri bulle d'une table décroissante)

```
create or replace procedure TestPerf is
  TYPE Table_N IS table OF pls_INTEGER INDEX BY pls_INTEGER;
  Tbl Table_N ;
  Taille pls_INTEGER :=4000;
  modified boolean;
  swap integer;
begin
  -- On remplit le tableau trié en décroissant
  for j in 1..taille
  loop
    Tbl(j):=taille-j;
  end loop;
  -- On tri le tableau
  loop
    modified:=false;
    for j in 1..taille-1
    loop
      if Tbl(j)>Tbl(j+1) then
        swap:=Tbl(j);
        Tbl(j):=Tbl(j+1);
        Tbl(j+1):=swap;
        modified:=true;
      end if;
    end loop;
    exit when not modified;
  end loop;
end TestPerf;
```

Cette procédure en code interprété s'exécute en 4,8 secondes. Après la compilation native, elle le fait en 1,9 seconde, ce qui constitue un gain notable.

La requête suivante permet de connaître le type de code et la présence d'informations de débogage de tout le code PL/SQL de l'utilisateur.

```
SELECT * FROM user_PLSQL_OBJECT_SETTINGS
ORDER BY TYPE, PLSQL_CODE_TYPE;
```

Axe 3

Autres pistes d'optimisation

Optimisation applicative (hors SQL)

En général, une application n'est pas seulement une base de données, mais un applicatif qui utilise une base de données. Côté applicatif, certains choix peuvent avoir de grosses répercussions sur l'interaction avec la base de données. Nous resterons au fil de ce chapitre assez généraliste, car les techniques sont propres à chaque environnement de développement.

8.1 Impact du réseau sur le modèle client/serveur

Si le réseau est peu performant et que l'application soit de type client lourd, on veillera à réduire le nombre de requêtes exécutées sur le serveur, nombre qui peut rapidement exploser avec des relations maître/détails.

On veillera à ne pas ramener des ensembles de données entiers s'il est possible de les ramener petit à petit. Un maximum de filtrages et de regroupements seront faits côté SGBDR afin de limiter les volumes transférés sur le réseau.

8.2 Regroupement de certaines requêtes

L'overhead de l'exécution d'une requête se définit par le temps nécessaire à son exécution qui n'est pas consacré au parcours des données – cela inclut les échanges interprocessus et réseaux, l'analyse, etc. Il est généralement considéré comme faible. Cependant, ce temps peut devenir très significatif par rapport au temps total sur des

requêtes simples. Si beaucoup de requêtes sont exécutées, des problèmes de latence peuvent surgir. L'apparition d'ORM (*Object-Relational Mapping*) et de *framework* a contribué à mettre ce problème en exergue.

Illustrons ce phénomène avec un exemple : imaginons que, dans notre base exemple, nous souhaitions récupérer les informations relatives à un client qui a passé 10 commandes de 5 articles pris dans notre catalogue de livres.

Une solution que pourrait apporter un ORM générant des requêtes SQL peu performantes se décomposerait ainsi :

- 1 requête pour récupérer le client ;
- 1 requête pour récupérer la collection de 10 commandes ;
- 10 requêtes pour récupérer les 5 articles de chacune des commandes ;
- 50 requêtes pour récupérer les 50 articles commandés (s'ils sont tous différents).

Soit 62 requêtes SQL.

Une autre approche, parfois plus efficace, serait :

- 1 requête pour récupérer le client ;
- 1 requête pour récupérer les commandes ;
- 1 requête pour récupérer les items de commandes joints avec les articles.

Soit 3 requêtes SQL.

Comme souvent lorsqu'on parle d'optimisation, la notion de performance dépend de pas mal de choses. Si l'ORM utilise des mécanismes de *lazy loading*, qui permet d'attendre le premier accès effectif à une donnée, il n'exécutera pas forcément toutes les requêtes. Si l'application a la capacité de maintenir un cache des objets, les articles resteront en mémoire et ne seront pas à extraire chaque fois.

Le regroupement de requêtes peut conduire à dupliquer des données et donc à gaspiller des ressources. Par exemple, si les mêmes articles sont présents dans plusieurs commandes, la seconde solution va, inutilement, les ramener plusieurs fois.

Certaines personnes pensent, à tort, que l'utilisation d'un ORM évite d'avoir à se plonger dans la base de données et que tout marchera tout seul. Les ORM n'ont rien de magique. Leurs atouts sont indéniables dans certains environnements, plus discutables dans d'autres. Il est impératif de se pencher sur la configuration des ORM

pour ne pas tomber dans des travers tels que celui décrit précédemment. Les ORM peuvent tout à fait générer du SQL performant s'ils sont correctement configurés. Hibernate, par ailleurs, intègre un langage HQL qui permet de faire des requêtes performantes dans le SGBDR au lieu de parcourir des collections d'objets qu'il faudrait préalablement récupérer.

Le problème de la multiplication des requêtes arrive aussi dans d'autres circonstances. J'ai eu plusieurs fois l'occasion de voir dans des applications des boucles dans l'applicatif là où une jointure aurait été pertinente. Ci-après figure un exemple de code multipliant les requêtes inutilement :

```
$ListeCmd=$db->GetArray("select * from cmd where noclient=$noclient");
foreach($ListeCmd as $cmd)
{
    $LignesCommandes=$db->GetArray("select * from lignes_cmd where
nocmd='".$cmd['nocmd']");
    // Traitement
}
```

8.3 Utilisation du binding

Le binding est un mécanisme qui permet d'exécuter plusieurs fois la même requête en changeant la valeur des paramètres. Sous Oracle, cette technique utilise des paramètres préfixés par le symbole deux points (:). La valeur effective est passée séparément du texte de la requête. Exemple d'une requête utilisant un binding sous Oracle :

```
Select * from clients where Noclient=:P1
```

Cette technique permet d'économiser le temps d'analyse de la requête sur les exécutions suivant la première, ce qui peut être intéressant si une même requête est utilisée de nombreuses fois.

L'exemple ci-après en PHP montre deux solutions possibles :

- La première modifie le texte de la requête et la re-parse donc chaque fois.
- La seconde utilise un binding.

Le résultat est sans appel : on passe de 7,68 secondes à 1,05 seconde soit un facteur 7 pour 10 000 exécutions. Sur 1 000 exécutions, le facteur n'est plus que de 2,5, ce qui est déjà un bon résultat. Bien évidemment, le facteur de gain dépend de la requête exécutée.

