IS-Net 29 DATA WEBHOUSE

Informatique de gestion et
systèmes d'information

Parallel Execution

Centre de compétences

Hes·so
Haute Ecole Spécialisée
de Suisse Occidentale

LIBD@hegne.ch / Mars 2001

ISNet29_parallel_execution.doc

TABLE DES MATIERES

1 Introduction

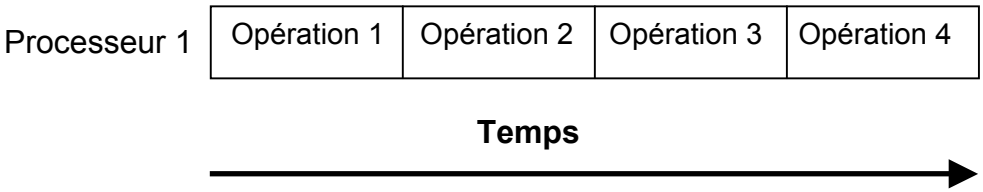
Les temps de réponse pour les opérations sur de grandes bases de données peuvent être drastiquement réduits grâce à l'utilisation de l'exécution d'opérations en parallèles (*Parallel execution* ou PX). Ce moyen d'exécution est notamment avantageux dans les systèmes dévoués aux data warehouses.

2 Les architectures parallèles

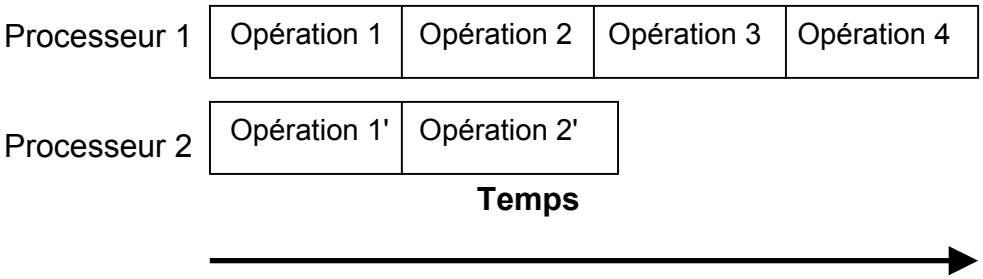
Les raisons d'une machine à architecture parallèle sont évidentes:

- Le fait d'exécuter *n* traitements sur *n* processeurs, les temps de réponse sont identiques et optimaux pour les *n* clients demandeurs.
- En faisant collaborer plusieurs processeurs pour un seul traitement que l'on peut découper, on diminue en théorie d'autant les temps de réponse.

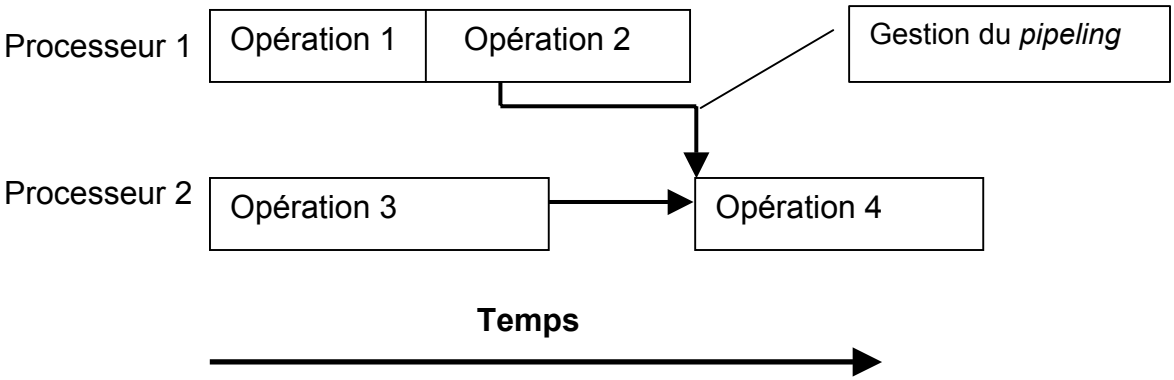
Le graphe suivant illustre la première constatation, il représente le temps d'exécution de deux opérations sur une machine monoprocesseur:



