



M1 STIC

SGBD

D'après une histoire originale de T. Grison

Systemes de gestion de bases de données

NOTES DE COURS

Sommaire

1	Gestion des données	7
1	Architecture globale d'Oracle	7
1	Exécution d'une requête	7
2	Requêtes d'extraction de données	7
3	Requêtes de mise à jour	9
4	Logs	9
5	Mémoire centrale	9
2	Organisation de fichiers	10
1	Organisation séquentielle	10
a	Avantages et inconvénients	10
b	Primitives	10
2	Organisation indexée	11
a	Introduction	11
b	Propriétés des index	11
c	La réalité	13
d	Création d'un index sous Oracle	13
e	Primitives	14
f	Propriétés d'un B-arbre	14
g	Organisation ISAM	14
h	Organisation VSAM	15
i	Organisation IS3	17
3	Organisation aléatoire	18
a	Organisation hash-code classique	18
b	Organisation hash-code dynamique	18
c	Organisation hash-code dynamique linéaire	19
3	Oracle	19
1	Schéma conceptuel de base de données sous Oracle	19
a	Table (ou relation)	19
b	Attributs et domaines	19
2	Organisation générale d'Oracle	21
3	Organisation des données sur disque	21
a	Organisation séquentielle	22
b	Organisation indexée	22

c	Organisation en clusters	22
d	Organisation en index secondaires	23
e	Organisation en index bitmap	24
f	Organisation en reverse key index	24
4	Tablespaces	25
a	Définition et création sur Oracle	25
b	Utilité	25
c	Renommer un <i>datafile</i>	26
5	Segments	27
a	Gestion des segments avec Oracle	28
b	Types de segments	28
c	Rollback segments	28
d	Segments de données	29
e	Gestion des lignes dans un bloc de données	30
f	Segments temporaires	31
2	Algorithmes des opérateurs de l'algèbre relationnelle	33
1	Projection π	33
2	Sélection σ (restriction)	33
1	Balayage séquentiel	33
a	Présentation	33
b	Coût	33
c	Algorithme	34
2	Balayage <i>via</i> un index	34
a	Présentation	34
b	Coût	34
3	Choix par Oracle	35
3	Jointure \bowtie	35
1	Algorithme du produit cartésien	35
2	Algorithme sort-merge	36
a	Algorithme de tri fusion externe	36
b	Algorithme	37
3	Algorithme key-lookup (ou nested loop ou boucles imbriquées)	37
4	Algorithme par hachage	38
5	Algorithme de jointure sur deux index secondaires	39
3	Optimisation de requêtes	40
1	Nécessité de l'optimisation	40
2	Optimisation <i>rule-based</i>	41
3	Optimisation <i>cost-based</i>	42
4	Choix par Oracle	42
1	Changement du choix global	42
2	Changement du choix pour une requête	43

4	Faire joujou avec Oracle	44
1	Dictionnaire de données	44
1	Tables	44
2	Vues	44
3	Vues couramment utilisées	45
2	Requêtes à essayer	46
1	Paramétrer la sortie	46
2	Regarder le résultat d'une fonction	46
3	Afficher le schéma de la base de données	47
4	Joujou sur tablespaces	47
a	Afficher les tablespaces dans lequel l'utilisateur U possède des tables ou des index	47
b	Afficher, pour chaque tablespace, le nombre d'extents libres, leur taille moyenne, leurs optimums de taille, leur taille totale	47
c	Afficher, pour chaque tablespace, et par intervalle de tailles le nombre d'extents libres	47
5	Chercher les tables pour lesquelles un nouvel extent ne pourra pas être alloué	48
6	Calculer la taille prise par les tables utilisateur dans chaque tablespace de la base	48
7	Calculer la taille prise par les données utilisateur d'une table T(a,b,c)	49
8	Calculer la taille prise par les données de gestion d'une table T(a,b,c)	49
5	Transactions, concurrence	50
1	Notion de transaction	50
2	Problématique posée par la concurrence	50
3	Verrouillage	52
1	Graphe de précédence	52
2	Protocole de verrouillage	52
3	Verrous mortels	54
4	Gestion de la concurrence	55
A	Résumé des séances	56
1	Cours magistral	56
2	Travaux dirigés	56
1	B-tree	56
2	Fichiers à organisation indexée	57
3	Fichiers à organisation indexée sous INGRES (B-tree)	57
4	Organisation aléatoire dynamique	57
5	Schéma du dictionnaire de données	57
6	Requêtes sur le dictionnaire de données	57
7	Organisation aléatoire partitionnée	57
8	Jointure	57

9	Optimisation de requêtes	58
10	Concurrence dans les SGBD	58
3	Travaux pratiques	58
B	Utiliser SQL*Plus	59
1	Utiliser le serveur Oracle de l'université	59
1	Connexion à une session	59
a	Depuis la fac	59
b	Depuis un autre endroit	59
c	Suite et fin	60
2	Préparatifs	60
3	Connexion au serveur Oracle	60
2	Rendre SQL*Plus vivable	61
1	Utiliser un éditeur externe	61
2	Facilités de la ligne de commande	61
C	Bibliobus	63

Pour commencer

L'objet de ce cours est d'étudier ce qui se produit derrière les actions de l'utilisateur dans un SGBD, à travers l'exemple d'Oracle. Par exemple, lorsque l'utilisateur crée une table, le SGBD stocke les informations dans un fichier, il y a donc une gestion complexe des fichiers, des concurrences d'accès...

Parmi les différents SGBD, on trouve deux catégories importantes. Premièrement, les haut-de-gamme : Oracle, Postgres, MySQL, Ingres, SQL Server, DB2, Informix. Suivent les grosses merdes : Paradox, DBase, Access, InterBase.

Court historique

En l'an de grâce 1974, IBM inventa SYSTEM-RCIBM. Un système concurrent du nom d'INGRES fut introduit en 1976 par des universitaires de Berkeley. Dans les années 80, d'autres systèmes firent leur apparition, dont Oracle, construit à partir des cendres de SYSTEM-RCIBM, le successeur de ce dernier, et DB2 d'IBM. INGRES continuait à survivre. Depuis on a connu des centaines d'ersatz plus ou moins dégueulassement faits comme Access.

Objectifs du cours

Nous allons essayer de répondre à plusieurs questions. Comment sont stockées les données ? Comment sont exécutés les opérateurs de l'algèbre relationnelle ? Comment sont exécutées les requêtes prises dans leur totalité ? Comment sont faits la gestion des accès concurrents et les optimisations de requêtes ? Autant de questions qui ne trouveront jamais de réponses car vous allez détruire ce document **maintenant** ! MAINTENANT !

Gestion des données

SECTION



Architecture globale d'Oracle

On va plus principalement se pencher sur Oracle, alors regardons un peu la figure 1.1. Penchez-vous encore, encore, voilà c'est parfait. Maintenant détendez-vous et pensez à quelque chose d'agréable. Ça ne durera qu'un court instant.

Database datafiles est l'endroit qui contient les tables et d'autres informations et la mémoire centrale contient des éléments indispensables.

1 Exécution d'une requête

Lorsqu'une requête est exécutée par l'utilisateur par le biais de ce formidable outil de destruction impossible à utiliser sans penser au suicide qu'est **SQL*Plus**¹, elle est envoyée au serveur qui vérifie la syntaxe puis la sémantique de celle-ci, c'est-à-dire qui vérifie si les objets concernés existent et si la requête a un sens. Si la requête est correcte, elle est traduite en procédural (pour pouvoir donner le résultat) et elle est stockée dans les *Context Area*. Si la même requête est exécutée plusieurs fois, le serveur ne la traite pas étant donné que la procédure existe déjà. Dans une *Context Area* cohabitent une zone privée et une zone publique. Cette distinction peut être utile pour cacher certaines informations à certains utilisateurs.

2 Requêtes d'extraction de données

Lorsqu'on lance une requête d'extraction de données comme SELECT, soit ces informations sont déjà en mémoire centrale, dans le *Database buffer cache*, et dans ce cas la mémoire les renvoie à l'utilisateur, soit l'information n'y est pas et la mémoire la récupère dans un *Database datafile* sur le disque, grâce au processus *Server*, qui met tout ça en mémoire centrale puis le chie sur l'utilisateur.

¹ Nous verrons comment le rendre utilisable vers la page 61.

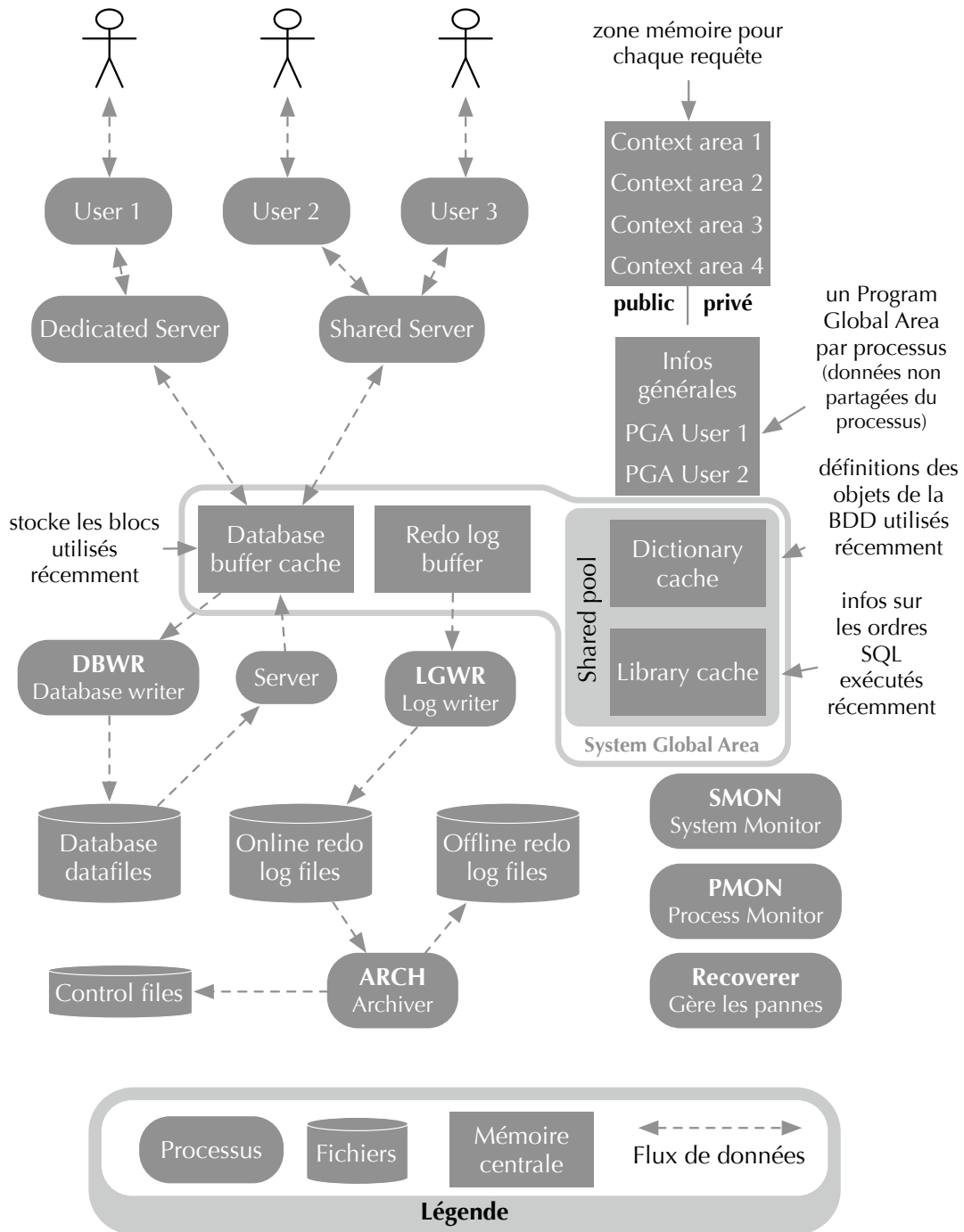


Figure 1.1. Architecture générale du SGBD Oracle

Notez que si les *Database datafiles* stockent aussi toutes les données de la base telles que les structures des tables et index, ce sont les *control files* qui contiennent les structures physiques et les états de la base de données.

3 Requetes de mise à jour

Lorsqu'il s'agit d'une requête de mise à jour comme INSERT ou UPDATE, le serveur va chercher l'information concernée, que ce soit dans la mémoire centrale ou dans un *Database datafile*, puis réécrit l'information changée ou créée dans *Database datafiles* avec le processus *DBWR*. Deux processus différents sont utilisés selon les requêtes car le processus *Server* fonctionne en temps réel, mais pas le processus *DBWR*. Ceci pourrait poser un problème si par exemple une panne machine ou une coupure de courant survient et que l'information a été modifiée mais pas enregistrée. Pour résoudre ce problème, cette information est gardée par *LGWR* dans un Redo log. Après la panne, il suffit de rejouer la requête (avec un *rollback segment*) s'il y a eu un *commit*.

4 Logs

Online redo log files concerne un nombre défini de fichiers, disons trois. Oracle les utilise de façon circulaire (le premier, le second, le troisième, puis on réutilise le premier en écrasant tout, etc). Il faut donc archiver les logs via le processus *ARCH*.

5 Mémoire centrale

La mémoire centrale est aussi appelée mémoire vive, primaire ou volatile. Vous en déduirez ses caractéristiques : elle est rapide et non-permanente. À l'opposé, la mémoire secondaire est aussi appelée mémoire secondaire, permanente, externe, stable, non volatile ou persistante. Elle est lente relativement à la mémoire centrale, mais les données y résident de manière permanente.

Dictionary cache est le cache des dictionnaires des données, c'est-à-dire les champs des tables, mais pas le contenu. *Dictionary cache* et *Database buffer cache* remplissent deux fonctions différentes ! En effet, le *dictionary cache* est responsable du stockage du dictionnaire de données². Les tailles des différentes zones de la mémoire centrale sont paramétrables sous Oracle. Avec Access, par contre, on ne sait rien du tout tellement c'est de la daube. Sous Oracle la taille conseillée pour le *Dictionary cache* est la moitié de celle du *Database buffer cache*.

Une zone PGA est allouée à chaque processus utilisateur et contient les variables liées à la session de connexion.

Vous trouverez tout plein de variables définissant la taille des différentes zones mémoire dans le fichier `init.ora`.

² Le dictionnaire de données, ou métabase, désigne la partie de la base de données contenant les méta-données, c'est-à-dire le schéma de la base, c'est-à-dire toutes les informations sur la structure de la base.

SECTION
2

Organisation de fichiers

Une organisation est le mode de placement des enregistrements dans un fichier. Il faut se souvenir qu'en bases de données, une relation ne correspond pas à un fichier. On trouvera plusieurs types d'organisation de fichiers :

- organisation séquentielle ;
- organisation indexée ;
- organisation aléatoire (hashcode) de deux sortes :
 - classique ;
 - organisation aléatoire dynamique.

1 Organisation séquentielle

Les enregistrements sont mis bout-à-bout (séquentiellement) où il y a de la place. Oracle utilise cette organisation par défaut lorsqu'il n'y a ni clause `CLUSTER` ni clause `ORGANIZATION INDEX`, et elle sert encore bien souvent car c'est la moins coûteuse. Or, la fragmentation est possible : lorsqu'un enregistrement est créé, on réserve de la place en mémoire, mais lorsqu'un enregistrement est supprimé sa place mémoire est libérée et on peut la réutiliser, ainsi on génère de la fragmentation. Dans ce cas on met une partie de l'enregistrement à un endroit et le reste ailleurs.

La clause `STORAGE` et les paramètres `PCTUSED` et `PCTFREE` contrôlent l'allocation et le remplissage des blocs. C'est un sujet que nous verrons en détail vers les pages 27 et 30.