Listing 8.1 : Comparaison de requêtes avec et sans binding

```

<?php
$conn = oci_connect('scott', 'tiger', 'orcl');
// Version sans binding
$start= microtime (true);
for($i=0;$i<10000;$i++)
{
    $query = 'SELECT * FROM clients where noclient='.(14797+$i);
    $stid = oci_parse($conn, $query);
    $r = oci_execute($stid, OCI_DEFAULT);
    $row = oci_fetch_row($stid);
    oci_free_statement ($stid);
}
echo microtime (true)-$start;
// Version Avec binding
$start= microtime (true);
$query = 'SELECT * FROM clients where noclient=:client';
$stid = oci_parse($conn, $query);
for($i=0;$i<10000;$i++)
{
    $client=14797+$i;
    oci_bind_by_name ( $stid , 'client', $client );
    $r = oci_execute($stid, OCI_DEFAULT);
    $row = oci_fetch_row($stid);
}
oci_free_statement ($stid);
echo microtime (true)-$start;
oci_close($conn);
?>

```

Dans le cadre de l'utilisation de paramètres bindés sur des prédicats d'intervalles dans une instruction SELECT, les bindings peuvent fausser les estimations car l'optimiseur recourt à des valeurs prédéfinies telles que 5 % pour les intervalles ouverts (>x) et 0,25 % pour les intervalles bornés (>x AND <y) [voir Chapitre 5, section 5.1.3, "Sélectivité, cardinalité, densité"].

Ces estimations pouvant poser des problèmes, la version 10g a introduit la notion de *Bind Peeking* qui a pour effet de déterminer le coût en fonction de la première valeur utilisée. Cette solution donne parfois de meilleurs résultats mais elle a provoqué une levée de boucliers dans la communauté car elle entraînait des instabilités du plan d'exécution utilisé pour une même requête en fonction de la première valeur soumise, ce qui avait un aspect aléatoire.

La version 11g a changé de méthode en introduisant les *Adaptive Cursors*. Cette technique permet de gérer plusieurs plans d'exécution pour une même requête utilisant le binding, l'optimiseur choisissant ensuite le plan le plus adapté à la valeur soumise.

8.4 Utilisation de cache local à l'application

Plutôt que de réinterroger la base de données chaque fois qu'on a besoin d'une donnée de référence, il peut être plus performant d'utiliser un cache local. Comme avec tous les caches, cette solution introduit le problème de la durée de validité des données du cache et des éventuels comportements inadéquats que cela peut entraîner.

8.5 Utilisation du SQL procédural

Parfois, la logique métier exige inévitablement de faire un grand nombre de requêtes. Plutôt que de coder toute la logique du côté client de l'application, il peut être pertinent de la coder avec du SQL procédural (bloc PL/SQL sous Oracle, Transact SQL sous SQL Server). L'avantage de cette solution est que les procédures seront exécutées d'un seul coup sans quitter le noyau, ce qui réduit l'overhead de chaque requête. Les performances sont généralement au rendez-vous s'il y a de nombreuses requêtes.

Au-delà des aspects relatifs aux performances, le SQL procédural est parfois utilisé pour implémenter la séparation d'un applicatif en plusieurs couches. Dans ce cas, la logique métier est implémentée à l'aide de procédures stockées et de packages.

8.6 Gare aux excès de modularité

Parfois, à vouloir trop bien faire, on peut introduire des problèmes de performances. Par exemple, dans un souhait de découplage de deux applications, vous décidez de mettre à disposition dans une des applications un package PL/SQL permettant de manipuler et d'interroger les données. Supposons que ce package contienne les fonctions suivantes :

- `GetCmdClient(NoClient)`. Retourne les commandes d'un client.
- `GetMontantCmd (NoCmd)`. Retourne le montant d'une commande.

Ces fonctions sont adaptées aux opérations pour lesquelles elles ont été définies. Cependant, il se peut que les besoins évoluent et que l'interface ne suive pas. Par exemple, si vous avez besoin de calculer le chiffre d'affaires d'un client, vous allez appeler `GetCmdClient` puis `GetMontantCmd` pour chaque commande. Cette approche

sera moins efficace que de faire une requête SQL calculant directement ce chiffre d'affaires.

De façon générale, la manipulation des bases de données se prête relativement peu à la présentation d'API pour agir sur les données sous-jacentes. Considérez donc les impacts si vous devez recourir à ce genre de mécanisme. L'utilisation de vues (avec les réserves présentées au Chapitre 6, section 6.2.4, "Réutilisation de vue") peut être une solution plus performante que des procédures PL/SQL.

Optimisation de l'infrastructure

9.1 Optimisation de l'exécution du SGBDR

Cette tâche est plutôt dans le périmètre de l'administrateur de la base de données, mais nous abordons ici quelques pistes de base. Des dizaines de paramètres sont disponibles pour adapter le comportement des SGBDR. Il faut les manipuler avec beaucoup de prudence car ils peuvent avoir de lourdes conséquences – seul, un DBA devrait normalement les modifier. Si vous n'avez pas de DBA, les modules AWR et ADDM étudiés au Chapitre 4, section 4.1.3, "Identifier les requêtes qui posent des problèmes", pourront vous donner des pistes d'amélioration.

9.1.1 Ajustement de la mémoire utilisable

La plupart des SGBDR intègrent un paramètre permettant de configurer la quantité maximale de mémoire qu'ils peuvent utiliser. Il faut veiller à ce que ces paramètres n'étouffent pas le serveur (il ne faut pas swapper ces zones), mais il faut que le SGBDR utilise au mieux la mémoire disponible sur le serveur.

- Sous Oracle 10g, le paramètre `sga_target` peut être utilisé.
- Sous Oracle 11g, les paramètres `memory_target` et `memory_max_target` peuvent être utilisés.
- Sous SQL Server, le paramètre `max server memory` peut être utilisé.
- Sous MySQL, de nombreux paramètres sont disponibles parmi lesquels : `key_buffer_size`, `innodb_buffer_pool_size`, `innodb_additional_memory_pool_size`, `innodb_log_buffer_size`, `query_cache_size`.

Normalement, sous Oracle et SQL Server, ces paramètres sont automatiquement configurés en fonction de la mémoire disponible au moment de l'installation. À la suite d'un changement de configuration ou pour des raisons diverses, ils peuvent avoir une valeur inadaptée et ainsi sous-utiliser les ressources disponibles du serveur. J'ai plusieurs fois eu l'occasion de constater chez des clients que la base de données manquait de mémoire alors que le serveur avait encore 1 ou 2 Go de mémoire vive disponible. Le cas le plus récent était sur un serveur destiné à la base de données avec 2 Go de RAM, la mémoire maximale utilisable était configurée à 512 Mo alors que 1,5 Go aurait été plus adapté.

9.1.2 Répartition des fichiers

Répartir les fichiers de données, de contrôle et de journalisation, sur des disques différents améliore généralement les performances en lecture et en écriture. Pour les fichiers de contrôle Oracle, cette pratique est recommandée pour des raisons de fiabilité.