Le graphe suivant montre le moyen le plus simple pour tirer parti d'une machine multiprocesseur. Dans ce cas, tout processeur peut prendre en charge l'exécution d'une opération ou d'un ensemble d'opérations. La limitation des processeurs tient ici à ce qu'ils ne peuvent travailler en parallèle que sur des opérations indépendantes les unes des autres (on parle alors de parallélisme *interrequête*). Les deux opérations étant indépendantes l'une de l'autre, elles peuvent s'exécuter en parallèle:



Le graphe suivant représente une étape supplémentaire dans la gestion du parallélisme: plusieurs processeurs peuvent travailler ensemble pour optimiser l'enchaînement d'un ensemble d'opérations éventuellement interdépendantes, dans le cadre d'une même requête. Les différentes opérations étant interdépendantes, elles ne peuvent pas s'exécuter totalement en parallèle, et sont synchronisées entre elles par un mécanisme de *pipelining*. Le *pipelining* est une méthode de transfert des résultats d'une opération à une autre aussitôt que ceux-ci sont disponibles, et non pas seulement à la fin de la première opération. Dans l'exemple, l'opération 4 bénéficie de ce mécanisme. Ainsi, un processeur peut travailler sur des données qui lui sont transmises par un autre processeur. On parle alors de parallélisme intra requête.



2.1 Types d'architectures

Il existe différents types de systèmes multiprocesseurs bénéficiant du principe d'exécution en parallèle, voici les plus important :

- systèmes SMP (S*ymmetric Multi-Processors*): tous les processeurs sont identiques et ils ont le même niveau de priorité.
- systèmes MP (*Multi-Processing*): les processeurs sont spécialisés pour certaines fonctions.
- systèmes Cluster: mise en commun de plusieurs machines.
- systèmes MPP (M*assively Parallel Systems*): dernière génération de machines.

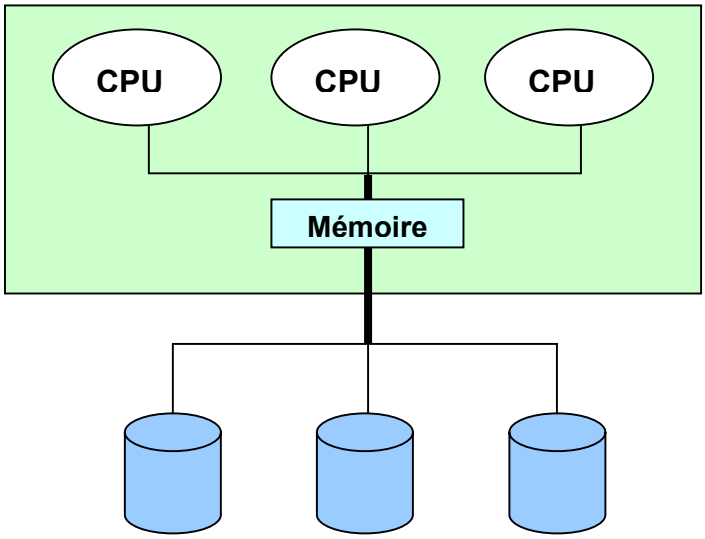
Ces systèmes devront en outre avoir une ou plusieurs caractéristiques suivantes pour bénéficier de l'exécution en parallèle:

- suffisamment de bande passante (E/S),
- la sous-utilisation des CPUs (typiquement dans les systèmes ou l'utilisation du CPU est moins de 30%),
- suffisamment de mémoire pour supporter des processus en plus, tels que des tris (*sort*), *hashing* et *I/O buffers.*

2.1.1 Architecture SMP

Les machines SMP sont les systèmes les plus répandus sur le marché. Ce type de système est constitué d'une seule mémoire et de *n* processeurs reliés entre eux, et la mémoire par un bus.