On peut construire des index secondaires sur les tables à organisation séquentielle.

a Avantages et inconvénients

L'organisation séquentielle est économe en place, performante, simple, rapide, répond à de nombreux besoins, permet de tout faire, sent bon et peut ramener la paix dans le monde. Par contre, lorsqu'on cherche un enregistrement, il faut tout parcourir jusqu'à l'avoir trouvé. C'est la raison d'exister des autres types d'organisation de fichiers.

b Primitives

Il y a trois opérations basiques :

Création d'un fichier `creerFichier`

Écriture `ecrSeq(fichier,enregistrement) → fichier`

Lecture `lecSeq(fichier, 1er/suiv) → enregistrement, code_err`

2 Organisation indexée

Avant de parler d'index, on va tout simplement définir ce que c'est. Une alternative à la recherche séquentielle d'un enregistrement est la recherche associative, c'est-à-dire qu'on va accéder à un enregistrement à partir d'une clé d'accès qui peut être un champ ou une combinaison de champs de l'enregistrement, par exemple chercher un client par son numéro de client, ou des étudiants par leur ville. Une structure de données qui permet de retrouver l'adresse d'un tuple à partir d'une valeur de la clé d'accès est appelée un **index**, et on appelle la clé **clé d'index**.

Il y a une raison pour laquelle on doit quelquefois utiliser les index. Avec une organisation séquentielle, la recherche à partir d'une valeur de champ nécessite au pire de parcourir toute la table, donc avec une complexité de $\mathcal{O}(n)$. Cela peut être handicapant quand le nombre de lignes d'une table est très grand, et qu'on doit lire en mémoire secondaire. C'est pourquoi Dieu a inventé l'indexation et le hachage. Grâce à tout ce bordel, on peut atteindre une complexité de $\mathcal{O}(\log n)$ et même $\mathcal{O}(1)$ avec le hachage³. Le deal ici est de transférer le nombre minimal de blocs en mémoire secondaire, et idéalement uniquement ceux qu'on cherche. L'indexation permet de se balader vite dans un intervalle de valeurs de clé, alors que le hachage n'est pas terrible pour cela. On verra pourquoi.

Vous vous douterez bien que si l'organisation indexée accélère grandement la recherche, en revanche, les opérations de modification de la table (ajout, modification, suppression d'une ligne...) prennent plus de temps en fonction du nombre d'index, car le SGBD s'amuse à mettre à jour les index correspondant à la table modifiée.

a Introduction

Pour créer un index, il faut un objet à indexer, pour nous c'est un fichier. Voici le fichier "CLIENT". On crée un index sur ville. L'index est créé pour retrouver facilement et directement un enregistrement à partir de la ville. Notre index sera une table à deux colonnes. Une colonne avec les villes (le champ "ville" de la relation CLIENT) et l'autre colonne accueillera les adresses physiques de la ligne du fichier qui correspond. Du coup, on n'a pas à charger tout le fichier en mémoire centrale mais juste l'index : c'est plus rapide. Dans l'absolu un index n'est pas forcément trié mais on les trie toujours pour obtenir des temps d'accès beaucoup plus courts.

Le point négatif des index, c'est que s'ils sont très avantageux pour rendre très rapides les recherches dans un fichier, les mises à jour sont très lentes.

b Propriétés des index

Plaçant Un index est primaire (ou plaçant) ou secondaire (ou non-plaçant). Quand un index est primaire, c'est qu'il sera trié dans le même ordre que le champ de l'index. Quand il est secondaire, le fichier restera comme il était.

Les index primaires peuvent améliorer les requêtes du style de :

³ Ça vous met le soldat au garde-à-vous, non ?

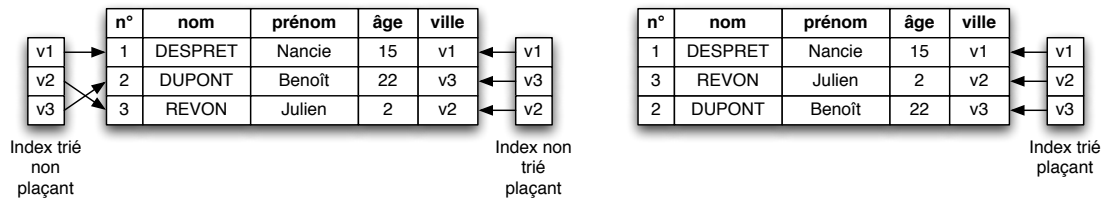


Figure 2.1. Le fichier CLIENT et des index divers

```
1 SELECT * FROM clients WHERE ville='v3';
```

De nos jours, c'est le SGBD qui décide s'il est utile d'utiliser l'index ou pas. Le SGBD est moins bon qu'un programmeur qui pourra forcer le SGBD à utiliser l'index même si le SGBD considèrerait que ce n'était pas nécessaire de s'en servir.

Les lignes de la table et l'index proprement dit sont tous placés dans un même arbre B+.

Attention ! Un index primaire est différent d'une clé primaire : il peut y avoir des doublons dans un index primaire, pas dans une clé primaire. Oracle ne permet de créer des index primaires que sur des clés primaires.

Densité Un index est dense s'il contient autant de valeurs que le champ de la table. Un index non-plaçant est toujours dense. Par contre, un index plaçant peut être non-dense. En effet, on peut ne stocker qu'une fois la valeur et parfois même pas. Par exemple, si on ne stocke que v_1 , v_3 et v_5 et qu'on cherche v_2 , on sait de toute façon qu'il sera entre v_1 et v_3 , donc on n'a qu'à faire un parcours séquentiel à ce niveau et on les trouvera.

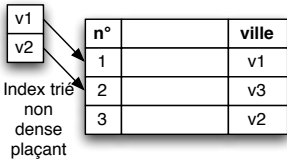


Figure 2.2. Un index non dense

Hiérarchie Un index est hiérarchisé : c'est un arbre, en réalité. Ce qui se passe, c'est que lorsque l'index est trop gros, il est indexé à son tour par un second niveau non-dense (oui, si c'était dense, ça ne servirait à rien).

Ordre Un index sur a et b est différent d'un index sur b et a. En effet, la distinction impose un ordre de recherche (d'abord a puis ensuite b).

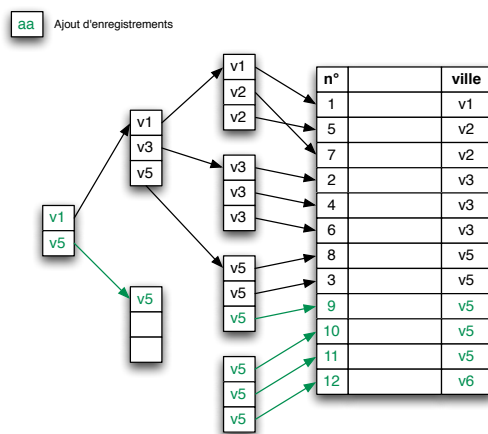


Figure 2.3. Un index hiérarchisé

c La réalité

Mettons qu'on veuille indexer un fichier non trié. Notre index est un fichier et est toujours trié sur la clé. Cet index est dense puisque tous les enregistrements doivent être représentés. Si on a un fichier d'un million d'enregistrements, pesant 120 mégaoctets, et qu'on fait un index sur un champ de 4 octets, alors en supposant qu'une adresse soit codée sur 8 octets l'index pèserait $1000000 * (4 + 8) = 12$ mégaoctets. C'est déjà plus léger à mettre en mémoire centrale, mais c'est déjà assez gros. Un index peut atteindre des dimensions suprenantes et c'est pour ne pas saturer la mémoire centrale qu'il est hiérarchisé.

En pratique, on économise de la place par rapport à un index secondaire puisque l'index peut être non-dense et on n'a pas besoin de stocker physiquement les ROWID des lignes de la table dans la partie index⁴. Cette organisation utilise des ROWID logiques, et il n'est pas possible de stocker des index primaires dans un cluster.

d Création d'un index sous Oracle

index secondaire On fait juste cela :

- 1 **CREATE INDEX** nom_index
- 2 **ON** nom_table(attr1, attr2...);

index primaire La clause **ORGANIZATION INDEX** et la présence d'une contrainte de clé primaire permet la création d'une table organisée en index :

- 1 **CREATE TABLE** nom_table (col1, col2...)
- 2 **PRIMARY KEY** (attr1, attr2...)
- 3 **ORGANIZATION INDEX**;

⁴ On me signale en régie qu'un ROWID, c'est dix octets par ligne.

e Primitives

```

1 lecInd(fichier, val_ind, ordre, nom_idx) {
2   si(ordre = 1er) alors
3     rech_val_ind(fichier.indexes(nom_idx), val_ind) enregistrement,
      code_erreur
4   sinon
5     rech_val_ind.suivant(fichier.index(nom_idx), val_ind) enregistrement,
      c_err
6   fin si
7   si(c_err != pas_enregistrement)
8     lec_adr(fichier.enregistrement, adr_err) enregistrement, c_err
9   fin si
10 }

```

Ecriture on ne spécifie pas l'index car il est contenu dans l'enregistrement
 ecrInd(fichier, enr) → fichier, code_err

f Propriétés d'un B-arbre

Nous allons bientôt jouer avec des B-arbres. Ce sont des arbres qui sont tout le temps équilibrés⁵. Il faut connaître ses propriétés :

- c'est un arbre dont toutes les branches ont la même longueur ;
- il a un ordre m ;
- toutes les feuilles et les nœuds de l'arbre, sauf la racine, peuvent contenir entre m et $2m$ valeurs (dans des entrées), donc être remplies à au moins 50% ;
- chaque nœud non terminal a autant de fils que de valeurs ;
- les valeurs sont réparties dans l'arbre de façon croissante, de gauche à droite et de bas en haut.

Lorsqu'on ajoute une valeur, on fait en sorte de respecter les propriétés, et donc de réorganiser l'arbre. On verra le détail des opérations sur une variante de B-arbre

Il y a des variantes multiples mais nous utiliserons fréquemment l'**arbre B+** dans lequel les feuilles contiennent toutes les données indexées, et sont reliées à leurs voisines, et l'**arbre B*** dans lequel les feuilles et nœuds de l'arbre ne doivent pas être remplies à 50% mais à 66% (aux deux tiers).

g Organisation ISAM⁶

Une organisation ISAM, c'est comme un carnet d'adresses (la structure est statique). Dans l'organisation ISAM, l'index est trié, non dense, plaçant (le fichier est trié physique-

⁵ Le B pourrait signifier *balanced*.

⁶ Index Sequential Access Method

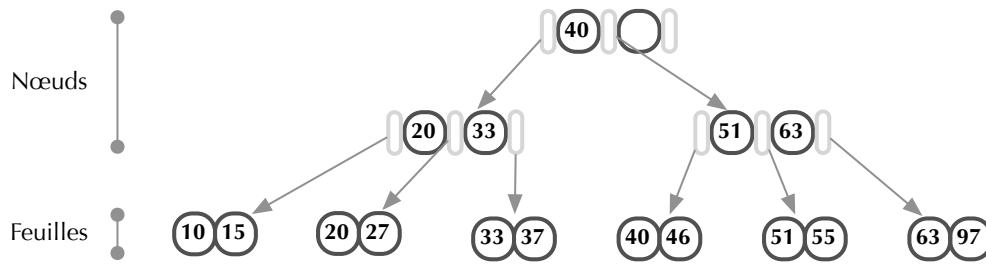


Figure 2.4. Un arbre ISAM

ment) et hiérarchisé. De plus, l'index est statique : on ne change pas le nombre et le niveau des feuilles.

Il y a une zone de débordement globale à la fin du fichier proprement dit et il y a toujours un certain pourcentage d'espace laissé vide dans chaque bloc : une zone de débordement local. Oui, les fichiers sont des ensembles de blocs contigus. L'index est un arbre maintenu à côté, de telle sorte que si on veut faire un accès séquentiel, c'est toujours possible.

La recherche commence à la racine et on fait une comparaison de clé pour arriver aux feuilles. Pour une relation R comportant B_R blocs et T_R tuples, avec un index comportant B_I blocs, le coût de recherche est $\log_{\frac{T_R}{T_I}} B_R$.

L'insertion consiste à rechercher le bloc des données correspondant à l'endroit où on veut insérer, puis à insérer l'entrée à l'endroit qu'on a trouvé.

La suppression consiste à rechercher la ligne puis à supprimer son bloc feuille dans l'index. Si jamais on en vient à vider un bloc de débordement, on le supprime.

Dans les opérations d'insertion et de suppression, seules les feuilles de l'arbre sont affectées, le reste de la structure est fixe.

Insertion et suppression Voyons un exemple d'ajout et de suppression. Soit un arbre ISAM à la figure 2.4. Nous allons insérer les entrées d'index 23, 48, 41 et 42 pour obtenir l'arbre de la figure 2.5. Supprimons maintenant les entrées d'index 42, 51 et 97 et nous obtenons l'arbre de la figure 2.6. Plutôt simple, hein ?

h Organisation VSAM ⁷

Dans l'organisation VSAM, l'index est plaçant, hiérarchisé, non dense, et les enregistrements et l'index sont organisés dans un seul B-arbre. On appelle même ça un arbre B+. Contrairement à l'ISAM, ici la structure de l'arbre peut être chamboulée à chaque mise à jour, dans l'optique de garder l'arbre constamment équilibré.

Comment jouer avec ce genre d'arbres ? C'est pas si simple que ça.

⁷ Virtual Sequential Access Method

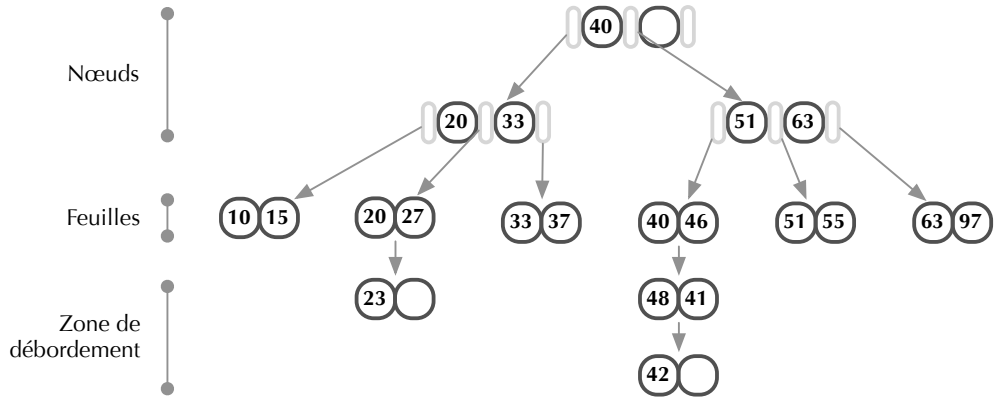


Figure 2.5. Un arbre ISAM, le retour

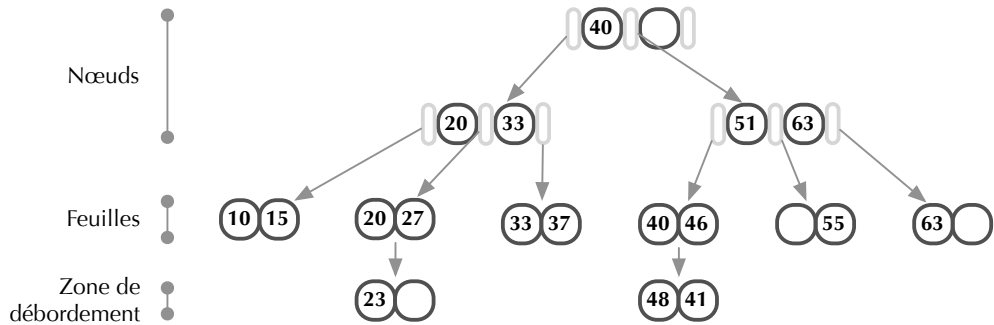


Figure 2.6. Un arbre ISAM, la revanche

Insertion Voyons les étapes nécessaires à l'**insertion** d'un élément dans un arbre B+ :

- On recherche la bonne feuille L .
- On met la nouvelle entrée d'index dans L :
 - Si L a assez d'espace, c'est terminé.
 - Sinon, il faut couper L en L et $L2$:
 - Il faut partager les entrées de L entre les deux feuilles : on prend les entrées à partir de l'entrée médiane m (celle du milieu) jusqu'à la fin et on les déplace dans $L2$.
 - On *insère* une entrée m au niveau au-dessus pour pointer sur $L2$ (on la fait remonter, ou on peut dire qu'elle est promue).
 - Lorsque m était une feuille, alors on a juste à copier cette entrée médiane au-dessus (*copy-up*).
 - Lorsque m était un nœud intermédiaire, alors on doit par contre la déplacer au niveau au-dessus (*push-up*).