Répartir les données et les index sur des tablespaces qui sont sur des disques différents est une bonne pratique assez répandue.

9.2 Optimisation matérielle

Cette tâche est plutôt dans le périmètre du DBA et de l'administrateur système, cependant nous allons ici survoler quelques pistes.

Un des moyens d'améliorer les performances est d'améliorer le matériel utilisé pour faire tourner le SGBDR. Cette solution devrait être envisagée seulement lorsque toutes les autres ont été épuisées (d'où la localisation de ce chapitre à la fin de l'ouvrage). Malheureusement, pour des raisons de simplicité, c'est souvent une des premières à être mises en œuvre. Nous allons rapidement étudier les impacts de l'amélioration des éléments clés.

9.2.1 Le CPU

De façon générale, l'amélioration du CPU aura toujours un résultat positif, du fait qu'il est un point de passage systématique pour toutes les opérations. Cependant, c'est rarement le point qui résoudra les gros problèmes de performances à moins d'avoir un très vieux serveur.

L'augmentation du nombre de CPU n'aura généralement d'impact significatif que s'il y a plusieurs utilisateurs ou des exécutions de requêtes en parallèle.

9.2.2 La mémoire vive (RAM)

La quantité de RAM sur les serveurs de base de données est souvent le nerf de la guerre. C'est elle qui permet d'augmenter la taille du cache (voir Chapitre 1, section 1.2.6, "Le cache mémoire"). Il est fréquent de voir des systèmes avec plusieurs gigaoctets de RAM (les SGBDR sont les plus demandeurs d'architectures 64 bits).

L'écart de performances entre une requête portant sur un volume de données tenant en RAM et une requête nécessitant de nombreux accès disque est énorme. Pour mémoire :

- Un accès disque moyen est de l'ordre de 10 ms de latence avec un débit de quelques Mo/s.
- Un accès RAM est de l'ordre de quelques nanosecondes avec un débit de quelques Go/s.

L'augmentation de la RAM est souvent l'amélioration matérielle la plus efficace.

9.2.3 Le sous-système disque

Les données viennent des disques et repartent sur les disques une fois qu'elles sont modifiées. Si elles sont trop grosses pour tenir en RAM, les performances du sous-système disque sont critiques.

Une première solution consiste à améliorer les performances intrinsèques des disques :

- vitesse rotation 15 000 tours/min ;
- interface haut débit SCSI/SAS ;
- gros cache interne ;
- disques SSD (*Solid State Disk, Flash*).

Une autre solution consiste à utiliser des systèmes RAID. La mise en œuvre des techniques de mirroring et de stripping permettra d'augmenter les débits. Certains

modes RAID peuvent légèrement dégrader les performances en écriture (RAID 5) et ne sont donc pas recommandés pour les fichiers de journalisation.

Une dernière solution, plus simple à mettre en œuvre, consiste à répartir les données sur plusieurs disques (avec ou sans RAID) afin de maximiser les débits.

9.2.4 Le réseau

Les échanges avec un SGBDR sont généralement composés de très nombreux petits échanges (même s'il peut y en avoir des gros). Le facteur clé côté réseau sera la latence.

Sur un LAN (*Local Area Network*), il n'y a généralement pas de problème, mais l'utilisation d'une application client/serveur sur un WAN (*Wide Area Network*) peut très vite devenir désastreuse.

Conclusion

Au long de cet ouvrage, nous avons étudié de nombreuses solutions pour résoudre des problèmes de performances. Je tiens, dans cette conclusion, à vous rappeler une dernière fois quelques principes.

Dans la vie, toute médaille a son revers :

- Souvent une optimisation aura des conséquences. Le tout est de savoir si elles sont mineures ou pas par rapport au gain apporté.
- Il faudra chaque fois réfléchir aux conséquences possibles et savoir si elles sont acceptables pour votre situation.
- Les conséquences peuvent être de nature différente :
 - espace disque supplémentaire requis ;
 - ralentissement des écritures ;
 - ralentissement d'autres requêtes ;
 - ralentissement de la même requête avec d'autres valeurs ;
 - réduction de la capacité à monter en charge ;
 - etc.

Certains principes sont formidables sur le papier mais, dans la réalité, ils peuvent déboucher sur des contre-performances. Faites preuve de pragmatisme et d'objectivité. Ayez toujours un regard critique sur ce qui se passe quand vous évaluez une optimisation.

Il faut toujours avoir en tête que la nature des données, les volumes concernés, la diversité des utilisateurs et bien d'autres paramètres peuvent déboucher sur des situations autres que celles que nous venons de rencontrer et que cela peut faire mentir les plus belles explications.

Cet ouvrage n'est pas un grimoire de magie plein de formules, mais un livre qui donne des clés pour comprendre les mécanismes des bases de données et les pistes d'amélioration les plus communes.

Le plus important est que vous développiez, au fil du temps, un talent d'analyse, qui vous permettra de comprendre des situations diverses et parfois complexes. L'élément le plus important pour optimiser une base de données, c'est vous.

J'espère que ce livre vous aura aidé à développer votre talent.

Annexes

A

Gestion interne des enregistrements

A.1 Le RowID

Le RowID est une information permettant d'adresser un enregistrement au sein de la base de données (voir Chapitre 1, section 1.2.2, "Le RowID").

Le RowID d'un enregistrement s'obtient en interrogeant la pseudo-colonne rowid et il ressemble à cela AAARnUAAGAAASIfAAG.

Quand les enregistrements sont dans un même bloc, les RowID se suivent. Lors des changements de blocs, les trois derniers caractères repartent de AAA, et la partie précédente du rowid change. On voit ci-après un changement de bloc :

```
SQL> select noclient, nom, prenom, rowid from clients;
-----
NOCLIENT NOM          PRENOM      ROWID
-----
. . . . .
14989 Humphrey        Carlos      AAARdSAAGAAAGg0ABA
14992 Harry           Anna        AAARdSAAGAAAGg0ABB
14995 Beckham         Bobbi       AAARdSAAGAAAGg0ABC
14998 Stone           Mos         AAARdSAAGAAAGg0ABD
15001 Foster          Boyd        AAARdSAAGAAAGg0ABE
15004 Iglesias         Lance       AAARdSAAGAAAGg0ABF
15007 Webb            Tia         AAARdSAAGAAAGg0ABG
15010 Viterelli        Hope        AAARdSAAGAAAGg0ABH
15013 Dern            Hope        AAARdSAAGAAAGg0ABI
15016 Sepulveda        Al          AAARdSAAGAAAGg1AAA
15019 Marshall        Pierce      AAARdSAAGAAAGg1AAB
15022 Leoni            Donna       AAARdSAAGAAAGg1AAC
15025 Navarro          Candice     AAARdSAAGAAAGg1AAD
15028 Krieger          Rip         AAARdSAAGAAAGg1AAE
```