Architecture SMP



Ce système est basé sur des processeurs bons marchés mais dont le rapport prix/performance est excellent. L'inconvénient est qu'il existe un goulet d'étranglement sur le bus pour accéder à la mémoire.

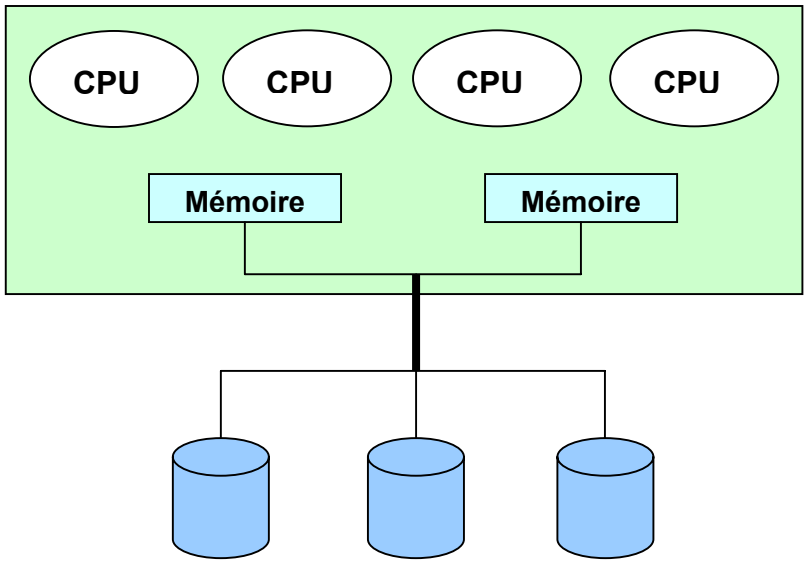
L'évolutivité de ce système reste limitée par le goulet d'étranglement que représente le bus. Ainsi la courbe des performances reste linéaire jusqu'à un certain nombre de processeurs, puis ensuite les performances se stabilisent. Le nombre de processeur dépendra en fait de la puissance du bus qui relie les processeurs et la mémoire.

C'est une technologie très répandue chez la plupart des fournisseurs.

2.1.2 Architecture NUMA

L'architecture NUMA (*Non Uniform Memory Acces*) est dérivée de SMP, c'est-à-dire que tous les processeurs ont la même priorité et partagent un unique espace adressable mémoire. L'objectif de NUMA est de contourner les limites du bus en reliant plusieurs grappes de processeurs SMP par un bus secondaire, au sein d'une même machine. Globalement, tous ces processeurs partagent la même mémoire logique, mais chacun sait reconnaître comme un unique espace adressable logique sa mémoire primaire, située dans sa grappe de processeurs, et l'ensemble des mémoires secondaires, situées dans les autres grappes. C'est une solution plus économique concernant les architectures avec beaucoup de processeurs, en effet un système constitué de deux cartes SHV (*Standard High Volume*, fabriqués par Intel) de quatre processeurs sera plus économique qu'un système SMP équipé de huit processeurs.