C'est un mécanisme qui est récursif, puisque l'insertion d'une entrée au niveau supérieur peut nécessiter une autre coupe, et ce jusqu'à la mort ! L'ajout de nœuds fait grossir l'arbre, et le fait de couper le nœud racine en deux rajoute un niveau à la hauteur de l'arbre.

Suppression Voyons maintenant les étapes nécessaires à la **suppression** d'un élément dans un arbre B+ :

- À partir de la racine, on recherche la bonne feuille L qui contient notre entrée à supprimer.
- On supprime l'élément :
 - Si L est au moins à moitié pleine, c'est terminé.
 - Sinon, c'est que L a $n - 1$ éléments (n étant l'ordre de l'arbre) :
 - On essaie de redistribuer les entrées des feuilles sœurs de L (donc qui ont le même parent) : cela implique que les entrées parentes seront changées
 - Si la redistribution ne donne rien (si L a toujours $n - 1$ éléments), alors on *fusionne* L avec le nœud voisin. Cela a pour effet de *supprimer* l'entrée d'index pointant sur L .

On a aussi affaire à de la récursion puisque la suppression d'un parent peut aussi engendrer le fait qu'il ait moins d'éléments que permis, et ce jusqu'à la fin des temps. La suppression de nœuds fait maigrir l'arbre (ce gros sac), et le fait de fusionner les nœuds peut remonter jusqu'à la racine et faire baisser d'un niveau l'arbre.

i Organisation IS3

C'est l'organisation des index secondaires sous Oracle. Seul l'index est dans un B-arbre, et cet index est évidemment... dense !

3 Organisation aléatoire

L'organisation aléatoire, c'est un mode d'adressage réalisé au moyen d'une fonction aléatoire (ou hash-code). On utilise donc une fonction h qui calcule, à partir de la clé d'accès, l'adresse d'un paquet (ou *bucket*) contenant l'enregistrement. En général, un paquet c'est un bloc du fichier.

a Organisation hash-code classique

La forme de hachage la plus simple est l'organisation hash-code statique. On doit connaître la plage d'adresses dont on dispose, c'est-à-dire le nombre de blocs B qu'on se donne. Les fonctions de hachage les plus courantes sont celles du style $h(v) = v \bmod B$. Par exemple, si $B = 3$, alors on aura une fonction $h(v) = v \bmod 3$, et l'enregistrement de clé 10 sera mis dans le bloc $h(10) = 10 \bmod 3 = 1$. La figure 2.7 résume l'organisation aléatoire statique.

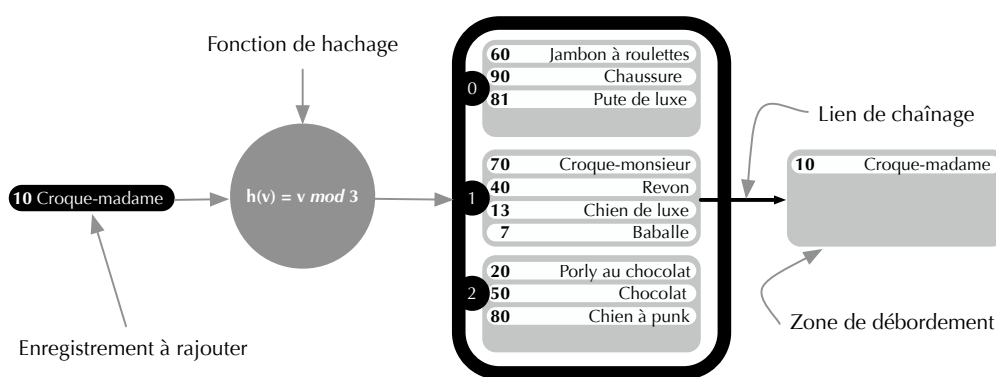


Figure 2.7. Organisation aléatoire classique

Bien qu'on puisse penser le contraire quand on est petit, il y a forcément un moment où il n'y a plus d'espace dans le bloc pour accueillir de nouveaux enregistrements. Lorsque deux valeurs de clé aboutissent à la même adresse, alors on dit qu'il y a **collision**. Quand il y a trop de collisions dans un bloc et qu'il n'y a plus de place, alors on dit qu'il y a un **débordement**. Lorsque cela se produit, on crée une zone de débordement qu'on va lier au bloc en débordement par un **lien de chaînage**.

Une bonne fonction hash-code est une fonction qui provoque le moins de collisions possible et utilise tout l'espace d'adressage alloué.

b Organisation hash-code dynamique

Des gens en ont eu marre de la contrainte d'espace d'adressage fixe du hachage statique. Avec l'organisation hash-code dynamique, le nombre de blocs alloués à la structure et la fonction de hachage sont adaptés dynamiquement aux variations du volume de données, afin de maintenir une bonne performance et un taux d'occupation d'espace raisonnable.

On part sur les mêmes bases que précédemment, mais on ajoute la notion de *répertoire d'adresses* qui est un tableau d'endroits où trouver les blocs : les indices sont les valeurs prises par h et les cases du tableau pointent vers un bloc particulier. Deux cases peuvent pointer vers le même bloc.

En pratique, quand on doit ajouter une valeur de clé à un bloc saturé, on augmente son répertoire d'adresses et on change la fonction de hachage. Par exemple on passe de $h(v) = v \bmod 2^i$ à $h(v) = v \bmod 2^{i+1}$. Il est bien sûr nécessaire de récupérer toutes les valeurs de clé du bloc plein et de leur réappliquer la nouvelle fonction de hachage. Si la valeur de clé ne va plus dans le bloc on change le répertoire d'adresses et on pointe sur un nouveau bloc alloué.

Si on ajoute souvent des valeurs d'index qui ont la même image par h , il vaut mieux faire des liens de chaînage, sinon on a trop de réorganisations qui ne servent à rien.

c Organisation hash-code dynamique linéaire

À faire !

SECTION

3

Oracle

1 Schéma conceptuel de base de données sous Oracle

a Table (ou relation)

La création d'une table en SQL est faite par la commande suivante :

```

1 CREATE TABLE nomtable (
2   colonne1 typec1 [contraintes_integrite_colonne1],
3   colonne2 typec2 [contraintes_integrite_colonne2],
4   ...,
5   [contraintes_integrite_table]
6 )
7 [clauses_politique_alloc_espace];

```

On peut modifier une table avec cette commande :

```

1 ALTER TABLE nomtable (
2   {ADD | MODIFY | DROP | RENAME} colonne, ...);

```

b Attributs et domaines

Chaque attribut (ou colonne) possède un type, ou domaine. Un domaine est l'ensemble des valeurs que peut prendre l'attribut. Il y a toutes sortes de choses magiques qu'on peut utiliser !

Chaînes de caractères

char(taille) Chaîne de caractères de taille fixe (2000 octets maximum) dont la valeur est éventuellement complétée par des blancs

varchar(taille) Chaîne de caractères de taille variable (standard SQL)

varchar2(taille) Chaîne de caractères de taille variable (spécifique à Oracle, 4000 octets maximum)

Nombres

number(nb_chiffres, nb_decimales) Un nombre en virgule flottante (ou non, mais stocké comme tel). Le stockage est fait sur 22 octets au maximum, avec deux chiffres par octet (20 octets pour la mantisse, 38 chiffres au maximum), un octet pour la longueur de la valeur stockée, et un octet pour le signe et l'exposant (ce dernier varie de -130 à 125). Ainsi, le nombre 5483,789 sera stocké différemment selon le type déclaré :

NUMBER 5483,789

NUMBER(7,3) 5483,789

NUMBER(6,2) 5483,789

NUMBER(6) 5484

NUMBER(5,-2) 5500

NUMBER(6,3) Erreur

Dates

date Date sur 7 octets : 2 octets pour l'année sur 4 chiffres, 1 octet pour le mois sur 2 chiffres, 1 octet pour le jour sur 2 chiffres, 1 octet pour l'heure sur 2 chiffres, 1 octet pour les minutes sur deux chiffres et un octet pour les secondes sur 2 chiffres. Le format externe par défaut est **JJ-MOI-AA**. Lorsqu'on met des dates hors du vingtième siècle, on utilise un masque. On peut spécifier un autre format externe avec la fonction `to_date(chaine[, masque, [langue_nls]])`.

Données

blob Binary Large Object, fait pour tout type de données binaires non structurées (images, vidéos, son), jusqu'à 4 Go. Le stockage est fait dans une table, ou directement dans un tablespace grâce à un *locator*. On peut y avoir accès par morceaux. Les blobs participent aux transactions, donc les changements qui y sont apportés peuvent être validés ou annulés. Les données de type BLOB doivent être manipulées avec les opérations définies dans le package `DBMS_LOB`.

bfile Pareil qu'un BLOB mais le stockage est externe à la base (dans le système de fichiers). Ces fichiers sont en lecture seule. Dans une colonne de type BFILE, on stocke le chemin du fichier grâce à un *locator*. On manipule les données à travers une API spécifique (dans le package `DBMS_LOB` ou avec *Oracle Call Interface*).

clob Pareil que BLOB mais pour les chaînes de caractères. Cela remplace le domaine LONG conservé uniquement pour des raisons de compatibilité ascendante. On les manipule de la même façon que les BLOB.

Identifiant de ligne ROWID Toute ligne d'une table Oracle a un ROWID unique. Ils servent d'adresses physiques (attention à ça) pour les lignes des tables et sont utilisés dans les index. Ils constituent une pseudo-colonne, ce qui fait qu'on puisse l'utiliser dans une sélection (`SELECT rowid FROM client`). Il est impossible de changer le ROWID d'une ligne sauf en exportant puis réimportant cette ligne. On distingue trois types de ROWID :

ROWID réduit Ce type a subsisté jusqu'à Oracle 7 mais n'est plus utilisé. Un rowid réduit est constitué d'un numéro de bloc (8 chiffres en hexadécimal), d'un numéro de ligne dans le bloc (4 chiffres en hexadécimal) et d'un numéro de fichier⁸ (4 chiffres en hexadécimal). Tout cela est donc codé en hexadécimal et tient sur 6 octets. À partir d'Oracle 8, on peut encore utiliser ces ROWID malgré quelques conversions, mais cela vaut-il vraiment le coup ?

ROWID étendu Ce type est celui qui sévit actuellement, à partir de Oracle 8. Un rowid étendu se compose d'un numéro de segment (*data object number*), d'un numéro de fichier, d'un numéro de bloc et d'un numéro de ligne. Il occupe 10 octets et est codé en base 64. Le package `DBMS_ROWID` permet d'extraire les différentes informations que contiennent les ROWID étendus.

ROWID logique Apprêtez-vous à rencontrer ce troisième type. Il est constitué de la valeur de la clé primaire et d'un identifiant de bloc. Ce dernier peut devenir obsolète suite au déplacement de la ligne dans un autre bloc.

2 Organisation générale d'Oracle

Regardez donc le schéma de la page 8.

3 Organisation des données sur disque

Un SGBD se doit d'organiser et de référencer ses données d'après le principe de *data independence*. Ce principe stipule que le schéma relationnel et le schéma physique des données doivent être indépendants. Ainsi l'organisation interne peut être changée sans que cela affecte le code relationnel.

⁸ C'est ce numéro qui identifie un *Datafile*.

a Organisation séquentielle

L'organisation séquentielle est celle qui est utilisée par défaut avec Oracle lorsqu'on ne donne pas de clé primaire. Il est possible de créer des index secondaires sur les tables organisées de telle manière.

b Organisation indexée

L'organisation indexée des données existe depuis Oracle 7.3. La table et l'index sont dans un même arbre B+. L'index est créé sur la clé primaire.

Dans l'arbre B+, les feuilles sont doublement chaînées entre elles. À cause du principe de data independence les ROWID ne sont pas stockés dans les index.

- 1 **CREATE TABLE** tbl (...)
- 2 ORGANIZATION **INDEX**;

Quand on rajoute des index secondaires sur une table, ces index sont stockés dans des segments d'index, donc à part.

c Organisation en clusters

L'objectif de l'organisation en clusters, c'est de mettre plusieurs tables dans le même espace physique parce que ça peut accélérer leur jointure. Dans les faits, il s'agit de rapprocher les tipes de tables différentes selon un critère appelé *clé du cluster*.

Par exemple, soient les tables suivantes :

Client (N°Client, NomC, AdresseC)

Commande (N°Commande, date, N°Client)

On peut créer un cluster entre les tables Client et Commande et la clé du cluster serait N°Client.

En fait, la notion de cluster permet de stocker un résultat de jointure. Lorsqu'on appelle ces jointures en SQL, elles sont précalculées car créées automatiquement.

- 1 **CREATE CLUSTER** nomcluster
- 2 (nom_colonne_virtuelle type_colonne)⁹
- 3 **SIZE** taille¹⁰
- 4 **{INDEX | [HASH IS colonne] HASHKEYS nb_de_buckets};**

Ce qu'on choisit de mettre ici à la dernière ligne dépend de l'organisation choisie.

⁹ Clé du cluster

¹⁰ Taille des données attendues en moyenne par valeur de clé. Si c'est inférieur à la taille d'un bloc, alors plusieurs valeurs de la clé de cluster et leurs lignes correspondantes peuvent être stockées dans un même bloc.

Clusters indexés Lorsqu'on veut indexer le cluster (pour des tables avec des accès plutôt séquentiels, sur des intervalles de valeur de clé) alors on fait plutôt ça :

- 1 **CREATE INDEX index**
- 2 **ON CLUSTER cluster**
- 3 `caract_physiques;`

À chaque valeur *val* de l'index est associée la liste des blocs (il n'y en a qu'un seul au début) qui contiennent l'ensemble des lignes des tables du cluster dont la clé est *val*.

Hash-clusters Par contre, si on souhaite plutôt utiliser une organisation aléatoire (pour des tables avec plein d'accès ponctuels), alors on utilisera **HASHKEYS** en précisant **HASH IS colonne** uniquement si la fonction aléatoire d'Oracle ne nous convient pas. À chaque valeur *val* délivrée par la fonction hash-code *h* est associée la liste des blocs qui contiennent tous les tuples des tables du cluster dont la clé *c* est telle que $h(c) = val$. Le nombre de *buckets* est arrondi au nombre premier immédiatement supérieur au nombre spécifié, et l'espace initial réservé au hash-cluster est supérieur ou égal à `nb_de_buckets*taille`.

Ajout de tables au cluster Ensuite, on peut utiliser notre cluster. Lorsqu'on veut créer une table que l'on stocke dans un cluster alors c'est :

- 1 **CREATE TABLE** `tbl (...)`
- 2 **CLUSTER** `nomclstr (colonne, ...)` ¹¹

On fait ça autant de fois qu'on veut mettre de tables dans notre cluster. Les résultats de la fonction hash-code interne à Oracle ne sont pas stockés.

d Organisation en index secondaires

On spécifie un index secondaire grâce à la commande SQL déjà survolée :

- 1 **CREATE [UNIQUE]** ¹² **INDEX** `nom_index`
- 2 **ON** `tbl (attr [ASC | DESC])`
- 3 **[NOSORT]** ¹³ **[PCTFREE** `pourcentage` ¹⁴ **];**

Si vous avez bien suivi, un index secondaire est dense, non plaçant et structuré en arbre B*. Les index sont créés sur une ou plusieurs colonnes prises dans leur entier ¹⁵.

