# Bases de Données

## C9 - Optimisations

Lina Soualmia

Université de Rouen
LITIS - Équipe TIBS-CISMeF
lina.soualmia@chu-rouen.fr

18 novembre 2015



# Optimisations



### Gestion physique des données

- Adaptation et enrichissement du SLD
  - pour tenir compte des spécificités du SGBD
  - des performances des traitements et de la sécurité
- À ce stade d'élaboration du schéma physique de données, on dispose :
  - du SLD normalisé ou non
  - du schéma physique de traitement (description détaillée des modules)



### Résultat : Schéma Physique des Données (SPD)

- Représentation de la structure définitive de la base de données telle qu'elle doit être implantée
- prise en compte des caractéristiques techniques du SGBD et du matériel
- des exigences en interfaces et en performances des modules de programmation
- Si le matériel ou le SGBD change alors il faut changer le SPD (les exigences changent)



### Gestion physique des données

- État des lieux : des tables en 3ème FN, ou autre etc. bien conçues

Objectifs :

- optimisation des accès en choisissant les colonnes qui doivent être indexées
- introduction (éventuellement) de redondances calculées
- définition de vues standards et des vues matérialisées
- définition des contraintes d'intégrité et des déclencheurs sur les tables



### Gestion physique des données

- création des tables et des colonnes techniques
- définition des droits d'accès aux données
- répartition des tables sur les sites (en cas de BD réparties)
- calcul des paramètres de stockage des tables et des index
- ...etc.



### Colonnes à indexer

- les clés primaires (concaténées ou pas) pour garantir l'unicité et accélérer les recherches
  - `INDEX UNIQUE`
- les clés étrangères (concaténées ou pas) pour accélérer les jointures
  - `INDEX NOT UNIQUE`
- les colonnes servant de critères de recherche (apparaissent souvent dans la clause `WHERE` de SQL)
  - `INDEX NOT UNIQUE`



### Gestion des méthodes d'accès aux données

- Méthodes d'accès :
  - Méthode séquentielle
  - Méthodes basées sur les index :
    - Index hiérarchisé,
    - Arbre-B (B-tree), Arbre-B+, Arbre-B+*, ...
  - Méthode de hachage
  - Mise en cluster

## Différents types d'index

- Logique
  - Index simple (sur une colonne)
  - Index concaténé (sur plusieurs colonnes)
  - Index unique (colonne ne comportant pas de doublon)
  - Index non unique (colonne comportant des doublons)
- Physique
  - B-tree
  - Bitmap
  - Hash



## Exemples

- Index simple

```
CREATE INDEX idx_no_four ON tt_four(no_four);
```

- Index concaténé

```
CREATE INDEX idx_commande ON (no_four, n_client);
```



- Catalogue Oracle pour la gestion des Index et des Clusters

```
SELECT * FROM sys.dba_users;
SELECT * FROM sys.dba_index_columns;

SELECT * FROM sys.dba_clusters;
SELECT * FROM sys.dba_clu_columns;
```



## Cluster

- Création du cluster

```
CREATE CLUSTER client_hash
    (Num_cli NUMBER (9)
     SIZE 512 HASHKEY 500
     STORAGE (initial 2M next 1M);
```

- Mise en cluster d'une table

```
CREATE TABLE Client(N_cli NUMBER, nom VARCHAR2(50));
CLUSTER client_hash(N_cli);
```

- Création de l'index sur le cluster

```
CREATE INDEX Idx_Client ON CLUSTER client_hash;
```



## Optimisation des requêtes

Visualisation :

- Expression Algébrique
- Arbre algébrique
- Plan d'exécution de la requête



## Plan d'exécution

- Sous Oracle, possibilité de connaître pour chaque requête son plan d'exécution
- Consultation du plan d'exécution selon lequel une instruction SQL sera exécutée par le SGBD dans une table système
- Stockage des plans d'exécution dans une table PLAN_TABLE dont le script de création (fourni par Oracle) se trouve dans $ORACLE_HOME/rdbms/admin/utlxplan.sql