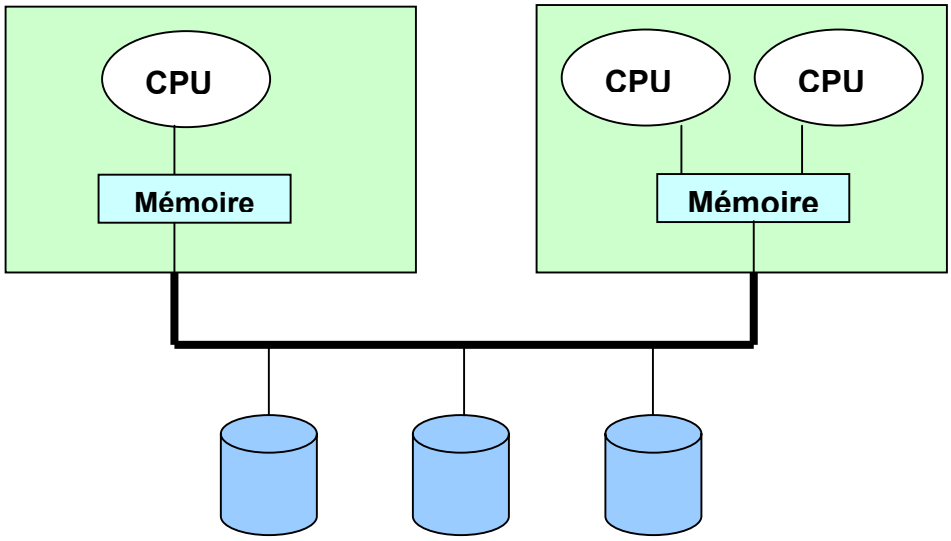
Architecture NUMA



2.1.3 Architecture à mémoire distribuée : clusters

SMP et NUMA sont toutes deux des architectures à mémoire dite partagée. Clusters et MPP sont des architectures à mémoire distribuée : tous les processeurs (ou chaque grappe de processeurs à mémoire partagée) sont alors regroupés dans une même machine logique. Il s'agit alors de faire qu'ils communiquent de façon à ce qu'ils réalisent des tâches communes. L'impact sur les applications logicielles (système d'exploitation, base de données, etc.) est qu'il y a plusieurs espaces mémoire adressables autonomes dans l'architecture. Ainsi plusieurs instances d'une base de données fonctionneront-elles en simultané, et le logiciel devra les synchroniser et les faire apparaître comme une seule et unique instance aux couches applicatives supérieures et aux administrateurs.

Les clusters sont des ensembles de machines (ou grappes) mises en réseau et présentant une image unique (*Single Image*) aux applications. Chaque machine est nommée un *nœud*. Le CPU et les entrées/sorties sont partagées, les machines sont reliées par un réseau à très large bande passante (utilisant le plus souvent la fibre optique).



Les machines se partagent les mêmes disques et sont dotées d'un mécanisme logiciel de reprise sur panne. Ce type de système est dit à haute disponibilité car si une machine tombe en panne, l'autre peut prendre le relais immédiatement. La haute disponibilité est ainsi le principal atout des clusters. La performance est leur second atout, sachant que ces deux machines pourront unir leur puissance au profit des applications.
Dans la pratique, cela se concrétise par des systèmes à 2, 4 ou 8 nœuds. Ils peuvent fédérer des machines multiprocesseurs (cluster de machines SMP ou NUMA). Le principal avantage c'est que si la machine hébergeant une base de données devient insuffisante et si elle n'est plus extensible, on peut ajouter d'autres machines pour en faire qu'elles apparaissent comme une seule et unique instance pour les applications clientes.

Nous ne parlerons pas des architectures MPP (*Massively Parallel Processing*) qui dérive de SMP dont le bus a été supprimé. Ces systèmes demandent une adaptation logicielle et une administration spécifique. Comme le SGBD doit s'adapter à ces systèmes, le coût est relativement élevé par rapport à SMP et NUMA, donc très peu utilisé.

*2.2 L'exécution en parallèle (*Parallel execution*)*

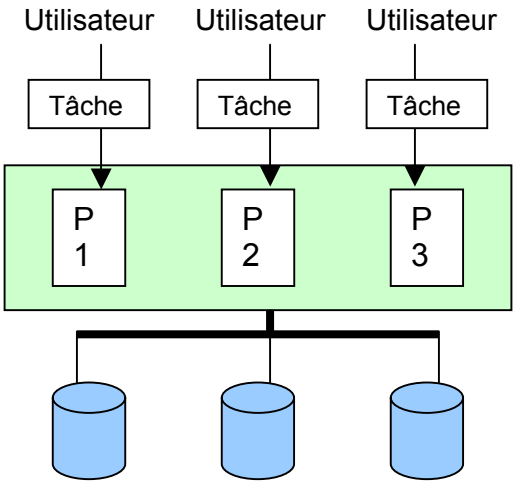
Les SGBD dans un environnement parallèle de type décisionnel sont confrontés à différents types d'opérations:

- Requêtes complexes en accès en lecture sur de gros volumes de données.
- Opérations de calcul lourd (batch) et des opérations de maintenance (chargement, sauvegarde, re-construction d'index …).