Les valeurs de l'index et les ROWID correspondants sont stockés dans l'index.

Pour créer un index, Oracle commence par trier la relation à indexer (ben oui, quand même), sauf si la clause **NOSORT** a été spécifiée (et vous savez quand, maintenant) ¹⁶.

¹¹ Colonnes placées dans la clé du cluster précédemment créé. Les colonnes doivent correspondre.

¹² On peut spécifier si l'index est unique, *ie.* s'il est possible d'avoir deux fois la même valeur dans l'index.

¹³ Demande à ne pas trier l'index, quand celui-ci est déjà trié.

¹⁴ Permet de spécifier le taux de remplissage des nœuds de l'index lors de sa création.

¹⁵ C'est vrai que ça choque... Moi je ne connaissais pas. ``Prises dans leur ensemble''.

¹⁶ Faites attention cependant, Oracle est malin, il va supposer que votre relation est triée mais il vérifie ! Vous voyez ? Un SGBD ne fait confiance à personne, alors pourquoi lui faire confiance ?

Les valeurs `NULL` ne sont pas stockées dans l'index. Ainsi, plusieurs lignes peuvent avoir une valeur d'index `NULL`, même pour un index *unique*.

e Organisation en index bitmap

Un index bitmap comporte autant de colonnes qu'il y a de valeurs possibles pour l'index, et autant de lignes que de lignes dans la table. Pour chaque ligne l de la table et chaque valeur c de l'index, on stocke un bit positionné à 0 ou à 1 qui indique si la ligne l a la valeur c pour la colonne indexée.

Les ROWID n'ont pas besoin d'être stockés dans l'index. Ils peuvent être recalculés à partir du numéro de ligne dans l'index puisque ce dernier est trié (les informations sont stockées dans l'index dans l'ordre où on les trouve sur le disque).

C'est un type d'index qui permet de trouver rapidement les lignes répondant à un critère particulier. Il est aussi utile pour combiner facilement plusieurs critères. Par exemple, si on a des index bitmap sur la ville et sur le sexe, on peut combiner les bitmaps avec un \wedge (et) logique pour obtenir directement la liste des étudiantes habitant Dijon ¹⁷.

	célibataire	marié	veuf
ligne1	1	0	0
ligne2	0	1	0
ligne3	0	0	1
ligne4	0	1	0

Figure 3.1. Un index bitmap

f Organisation en reverse key index

Les valeurs des colonnes clés sont stockées à l'envers : deux valeurs successives selon l'ordre croissant ne sont plus physiquement consécutives. Cela peut être intéressant pour mieux répartir les valeurs dans des feuilles d'index différentes.

¹⁷ Et une liaison avec un langage de plus bas niveau comme Pro*C vous permettra de contacter directement, par le biais d'un modem, chacune de ces étudiantes pour leur faire des avances.

4 Tablespaces

a Définition et création sur Oracle

Un **tablespace** est un espace de stockage constitué d'un ou plusieurs fichiers. En fait, on peut même dire qu'une base de données Oracle, c'est un ensemble de tablespaces.

Sur Oracle on crée un tablespace avec la syntaxe suivante :

```

1 CREATE TABLESPACE tblspace
2 DATAFILE fichier
3 SIZE taille {K | M}
4 [REUSE]
5 [DATAFILE ...]
6 clause_stockage
7 {ONLINE | OFFLINE};

```

La taille est indiquée par un nombre suivi de **K** pour kilo-octets ou **M** pour mégaoctets. On spécifie **REUSE** pour dire que le fichier est réutilisable. Si on veut spécifier plusieurs fichiers on ré-enchaîne avec **DATAFILE**. La clause de stockage dit comment les objets sont stockés (vous la verrez un peu plus loin). Enfin, on doit choisir entre **ONLINE** et **OFFLINE**. Si un tablespace est **OFFLINE**, alors les tables et index qui y sont stockés sont rendus inaccessibles. On ne peut pas mettre offline les tablespaces **SYSTEM** et les tablespaces qui contiennent un rollback segment actif.

Une fois le tablespace créé, on peut l'utiliser pour créer une table, un index ou un cluster de cette façon ¹⁸ :

```

1 CREATE {TABLE | INDEX | CLUSTER} ... TABLESPACE tblspace;

```

On peut rajouter un fichier à un tablespace existant grâce à l'instruction suivante :

```

1 ALTER TABLESPACE ts
2 [
3   ADD DATAFILE fichier
4   SIZE taille [REUSE]
5   |
6   RENAME DATAFILE fichier,... TO fichier,...
7 ]
8 {ONLINE | OFFLINE [NORMAL | IMMEDIATE]}
9 ;

```

b Utilité

Notons que le tablespace a deux principales utilisations. Tout d'abord c'est un outil pratique pour le stockage par morceaux, ainsi c'est pratique pour éviter les conséquences néfastes des pannes. Cela permet de déterminer des politiques d'allocation d'espace personnalisées pour certains objets, de regrouper différents objets dans le même fichier...

¹⁸ Chaque table et chaque index est localisé dans un seul tablespace.

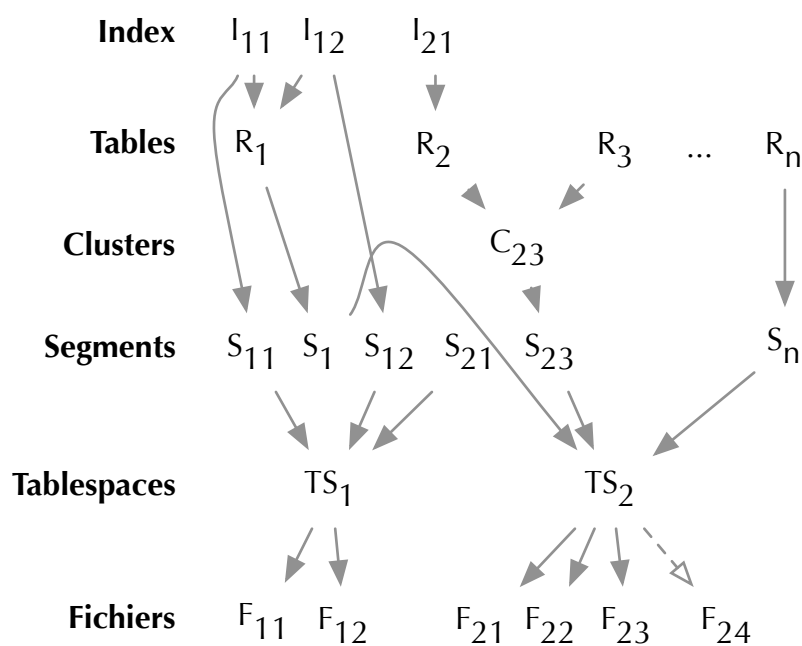


Figure 3.2. Un schéma conceptuel

Ensuite, cela peut être très utile pour accélérer l'accès disque. Mettons qu'on a une table et son index sur un disque. Avec une tête de lecture, on accède à l'index puis à la table : la tête de lecture bouge beaucoup entre la table et l'index, et le temps perdu est considérable. Si on a la possibilité d'avoir deux disques, alors il n'y a qu'à créer sur le premier disque un tablespace accueillant la table, et sur le deuxième disque un autre tablespace accueillant l'index. De cette façon les deux têtes ne bougent presque pas pour lire l'ensemble de la table. On gagne en temps et en usure des têtes de lecture.

On conclut donc que les tablespaces servent à spécifier l'endroit ou les endroits où sont stockées les informations de la base de données, et permettent à l'administrateur d'optimiser sa base de données en indiquant au SGBD où stocker certains éléments.

Pour ce qui est des tablespaces *offline*, ça sert à plusieurs choses. On peut choisir de mettre online une partie de la base seulement. Mais on peut aussi utiliser des tablespaces *offline* pour faciliter la maintenance (sauvegardes et restauration), atteindre une taille supérieure à la capacité des disques *online*, ou arrêter partiellement une partie de la base de données.

C Renommer un datafile

Il est possible de renommer un fichier de données de la base. Pour cela il faut :

- mettre *offline* le tablespace auquel ce fichier appartient, avec `ALTER TABLESPACE` ;

- copier le fichier sous son nouveau nom dans le système de fichiers (avec `cp`) ;
- modifier le nom du fichier enregistré dans le dictionnaire, avec `ALTER TABLESPACE` ;
- remettre *online* le tablespace.

5 Segments

Chaque table, index ou cluster est stocké en mémoire physique sous la forme d'un segment.

Un **segment** est une liste d'extensions (ou *extents*). C'est bien beau, tout ça, si on ne précise pas qu'un **extent** est simplement une suite de blocs consécutifs. Le segment est du même nom que la table, l'index ou le cluster qui lui est associé. On peut être plus précis en disant qu'un segment contient toutes les données pour une structure spécifique à l'intérieur d'un tablespace.

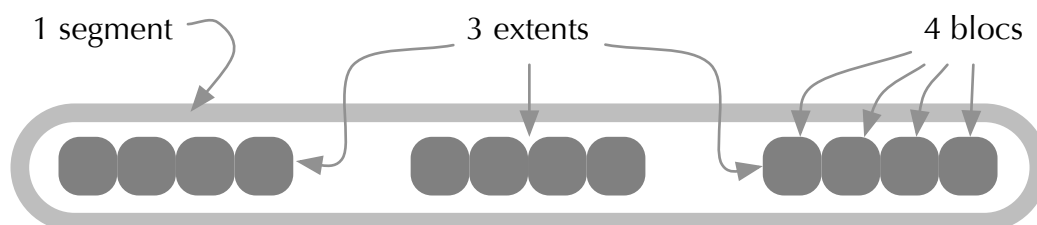


Figure 3.3. Segment, extents, blocs

Dans nos requêtes de création on peut utiliser la clause `STORAGE` qui permet de définir la politique d'allocation d'espace aux objets concernés. Voici ce qu'on met dedans :

```

1 STORAGE (
2   INITIAL nb_oct1 19
3   NEXT nb_oct2 20
4   PCT_INCREASE pourcentage 21
5   MINEXTENTS nb 22
6   MAXEXTENTS nb 23
7   OPTIMAL nb 24

```

¹⁹ Taille du premier extent alloué (par défaut 5 blocs).

²⁰ Taille du deuxième extent alloué (par défaut 5 blocs).

²¹ Pourcentage par lequel on multiplie la taille des extents suivants ($taille_n = taille_{n-1} \times PCT_INCREASE$). Il est toujours de zéro pour un rollback segment, mais par défaut c'est 50%.

²² Nombre d'extents alloués à la création (par défaut 1).

²³ Limite du nombre d'extents. Peut être UNLIMITED mais ça n'a pas à être utilisé pour les rollback segments (pour ces derniers le minimum de MAXEXTENTS est 2, et pour le reste c'est 1, et par défaut c'est 99).

²⁴ Cet attribut spécifiant une taille optimale n'est utilisé que pour les rollback segments.

a Gestion des segments avec Oracle

Pour connaître l'espace total alloué à chaque segment, consultez la table `USER_SEGMENTS` et en particulier la colonne `blocks` ou `bytes`.

Pour connaître les extents alloués à chaque segment, consultez la table `USER_EXTENTS`.

Pour connaître l'espace non encore affecté à un segment particulier, dans un tablespace, il faudra consulter la table `USER_FREE_SPACE` et particulièrement les colonnes `file_id` et `blocks`.

b Types de segments

Il y a six types de segments :

rollback segment Contient les données de rollback pour les rollbacks, la consistance en lecture ou la restauration, en fait c'est tout ce qui concerne les transactions

segment de données Contient toutes les données d'une table ou d'un cluster

segment des tables avec index primaire Contient les données d'une table avec son index primaire

segment d'index Contient toutes les données d'un index secondaire ou bitmap

segment temporaire Contient les données qui appartiennent à des objets temporaires (opérations de tri...)

bootstrap segment Il est unique, pèse environ 50 blocs et contient, entre autres, la description du dictionnaire de données

c Rollback segments

Ce sont les journaux d'annulation d'Oracle. Chaque mise à jour d'une donnée génère une entrée dans un rollback segment (c'est l'état avant que la donnée soit modifiée). Une transaction ²⁵ est associée à un unique rollback segment (explicitement ou implicitement).

Ça, c'est **N2** qui le dit, quelqu'un de chez Oracle, en qui on devrait pouvoir se fier les doigts dans le nez ! Pourtant, du côté de chez **D2**, on rapporte que notre maître T. Grison a dit qu'Oracle se sert des rollback segments pour stocker des valeurs nécessaires pour calculer une requête. Qui croirez-vous à l'examen ?

Croyez la documentation d'Oracle 10g, qui dit qu'à partir de cette version Oracle préfère l'*undo tablespace* plutôt que les rollback segments. Donc ces segments ne sont pas très importants, malgré la taille de cette section. En plus, la boîte à chaussures avait tout faux.

²⁵ On définira vraiment ce qu'est une transaction page 50, mais il y a juste à savoir que c'est une suite d'instructions SQL entre deux `COMMIT`.

d Segments de données

Les segments de données contiennent les lignes des relations de la base. La figure 3.4 montre c'qu'un bloc d' données a dans l'bide, et c'est pas du foie gras !

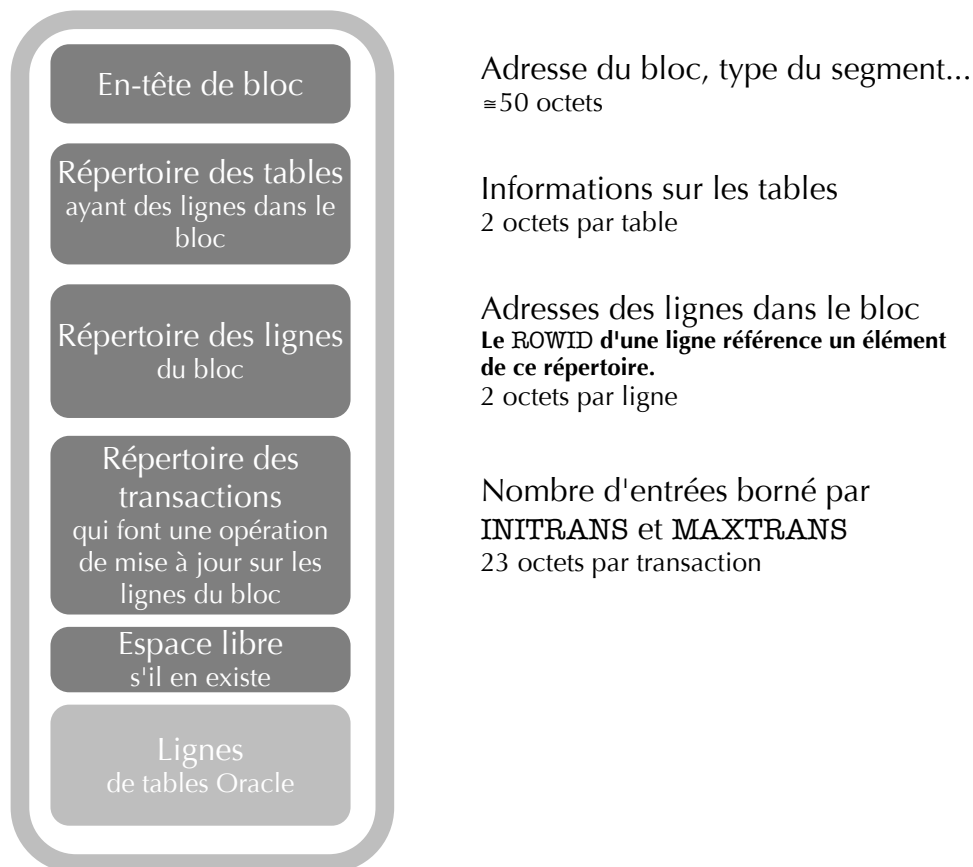


Figure 3.4. Ceci n'est pas du foie gras.

