

# TP – Bases de données réparties

## requêtes réparties

### Version corrigée

Auteur : Hubert Naacke, révision 5 mars 2003

Mots-clés: bases de données réparties, fragmentation, schéma de placement, lien, jointure inter-site, JDBC, plan d'exécution, performance.

### Configuration de l'environnement de travail

Installer sur votre compte l'archive tpbdr.tar : ouvrir une fenêtre shell, puis saisir les commandes :

```
cd
cp /infres/may/testbd/bdtest/tpbdr/tpbdr.tar . // copier l'archive
tar xvf tpbdr.tar // extraire les fichiers de l'archive
cd tpbdr // travailler dans le répertoire tpbdr
source oracle.shenv // configuration pour utiliser Oracle
sqlplus bdaxx/bda@chop1 // client Oracle (choisir un user bdaxx dans {bda02 à bda36}
@creation-plan-table // création de la table pour visualiser les plans d'exécution
set autotrace trace // activer le mode de visualisation du plan d'exécution des requêtes
```

Editer un fichier texte *exemple.sql* qui contiendra les ordres SQL que vous écrirez :

```
xemacs exemple.sql
saisir les réponses du TP, puis sauvegarder
Remarque: les lignes de commentaires commencent par deux tirets :
-- ceci est un commentaire dans un fichier SQL
```

Puis exécuter ensuite les ordres SQL du fichier *exemple.sql* dans la fenêtre du client Oracle (Sql\*Plus) :

```
SQL> @exemple // ne pas saisir le suffixe du fichier (.sql)
```

### Introduction

Le schéma relationnel de l'application est :

**Producteurs**(np, region, ...)

**Recoltes**(np, nv, qte)

**Achat**(nb, nv, date, lieu)

**Buveurs**(nb, nomb, prenom, type)

**BuveursBig** a les mêmes attributs que Buveurs, mais est beaucoup plus volumineuse (200 000 tuples).

Le domaine de l'attribut *type* d'un buveur est {'rare', 'petit', 'moyen', 'gros'}

Les données de l'application sont réparties sur les bases de données suivantes :

Site	nom du service	utilisateur	mot de passe	nom de la BD (SID)
S1	chop1	bda02 à bda36	bda	cours
S2	chop2.enst.fr	bda01	bda	coursv

Pour faciliter la mise en œuvre du TP, certaines tables sont **répliquées** sur les 2 sites, mais Oracle n'a pas connaissance de la réplification pendant l'optimisation des requêtes

Le schéma global (schéma de placement) sera construit sur le site **S1**.

### Exercice 1 : Placement des relations

#### 1.1) Exécution de requête répartie : transfert de requête / transfert de données

Donner les ordres SQL pour créer dans S1 le schéma de placement suivant :

S1 contient Producteurs et Recoltes

S2 contient Achats et Buveurs

Données **locales** : Producteurs et Recoltes existent déjà sur S1.

Données **distantes** :

```
créer un lien S1 ← S2 (voir le fichier ls2.sql)
create database link cours
```

```
connect to bda01 identified by bda
using 'chop2.enst.fr';
```

```
créer dans S1 des synonymes S2Achats et S2Buveurs.
create synonym S2Buveurs for buveurs@coursev;
create synonym S2Achats for achats@coursev;
```

Traiter sur S1 la requête R0 : *Quels sont les buveurs qui ont fait des achats ?*  
Mesurer le temps d'exécution de la requête

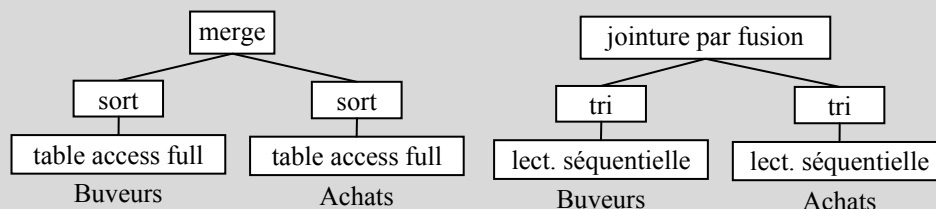
```
(voir le fichier r0a.sql).
set timing on
select * from S2Achats a, S2Buveurs b
where a.nb = b.nb
```