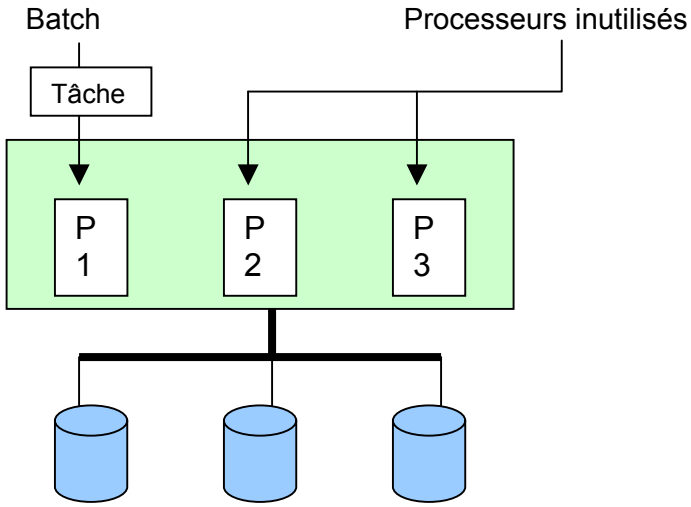
L'objectif du SGBD sera de faire en sorte que les opérations soient le plus partagées entre les ressources du système:

- Partager les utilisateurs sur les processeurs pour ne pas les pénaliser mutuellement,
- Partager les données sur plusieurs processeurs pour effectuer des traitements en parallèle.

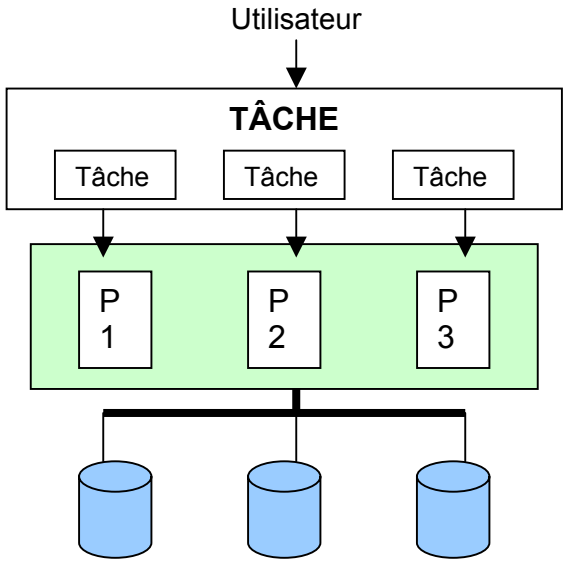
Dans un système simplifié, on peut affecter un processeur à une tâche, et donc à un utilisateur.



Dans un système décisionnel, si le nombre d'utilisateur est faible, il y aura une sous utilisation des processeurs:



La solution à ce problème consiste à découper les tâches en sous-tâches, c'est-à-dire à diviser une requête en opérations plus élémentaires, et à répartir ces dernières sur les différents processeurs. L'optimiseur de la base de données sera chargé de cette décomposition. La capacité de parallélisation du SGBD dépendra, non seulement de la qualité de l'optimiseur, mais aussi des algorithmes de traitements de chaque opération ensembliste.



Les opérations, comme la sélection, sont divisées en autant de tâches parallèles que possible, chacune traitant une partie (un fragment) de table.
Les opérations plus complexes (impliquant plusieurs tables), comme les jointures, demandent des algorithmes de jointure parallèle par hachage (en général), car ils favorisent le traitement en parallèle.

Un problème qui se pose aux SGBD dans un environnement de type Cluster ou MPP, c'est celui des mémoires multiples. En effet, plusieurs machines se partagent les même données, se qui pose un problème de verrouillage (chaque processeur possède une copie d'une page dans sa propre mémoire). Pour résoudre ce

problème, il est apparu les *Lock Manager*, qui coordonnent le verrouillage d'une manière centralisée, soit distribuée. Il existe aussi une autre solution qui consiste en une synchronisation par échanges de message. Nous n'allons pas étudier ce cas de figure.