15031 Waite	Winona	AAARdSAAGAAAg1AAF
15034 Sylvian	Carolyn	AAARdSAAGAAAg1AAG
15037 Paige	Curtis	AAARdSAAGAAAg1AAH
.		

Le package DBMS_ROWID permet de manipuler et de décoder ces données, comme dans cet exemple :

```

DECLARE
ridtyp NUMBER;
objnum NUMBER;
relfno NUMBER;
blno    NUMBER;
rowno   NUMBER;
rid     ROWID;
infos   varchar2(200);
tabspace varchar2(200);
BEGIN
  SELECT rowid INTO rid FROM bigemp where empno=52900;

  dbms_rowid.rowid_info(rid,ridtyp,objnum,relfno,blno,rowno);
  select t.object_type||' '||t.owner||'. '||t.object_name,ta.tablespace_name
  into infos,tabspace
  from sys.dba_objects t,sys.dba_tables ta
  where t.object_id=objnum and t.owner=ta.owner
  and t.object_name=ta.table_name;

  dbms_output.put_line('Type RowID:' || TO_CHAR(ridtyp));
  dbms_output.put_line('No Objet : ' || TO_CHAR(objnum) || ' - ' ||infos||'
  Tablespace:' ||tabspace);
  select file_name into infos from sys.dba_data_files
  where relative_fno=6 and tablespace_name=tabspace;
  dbms_output.put_line('Datafile : ' || TO_CHAR(relfno) || ' - ' ||infos);
  dbms_output.put_line('No Block : ' || TO_CHAR(blno));
  dbms_output.put_line('No ligne : ' || TO_CHAR(rowno));
END;
```

Le résultat est le suivant :

```

Type RowID:1
No Objet :71232 - TABLE SCOTT.BIGEMP Tablespace:BIG_TABLESPACE
Datafile :6 - E:\ORACLE\ORADATA\ORCL\BIG_TABLESPACE1
No Block :75965
No ligne :2
```

A.2 Row Migration et Row Chaining

Nous détaillons ici les mécanismes de Row Migration et de Row Chaining (voir Chapitre 1, section 1.2.5, "Row Migration et Row Chaining") en les étudiant de façon pratique au moyen de la table suivante :

```
CREATE TABLE row_mig_chain_demo (
  cle int PRIMARY KEY,
  col1 VARCHAR2(4000),
  col2 VARCHAR2(4000),
  col3 VARCHAR2(4000),
  col4 VARCHAR2(4000) );
```

Création de deux enregistrements qui tiennent dans le même bloc

```
INSERT INTO row_mig_chain_demo (cle,col1) VALUES (1,lpad('A',3000,'X'));
INSERT INTO row_mig_chain_demo (cle,col1) VALUES (2,lpad('A',3000,'X'));
select t.cle,t.rowid from row_mig_chain_demo t;
CLE  ROWID
---  -----
  1  AAASGpAAHAAAIOnAAA
  2  AAASGpAAHAAAIOnAAB
```

Si on analyse les données de cette table :

```
analyze table ROW_MIG_CHAIN_DEMO compute statistics;
SQL> select table_name,num_rows,blocks,empty_blocks,avg_space,chain_cnt,avg_
row_len
  2  from sys.user_tables where table_name='ROW_MIG_CHAIN_DEMO';
```

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
ROW_MIG_CHAIN_DEMO	2	5	3	6869	0	3009

on constate que, dans Chain_Cnt, il n'y a pas de chaînage dans cette table.

Nous allons agrandir un enregistrement afin que les deux ne puissent plus tenir dans le bloc de 8 Ko :

```
UPDATE row_mig_chain_demo SET col2 = lpad('A',4000,'X') WHERE cle = 1;
```

L'analyse des données de cette table :

```
analyze table ROW_MIG_CHAIN_DEMO compute statistics;
SQL> select table_name,num_rows,blocks,empty_blocks,avg_space,chain_cnt,avg_
row_len
  2  from sys.user_tables where table_name='ROW_MIG_CHAIN_DEMO';
```

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
ROW_MIG_CHAIN_DEMO	2	5	3	6059	1	5013

montre que, à présent, Chain_Cnt=1. Il y a donc eu Row Migration.

Si on "défragmente" la table :

```
alter table row_mig_chain_demo move;
```

et qu'on analyse de nouveau les données de la table :

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
ROW_MIG_CHAIN_DEMO	2	5	3	3033	0	5010

on voit que Chain_Cnt revient à 0. Il n'y a plus de chaînage et les données sont dans deux blocs distincts, comme le montrent les RowID ci-après.

```
select t.cle,t.rowid from row_mig_chain_demo t;
CLE  ROWID
---
1  AAASGuAAHAAAI01AAA
2  AAASGuAAHAAAI0KAAA
```

Si on agrandit une ligne au point qu'elle ne puisse plus tenir dans un bloc :

```
UPDATE row_mig_chain_demo SET col3 = lpad('A',3000,'X') WHERE cle = 1;
UPDATE row_mig_chain_demo SET col4 = lpad('A',3000,'X') WHERE cle = 1;
```

et qu'on analyse une nouvelle fois les données de la table :

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
ROW_MIG_CHAIN_DEMO	2	8	0	4841	1	8016

nous voyons qu'un chaînage réapparaît dans la colonne Chain_Cnt. Si on "défragmente" la table à nouveau, il y a toujours un chaînage (voir ci-après). Cela est dû au fait que les données ne peuvent pas tenir dans un bloc unique car leur taille est supérieure. Le chaînage sera permanent.

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
ROW_MIG_CHAIN_DEMO	2	6	2	2688	1	8025

B

Statistiques sur les données plus en détail

Comme nous l'avons vu au Chapitre 5, section 5.1, "Statistiques sur les données", les statistiques sont un élément fondamental pour l'utilisation du CBO (*Cost Based Optimiser*). Nous allons étudier ici plus en détail la nature des statistiques utilisées par Oracle.

B.1 Statistiques selon l'ancienne méthode de collecte

L'ancienne méthode de collecte des statistiques les stocke dans les tables du dictionnaire de données, grâce à l'instruction suivante.

```
analyze table <NomTable> compute statistics;
```

Les statistiques sont consultables avec les requêtes suivantes :

Listing B.1 : Statistiques au niveau de la table

```
select t.table_name,t.num_rows,t.blocks, t.empty_blocks,t.avg_space,t.chain_
cnt,t.avg_row_len
from sys.user_tables t where t.table_name like 'BIG%';
```

TABLE_NAME	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVG_SPACE	CHAIN_CNT	AVG_ROW_LEN
BIGDEPT	400004	1630	0	0	0	22
BIGEMP	1400014	10097	0	0	0	44

On y voit le nombre de lignes et de blocs utilisés ainsi que la taille moyenne d'une ligne. Ces informations contribuent à estimer le nombre d'entrées/sorties disque.