Notez les paramètres **INITRANS** et **MAXTRANS**. **INITRANS** détermine la taille de la partie variable de l'en-tête d'un bloc. Si **INITRANS** = n alors Oracle réservera $23 * n$ octets pour cette partie et laissera alors de la place pour que n transactions puissent travailler simultanément sur le bloc. Lorsque $n + 1$ transactions veulent travailler sur le bloc, alors la dernière pourra travailler en même temps que les autres s'il existe de la place dans le bloc (à l'extérieur de l'en-tête), et sinon elle attend son tour comme une grande. Il est donc possible qu'il y ait un nombre indéterminé de transactions en même temps sur un bloc, mais on sait que s'il est plus grand que **INITRANS** alors il sera inférieur à **MAXTRANS** qui est la limite de transactions simultanées pouvant modifier le bloc.

e Gestion des lignes dans un bloc de données

Les paramètres PCTFREE et PCTUSED On les spécifie quand on crée une table.

1 **CREATE TABLE** nomtable ... PCTFREE pct_free PCTUSED pct_used;

Qu'est-ce que ça veut dire ? PCTFREE indique un seuil à partir duquel les insertions de lignes dans un bloc ne sont plus autorisées. C'est un pourcentage de la taille *utile* (hors données système) d'un bloc. Cela sert à réserver de l'espace pour les mises à jours "extensives" des lignes. Il est fixé à 10% par défaut.

PCTUSED indique à partir de quel moment de nouvelles lignes peuvent être insérées dans le bloc. Il s'exprime en pourcentage de la taille *utile* d'un bloc, et fixe un seuil minimal au taux de remplissage. Il est fixé à 40% par défaut.

Bien entendu, on doit toujours vérifier que $PCTFREE + PCTUSED < 100\%$.

Fonctionnement Le fonctionnement repose sur le phénomène d'hystérésis²⁶, que vous pouvez retrouver en électronique dans la bascule de Schmitt, et en bases de données comme ceci :

1. Lorsqu'un bloc est affecté à une relation, il est naturellement vide de toute ligne. Le pourcentage d'occupation effectif, nommé *Pcto*, est égal à zéro. Le bloc fait donc partie de la liste des blocs libres (la *free list*).
2. Des lignes sont insérées dans le bloc jusqu'à ce que $Pcto \geq 1 - PCTFREE$.
3. Lorsque *Pcto* dépasse ce seuil, le bloc est retiré de la *free list*.
4. Des lignes peuvent être mises à jour ou supprimées dans les blocs non libres.
5. Lorsque *Pcto* descend sous le seuil de PCTUSED, ce bloc est remis dans la *free list*.

À quel niveau fixer PCTFREE et PCTUSED ? Je ne sais pas. Ah si ! En fait, PCTFREE sert à éviter d'avoir un chaînage inter-blocs trop important (le chaînage est dû au fait qu'une ligne, après mise à jour, ne tient plus entièrement sur un bloc). Il faut donc un PCTFREE élevé pour les tables dont la taille des lignes, suite à des modifications, augmente sensiblement. C'est le cas des tables dont certaines colonnes ne sont pas renseignées à la création des lignes.

PCTUSED permet de contrôler la fréquence selon laquelle les blocs d'une relation entrent et sortent de la *free list* suite aux mises à jour qu'ils subissent.

Un PCTUSED faible assure qu'un bloc remis dans la *free list* dispose d'un espace libre important pour permettre d'insérer des nouvelles lignes.

Structure d'une ligne de données Oui, les blocs contiennent des lignes de données. Mais que contient une ligne de données ? Du foie gras ? La figure 31 répond à cette question.

²⁶ À ne pas confondre avec l'hystérectomie qui ne devrait rien vous rappeler (ou alors c'est malheureux).

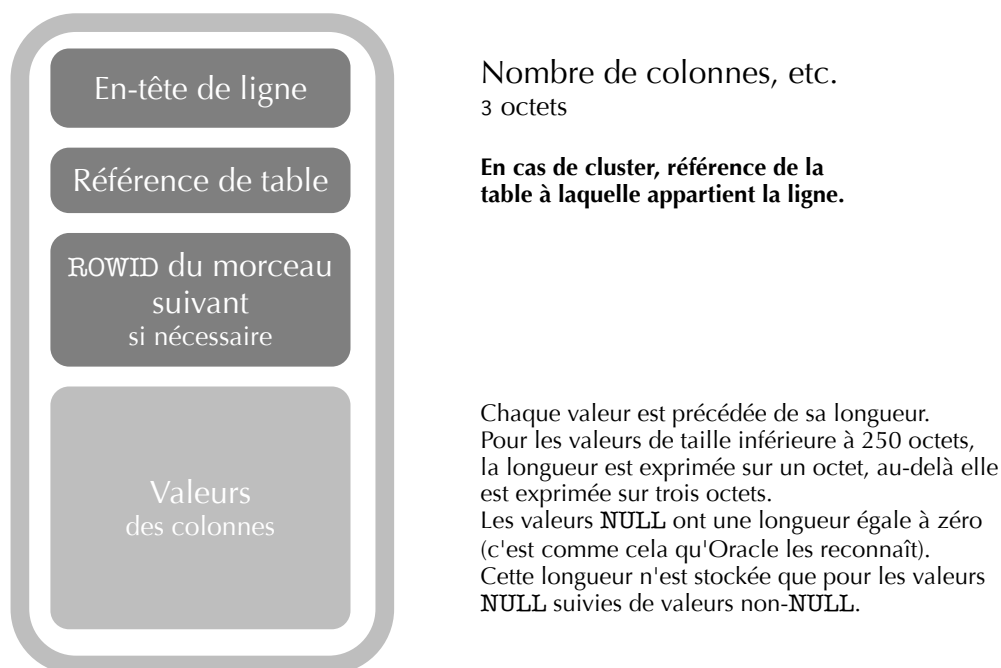


Figure 3.5. Ceci n'est pas *non plus* du foie gras.

Compactage dans un bloc et morcellement des lignes L'espace libre dans un bloc peut être fragmenté à la suite de mises à jour comme la suppression de lignes et la diminution de valeurs. L'espace libre fragmenté n'est pas systématiquement compacté.

En fait, le **compactage** a lieu dans deux cas :

- si de l'espace est requis pour une insertion de ligne ou une modification extensive d'une valeur ;
- ou si aucun fragment d'espace libre n'est suffisant pour cette modification.

Pour faciliter un éventuel rollback, l'espace libéré pour des lignes supprimées n'est pas immédiatement récupérable. Il ne le devient qu'à la validation de la transaction qui a opéré la suppression. Ben ouais, imaginez les conséquences sinon !

Ainsi, le nombre d'extents qui sont affectés à une table peut grossir plus vite que prévu.

Une ligne est **morcelée** pour être stockée sur plusieurs blocs dans deux cas :

- si la ligne insérée a une taille supérieure à celle du bloc (logique) ;
- ou après mise à jour, si la ligne ne tient plus entièrement sur son bloc d'origine (espace libre insuffisant).

Les différentes portions de lignes sont stockées sur des blocs chaînés entre eux.

f Segments temporaires

Les segments temporaires sont créés automatiquement par le serveur Oracle pour réaliser des opérations complexes du genre de **GROUP BY**, **ORDER BY**, **CREATE INDEX**,

des trucs où il y a du tri, quoi.

Ces segments temporaires sont créés dans les fichiers de données lorsque l'espace nécessaire n'est pas disponible dans la mémoire centrale, mais ça vous n'êtes pas obligés de le savoir.

Algorithmes des opérateurs de l'algèbre relationnelle

SECTION

1

Projection π

L'**opérateur projection** consiste juste à ne sélectionner que quelques colonnes dans une table. Cela se fait très simplement de manière séquentielle.

SECTION

2

Sélection σ (restriction)

L'**opérateur sélection** est implémenté de deux manières. Le **coût** de chaque algorithme présenté est le nombre de blocs transférés entre la mémoire centrale et la mémoire secondaire pour effectuer la sélection. Soit B_R le nombre de blocs utilisés par la relation R .

1 Balayage séquentiel

a Présentation

Pour chaque tuple de la table, on vérifie si le critère de sélection est vérifié.

b Coût

Le coût de cette méthode est B_R , puisqu'on parcourt toute la relation pour faire la sélection.

c Algorithme

L'algorithme de sélection par balayage séquentiel est très simple :

```

1 selection(relation R, condition CS) {
2   pour chaque tuple t de R
3     si verifier(t, CS) = vrai alors
4       resultat = union(resultat, t)
5   finsi
6   finpour
7 }
```

2 Balayage via un index**a Présentation**

Cette méthode s'avère beaucoup plus rapide que le balayage séquentiel. Par contre il faut mettre en place l'index et la méthode a ses limites. Par exemple, si on crée un index sur l'âge dans une table et qu'on cherche les étudiants qui sont plus vieux que 20 ans, ce n'est pas intéressant.

b Coût

Critère de sélection simple Un critère de sélection simple est de la forme ($a = valeur$) avec a un attribut de R. Le coût est alors $\frac{T_R}{I_a}$ où T_R est le nombre de tuples de R, et I_a est le nombre de valeurs distinctes pour l'attribut a dans R,

Critères de sélection complexe Il y a plusieurs types de critères de sélection complexe. Nous en verrons trois pour a un attribut de R et T_R le nombre de tuples de R.

$a > valeur$ Le coût est le suivant :

$$coût_{ind} = \frac{valmax - valeur}{valmax - valmin} * T_R$$

En résumé, moins il y aura de tuples qui satisferont le critère de sélection, et moins le coût de l'index sera grand, et vice versa ^❶.

$a = v_1$ et $b = v_2$ Il est possible d'utiliser deux index si disponible. On récupère les adresses pour $a = v_1$ via le premier index, on fait de même pour $b = v_2$ puis on fait l'intersection des deux ensembles.

$a = v_1$ ou $b = v_2$ Si l'un des deux attributs a ou b n'a pas d'index, alors on devra revenir à un balayage séquentiel.

❶ Mais tu dis (mais tu dis) que le bonheur est irréductible, et je dis (et il dit) que ton espoir n'est pas si désespéré, à condition d'analyser que l'absolu ne doit pas être annihilé par l'illusoire précarité de nos amours... destitués ! Et vice versa !

3 Choix par Oracle

Soit deux sélections :

- a $\sigma_{ville="Paris"}Etudiant$
- b $\sigma_{ville="Villeneuve"}Etudiant$

Alors Oracle ne fera pas appel aux mêmes algorithmes pour les requêtes de ces deux sélections. Il choisira ces algorithmes :

- a Table access full (balayage séquentiel)
- b Access by ROWID, Index range scan (balayage par index)

Et ça, croyez-le ou non, c'est pareil partout ! On verra ça vers la page 40, dans un chapitre qui traite de l'optimisation de requêtes.

SECTION
3

Jointure \bowtie

Nous verrons quatre algorithmes pour l'opérateur de jointure.

Soient R_1 et R_2 deux relations. Alors B_{R_1} et B_{R_2} seront le nombre de blocs utilisés par les relations respectivement R_1 et R_2 . On notera M le nombre de blocs dont on dispose en mémoire centrale.

1 Algorithme du produit cartésien

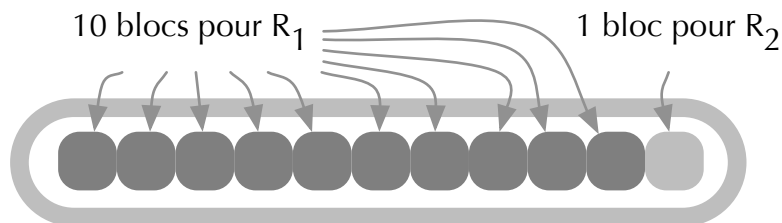


Figure 3.1. La mémoire centrale (11 blocs) lors de l'algorithme du produit cartésien

Imaginons qu'on ait $B_{R_1} = 30$ et $B_{R_2} = 50$. Si on a 11 blocs en mémoire centrale, alors on en prend 10 pour charger les blocs de R_1 dix par dix, et le dernier bloc disponible est là pour qu'on puisse lire un à un les blocs de R_2 et les comparer avec les dix blocs de R_1 actuellement en mémoire centrale. Du coup, on choisit comme R_1 la table qui a le plus petit nombre de blocs.

Le coût de la lecture de R_1 est donc B_{R_1} , et le coût de la lecture de R_2 est donc $\frac{B_{R_1}}{M-1} \times B_{R_2}$. Ainsi le coût total de la lecture par l'algorithme du produit cartésien est la somme

des deux coûts, soit :

$$\hat{c}_{pc} = \frac{B_{R_1}}{M-1} \times B_{R_2} + B_{R_1}$$

2 Algorithme sort-merge

L'algorithme du sort-merge est composé de deux phases :

1. le tri des deux relations R_1 et R_2 sur le critère de jointure, sachant qu'un tel tri sur la relation R a pour coût (au pire) $\hat{c}_{tri}(R) = 2B_R \log_{2M} B_R$;
2. la pseudo-fusion des deux relations triées, ce qui a pour coût $\hat{c}_f = B_{R_1} + B_{R_2} + \delta$ (avec δ le coût négligeable des retours en arrière) puisqu'on lit en entier R_1 et R_2 au moins une fois

Le coût total de la jointure sort-merge est donc la somme des coûts de deux tris et d'une fusion :

$$\hat{c}_{sm} = 2B_{R_1} \log_M B_{R_1} + 2B_{R_2} \log_M B_{R_2} + B_{R_1} + B_{R_2}$$

On pourrait ne pas utiliser cet algorithme car il est très coûteux en mémoire, mais il est très pratique dans la vie car on peut tirer parti du fait qu'une des tables (ou les deux) dont on fait la jointure est déjà triée. Cela peut arriver, par exemple, quand on vient de faire une jointure sort-merge, ou lorsqu'on a utilisé un index, ou que la requête a précisé ORDER BY colonneJointure.

a Algorithme de tri fusion externe

Il serait amusant de voir comment retrouver le calcul du coût du tri. Cela nous permettra de savoir comment on fait un tri fusion externe (qui est utile pour plein de choses en dehors des algorithmes de jointure).

Soit R une relation. Si la taille de la mémoire centrale disponible est suffisante, alors le tri sera effectué directement en mémoire centrale.

Sinon on doit opérer en mémoire secondaire et trier par groupe de blocs en faisant un tri fusion externe.

Voyons donc les étapes qu'on franchit lorsqu'on est un gentil Oracle qui trie notre relation R .

Etape 1 La relation contient B_R blocs et on la partage en tas de M blocs (puisque c'est la taille de la mémoire centrale). On a donc $\left\lceil \frac{B_R}{M} \right\rceil$ tas. Pour chaque tas on le lit en mémoire centrale, puis on le trie, et on le réécrit trié en mémoire secondaire. Au total on aura donc fait B_R lectures et B_R écritures, donc $2B_R$ accès. On obtient donc, à la fin, des tas de M blocs triés.

Etape 2 On va fusionner les tas en les triant deux par deux et à la fin de l'étape il y a deux fois moins de tas. Dans cette étape, on aura donc aussi fait $2B_R$ accès, et on aura, au final, des tas de M^2 blocs triés.

...

Etape n On obtient des tas triés de M^n pages.