Analyser le plan d'exécution :

```
set timing off
set autotrace trace
select * from S2Achats a, S2Buveurs b
where a.nb = b.nb ;
```

OPERATIONS	OPTIONS	OBJECTS
MERGE JOIN		
SORT	JOIN	
TABLE ACCESS	.....FULL	<b>BUVEURS</b> COURSV.WORLD
SORT	JOIN	
TABLE ACCESS	FULL	<b>ACHATS</b> ..... COURSV.WORLD

La tabulation (espace en début de ligne) indique l'arborescence des opérateurs.



Nom des opérateurs :

```
merge-join = jointure par fusion
sort join = tri préliminaire (avant d'effectuer une jointure par fusion)
table access full nom_relation nom_BD = lecture séquentielle de tous les tuples de la relation
nom_relation sur la base nom_BD.
```

Plan = Jointure-fusion( tri(lecture-séq(Buveurs)), tri(lecture-séq(Achat)) )

Le nom de la base de données est COURSV, la requête est exécutée entièrement sur la BD coursev du site 2.

Quelles sont les opérations traitées sur S2 et S1 ? Quelles sont les données transmises entre S1 et S2 ?

```
sur S2 : T1= (Achat ∞ Buveurs)
puis transfert de T1 de S2 vers S1
puis résultat sur S1
```

Oracle détecte que les données sont sur le même site et transmet la requête sur le site S2.

Seul le résultat de la requête est transféré de S2 vers S1.

Cette solution est plus performante que la solution naïve vue en cours : envoyer toutes les données sur le site où l'utilisateur demande la requête (site S1) pour y traiter la requête.

Remarque :

L'algorithme de jointure locale utilisé dépend de la présence d'un index sur l'attribut de jointure :

- un index : jointure par boucles + index : (index-nested-loop) (voir le fichier *r0.sql*)  
- pas d'index : jointure par tri fusion (sort + merge join) : (voir le fichier *r0\_noindex.sql*)

## 1.2) Exécution de requête avec jointure inter-site

Modifier le schéma de placement pour que la relation Achat soit locale sur S1 :

Sur S1 : Producteurs, Recoltes, Achats

Sur S2 : S2Buveurs

Mesurer le temps d'exécution de la requête R1 : *Donner le nombre de buveurs qui ont fait des achats ?*  
(voir le fichier *r1a.sql*).

```
set timing on
select count(*) from Achats a, S2Buveurs b
where a.nb = b.nb ;
```

Analyser le plan d'exécution :

```
set timing off
set autotrace trace
select * from Achats a, S2Buveurs b
where a.nb = b.nb ;
```

Quelles sont les opérations traitées sur S2 et S1 ? Quelles sont les données transmises entre S1 et S2 ?

sur S2 : T1 = Lecture séquentielle de Buveurs

S2 -> S1 : T1

sur S1 : Achat  $\infty$  T1

Tous les tuples de Buveurs sont transmis de S2 à S1

L'objectif est de mesurer le coût du transfert des données entre deux sites en étudiant une requête de jointure inter-site avec une relation très volumineuse (cardinalité élevée). Modifier le schéma de placement :

- les achats sont sur S1

- les buveurs sont dans la relation **BuveursBig** sur S2. Cette relation contient 200.000 buveurs.

Données distantes :

créer dans S1 le synonyme S2BuveursBig.

```
create synonym S2BuveursBig for BuveursBig@coursev;
```

Analyser la requête R2 : *Donner le nombre de buveurs dans BuveurBig qui ont fait des achats ?*

```
(voir le fichier r1.sql).
set timing on
set autotrace trace
select * from Achats a, S2BuveursBig b
where a.nb = b.nb ;
```

Comparer les temps d'exécution de R1 et R2 : Quelle est l'origine de la baisse de performance entre R2 et R1 ?