Listing B.2 : Statistiques au niveau des index

```
select t.index_name, t.num_rows ,t.blevel, t.leaf_blocks, t.distinct_keys ,t.
avg_leaf_blocks_per_key, t.avg_data_blocks_per_key,t.clustering_factor
from sys.user_indexes t where t.table_name like 'BIG%';
```

INDEX_NAME	NUM_ROWS	BLEVEL	LEAF_BLOCKS	DISTINCT_KEYS	AVG_LEAF_BLOCKS_PER_KEY	AVG_DATA_BLOCKS_PER_KEY	CLUSTERING_FACTOR
PK_BIGEMP	1400014	2	3103	1400014	1	1	9502
IS_BIGEMP_DEPT	1400014	2	3298	301920	1	1	38416
PK_BIGDEPT	400004	2	886	400004	1	1	1540
IS_BIGDEPT_LOC	400004	2	1375	400004	1	1	6261

Listing B.3 : Statistiques au niveau des colonnes

```
select t.table_name,t.column_name,t.num_distinct,t.low_value,t.high_value
,t.density,t.num_nulls
from sys.user_tab_cols t where t.table_name like 'BIG%'
order by t.table_name,t.column_id;
```

TABLE_NAME	COLUMN_NAME	NUM_DISTINCT	LOW_VALUE	HIGH_VALUE	DENSITY	NUM_NULLS
BIGDEPT	DEPTNO	400004	C10B	C40B010129	2,49997E-6	0
BIGDEPT	DNAME	4	4143434F554E54494E47	53414C4553	0,25	0
BIGDEPT	LOC	5	424F53544F4E	546F756C6F757365	0,2	0
BIGEMP	EMPNO	1400014	C24A46	C50201015023	7,14278E-6	0
BIGEMP	ENAME	14	4144414D53	57415244	0,0714285	0
BIGEMP	JOB	5	414E414C595354	53414C45534D414E	0,2	0
BIGEMP	MGR	597824	C24C43	C50201015003	1,67273E-6	100001
BIGEMP	HIREDATE	13	77B40C11010101	77BB0517010101	0,07692307	0
BIGEMP	SAL	34	C211	C3020F1B	0,02941176	0
BIGEMP	COMM	4	80	C20F	0,25	1000010
BIGEMP	DEPTNO	301920	C10B	C40B01011F	3,31213E-6	0

Ces données serviront principalement à estimer l'impact de chaque condition.

B.2 Statistiques selon la nouvelle méthode de collecte

À présent, il est recommandé d'utiliser le package DBMS_STATS qui donne un niveau d'information plus élevé. Les statistiques sont stockées dans des tables dédiées.

Listing B.4 : Statistiques DBMS_STATS sur les tables

```
select * from sys.user_tab_statistics
where table_name='BIGEMP'
```

<i>Nom champ</i>	<i>Valeur</i>
TABLE_NAME	BIGEMP
PARTITION_NAME	
PARTITION_POSITION	
SUBPARTITION_NAME	
SUBPARTITION_POSITION	
OBJECT_TYPE	TABLE
NUM_ROWS	1400014
BLOCKS	10097
EMPTY_BLOCKS	0
AVG_SPACE	0
CHAIN_CNT	0
AVG_ROW_LEN	44
AVG_SPACE_FREELIST_BLOCKS	0
NUM_FREELIST_BLOCKS	0
AVG_CACHED_BLOCKS	
AVG_CACHE_HIT_RATIO	
SAMPLE_SIZE	1400014
LAST_ANALYZED	27/03/2010 11:00:44
GLOBAL_STATS	YES
USER_STATS	NO
STATTYPE_LOCKED	
STALE_STATS	YES

Listing B.5 : Statistiques *DBMS_STATS* sur les index

```
select * from sys.user_ind_statistics
where table_name='BIGEMP'
```

<i>Nom champ</i>	<i>Valeur</i>
INDEX_NAME	PK_BIGEMP
TABLE_OWNER	SCOTT
TABLE_NAME	BIGEMP
PARTITION_NAME	
PARTITION_POSITION	
SUBPARTITION_NAME	
SUBPARTITION_POSITION	
OBJECT_TYPE	INDEX
BLEVEL	2
LEAF_BLOCKS	3103
DISTINCT_KEYS	1400014
AVG_LEAF_BLOCKS_PER_KEY	1
AVG_DATA_BLOCKS_PER_KEY	1
CLUSTERING_FACTOR	9502
NUM_ROWS	1400014
AVG_CACHED_BLOCKS	
AVG_CACHE_HIT_RATIO	
SAMPLE_SIZE	1400014
LAST_ANALYZED	27/03/2010 11:00:49
GLOBAL_STATS	YES
USER_STATS	NO
STATTYPE_LOCKED	
STALE_STATS	YES

Listing B.6 : Statistiques *DBMS_STATS* sur les colonnes

```
select * from sys.user_tab_col_statistics t
where t.table_name like 'BIG%';
```

<i>Nom champ</i>	<i>Valeur</i>
TABLE_NAME	BIGEMP
COLUMN_NAME	EMPNO
NUM_DISTINCT	1400014
LOW_VALUE	C24A46
HIGH_VALUE	C50201015023
DENSITY	7,14278571499999E-7
NUM_NULLS	0
NUM_BUCKETS	1
LAST_ANALYZED	27/03/2010 11:00:23
SAMPLE_SIZE	1400014
GLOBAL_STATS	YES
USER_STATS	NO
AVG_COL_LEN	6
HISTOGRAM	NONE

Les tables contenant des statistiques DBMS_STATS sont :

DBA_TABLES, DBA_OBJECT_TABLES, DBA_TAB_STATISTICS
DBA_TAB_COL_STATISTICS, DBA_TAB_HISTOGRAMS, DBA_INDEXES
DBA_IND_STATISTICS, DBA_CLUSTERS, DBA_TAB_PARTITIONS
DBA_TAB_SUBPARTITIONS, DBA_IND_PARTITIONS,
DBA_IND_SUBPARTITIONS, DBA_PART_COL_STATISTICS,
DBA_PART_HISTOGRAMS,
DBA_SUBPART_COL_STATISTICS, DBA_SUBPART_HISTOGRAMS

Avec leurs déclinaisons USER_* et ALL_*

Au-delà de l'optimisation, certaines de ces statistiques peuvent être utiles pour prendre en main une base de données. En effet, elles aident à repérer facilement les plus grosses tables, les colonnes qui ne sont jamais remplies, etc.