## Plan d'exécution

- Étapes en vue de la consultation :

1. Création la table de nom `PLAN_TABLE` (éventuellement la détruire et la recréer)
2. Exécution d'une commande `EXPLAIN PLAN` sur une requête
3. Stockage (description) du plan d'exécution de cette dernière dans la table `PLAN_TABLE`
4. Interrogation de la table `PLAN_TABLE`



## Exécution d'une commande EXPLAIN PLAN sur une requête

```
EXPLAIN PLAN
SET statement_id='exple1' --identifiant
FOR  --exemple de requête
SELECT [] FROM []WHERE ;
```

Exemple :

```
EXPLAIN PLAN
SET statement_id ='requete1'
FOR SELECT * FROM eleves WHERE nom LIKE 'DUPON%';
```

## Affichage du plan d'exécution

Interrogation de la table PLAN_TABLE

```
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options||' '||object_name "Plan d'execut
FROM plan_table
START WITH id = 0 AND statement_id = 'requete1'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete1';
```

Résultat affiché :

```
Plan d'exécution
----------------------------------------------
SELECT STATEMENT
   TABLE ACCESS BY INDEX ROWID ELEVES
        INDEX RANGE SCAN I_NOM
```





## Remarques et explications

- Représentation d'une requête par un arbre
- Sous Oracle, possibilité de représentation et de manipulation de données ayant une structure de données récursive (arbre) par une réprésentation tabulaire et avec la commande SQL : `CONNECT BY PRIOR`





- Requête simple sur une table indexée

```
DROP INDEX i_nom;
CREATE INDEX i_nom ON eleves(nom);

DELETE plan_table; --la table étant créée
-- il suffit de la vider

EXPLAIN PLAN
SET statement_id ='requete1'
FOR SELECT * FROM eleves WHERE nom LIKE 'DUPON%';

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options||' '||object_name
FROM plan_table
START WITH id = 0 AND statement_id = 'requete1'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete1';
```





```
Plan d'exécution
----------------------------------------------
SELECT STATEMENT
   TABLE ACCESS BY INDEX ROWID ELEVES
        INDEX RANGE SCAN I_NOM
```





- Requête avec un `order by` sur une table non indexée

```
DROP INDEX i_nom;
DELETE plan_table;

EXPLAIN PLAN
SET statement_id ='requete2'
FOR SELECT * FROM eleves WHERE nom LIKE 'DUPON%'
ORDER BY nom;

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||option||' '||object_name "
FROM plan_table
START WITH id = 0 AND statement_id='requete2'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete2';
```





```
Plan d'exécution
----------------------------------
SELECT STATEMENT
  SORT ORDER BY
    TABLE ACCESS FULL ELEVES
```





- Requête avec un `order by` sur une table indexée

```
DROP INDEX i_nom;
CREATE INDEX i_nom ON eleves(nom);

DELETE plan_table;

EXPLAIN PLAN
SET statement_id ='requete2'
FOR SELECT * FROM eleves WHERE nom LIKE 'DUPON%'
ORDER BY nom;

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||option||' '||object_na
FROM plan_table
START WITH id = 0 AND statement_id='requete2'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete2';
```





```
Plan d'exécution
-----------------------------------------
SELECT STATEMENT
  SORT ORDER BY
    TABLE ACCESS BY INDEX ROWID ELEVES
      INDEX RANGE SCAN I_NOM
```

## Optimisation des requêtes : Comparaison

- Requête avec un `order by` sur une table non indexée

```
Plan d'exécution
-----------------------------------
SELECT STATEMENT
  SORT ORDER BY
    TABLE ACCESS FULL ELEVES
```