Le temps prépondérant est le transfert des tuples de la relation BuveursBig depuis S2 vers S1.

Proposer une solution pour améliorer le temps de réponse de R2.

Objectif : réduire le volume des données transférées entre les sites S1 et S2.

Exigence : la requête est posée sur S1 donc le résultat de la requête doit être sur S1.

On constate que

$\text{volume}(\text{Achat}) + \text{volume}(\text{résultat de la requête}) < \text{volume}(\text{BuveursBig})$

Donc il est avantageux de transférer Achat de S1 vers S2 pour traiter la jointure sur S2 puis de transférer le résultat de la requête sur S1.

S1 doit transmettre à S2 une requête inter-sites pour traiter la jointure sur le site S2 qui contient BuveursBig (la plus grosse relation). Le scénario à construire est :

- L'utilisateur se connecte à S1 et pose la requête globale RG1
- Pour traiter RG1, S1 se connecte à S2 et pose la requête T2
- Pour traiter T2, S2 se connecte à S1 et pose la requête T1

Exécution du scénario :

Sur S1: T1 = select \* from Achats, puis transfert vers S2

Sur S2: T2 = T1 >< BuveursBig, puis transfert vers S1

Sur S1: résultat

**Mise en œuvre du scénario :**

Dans S2 :

créer un lien S2 ← S1 // un lien n'est **pas** bi-directionnel  
créer un synonyme S1Achats représentant les Achats de S1

Dans S1 :

créer un synonyme S2S1Achats représentant le synonyme S1Achats de S2  
traiter la requête :  
select count(\*) from S1S2Achats a, S2BuveursBig b  
where a.nb = b.nb

Le plan d'exécution déterminé par l'optimiseur d'Oracle est :

Depuis S1 : transmettre à S2 la requête de jointure inter-site entre S1Achats et BuveursBig

Sur S2 : récupérer les Achats de S1 vers S2

traiter la jointure

renvoyer le résultat à S1

## Exercice 2 : Requêtes inter-site avec JDBC

L'objectif est d'utiliser JDBC pour traiter la requête R2 plus rapidement.

Le plan d'exécution optimal est :

lecture séquentielle de Achats : pour chaque tuple  $t$  de Achat :  
accéder à la relation BuveursBig de S2 en utilisant l'**index** sur l'attribut  $nb$  (attribut de jointure). La  
requête d'accès à S2 est : select nb from BuveurBig where nb =  $t.nb$   
transférer le résultat sur S1.

Configurer l'environnement java :

```
cd jdbc  
source config-jdbc // configuration de l'environnement pour utiliser le pilote jdbc
```

Implémentation du plan : voir le fichier AccesInterbase.java

```
xemacs AccesInterbase.java  
M-x font-lock-mode // fichier java avec couleurs syntaxiques
```

Quelle est le gain de performance par rapport au traitement de la même requête avec Oracle (cf. exercice 1)?

```
javac AccesInterbase.java // compilation du programme java  
time java AccesInterbase // exécution + chronométrage
```

Modifier le fichier AccesInterbase.java pour remplacer la relation BuveursBig par

une relation identique mais qui n'a **pas d'index** sur l'attribut  $nb$  (relation BuveursBig\_noindex).

Comparer le temps de réponse avec la solution précédente : quel est le gain de performance apporté par l'index ?

Proposer un algorithme pour traiter efficacement la requête R3 :

*Quels sont tous les achats des buveurs (de BuveursBig) dont de numéro de buveur est supérieur à 100 ?*

Traiter la sélection ( $nb > 100$ ) d'abord sur Achat. Le résultat étant vide, il est inutile d'accéder à BuveursBig.

Retourner directement le résultat vide.