En fait, on s'arrête lorsqu'on n'a plus qu'un seul gros tas suintant, puisque ce gros tas suintant c'est notre relation complètement triée. La taille de ce tas est donc au moins le nombre de blocs de R , ainsi l'algorithme s'arrête quand $M^n \geq B_R$. On en déduit le nombre d'étapes : $n = \lceil \log_M B_R \rceil$, et le coût du tri fusion externe :

$$\hat{c}_{trif} = 2B_R \log_M B_R$$

b Algorithme

Voici la fameuse recette de la jointure sort-merge à la sauce piquante :

```

1 jointure_sort_merge(relation R1, relation R2, attribut A) {
2   trier R1 sur A
3   trier R2 sur A
4   posR1 = 0
5   posR2 = 0
6   tantque posR1 = taille(R1)-1 ou posR2 = taille(R2)-1
7     si R1[lig:posR1, col:A] = R2[lig:posR2, col:A] alors
8       nouvelle_ligne(concat(R1[lig:posR1],R2[lig:posR2]))
9       posR2 = posR2 + 1
10    sinon si R1[lig:posR1, col:A] > R2[lig:posR2, col:A] alors
11      posR2 = posR2 + 1
12    sinon
13      posR1 = posR1 + 1
14    fin si
15  fin tantque
16 }
```

3 Algorithme key-lookup (ou nested loop ou boucles imbriquées)

On commence par lire séquentiellement une des deux relations. Pour chaque tuple de cette relation, on accède directement *via* un index aux tuples de l'autre relation qui vérifient le critère de jointure.

Le coût d'un tel algorithme est :

$$\hat{c}_{kl} = B_{R_1} + T_{R_1} \times \frac{T_{R_2}}{I_a}$$

Avec I_a le nombre de valeurs différentes que peut prendre le critère de jointure.

Pourquoi un tel coût ? B_{R_1} est clairement le coût de la lecture séquentielle de R_1 , à laquelle on ajoute le coût de la lecture par sélection à R_2 ($\frac{T_{R_2}}{I_a}$) autant de fois qu'il y a de tuples dans R_1 .

Vous voulez la recette ?

```

1 jointure_key_lookup(relation R1, relation R2, critère C) {
2   pour chaque ligne lr1 de R1
3     pour chaque ligne lr2 de R2 vérifiant C (balayage par index)
4       nouvelle_ligne(concat(lr1, lr2))
5     finpour
6   finpour
7 }
    
```

4 Algorithme par hachage

La jointure par hachage ne s'applique que lorsqu'on a une égalité comme critère de jointure. Comme dans le sort merge, on groupe les lignes en formant une partition des relations. Dans le sort merge, la partition est le résultat des groupes de lignes de même valeur de clé formés par le tri, alors que dans la jointure par hachage les groupes correspondent aux paquets produits par une fonction de hachage.

L'algorithme par hachage comprend deux étapes.

Premièrement, on hache les deux relations sur le critère de jointure $R_1.a = R_2.b$. Ainsi on obtient :

$$\begin{aligned}
 R_1 &\xrightarrow{h(a)} (R_{11}, R_{12}, \dots, R_{1n}) \\
 R_2 &\xrightarrow{h(b)} (R_{21}, R_{22}, \dots, R_{2n})
 \end{aligned}$$

Secondement, on calcule chaque jointure partielle $R_{1i} \bowtie R_{2i}, \forall i$, puis on a juste à les réunir :

$$R_1 \bowtie R_2 = \bigcup_{i=1}^n R_{1i} \bowtie R_{2i}$$

Du coup, lorsque chaque couple (R_{1i}, R_{2i}) tient en mémoire centrale, on a le coût total :

$$\hat{c}_{hash} = 3B_{R_1} + 3B_{R_2}$$

Le coût peut être expliqué comme ceci : la partition par hachage coûte $2B_{R_1} + 2B_{R_2}$, et le reste coûte $B_{R_1} + B_{R_2}$.

Algorithme par hachage simple Au premier passage, on lit R_1 une première fois. On extrait R_{11} , que l'on conserve en mémoire centrale, tandis que l'ensemble $R_1 - R_{11}$ est réécrit sur la mémoire secondaire. On lit R_2 bloc par bloc. On peut calculer $R_{11} \bowtie R_{21}$.

Au deuxième passage, on lit $R_1 - R_{11}$ pour extraire R_{12} qui est conservé en mémoire centrale. On lit ensuite $R_2 - R_{21}$ bloc par bloc pour pouvoir calculer $R_{12} \bowtie R_{22}$.

Et ainsi de suite, jusqu'à la fin des temps !

Recette du hachage Voyons plutôt la recette.

```

1 jointure_hachage(relation R1, relation R2, attribut A) {
2   partitionner_h(R1, A) -> (R11, ..., R1n)
    
```

```

3  partitionner_h(R2, A) -> (R21, ..., R2n)
4  pour chaque i = 1 à n
5      lire tous les blocs de R1i en mémoire centrale
6      pour chaque bloc b de R2i
7          pour chaque ligne lR2 dans b
8              pour chaque ligne lR1 dans R1i
9                  nouvelle_ligne(concat(lR1,lR2))
10             finpour
11         finpour
12     finpour
13 finpour
14 }
15
16 partitionner_h(relation R, attribut A) {
17     pour chaque ligne lR de R
18         i <- h(R(lR).A)
19         ajouter lR au bloc de Ri
20     finpour
21 }

```

5 Algorithme de jointure sur deux index secondaires

Ce n'est pas un algorithme si commun, mais il aurait dû être vu en TD. On fait une jointure sur les tables R et S ($R \bowtie_{R.a=S.b} S$) mais en exploitant la présence d'un index secondaire sur $R.a$ et sur $S.b$.

La première idée qui nous viendrait à l'esprit est mauvaise, et il s'agirait de fusionner les deux index pour avoir des couples de ROWID de tuples qui vérifient le critère de jointure, puis pour chaque couple de ROWID, de lire les deux tuples correspondants.

Une autre idée est de construire un graphe où les nœuds sont les blocs des deux index et un arc est fait entre deux blocs s'ils contiennent des tuples vérifiant ensemble la condition de jointure.

On va voir encore une autre solution qui nous vient de Mr T :

```

1  trier R sur l'attribut a
2  trier S sur l'attribut b
3  cR <- 1
4  cS <- 1
5  tantque cR < nb_tuples(R)
6      tantque R(cR).a != S(cS).b
7          cS <- cS + 1
8      fin tantque
9      nouvelle_ligne(concat(R(cR), S(cS)))
10     cR <- cR + 1
11 fin tantque

```

Optimisation de requêtes

SQL est un langage déclaratif, comme le HTML, et non pas un langage procédural (ben ouais, il n'y a pas de structures de contrôle, par exemple, et de manière générale la déclaration des éléments que l'on fait n'est pas une séquence d'instructions). Pour chaque requête SQL, Oracle doit calculer un plan d'exécution, c'est-à-dire une procédure pour exécuter la requête. Le nombre de plans d'exécution possibles pour une requête donnée peut être gigantesque, et tout naturellement, tous les plans d'exécution ne se valent pas !

SECTION



Nécessité de l'optimisation

Voyons un exemple avec une requête et deux plans d'exécution :

Requête $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$

Plan 1 $((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$

Plan 2 $((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4$

Plan 9 Un concentré de génie pur^① !

Même si, par exemple, une jointure est commutative et associative, et que le résultat est strictement identique avec chaque plan, il est simple de comprendre que l'ordre des opérations a un impact sur les performances. Ce n'est pas le seul critère qui peut jouer : étant donné qu'une opération (ici une jointure) peut être exécutée de plusieurs manières différentes, un plan d'exécution est aussi déterminé par l'algorithme de chaque opération. Les grands anciens notent les plans d'exécution sous forme d'arbres. La figure 1.1 présente de tels arbres.

^① Plan 9 était :

- *Plan 9 from Outer Space*, un film d'Ed Wood considéré comme un des plus grands navets du cinéma (je ne l'ai pas vu mais pour avoir déjà vu un film d'Ed Wood, ça semble crédible) ;
- *Plan 9 from Bell Labs*, un système d'exploitation dont un des grands principes est d'appliquer coûte que coûte la maxime UNIX *tout est fichier*.

Dans les deux cas c'est de l'art.

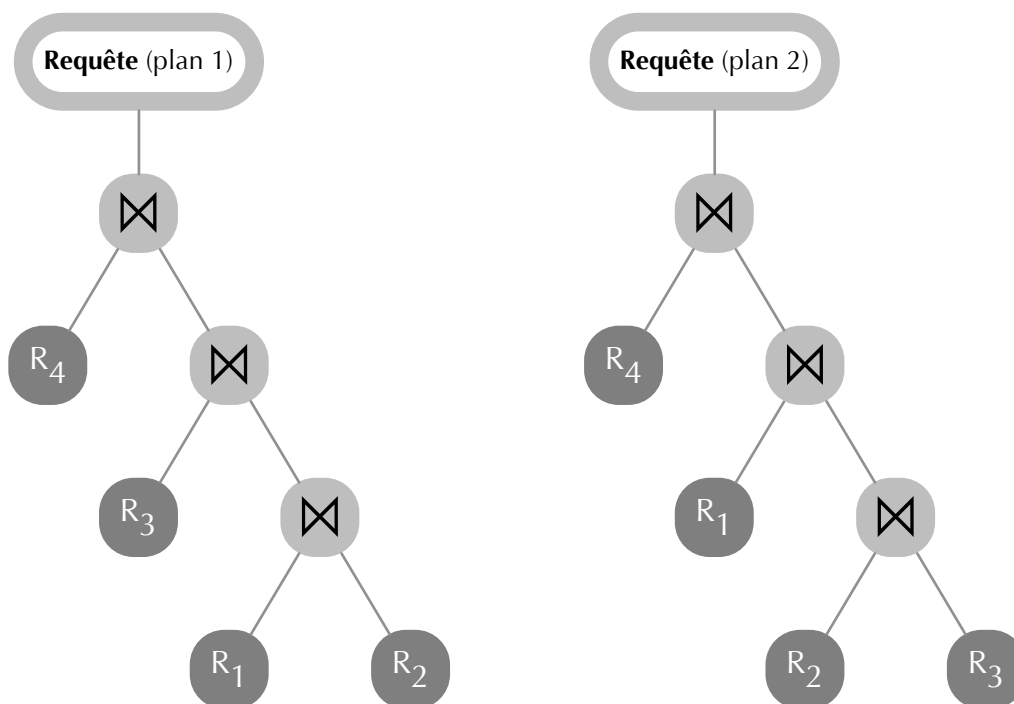


Figure 1.1. Deux plans d'exécution pour la même requête

L'optimiseur de requêtes d'Oracle est chargé de trouver le meilleur plan d'exécution pour une requête donnée. Ça signifie qu'il doit connaître le meilleur ordre d'exécution (lorsqu'on peut changer l'ordre sans affecter le résultat) et le meilleur algorithme pour chaque opération à effectuer. Évidemment, trouver le meilleur plan d'exécution pour une requête donnée implique de les envisager tous : c'est beaucoup trop long² ! Heureusement, Oracle ne manque pas de techniques de sioux pour savoir approximativement quel plan d'exécution irait bien pour une requête donnée.

SECTION
2

Optimisation *rule-based*

Jusqu'à Oracle 6, on effectuait uniquement une optimisation *heuristique* (rule-based). Cela consistait en l'application d'un certain nombre de règles appelées *heuristiques* (effectuer les sélections le plus tôt possible, les projections le plus tôt possible, effectuer les jointures les plus restrictives en premier...) qui, lorsqu'elles sont appliquées, permettent d'augmenter significativement les chances d'avoir des bons résultats.

² Pour vous donner un ordre d'idée, pour une jointure sur n tables, on peut avoir $\frac{(2(n-1))!}{(n-1)!}$ façons d'ordonner l'exécution de la jointure.

SECTION

3

Optimisation *cost-based*

Dorénavant, on peut choisir d'appliquer un autre type d'optimisation, non plus basé sur des règles prédéfinies, mais sur une estimation du coût des plans d'exécution. On l'appelle l'optimisation par coût (*cost-based*). Oracle générera ici tous les plans d'exécution (de manière exhaustive) et retiendra le moins coûteux. Tout cela devient un problème d'optimisation classique et peut se résoudre, par exemple, par les stratégies suivantes :

Programmation dynamique On construit un ensemble de plans pour une table à la fois, puis on garde les meilleurs. Ensuite, on construit un ensemble de plans pour deux tables à la fois, *et caetera* jusqu'à n tables. On réutilise les calculs précédents à chaque fois. Cette stratégie n'est pas viable pour les expressions complexes car elle a une complexité exponentielle.

Amélioration itérative On va d'abord trouver un plan de départ de manière aléatoire ou de manière heuristique, et ensuite on cherchera à améliorer cette solution de manière itérative en regardant les plans *voisins*, construits en appliquant des règles d'équivalence. On fait ça jusqu'à ce qu'on ne trouve pas d'amélioration possible.

Recuit simulé C'est pareil que précédemment, sauf qu'on considère également les plans voisins plus coûteux, afin d'augmenter les chances de trouver quelque chose de différent et de ne pas converger trop rapidement.

Algorithme génétique C'est un truc horrible où on applique la théorie de l'évolution pour que les solutions les plus adaptées survivent alors que les moins adaptées meurent.

On n'a pas à retenir tout ça, c'est juste pour la culture, bande de moules. Sinon, il faut savoir qu'il y a deux variantes du mode *cost-based*, qui déterminent comment le coût est calculé. Le mode `COST FIRST ROWS` va regarder le temps nécessaire pour la première ligne du résultat (donc le temps d'attente), alors que le mode `COST ALL ROWS` estimera le temps total.

SECTION

4

Choix par Oracle

En pratique, l'optimiseur peut être réglé pour utiliser un type d'optimisation voulu.

① Changement du choix global

Voici ce qu'il faut faire pour changer le type d'optimisation voulu au niveau global :

- 1 **ALTER SESSION SET** OPTIMIZER_GOAL = {RULE | ALL_ROWS | FIRST_ROWS | CHOOSE};

Un petit rappel :

RULE Optimisation heuristique

ALL_ROWS Optimisation en fonction du temps total (par coût)

FIRST_ROWS Optimisation en fonction du temps d'attente (par coût)

CHOOSE Laisser le choix à Oracle ³

2 Changement du choix pour une requête

Il faut utiliser ce qu'on appelle un **hint**, c'est-à-dire une indication pour optimiser. Voici comment forcer l'utilisation du mode **RULE** :

```
1 SELECT /*+ RULE*/ nom FROM client WHERE noClient = 10;
```

Sinon, en plus de pouvoir forcer un mode d'optimisation, il existe de nombreuses petites astuces et des livres entiers pour suggérer à Oracle l'utilisation de tel ou tel algorithme.

³ Cela dépendra de la disponibilité des statistiques, qui sont générées avec la commande **ANALYZE**.

Faire joujou avec Oracle

Vous trouverez en annexe de quoi vous occuper si vous ne savez pas vous connecter sur Oracle. Ensuite vous pourrez jouer.

SECTION

1

Dictionnaire de données

Comme vous le savez déjà, le dictionnaire de données, c'est des informations sur les informations de votre base de données. Il y a tout sur la structure de la base dans ce truc. Comme vous vous en doutez, c'est magique ! Et pour tout vous dire, c'est extra-magique : ce dictionnaire de données est organisé en tables et en vues, et on peut regarder ce qu'il y a à l'intérieur avec du simple SQL^①. Par contre, personne, même un administrateur, ne peut modifier explicitement ce dictionnaire.

L'utilisateur à qui appartient le dictionnaire est **SYS** (pas un humain).

Vous pouvez vous balader dans le dictionnaire de données en utilisant la vue **DICTIONARY** ou son petit raccourci **DICT**. Cela vous donnera la liste des vues et tables associées au dictionnaire de données, avec leur description.

Utilisez **SELECT** pour récupérer des lignes et **DESCRIBE** pour avoir des informations sur les champs de votre vue ou de votre table.