- Requête avec un `order by` sur une table indexée

```
Plan d'exécution
-----------------------------------------
SELECT STATEMENT
  SORT ORDER BY
    TABLE ACCESS BY INDEX ROWID ELEVES
      INDEX RANGE SCAN I_NOM
```





- Requête de jointure avec les deux tables indexées

```
DELETE plan_table;
EXPLAIN PLAN
SET statement_id ='requete3'
FOR SELECT nom, prenom, num_cours, points
FROM eleves e, resultats r
WHERE r.num_eleve=e.num_eleve;

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||option||' '||object_name "Plan
FROM plan_table
START WITH id = 0 AND statement_id='requete3'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete';
```





```
Plan d'exécution
-----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS FULL RESULTATS
    TABLE ACCESS BY INDEX ROWID ELEVES
```





- L'ordre des 2 tables change

```
DELETE plan_table;
EXPLAIN PLAN
SET statement_id ='requete3'
FOR SELECT nom, prenom, num_cours, points
FROM resultats r, eleves e
WHERE r.num_eleve=e.num_eleve;

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||option||' '||object_name "I
FROM plan_table
START WITH id = 0 AND statement_id='requete3'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete3';
```





```
Plan d'exécution
-------------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS FULL ELEVES
    TABLE ACCESS BY INDEX ROWID RESULTATS
```





- Requête de jointure avec les deux tables indexées

```
Plan d'exécution
-----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS FULL RESULTATS
    TABLE ACCESS BY INDEX ROWID ELEVES
      INDEX UNIQUE SCAN PK_ELEVES
```

- Requête de jointure avec les deux tables indexées (l'ordre des 2 tables change)

```
Plan d'exécution
-------------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS FULL ELEVES
    TABLE ACCESS BY INDEX ROWID RESULTATS
      INDEX RANGE SCAN PK_RESULTATS
```





- Requête de jointure avec les deux tables indexées et une sélection

```
DELETE plan_table;
EXPLAIN PLAN
SET statement_id ='requete4'
FOR SELECT nom, prenom, num_cours, points
FROM eleves e, resultats r
WHERE r.num_eleve=e.num_eleve
AND e.num_eleve=10;

SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||option||' '||object_name "I
FROM plan_table
START WITH id = 0 AND statement_id='requete4'
CONNECT BY PRIOR id = parent_id
AND statement_id ='requete4';
```



```
Plan d'exécution
-------------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS BY INDEX ROWID ELEVES
      INDEX UNIQUE SCAN PK_ELEVES
    TABLE ACCESS BY INDEX ROWID RESULTATS
      INDEX RANGE SCAN PK_RESULTATS
```

## Autre requête `EXPLAIN PLAN` équivalente

```
EXPLAIN PLAN FOR
SELECT *
FROM   emp e, dept d
WHERE  e.deptno = d.deptno
AND    e.ename  = 'SMITH';

SET LINESIZE 130
SET PAGESIZE 0
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY);
```





```
-----------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    58 |     4   (0)| 00:00:01
|   1 |  NESTED LOOPS                |         |       |       |            |
|   2 |   NESTED LOOPS               |         |     1 |    58 |     4   (0)| 00:00:01
|*  3 |    TABLE ACCESS FULL         | EMP     |     1 |    38 |     3   (0)| 00:00:01
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |     1 |       |     0   (0)| 00:00:01
|   5 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     1 |    20 |     1   (0)| 00:00:01
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("E"."ENAME"='SMITH')
   4 - access("E"."DEPTNO"="D"."DEPTNO")

18 rows selected.
```





```
EXPLAIN PLAN SET STATEMENT_ID='TSH' FOR
SELECT *
FROM   emp e, dept d
WHERE  e.deptno = d.deptno
AND    e.ename  = 'SMITH';

SET LINESIZE 130
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY('PLAN_TABLE','TSH','BASIC'));
```





```
-----------------------------------------------
| Id  | Operation                    | Name    |
-----------------------------------------------
|   0 | SELECT STATEMENT             |         |
|   1 |  NESTED LOOPS                |         |
|   2 |   NESTED LOOPS               |         |
|   3 |    TABLE ACCESS FULL         | EMP     |
|   4 |    INDEX UNIQUE SCAN         | PK_DEPT |
|   5 |   TABLE ACCESS BY INDEX ROWID| DEPT    |
-----------------------------------------------

12 rows selected.
```