Comparer les temps d'exécution de R3 avec Oracle et avec JDBC.

## Exercice 3 : Fragmentation horizontale

Définir sur S1 le schéma de fragmentation suivant :

Tous les buveurs de la relation BuveurBig sont fragmentés horizontalement en deux fragments selon

le type du buveur :

- le fragment BuvRare contient les buveurs dont le type est 'rare',
- le fragment BuvAutre contient les autres buveurs.

L'allocation des fragments est la suivante :

sur S1 : le fragment BuvAutre

sur S2 : le fragment BuvRare

Quel est l'ordre SQL pour créer le fragment BuvAutre sur S1 ? (voir le fichier f1.sql)

```
copy from bda01/bda@chop2 create BuvAutre using ( select * from BuveursBig where type <> rare);
commit;
```

Le fragment BuvRare est déjà créé sur S2.

Définir sur S1 la vue VueBuveurs qui représente l'union des fragments BuvRare et BuvAutre.

```
create view VueBuveurs as
select * from BuvAutre
union
select * from BuvRare@coursv;
```

Soit la requête RV1 accédant à la vue des buveurs : *Donner le nombre de gros buveurs.*  
Analyser le plan d'exécution (voir le fichier rv1.sql) :

```
select count(*)
from VueBuveurs
where type = 'gros';
```

Le plan d'exécution est-il optimal ? Proposer un plan d'exécution simplifié (voir le fichier rv2.sql)

Le plan initial avant simplification est :

$(\sigma_{type='gros'}(BuvAutre)) \text{ union } (\sigma_{type='gros'}(BuvRare))$

La sélection est distribuée avant l'union, mais Oracle n'élimine pas l'accès au fragment BuvRare car il n'a pas connaissance de la définition algébrique des fragments.

Pour simplifier le plan, il faut enlever l'accès au fragment BuvRare car

$\sigma_{type='gros'}(BuvRare) = \sigma_{type='gros'}(\sigma_{type='rare'}(BuveursBig))$   
 $= (\sigma_{type='rare' \text{ AND } type='gros'}(BuveursBig)) = \text{ensemble vide}$

Le plan simplifié est :

$\sigma_{type='gros'}(BuvAutre)$

Dans la relation Achats, le plus grand numéro de buveurs est 100.

Quelle est la cardinalité du résultat de R3 :

```
select * from Achats a, BuveursBig b
where a.nb = b.nb and a.nb > 100
```

Résultat vide : cardinalité nulle

Comparer les temps d'exécution des 3 requêtes suivantes. Expliquer pourquoi les temps d'exécution diffèrent.  
En déduire les heuristiques que l'optimiseur d'Oracle utilise.

R3a :

```
select * from S2BUVEURSBIG b, ACHATS a
where a.nb = b.nb and a.nb > 100;
```

R3a est rapide car la collection intérieure (selection nb>100 de Achat)  
ne produit aucun tuple  
donc pas d'accès aux buveurs

R3b:

```
select * from ACHATS a, S2BUVEURSBIG b
where a.nb = b.nb and a.nb > 100 ;
```

R3b: requête lente car la collection intérieure est BuveursBig.  
Oracle n'utilise pas la commutativité de la jointure :  
 $Achats \infty BuveursBig = BuveursBig \infty Achats$

R3c:

```
select * from S2BUVEURSBIG b, ACHATS a  
where a.nb = b.nb and b.nb > 100;
```

R3c : requête lente. La sélection (nb>100) n'est pas poussée sur Achats.

Oracle ne fait pas la déduction suivante :

si (a.nb=b.nb et b.nb > 100) alors a.nb > 100

donc jointure avec la totalité des Achats, puis sélection après la jointure.

Rmq :

Le fait d'ajouter des contraintes d'intégrité référentielle (Buveur.nb est clé primaire, et Achat.nb est une clé étrangère) a-t-il une influence sur les choix de l'optimiseur ?