① Tables

Les tables qui le composent se terminent généralement par un **\$**. Ainsi, on a **SYS.TAB\$** la table des tables, **SYS.TS\$** la table des tablespaces... On peut ou non les consulter, cela dépend de vos privilèges.

② Vues

Il y a trois catégories de vues statiques qu'on distingue par leur préfixe :

① SQL est une abbréviation qui signifie : le langage le plus merveilleux du monde.

USER_* Décrit les objets qui appartiennent à l'utilisateur courant

ALL_* Décrit les objets qui sont accessibles à l'utilisateur courant

DBA_* Décrit tous les objets (ces vues ne sont accessibles que par l'administrateur)

Ainsi on aura **USER_TABLES**, **ALL_SYNONYMS**... Vous verrez donc que pour récupérer toutes les tables de la base de données, il faut faire **DBA_TABLES** et non pas **ALL_TABLES**.

Notez que vous pouvez récupérer la définition exacte de chaque vue, c'est-à-dire la requête **SELECT** qui sert à constituer la vue. Voici un exemple avec la vue **DBA_USERS** :

- 1 SQL> **SET** LONG 2000
- 2 SQL> **SELECT** text **FROM** DBA_VIEWS **WHERE** view_name = 'DBA_USERS'

3 Vues couramment utilisées

Vous aurez souvent à remplacer * par **USER** (ou par **ALL**, selon votre but) étant donné que vous n'aurez pas d'accès **DBA** au serveur de la fac.

DICT Un raccourci pour **DICTIONARY**

DICTIONARY Vous savez déjà ce que c'est, non ?

***_TABLES** Tables (nom, namespace, stockage, statistiques, cluster éventuel).

TABS Un raccourci pour **USER_TABLES**

***_TAB_COLUMNS** Colonnes des tables (nom de la colonne, type, longueur, obligatoire)

COLS Un raccourci pour **USER_TAB_COLUMNS**

***_VIEWS** Vues (nom, texte de l'ordre SQL associé, type)

***_INDEXES** Indexs (nom, table indexée, unicité, stockage, statistiques)

IND Un raccourci pour **USER_INDEXES**

***_IND_COLUMNS** Nom de l'index, nom de la table, nom de la colonne, position et longueur

***_CLUSTERS** Clusters (nom, stockage, statistiques)

CLU Un raccourci pour **USER_CLUSTERS**

***_OBJECTS** Objets (tables, vues, index, clusters, synonymes, procédures, fonctions, packages, séquences)

OBJ Un raccourci pour **USER_OBJECTS**

***_SEQUENCES** Séquences (valeur minimum, valeur maximum, incrément, cycle, cache)

SEQ Un raccourci pour **USER_SEQUENCES**

- *_USERS** Caractéristiques générales de l'utilisateur (nom, tablespace par défaut, tablespace temporaire)
- *_CONSTRAINTS** Contenu des contraintes (nom, type, table d'accueil, statut)
- *_DB_LINKS** Contenu des liens vers des bases distantes (nom, user distant, mot de passe, serveur distant, date de création)
- *_TAB_PRIVS** Contenu des privilèges donnés ou reçus (bénéficiaire, propriétaire, créateur)
- *_EXTENTS** Caractéristiques des extents (nom du segment, nom de la partition, nom du tablespace, taille en octets, taille en blocs)
- *_SEGMENTS** Caractéristiques des segments (nom du segment, type du segment, nom du tablespace, taille en octets et en blocs, nombre maximum d'extents)
- *_TS_QUOTAS** Quotas d'écriture autorisée sur les tablespaces (nom du tablespace, taille maximum en octets et en blocs)

SECTION
2

Requêtes à essayer

1 Paramétrer la sortie

On utilisera plusieurs fonctions de SQL*Plus pour paramétrer la sortie.

```

1 % 150 caractères par ligne
2 SET LINES 150
3 % 40 lignes par page
4 SET PAGESIZE 40
5 % Attend appui sur une touche entre chaque page
6 SET PAUSE ON
7 % Connaître le temps exécution des requêtes
8 SET TIMING ON

```

2 Regarder le résultat d'une fonction

On peut regarder le résultat d'une fonction en faisant une sélection sur la table DUAL, qui est une table bidon faite pour ça.

```

1 SQL> SELECT substr('coucou',1,2) FROM dual;
2
3 SU
4 --
5 CO

```

3 Afficher le schéma de la base de données

Vous voulez savoir quel est le schéma de votre base sous la forme :

TABLE	COLONNE	TYPE	TAILLE
...

Et vous voulez qu'on n'affiche que les 30 premiers caractères du champ COLONNE. C'est possible. Par ici la compagnie !

```

1 SELECT
2   user_tab_columns.table_name AS "table",
3   user_tab_columns.column_name AS "colonne",
4   user_tab_columns.data_type AS "type",
5   user_tab_columns.data_precision AS "taille"
6 FROM user_tab_columns;
```

4 Joujou sur tablespaces

a Afficher les tablespaces dans lequel l'utilisateur U possède des tables ou des index

```

1 SELECT all_tables.tablespace_name as "tablespace"
2 FROM all_tables
3 WHERE owner = 'U'
4 UNION
5 SELECT all_indexes.tablespace_name as "tablespace"
6 FROM all_indexes
7 WHERE owner = 'U';
```

b Afficher, pour chaque tablespace, le nombre d'extents libres, leur taille moyenne, leurs opt

```

1 SELECT
2   count(*) as nb_ext_libres,
3   tablespace_name,
4   avg(bytes),
5   min(bytes),
6   max(bytes),
7   sum(bytes)
8 FROM user_free_space
9 GROUP BY tablespace_name;
```

c Afficher, pour chaque tablespace, et par intervalle de tailles le nombre d'extends libres

On cherche à savoir, pour chaque tablespace et pour chaque intervalle de taille ([0; 5000[, [5000, 10000[...]) le nombre d'extents libres.

```

1 SELECT
2   count(*) as nb_ex_lib,
3   tablespace_name,
4   '[',
5   trunc(bytes/5000) * 5000 AS binf,
6   '-',
7   (trunc(bytes/5000)+1) * 5000 AS bsup,
8   ']'
9 FROM user_free_space
10 GROUP BY tablespace_name, trunc(bytes/5000);

```

5 Chercher les tables pour lesquelles un nouvel extent ne pourra pas être alloué

```

1 SELECT table_name
2 FROM user_tables
3 WHERE max_extents = (
4   SELECT count(*) FROM user_extents
5   WHERE segment_name = table_name
6 );

```

On peut faire bien plus compliqué en se souvenant comment sont alloués les extents. On sait que la taille du n -ième extent ($n > 2$) est $\text{NEXT_EXTENT} * (1 + \text{PCT_INCREASE})^{n-2}$.

```

1 SELECT table_name
2 FROM user_tables
3 WHERE NOT EXISTS (
4   SELECT * FROM user_free_space
5   WHERE bytes >= (
6     SELECT next_extent * power(1+pct_increase,count(*)-1)
7     FROM user_extents
8     WHERE segment_name = table_name
9   )
10 );

```

6 Calculer la taille prise par les tables utilisateur dans chaque tablespace de la base

La fonction `nvl(a,v)` permet de substituer `v` à `a` si `a` est nul.

```

1 SELECT
2   tablespace_name,
3   nvl(sum(bytes), 0)
4 FROM user_segments AS us, dba_tablespaces AS dba_t
5 WHERE
6   segment_type = 'table'
7 AND
8   us.tablespace_name = dba_t.tablespace_name
9 GROUP BY tablespace_name;

```

7 Calculer la taille prise par les données utilisateur d'une table T(a,b,c)

```
1 SELECT  
2   sum(vsize(a) + vsize(b) + vsize(c))  
3 FROM t;
```

8 Calculer la taille prise par les données de gestion d'une table T(a,b,c)

```
1 SELECT  
2   count(distinct DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid))*50  
3     + count(*) * (3+2+3)  
4 FROM t;
```

Transactions, concurrence

SECTION



Notion de transaction

On peut imaginer une transaction comme un programme qui commence par un commit, un rollback ou un début de session, et qui se termine par un commit, un rollback ou une fin de session.

Les transactions ont quatre propriétés qu'on regroupe sous le terme mnémotechnique ACID :

Atomicité Une transaction est exécutée complètement, ou pas du tout.

Cohérence Les transactions doivent respecter les contraintes d'intégrité.

Isolement Chaque transaction doit fonctionner comme si elle était seule (il n'y a pas d'effet de bord entre elles).

Durabilité L'effet des transactions est pérenne, même en cas de panne.

Certains ajoutent une cinquième propriété :

Légalité La transaction doit respecter les privilèges d'accès qui ont été spécifiés.

SECTION



Problématique posée par la concurrence

Il peut y avoir des problèmes posés par la concurrence (exécution simultanée) au sein d'une base de données. Nous allons voir quelques exemples.

On peut tout d'abord perdre des mises à jour, comme on peut le voir sur la figure 2.1.

Transaction 1	Transaction 2
lire A	
	lire A
modif A=A*2	
	modif A=A*3
écrire A	
	écrire A

Transaction 1 n'a servi à rien !

Figure 2.1. Perte de mise à jour

On peut aussi lire la base de données dans un état incohérent, comme montré sur la figure 2.2.

Avant, A=B

Transaction 1	Transaction 2
lire A	
	lire B
	modif B=B+10
	écrire B
	lire A
	modif A=A+10
	écrire A
lire B	

Après, A≠B

Figure 2.2. Lecture dans un état incohérent

Et de la même manière on peut introduire des incohérences dans la base.

En fait, l'exécution en parallèle de deux transactions est correcte si et seulement si on a le même résultat avec l'exécution en série des deux transactions. On réunit tout cela dans le concept de **sérialisation**.

Une exécution parallèle de transactions est dite **sérialisable** s'il existe une exécution en série de ces mêmes transactions qui donne un résultat équivalent.

Verrouillage

Pour ce cours on verra deux types de verrous ^① :

- les verrous en lecture seule (LR);
- et les verrous en lecture/écriture (LRW).

Vous comprendrez qu'il peut y avoir plusieurs LR en même temps (on ne fait que lire, quoi), alors qu'un LRW est exclusive et empêche de poser tout autre verrou LR ou LRW. C'est le minimum à faire pour éviter les problèmes qu'on a vu avant. Notez que LR n'existe pas sous Oracle.

① Graphe de précédence

On va devoir déterminer si une exécution parallèle donnée est sérialisable pour pouvoir l'exécuter sans problème, et pour cela on utilise un outil nommé **graphe de précédence**. Dans ce graphe, les nœuds sont les transactions et les arcs représentent la précédence entre les opérations de ces transactions.

Si on a $T_i \rightarrow T_j$, alors :

- T_i a écrit une ressource g avant que T_j ne l'ait lue;
- ou T_i a lu une ressource g avant que T_j ne l'ait écrite;
- ou bien T_i (resp. T_j) a écrit une ressource g avant que T_j (resp. T_i) ne l'ait aussi écrite.

En gros, on a un arc entre T_i et T_j quand ils utilisent la même ressource et que le mode est incompatible.

Si on imagine la situation de la figure 3.1, alors le graphe de précédence sera celui de la figure 3.2.

Que faire de ces outils? Hé bien sachez qu'une exécution parallèle de transactions est sérialisable si son graphe de précédence est sans circuit (sans prendre en compte les étiquettes). Dans la figure 3.2 il y a malheureusement des circuits.

② Protocole de verrouillage

Le **protocole de verrouillage en deux phases**, c'est de dire que tous les verrous doivent être posés avant la levée qu'un quelconque verrou. Il garantit la sérialisabilité des exécutions parallèles des transactions.

Oracle pose des **verrous implicites** lors des `INSERT`, `SELECT`, `DELETE`, `UPDATE` ou ce que vous voulez, et ne les libère que lors d'un `COMMIT`, un `ROLLBACK`, ou une fin de session. C'est donc un protocole strict de verrouillage en deux phases et ça augmente drastiquement le nombre de verrous dans le temps.

En plus de ces verrous implicites, on a des **verrous au niveau table**, qui empêchent les autres de faire des `ALTER TABLE`. Et en plus de ça, il y a même des **verrous au niveau**

^① À ce sujet, répétez plusieurs fois très vite : ``Verra, verrous!''.

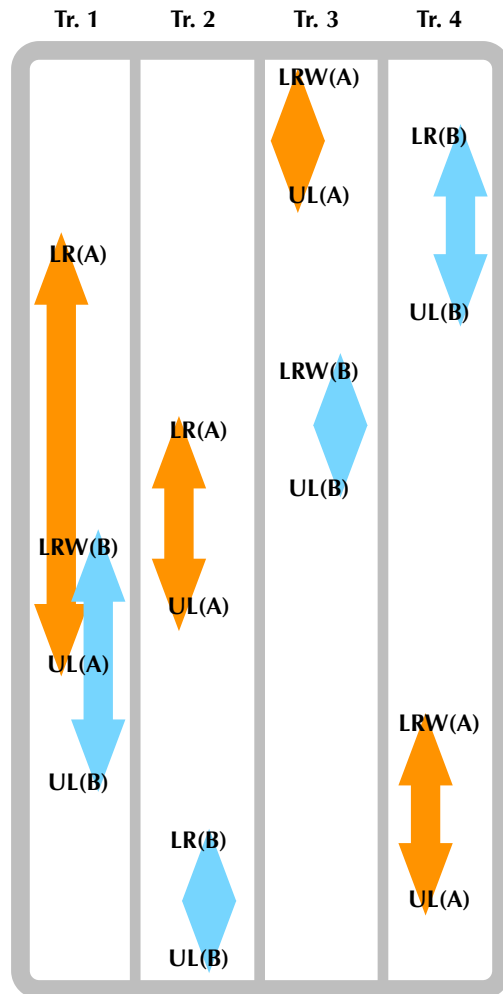


Figure 3.1. UL ça veut dire Unlock

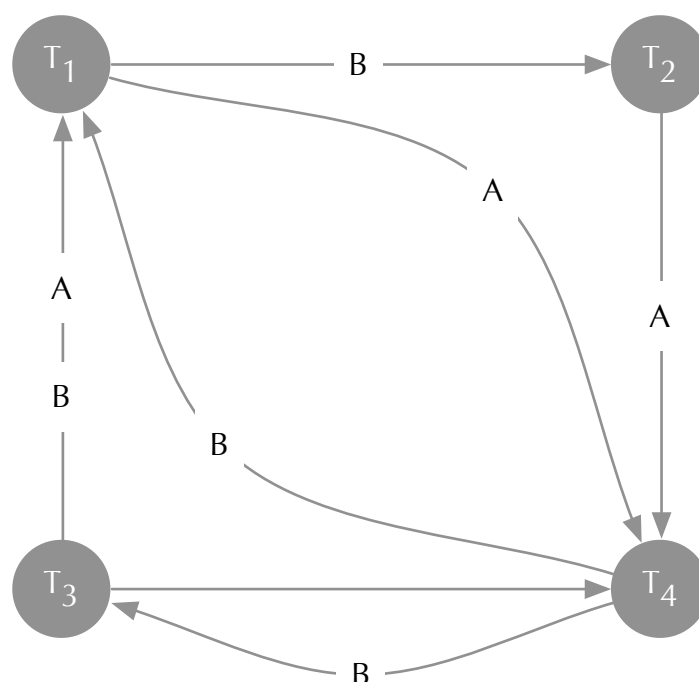


Figure 3.2. Graphe de précédence

ligne, dont le plus courant représentant (on dit aussi que c'est le mieux) est le **row exclusive (RX)**.

Lorsqu'on fait un **SELECT** dans une transaction T_1 , Oracle travaille sur un cliché (*snapshot*). Pendant ce temps-là (entre le début et la fin du **SELECT** de T_1), si un **UPDATE** se produit sur les mêmes données dans une autre transaction T_2 , alors Oracle garde des rollback segments, non seulement pour pouvoir retourner en arrière, mais également pour que T_1 continue de voir les données à l'état où elles étaient avant le **UPDATE**. Ces rollback segments sont supprimés lors du prochain **COMMIT** de T_2 , alors T_1 a l'erreur **Snapshot too old**. Et c'est foutu, voilà ! Vous voyez ainsi qu'Oracle gère les concurrences à un niveau très fin.