2.3 Parallélisme disque

Nous venons de voir que l'on peut paralléliser des requêtes grâce aux CPU. Il est aussi possible de le faire sur des machines monoprocesseurs: c'est le parallélisme des entrées/sorties. Comme l'accès aux disques est souvent un goulet d'étranglement, il est intéressant de pouvoir accéder en parallèle sur ceux-ci.

Dès l'instant qu'une machine possède plusieurs disques, les entrées/sorties peuvent être parallélisées.

En répartissant les données sur les différents contrôleurs, les bénéfices sont :

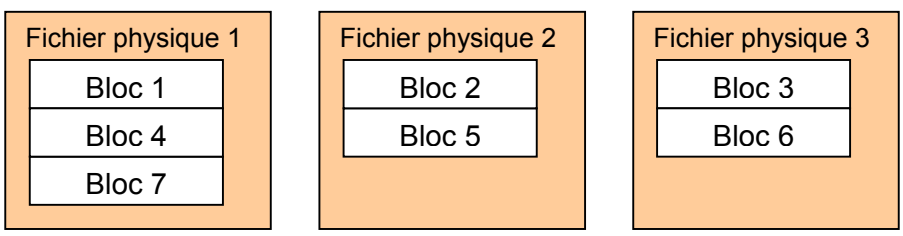
- Les contentions en écriture sont réduites, la charge se répartit sur tous les contrôleurs.
- Les lectures sont distribuées sur tous les contrôleurs qui ne lisent que la partie de données qu'ils ont stockée. Le temps de lecture des données est donc divisé par le nombre de contrôleurs sollicités (les contrôleurs lisent en parallèle). Il faut pour cela bien répartir les données sur tous les contrôleurs.
- Si la fragmentation physique correspond au critère de recherche de l'utilisateur, seul le contrôleur stockant la partie recherchée est sollicité. Les autres sont libres pour d'autres requêtes. Seule la partie de la table intéressante pour la requête est lue et traitée.

La méthode de répartition la plus simple consiste à séparer les différents types de données de la base : tables, index, journaux des mises à jour, objets longs (BLOB). Ainsi les index et les données indexées seront accédés en parallèle.

Une autre méthode consiste à utiliser le *striping* du système d'exploitation ou du SGBD. Le *striping* logiciel est une méthode de répartition cyclique et automatique d'écriture des blocs de données sur les différents contrôleurs disques de la machine. Chaque demande d'écriture (ce nombre est généralement configurable) est effectuée cycliquement sur un contrôleur différent.

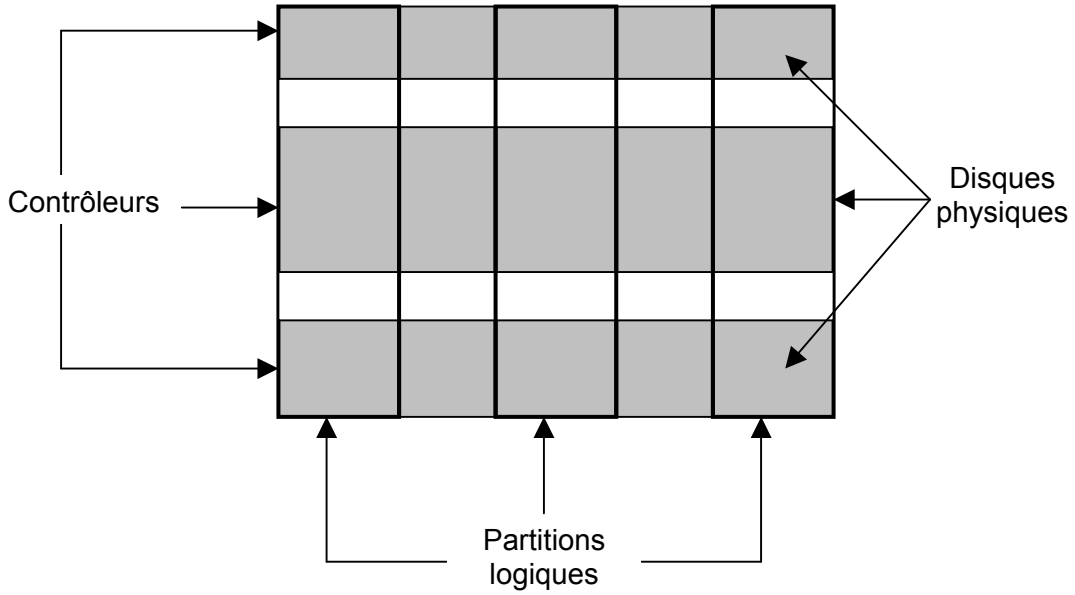
Lorsqu'il y a plusieurs écritures simultanées, elles ne se bloquent pas, puisque les contrôleurs ciblés sont différents. De même, les demandes de lectures de blocs de données se répartissent sur les contrôleurs.