B.3 Histogrammes

Nous détaillons ici les informations stockées dans les histogrammes étudiés au Chapitre 5, section 5.1.3, "Sélectivité, cardinalité, densité".

Listing B.7 : Statistiques *DBMS_STATS* sur les histogrammes de fréquence

```
select  endpoint_number-nvl(lag(endpoint_number,1)OVER (ORDER BY endpoint_
number ),0) TailleIntervalle
        ,endpoint_number, endpoint_value
from user_tab_histograms t
where table_name='BIGDEPT' and column_name = 'LOC'
```

La colonne TailleIntervalle est calculée à partir des valeurs courante et précédente du Endpoint.

endpoint_value est une représentation numérique de la chaîne de caractères.

<i>TailleIntervalle</i>	<i>endpoint_number</i>	<i>endpoint_value</i>
99900	99900	344 300 505 052 090 000 000 000 000 000 000 000
99900	199800	349 350 027 483 572 000 000 000 000 000 000 000
99900	299700	354 400 587 944 790 000 000 000 000 000 000 000
99900	399600	406 405 544 089 997 000 000 000 000 000 000 000
404	400004	438 413 586 837 071 000 000 000 000 000 000 000

L'histogramme de distribution coupe les données en n tranches égales et détermine les bornes des tranches.

Listing B.8 : Statistiques *DBMS_STATS* sur les histogrammes de distribution

```
select  endpoint_number, endpoint_value from user_tab_histograms
where table_name='BIGDEPT' and column_name = 'DEPTNO';
```

<i>endpoint_number</i>	<i>endpoint_value</i>
0	10
1	1000010
2	2000020

<i>endpoint_number</i>	<i>endpoint_value</i>
3	3000030
4	4000040
5	5000040
6	6000040
7	7000040
8	8000040
9	9000040
10	10000040

Normalement, le SGBDR détecte les colonnes sur lesquelles il est pertinent de calculer un histogramme. Cependant, vous pouvez forcer les histogrammes avec l'instruction suivante :

```
execute dbms_stats.gather_table_stats(user,'bigdept',cascade => true
,estimate_percent =>100,method_opt => 'for all columns size 100');
```

C'est le SGBDR qui déterminera le type d'histogramme en fonction du nombre de valeurs. Il est possible que votre histogramme forcé disparaisse lors de la prochaine collecte automatique des statistiques.

B.4 Facteur de foisonnement (*Clustering Factor*)

Nous allons étudier ici l'impact du facteur de foisonnement.

La constitution de la table BIGEMP a été consécutive à la création séquentielle des enregistrements ; la table BIGEMP2 a été créée à la suite du tri des données par la colonne Ename. Les séquences d'Empno sont donc réparties de façon à occuper successivement un 1/14 de tables différentes¹.

```
create table bigemp2 as select * from bigemp order by ename;
alter table bigemp2 add constraint PK_bigemp2 primary key(empno);
execute dbms_stats.GATHER_TABLE_STATS('SCOTT','BIGEMP2');
```

1. Rappel : la table BIGEMP est 100 000 fois la répétition des 14 lignes de la table EMP avec des offsets sur les Empno et Deptno.

Listing B.9 : Statistiques des index de clé primaire de *BIGEMP* et *BIGEMP2*

```
select t.INDEX_NAME,t.LEAF_BLOCKS,t.DISTINCT_KEYS,t.CLUSTERING_FACTOR from
sys.user_ind_statistics t
where index_name like 'PK_BIGEMP%';
```

INDEX_NAME	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTOR
PK_BIGEMP	3103	1400014	9502
PK_BIGEMP2	3103	1400014	1399950

On voit que l'index sur BIGEMP a un bon facteur de foisonnement puisqu'il y a en moyenne trois liens par feuille, alors que le deuxième index a presque autant de liens que d'enregistrements. Dans ce cas, dans chaque feuille, chaque clé pointe sur un bloc du tas de la table différent.

Démonstration par la pratique : cette requête, exécutée sur les deux tables, met un critère sur Empno pour utiliser l'index et un autre sur la colonne Ename afin de forcer un accès au tas, sinon seul l'index serait utilisé vu que nous effectuons un count (*).

```
Select count(*) from BigEmp
where empno between 10000000 and 20000000
and ename='MILLER';
```

Figure B.1

Sur table BIGEMP avec index ayant un faible taux de foisonnement (temps d'exécution 160 ms).

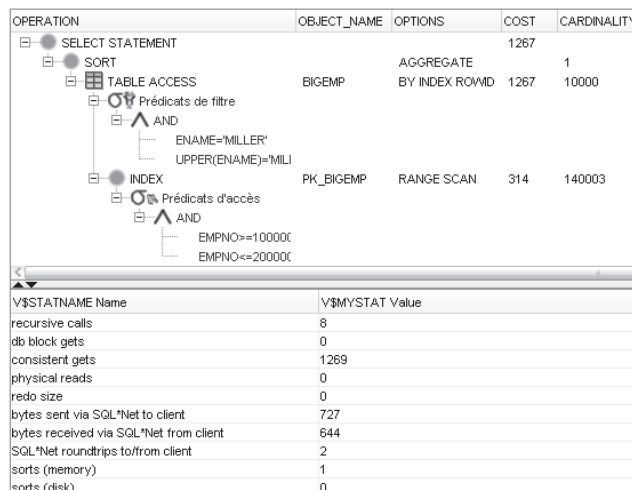


Figure B.2

Sur table BIGEMP2 avec index ayant un fort taux de foisonnement (temps d'exécution 440 ms).

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINAL
SELECT STATEMENT			140434	
SORT		AGGREGATE		1
TABLE ACCESS	BIGEMP2	BY INDEX ROWID	140434	10000
Prédicats de filtre				
AND				
ENAME='MILLER'				
UPPER(ENAME)=MILI				
INDEX	PK_BIGEMP2	RANGE SCAN	384	140003
Prédicats d'accès				
AND				
EMPNO>=100000				
EMPNO<=200000				

V\$STATNAME Name	V\$MYSTAT Value
recursive calls	8
db block gets	0
consistent gets	140384
physical reads	0
redo size	0
bytes sent via SQL*Net to client	730
bytes received via SQL*Net from client	663
SQL*Net roundtrips to/from client	2
sorts (memory)	1
sorts (disk)	0

On voit ici l'impact d'un mauvais Clustering Factor (Timing $\times 3$ et Consistent Gets $\times 100$).