## Comment améliorer le temps de réponse des applications ?

- Partitionnement **
- Exécution parallèle
- Tuning des entrées/sorties
- Détection des Index inutilisés
- Réorganisation des index
- Utilisation des Hints
- Codes SQL à ne pas utiliser **





## Création d'une table partitionnée sous Oracle :

```
Create table facture
( num_fact number (5),
  num_article number(5),
  date_vente date
)
Partition by range (date_vente)
 (
   partition p_200312 value less than
            (to_date('2004-01-01','YYYY-MM-DD'))
   partition p_200406 value less than
            (to_date('2004-07-01','YYYY-MM-DD'))
   partition p_200412 value less than
            (to_date('2005-01-01','YYYY-MM-DD'))
)
```





## Stratégies d'indexation

- Index de fonction :

```
CREATE INDEX idx_clts_nompren
ON client(upper(nom), prenom);

CREATE INDEX idx_clts_ca
ON client(calcul_ca(id_client));
-- calcul_ca est une procédure PL/SQL
```



## Les codes SQL à ne pas écrire

- Éviter l'utilisation des index dans les clauses `where` : consultation par les instructions SQL d'un plus grand nombre de blocs de données
  - en utilisant l'index
  - plutôt qu'en exécutant un balayage complet de la table
- Amélioration :
  - ajout d'une expression anodine à la colonne d'index : ajout de +0 à une colonne numérique ; concaténation d'une chaîne vide à une colonne alphanumérique

## Les codes SQL à ne pas écrire

- Éviter de mélanger ou de comparer des valeurs et des types de données de colonnes : l'optimiseur ignorera l'index
- Améliorations :
  - Si le type de colonne est NUMBER : ne pas utiliser de guillemets pour encadrer la valeur
  - Ne pas oublier d'encadrer une valeur avec des guillemets lorsqu'elle est définie comme de type alphanumérique

```
SELECT ... FROM EMP WHERE SALARY>'1000'; --Mauvais
SELECT ... FROM EMP WHERE SALARY>1000; --Bon
SELECT ... FROM EMP WHERE MANAGER=SMITH; --Mauvais
SELECT ... FROM EMP WHERE MANAGER='SMITH'; --Bon
```





## Les codes SQL à ne pas écrire

- Ne pas utiliser l'opérateur IS NULL dans une colonne indexée : l'optimiseur ignorera l'index
  - SELECT Nemp, name, adr FROM EMP WHERE name IS NULL;
- En cas d'utilisation de curseurs ou du SQL dynamique :
  - ne pas coder pas les instructions avec des valeurs en dur
  - empêche la réutilisation du code SQL dans le pool partagé : utiliser des variables liées.

```
SELECT ... FROM EMP WHERE EMPNO=2324; --Mauvais
SELECT ... FROM EMP WHERE EMPNO=:1; -- Bon

EXECUTE IMMEDIATE 'DELETE FROM EMP WHERE EMPNO=2324'; --Mauvais
EXECUTE IMMEDIATE 'DELETE FROM EMP WHERE EMPNO=:1' USING VARIABLE; --Bon
```





## Les codes SQL à ne pas écrire

- Pas d'instruction INSERT, UPDATE ou DELETE dans une itération (une boucle PL/SQL) si les opérations peuvent être réalisées en vrac

```
DECLARE CURSOR ... SELECT ... FROM
BEGIN
    FOR CURSOR IN ... LOOP
        INSERT TO Z VALUES (CURSOR.Col1, CURSOR.Col2*1.5, ...);
        COMMIT;
    END LOOP;
END; --Mauvais

BEGIN
    INSERT INTO Z SELECT Col1, Col2*1.5 FROM X WHERE;
    COMMIT
END; --Bon
```