3 Verrous mortels

Un **interblocage**, c'est quand par exemple T_1 attend T_2 , qui lui-même attend T_3 , qui lui-même attend T_1 . Chacun attend l'autre et du coup tout le monde est bloqué. On détecte ce genre de choses avec un graphe des attentes : s'il y a un circuit dans le graphe des attentes, alors on a affaire à un **verrou mortel**.

On a deux façons d'empêcher ça :

Méthode de prévention DIE_WAIT Une transaction T_i est mise en attente seulement si elle est plus vieille que T_j , et sinon elle est tuée. Donc en gros, les jeunes transactions meurent et les plus vieilles attendent.

Méthode de prévention WOUND_WAIT Les jeunes transactions attendent et les plus vieilles sont blessées (*wounded*). Une vieille transaction peut être en attente d'une jeune seulement si cette dernière n'est pas en attente d'autre chose. Sinon, la jeune est tuée. Kaput!

SECTION

4

Gestion de la concurrence

La gestion de la concurrence, ce n'est pas qu'une question de verrous. Il y a également des problèmes de vision incohérente de la base de données, de contrainte d'intégrités. Il y a aussi des problèmes d'intrusion d'incohérences, c'est-à-dire que deux transactions travaillant sur les mêmes variables, mais pas en même temps, peuvent causer une intrusion d'incohérences.

Pour vérifier qu'une exécution parallèle fonctionne, il faut agir en plusieurs étapes.

- Etape 1** On doit savoir si l'exécution fonctionne, en regardant s'il n'y a pas plusieurs verrouillages LRW sur la même variable en même temps.
- Etape 2** On construit le graphe de précédence, où les nœuds sont les transactions, et où il y a un arc entre deux transactions s'ils utilisent la même variable avec des modes incompatibles.
- Etape 3** On détermine si l'exécution est sérialisable, en regardant si le graphe de précédence est bien sans circuit.
- Etape 4** On vérifie si le protocole de verrouillage en deux temps est respecté, c'est-à-dire que tous les verrous doivent être posés avant un quelconque déverrouillage. Si ce n'est pas respecté, on peut avancer la pose d'un verrou, ou retarder le déverrouillage : il faut en général regarder la ressource la plus utilisée pour diminuer son temps de verrouillage.
- Etape 5** On construit un graphe des attentes où les nœuds sont les transactions et les arcs représentent une attente de la source vers la destination au niveau d'une variable. S'il y a un circuit dans ce graphe, alors il se produira un verrou mortel et la galaxie sera détruite. Pour éviter ça, il faudra tuer une transaction. C'est sacrifier un être pour sauver tout un peuple!

Résumé des séances

ANNEXE

A

Voici un court résumé plus ou moins correct de ce qui a été fait dans les séances de cours de SGBD.

ANNEXE A



Cours magistral

Vous avez le plan du CM dans le sommaire de ce document, voilà tout !

ANNEXE A



Travaux dirigés

Je me base sur la session 2009-2010 pour les séances, mais le contenu est basé sur les TD de la session 2008-2009. J'indique si on peut trouver les réponses en se servant de ce document ou pas en mettant ♠ à la fin d'un thème.

1 B-tree

- Construction d'un B-tree d'ordre 1 qui contient quelques valeurs ♠
- Suppression de quelques valeurs dans le B-tree ♠
- La même à l'ordre 2 (pas trop ♠ mais en fait si)
- Combien de valeurs peut-on stocker au maximum dans un B-tree d'ordre m de niveau n et avec un taux de remplissage t ?

Pour un arbre à un seul niveau, ça fait $2mt$ valeurs, pour un arbre à deux niveaux ça fait $2mt + (2mt + 1) * 2mt$ valeurs, et donc en généralisant on a $\sum_{i=0}^{n-1} 2mt(2mt + 1)^i$ valeurs.

2 Fichiers à organisation indexée

- Rappels ♠
- Les primitives ♠
 - Iecind
 - ecrind
- Algorithmes

3 Fichiers à organisation indexée sous INGRES (B-tree)

- Généralités ♠
- Avantages et inconvénients (moyen ♠)
- Algorithmes (pas ♠ mais on sait comment ça fonctionne)

4 Organisation aléatoire dynamique

- Construction d'un fichier à organisation aléatoire dynamique (non linéaire) qui contient quelques enregistrements ♠
- Même chose avec une organisation aléatoire dynamique linéaire
- Algorithmes (pas vu mais pour l'organisation aléatoire statique et dynamique non linéaire, ça peut se trouver)

5 Schéma du dictionnaire de données

- Explications sur chaque vue du dictionnaire de données ♠

6 Requêtes sur le dictionnaire de données

- Plein de requêtes ♠

7 Organisation aléatoire partitionnée

- Présentation
- Traitement d'une requête avec l'organisation aléatoire partitionnée sous contraintes de temps
- Algorithmes

8 Jointure

- Algorithmes et coûts ♠
- Jointure avec deux index secondaires ♠

9 Optimisation de requêtes

- Exemple avec des jointures ♠
- Deux approches d'optimisation ♠
- Arbre algébrique d'une requête ♠
- Chercher parmi trois algorithmes classiques le moins coûteux pour exécuter une jointure sur deux tables, avec toutes les données (nombre de tuples par bloc, nombre d'enregistrements, taille de la mémoire centrale...) (pas ♠ mais ça se fait)

10 Concurrence dans les SGBD

- Exemple d'exécution parallèle de transactions avec des verrous ♠
- Vérification de la validité d'une séquence ♠
- Vérification de la sérialisabilité d'une exécution avec graphe de précédence ♠
- Vérification du protocole de verrouillage en deux phases ♠
- Réécriture d'une transaction pour vérifier le protocole sus-cité ♠
- Trouver une exécution en série équivalente à une exécution parallèle ♠
- Prévention de verrous mortels par `DIE_WAIT` et `WOUND_WAIT` ♠

ANNEXE A

3

Travaux pratiques

Les trois premières séances ont été consacrées à la création d'une base de données recensant des étudiants. Il fallait créer des procédures PL/SQL pour remplir les tables avec des données extrêmement diverses, dans le but de plus tard étudier l'optimisation de requêtes avec une bonne table bien grasse. Une séance a permis de faire de l'optimisation de requêtes, et une autre à faire du JDBC pour connecter un programme Java à une base de données Oracle.



Utiliser le serveur Oracle de l'université

1 Connexion à une session

Vous devez d'abord vous connecter à une session sur votre compte utilisateur à partir d'un serveur de la faculté de sciences.

a Depuis la fac

Entrez dans une salle de machines quelconque disposant d'un bon vieux système d'exploitation plutôt UNIX, virez tout le monde à grands coups de pompes dans le boule et ouvrez un terminal. Voilà.

b Depuis un autre endroit

Il vous faudra vous connecter à l'aide de **ssh**, un outil magique établissant une connexion à votre compte utilisateur droit votre shell préféré.

Sous Windows, téléchargez **PuTTY** et arrangez-vous pour vous retrouver sur ufrsciencestech.u-bourgogne.fr et pour rentrer vos infos de connexion.

Sous un système d'exploitation plus respectable et forcément UNIX, ouvrez un Terminal si vous n'êtes pas déjà en train de vous baigner dedans, et exécutez :

```
1 $ ssh ab123456@ufrsciencestech.u-bourgogne.fr
```

Remplacez **ab123456** par votre propre nom d'utilisateur¹. Le serveur vous laissera rentrer dans un shell sur le serveur kundera en échange de votre mot de passe. Si vous voulez utiliser des programmes utilisant X11 et que vous n'avez peur de rien, alors vous pouvez rajouter l'option **-X** qui connectera l'export display (c'est extrêmement lent).

¹ Vous pouvez également, si vous en avez l'audace, remplacer `ufrsciencestech` par `depinfo`. Si vous avez l'audace de constater que ça marche pas, sachez que c'est parce que le caractère `$` au début de la ligne n'est pas à taper : il indique seulement que vous tapez cette ligne dans un *shell* en tant qu'utilisateur.

c Suite et fin

Il ne vous reste plus qu'à vous prendre un bon pavé de bœuf. Ah non, j'écris dans les mauvaises notes de cours... Il faut maintenant se connecter sur le serveur **stendhal** (ou **butor**) avec **ssh**. Vous devinez la commande ?

```
1 $ ssh stendhal
```

Votre mot de passe tapé, il ne vous reste plus qu'à lire le paragraphe suivant.

2 Préparatifs

Vous n'avez encore jamais utilisé Oracle? Méchant! Préparez-vous. Tapez tout ceci dans votre shell :

```
1 $ echo "export TERM=xterm; export ORACLE_BASE=/opt/oracle; export
  ORACLE_HOME=/opt/oracle/product/10r2; export ORACLE_SID=ens091;" |
  cat - /export/home/oracle/oraenv.sh > ~/oraenv.sh
```

Il s'agit en réalité de modifier à la volée un script qui définit quelques variables d'environnement bien utiles pour accéder à l'installation d'Oracle. Oui, c'est pas si facile d'utiliser Oracle, il faut le mériter. Pourquoi? C'est une question posée à l'examen. Une fois ce fichier créé, vous pouvez passer à l'étape suivante.

3 Connexion au serveur Oracle

Vous êtes dans votre session *shell*, tranquille, et vous connaissez l'existence du fichier **oraenv.sh** puisque vous l'avez déjà créé auparavant. Vous n'avez plus qu'à le *sourcer* :

```
1 $ source ~/oraenv.sh
```

Une fois ceci fait, vous pouvez lancer SQL*Plus :

```
1 $ sqlplus ab123456
```

Ce joli petit amour de programme vous demande mot de passe, c'est également votre **ab123456**.

Vous avez donc remarqué qu'après cette petite dizaine d'opérations, vous pouvez enfin travailler. C'est juste que vous n'en avez plus envie. Ne vous inquiétez pas, c'est encore pire après!

Habituez-vous à deux petits raccourcis qui fonctionnent dans SQL*Plus par défaut. Il s'agit de **C-c** suivi de l'appui sur **Entrée** pour annuler la requête qui est en cours sans forcément déclencher une erreur ni tout effacer. Et le plus utile : **C-d** pour vous échapper de ce terrible programme et retourner à la vie réelle.



Rendre SQL*Plus vivable

Je ne sais pas ce qui est le pire entre le fait que SQL*Plus est un invite de commandes SQL conçu pour être tout à fait désagréable et le fait que presque aucun prof ne vous dira comment faire pour bien utiliser ce programme comme un pro. Ce n'est pas le but de cette section, à moins que vous ne vouliez soumettre vos idées. Ici nous allons juste voir deux petites astuces pour que votre SQL*Plus paraisse, sinon agréable, au moins vivable.

1 Utiliser un éditeur externe

Voilà qui vous facilitera la vie lorsque vous éditez sur la fac. SQL*Plus permet d'éditer le buffer courant avec un éditeur de texte quelconque que vous pouvez définir. Ce n'est pas trop utile en dehors de la fac où les éditeurs nécessitent l'actualisation d'au moins toute la fenêtre du terminal, et avec la lenteur de la connexion c'est un calvaire. Comment faire ? Déjà, définissons notre éditeur. Prenons le meilleur :

```
1 SQL> def _editor = vi
```

Comme d'habitude, si ça ne marche pas, alors il faudra éviter de taper `SQL>` qui est le *prompt*. Ensuite, où vous voulez, lorsque vous en ressentez le besoin pressant, éditez votre buffer courant avec :

```
1 SQL> edit
```

Trop facile ! Sauvegardez le fichier et quittez l'éditeur pour revenir à SQL*Plus avec le buffer modifié.

2 Facilités de la ligne de commande

Vous avez très probablement l'habitude de pouvoir effectuer certaines actions lors de l'édition de commandes sur le shell, par exemple revenir quelques caractères avant le curseur avec la flèche gauche, ou revenir dans l'historique avec la flèche du haut et le raccourci-clavier `C-r` de recherche incrémentale inversée. Vous déplorez également que ce genre de facilités ne soit pas présent partout, dans tous les éditeurs. Le temps des malheurs est terminé ! Toutes ces facilités sont offertes par la bibliothèque GNU **readline** et il est possible d'assigner cette dernière à l'entrée standard de tout programme grâce à l'utilitaire `rlwrap`². Cette astuce a fait des ravages dans la communauté Oracle. Pour que ça marche avec SQL*Plus, exécutez-le en préfixant `rlwrap` comme suit :

```
1 $ rlwrap sqlplus ab123456
```

Le plus beau, c'est que ce programme magique sert aussi pour tous les programmes chiants du type. Je pourrais citer au hasard le mode interactif d'Objective Caml (enfin il sert à quelque chose).

² <http://utopia.knoware.nl/~hlub/uck/rlwrap/>

Ah oui, il y a encore plus beau : ce programme n'est pas installé sur le serveur **stendhal** et ce dernier ne possède même pas les outils GNU pour compiler notre programme ! Vous pouvez demander l'installation de cet outil fort pratique aux administrateurs en passant la requête par les profs, par exemple ^③ .

^③ L'adresse suivante donne les packages et les commandes pour installer **rlwrap** sur Solaris vite fait bien fait : http://www.pinegin.com/2009/08/26/install-oracle_10g-opensolaris-sunos-5-11/

L'élaboration de ces notes de cours a nécessité l'ensemble des documents suivants :

- D1 Planning de cours, T. Grison, 2009-2010**
Une jolie feuille très généraliste reçue du maître en personne.
- D2 Notes de cours de la boîte à chaussures, 2008-2009**
Les notes de cours incomplètes et baveuses d'un inconnu.
- D3 Notes de cours de Nancie, 2009**
Le début des notes de cours d'une fille qui s'enflamme dès qu'on prononce son nom.
- D4 Notes de cours de JB, 2009-2010**
Des notes d'un mec qui humainement, avait la classe. Je préférerais avoir sa classe plutôt qu'avoir la mienne. Moi, je suis un peu just...
- L1 Systèmes de gestion de bases de données par l'exemple, R. Godin, 2004, éd. Loze-Dion**
Un énorme livre qui se veut généraliste mais qui profite de chaque exemple (et il y en a plein, si vous regardez le titre du livre) pour se la ramener sur Oracle. Un coup de bol.
- L2 Oracle, H. Smine, 1995, éd. Eyrolles**
Un livre assez vieux, écrit par un mec d'Oracle France, qui croit très fort en SQL (qui est un langage génial). Bien pour les généralités, les choses qui ne changeront jamais et parce que c'est écrit gros.
- L3 Oracle Performance Tuning, E. Whalen et M. Schroeter, 2002, éd. Addison-Wesley**
Quelques trucs de cowboy pour mettre des néons sur son installation d'Oracle. Contient aussi des petits rappels sur les organisations de fichier, l'optimisation des requêtes, les algorithmes des opérateurs de l'algèbre relationnelle et l'utilisation d'`EXPLAIN PLAN` en détail.
- N1 Support de cours de SGBD, P. Rigaux, 2004**
Plusieurs points du cours sont abordés là-dedans, c'est du bon.
- N2 Support de cours sur le noyau d'Oracle 8, P. Borla-Salamet, Oracle France, 2000**
Des généralités avec un plan qui se voulait plus clair que celui de ce cours (du général vers le détail).

NS **Architecture Oracle 8, T. Grison, 2010**

Les parchemins sacrés. Une référence indispensable mais incomplète. Heureusement, sinon ça voudrait dire que j'ai travaillé pour rien.

W1 **Administration Oracle, D. Deléglise, 2006**

Des bons tuyaux sur le dictionnaire de données, mais aussi tout plein de choses.