L'optimiseur redoute tellement ces résultats qu'il n'utilise pas l'index dès que le clustering factor est mauvais. D'ailleurs, pour obtenir ce plan, nous avons dû insérer le hint `index_rs` dans la requête. Sans cela, l'optimiseur choisirait de faire un Full Table Scan qui provoque moins de Consistent Gets (9 600) mais un temps d'exécution de 1,37 seconde.

```
Select /*+index_rs(e)*/ count(*) from BigEmp2 e
where empno between 10000000 and 20000000
and ename='MILLER'
```


Scripts de création des tables de test BIGEMP et BIGDEPT

Les tables de test nous permettent d'effectuer des tests sur une base simple mais de taille suffisamment importante pour mettre en évidence l'intérêt des optimisations. La base LIVRE est téléchargeable sur le site de l'auteur <http://www.altidev.com/livres.php>. Sous Oracle, les bases BIGDEPT et BIGEMP sont générées à l'aide des scripts suivants.

Ces tables sont fondées sur les tables exemples d'Oracle présentes dans le schéma SCOTT de la base d'exemple mais dupliquées 100 000 fois, dans laquelle nous avons introduit quelques valeurs particulières.

Je vous conseille de les créer dans un tablespace séparé, par exemple un tablespace nommé BIG_TABLESPACE qui aurait une taille de 1 Go.

N'hésitez pas à adapter les tailles à votre environnement, pour que cela soit le plus significatif sans que chaque test soit trop long.

Listing C.1 : Fichier CreateBig.SQL

```
-- Création de la table BigDept
Create Table bigDEPT (
  DEPTNO      Number ,
  DNAME       VARCHAR2(14),
  LOC         VARCHAR2(13) )
Tablespace BIG_TABLESPACE ;
-- Remplissage de la table
begin
  for i in 0..100000
  loop
    insert into bigdept
```

```

        select i*100+deptno, dname, decode(mod(i, 1000),0,'Toulouse',loc)
        from dept;
end loop;
commit;
end;
/
-- Ajout de la clé primaire
alter table bigDEPT add constraint PK_bigDEPT Primary Key(Deptno);

-- Création de la table BigEmp
CREATE TABLE bigEMP (
EMPNO      NUMBER ,
ENAME      VARCHAR2(10) NOT NULL CHECK (ename=UPPER(ename)),
JOB        VARCHAR2(9),
MGR        NUMBER ,
HIREDATE   DATE,
SAL        NUMBER(7,2) CHECK (SAL > 500 ),
COMM       NUMBER(7,2),
DEPTNO     NUMBER NOT NULL )
Tablespace BIG_TABLESPACE ;

-- Insertion des lignes
begin
for i in 0..100000
loop
    insert into bigemp
    select i*1000+empno, ename, job, i*1000+mgr, hiredate, sal, comm,
i*100+deptno
    from emp;
end loop;
commit;
end;
/

-- Ajout de la clé primaire et des foreign key
alter table bigEMP add (
constraint PK_BIGEMP primary key(empno),
constraint fk_bigemp_dpt foreign key (deptno) REFERENCES bigdept (deptno),
constraint fk_bigemp_emp foreign key (mgr) REFERENCES bigemp (empno)
);

```

Si vous n'avez pas le schéma SCOTT sur votre base Oracle, et donc pas les tables DEPT et EMP, voici le script pour les recréer.

Listing C.2 : Fichier CreateEmpDept.SQL

```

Create Table DEPT (
DEPTNO     Number(2,0) constraint PK_DEPT Primary Key,
DNAME      VARCHAR2(14),
LOC        VARCHAR2(13));

```

```
Insert into DEPT values ('10','ACCOUNTING','NEW YORK');
Insert into DEPT values ('20','RESEARCH','DALLAS');
Insert into DEPT values ('30','SALES','CHICAGO');
Insert into DEPT values ('40','OPERATIONS','BOSTON');
Commit;
```

```
CREATE TABLE EMP (
EMPNO      NUMBER(4) PRIMARY KEY,
ENAME      VARCHAR2(10) NOT NULL CHECK (ename=UPPER(ename)),
JOB        VARCHAR2(9),
MGR        NUMBER(4) REFERENCES emp (empno),
HIREDATE   DATE,
SAL        NUMBER(7,2) CHECK (SAL > 500 ),
COMM       NUMBER(7,2),
DEPTNO     NUMBER(2) NOT NULL REFERENCES dept (deptno) );
```

```
Insert into EMP values (7839,'KING','PRESIDENT',NULL ,TO_
DATE('17/11/1981','DD/MM/YYYY'),5000,NULL,10);
Insert into EMP values (7566,'JONES','MANAGER',7839 ,TO_DATE('02/04/1981',
'DD/MM/YYYY'),2975,NULL,20);
Insert into EMP values (7698,'BLAKE','MANAGER',7839 ,TO_DATE('01/05/1981',
'DD/MM/YYYY'),2850,NULL,30);
Insert into EMP values (7782,'CLARK','MANAGER',7839 ,TO_DATE('09/06/1981',
'DD/MM/YYYY'),2450,NULL,10);
Insert into EMP values (7902,'FORD','ANALYST',7566 ,TO_DATE('03/12/1981',
'DD/MM/YYYY'),3000,NULL,20);
Insert into EMP values (7369,'SMITH','CLERK',7902 ,TO_DATE('17/12/1980',
'DD/MM/YYYY'),800,NULL,20);
Insert into EMP values (7499,'ALLEN','SALESMAN',7698 ,
TO_DATE('20/02/1981','DD/MM/YYYY'),1600,300,30);
Insert into EMP values (7521,'WARD','SALESMAN',7698 ,TO_DATE('22/02/1981',
'DD/MM/YYYY'),1250,500,30);
Insert into EMP values (7654,'MARTIN','SALESMAN',7698 ,
TO_DATE('28/09/1981','DD/MM/YYYY'),1250,1400,30);
Insert into EMP values (7788,'SCOTT','ANALYST',7566 ,TO_DATE('19/04/1987',
'DD/MM/YYYY'),3000,NULL,20);
Insert into EMP values (7844,'TURNER','SALESMAN',7698 ,TO_
DATE('08/09/1981','DD/MM/YYYY'),1500,0,30);
Insert into EMP values (7876,'ADAMS','CLERK',7788 ,TO_DATE('23/05/1987',
'DD/MM/YYYY'),1100,NULL,20);
Insert into EMP values (7900,'JAMES','CLERK',7698 ,TO_DATE('03/12/1981',
'DD/MM/YYYY'),950,NULL,30);
Insert into EMP values (7934,'MILLER','CLERK',7782 ,TO_DATE('23/01/1982',
'DD/MM/YYYY'),1300,NULL,10);
commit;
```

D

Glossaire

DBA (*DataBase Administrator*). Désigne les personnes qui sont chargées de l'administration de la base de données.

DDL (*Data Definition Language*). Voir LDD.

DML (*Data Manipulation Language*). Voir LMD.

E/S (Entrées/Sorties). Désigne les échanges vers le sous-système de stockage, c'est-à-dire généralement les disques durs. On peut aussi parler d'E/S réseau pour désigner les échanges de données vers d'autres machines à travers le réseau.