## Les codes SQL à ne pas écrire

- Évitez l'utilisation des sous-requêtes : impact négatif sur les performances du système en consommant beaucoup de ressources CPU
- Solution : utilisation des vues en ligne (inline views) : des sous-requêtes dans la clause FROM de l'instruction SELECT

```
SELECT e.* FROM EMP e
WHERE e.salary > (SELECT AVG(salary)
                  FROM EMP i
                  WHERE i.dept_id = e.dept_id); --Mauvais

SELECT e.* FROM EMP e,
    (SELECT i.dept_id DEP, AVG(i.salary) SAL
    FROM EMP I GROUP BY dept_id) EMP_VUE
WHERE e.dept_id = EMP_VUE.deptid
AND e.salary > EMP_VUE.SAL; --Bon
```





## Les codes SQL à ne pas écrire

- Évitez les produits cartésiens : ne pas construire la clause from d'une instruction select avec des tables qui n'apparaissent pas dans les conditions de jointure de la clause where
- Regrouper la création des tables et l'insert
  - Si la table est créée et alimentée, regrouper l'opération

```
CREATE TABLE X(col1 NUMBER, col2 VACHAR2(30)..);
INSERT INTO X SELECT col1, col2 FROM Y ...; --MAUVAIS

CREATE TABLE X(col1 NUMBER, col2 VARCHAR2(30)...)
AS SELECT col1, col2 FROM Y...; --BON
```





## Les codes SQL à ne pas écrire

- Éviter l'utilisation de select x from dual : elle peut consommer l'essentiel des performances du système

```
FOR i IN 1..10000 LOOP
   SELECT SEQ.NEXTVAL INTO var FROM DUAL;
   INSERT INTO X VLUES (var, ...);
END LOOP; --MAUVAIS

FOR i IN 1..10000
   INSERT INTO X VALUES(SEQ.NEXTVAL,...);
END LOOP; --BON
```





## Les codes SQL à ne pas écrire

- Eviter l'utilisation de NOT IN
- Privilégier l'utilisation de NOT EXISTS plutôt que NOT IN dans les clauses where
- utiliser une jointure extene (outer join)

```
--not in (lent)
SELECT a.nom, a.prenom
FROM SALARIE a
WHERE a.nom NOT IN (SELECT nom FROM FONCTION WHERE job='INFORMATICEN');

--avec jointure externe (plus rapide)
SELECT a.nom, a.prenom
FROM SALARIE a, FONCTION b
WHERE a.nom = b.nom(+) and b.nom IS NULL
AND b.job = 'INFORMATICIEN';
```





## Les codes SQL à ne pas écrire

- Utiliser l'opérateur LIKE avec un caractère initial plutôt que la fonction SUBSTR()
- Dans le cas de requêtes très complexes contenant de nombreuses conditions OR
  - Envisager leur réécriture à l'aide de UNION ALL
  - Possibilité de découpage de la requête en modules bien dimensionnés qui pourront être optimisés plus facilement
- Rappel :
  - Opérateur UNION : récupération de l'ensemble des enregistrements retournés par les deux requêtes. Les doublons ne sont pas pris en compte
  - Opérateur UNION ALL : récupération de tous les enregistrements y compris les doublons

### Les codes SQL à ne pas écrire

- Utiliser les index appropriés, les plus sélctifs
- La sélectivité des données est le ratio entre le nombre de clés uniques et le nombre de lignes
- Plus elle approche 1.00, meilleur est l'index
- Créer des index sur les colonnes de clés étrangères
- Utiliser des index composites : ceux-ci doivent être classés dans l'ordre décroissant de sélectivité.





À suivre (GIL/ITA) :

- Data Warehouses ; Fouille de données