Entité. Désigne une table dans la terminologie du modèle entité/association.

I/O (*Input/Output*). Voir E/S.

Index. Principal objet d'optimisation des performances, largement abordé dans cet ouvrage.

Heap (*tas*). Type d'organisation des données.

HOT (*Heap Organized Table*). Désigne les tables organisées en tas.

LDD (langage de définition des données). Sous-ensemble du SQL couvrant les instructions permettant de modifier les structures des données, principalement les instructions CREATE, ALTER et DROP.

LMD (langage de manipulation des données). Sous-ensemble du SQL couvrant les instructions permettant de modifier les données, principalement les instructions INSERT, UPDATE et DELETE.

LOB. Désigne les types de grande capacité (Large Object); ils peuvent être textuels ou binaires.

OLAP (*OnLine Analytical Processing*). Désigne les bases de données dites "analytiques", servant généralement à l'analyse des données. Elles contiennent généralement des données issues de bases de données OLTP. Elles en diffèrent par leur structure, le volume ou le niveau de détails des informations qu'elles contiennent.

OLTP (*OnLine Transaction Processing*). Désigne les bases de données dites "transactionnelles", servant généralement pour le traitement opérationnel des données.

Oracle RAC (*Real Application Cluster*). Désigne l'utilisation d'Oracle sur plusieurs serveurs qui partagent un disque dur.

RDBMS (*Relational Database Management System*). Voir SGBDR.

Relation. Désigne une table dans la terminologie du modèle relationnel tel qu'il a été établi par Edgar Frank Codd.

SGBDR (système de gestion de base de données relationnelle). Désigne un système (logiciel) qui permet de gérer une base de données relationnelle.

Table. Structure principale permettant de stocker les données dans une base de données.

Index

A

Analytique 194
ANALYZE 78
Anti-jointure 156
Autotrace 51
AWR 62

B

Binding 219
Buffer cache 13
BULK COLLECT 200

C

Cache de données 210
Cache mémoire 13
Cardinalité 80
CBO 16
Cluster 132
Clustered Index 127
Clustering Factor 98
Compilation 212
Compression
 Index 94
 index bitmap 101
 LOB 121
 Table 119
Consistent Gets 53
CONTAINS 113
CONTEXT 112
CTXCAT 113
Cube Group By 189

D

Datafile 7
DBMS_APPLICATION_INFO 68
DBMS_MONITOR 68
DBMS_PROFILER 212
DBMS_STATS 78
Dénormalisation 38
Densité 80
Direct Path 198
Données, cache de 210
DYNAMIC_SAMPLING 82

E

explain plan for 55
Extent 8

F

Facteur de foisonnement 98
Fast Full Scan 57
fonction analytique 194
FORALL 203
Full Table Scan 56

G

Grouping Sets 186

H

Hash Join 58
HAVING 162
Heap 11, 87
Hint 177
Histogramme 83

I

Index 14, 85
 bitmap 101
 Bitmap Join 108
 B*Tree 86
 clustered 127
 colonnes incluses 97
 composite 90
 compression 94
 fonction 99
 full text 113
 hints 180
 multicolonnes 90
 reconstruction 150
 Reverse 100
IOT 122

L

LIKE 80, 112

M

Materialized Views 146
MCD 23
Médiane 194
mémoire, cache 13
MERGE 195, 196
Merge Join 58
MLD 24
MPD 24

N

Nested Loop 57
NOLOGGING 165
Normalisation 33

O

Optimizer goal 17
Or Expansion 154

P

PARRALEL QUERY 184
Parsing 15
Partitionnement 135
PCTFREE 118
Physical Reads 53
Plan d'exécution 15, 55
PL/SQL 200
 compilation 212
Predicate push 154
Profiling 212

Q

Query Rewriting 147
Query Transformation 154

R

Range Scan 57
Ranking 192
Reconstruction
 index 150
 table 149
requêtes, réécriture 153
Rollup Group By 188
Row Chaining 12
RowID 10
Row Migration 12

S

Segment 8
Sélectivité 80
SGBDR 6
Skip Scan 57, 91
SQL Access Advisor 66
SQL Trace 67
SQL Tuning Advisor 66
Statistiques 77
 étendues 84
Statspack 62
Subquery unesting 154

T

Tablespace [7](#), [118](#)
Tas [11](#), [87](#)
tkprof [69](#)
Trace Analyzer [71](#)
Transformation de requêtes [154](#)
Trigger [198](#)
Typage [25](#)

U

Unique Scan [57](#)

V

V\$sql [61](#)
View merging [154](#)
Vue matérialisée [146](#)

W

WITH [191](#)

Optimisation des bases de données

Mise en œuvre sous Oracle

Cet ouvrage a pour objectif de mettre à la portée des développeurs les connaissances utiles à l'optimisation des bases de données. Cette activité est souvent confiée aux administrateurs de bases de données (DBA) une fois que les projets sont terminés, alors que c'est au niveau du développement qu'il faut se pencher sur la problématique des performances.

De manière claire et pragmatique, l'auteur expose les différentes techniques en les présentant en situation. Pour chacune d'elles, il montre à l'aide d'un cas concret ce qu'elle améliore et dans quel contexte elle agit efficacement. En homme du terrain, il les compare et prend parti.

L'ouvrage se fonde pour une grande part sur le système de bases de données Oracle (versions 9i, 10g, 11g Release 1&2), toutefois des parallèles sont fait régulièrement avec Microsoft SQL Serveur (versions 2005 et 2008) et MySQL (version 5.1) par le biais d'encadrés et de paragraphes dédiés. Les techniques présentées pour ces trois systèmes sont communes à de nombreux autres SGBDR, le lecteur pourra ainsi appliquer les conseils de ce livre à quasiment toutes les bases de données relationnelles du marché.

TABLE DES MATIÈRES

- Introduction aux SGBDR
- Modèle relationnel
- Normalisation, base du modèle relationnel
- Méthodes et outils de diagnostic
- Techniques d'optimisation standard au niveau base de données
- Techniques d'optimisation standard des requêtes
- Techniques d'optimisation des requêtes avancées
- Optimisation applicative (hors SQL)
- Optimisation de l'infrastructure
- Annexes

À propos de l'auteur :

Laurent Navarro est un développeur d'application de base de données depuis plus de 15 ans. Codirigeant de la société Altidev, il accompagne des industriels dans le développement d'applications de gestion et d'informatique industrielle.

Programmation

Niveau : Intermédiaire
Configuration : Multiplate-forme

PEARSON

Pearson Education France
2 rue des Vinaigriers
75004 Paris
01 43 05 11 17
www.pearson.fr

ISBN : 978-2-7440-4156-3