***Striping* logiciel**



Le *striping* matériel consiste quant à lui à créer, au travers du système d'exploitation, un disque logique composé de parties de plusieurs disques physiques. Le SGBD utilise le disque logique. Les écritures de blocs disques sont réparties de manière transparente par le système d'exploitation sur les disques physiques.

***Striping* matériel**



Dans l'exemple ci-dessus, trois partitions logiques sont physiquement *stripées*, c'est-à-dire équitablement réparties, sur trois disques.

La zone système utilisée pour stocker les tables temporaires doit être *stripée* en priorité, et ce afin d'optimiser les tris, opérations très gourmandes en entrées/sorties.

Une autre technique, consiste à fragmenter verticalement une table. Cela consiste à découper physiquement une table en un sous-ensemble de tables plus réduites. Un exemple fréquent est de fragmenter une table par unité de temps, ce qui provoquera une lecture d'une seule partition dans le cas ou l'on sélectionne des données pour un mois.

3 Quand faut-il implémenter le Parallel execution

Le *Parallel execution* assure de bonnes performances dans les systèmes décisionnels et les *datawarehouses*. Les systèmes de types OLTP peuvent également en bénéficier lors de l'utilisation de *batch*.

D'une manière général, les processus bénéficant de l'exécution en parallèle sont :

- les requêtes requérant de *large table scan*, des *joins* ou pour les *partitioned index scan*,
- la création de grands index,
- la création de grandes tables (incluant les vues matérialisées),
- l'insertion (*insert*), la mise à jour (*update*), la fusion (*merge*) et l'effacement (*delete*) de gros volumes de données, l'accès à de grands objets tels que les *LOBs*.

3.1 Les opérations qui peuvent être parallélisées

Les types d'opérations pouvant être parallélisés sont:

1. les méthodes d'accès
   a. *Table Scans*, *Index Full Scans* et *Partioned Index Range Scans*
2. les méthodes de jointure (*Join*)
   a. *Nested Loop*, *Sort Merge*, *Hash* et *Star Transformation*
3. les instructions DDL
   a. *CREATE TABLE AS SELECT, CREATE INDEX, REBUILD INDEX, REBUILD INDEX PARTITION et MOVE SPLIT COALESCE PARTITION*
4. les instructions DML
   a. *Inserts as Select, Updates, Deletes et Merges*
5. les diverses opérations SQL
   a. *GROUP BY, NOT IN, SELECT DISTINCT, UNION, UNION ALL, CUBE* et *ROLLUP* ainsi que les fonctions d'agrégat et de table.

3.2 Le Pool du Parallel Execution

Lorsqu'une instance démarre, Oracle crée un *pool* (une zone commune) pour les opérations en parallèles. Le paramètre d'initialisation `PARALLEL_MIN_SERVERS` (zéro par défaut) spécifie le nombre d'exécution en parallèle qui sont créées au démarrage de l'instance.

Lorsqu'une requête SQL est exécutée, l'*optimizer* décide si l'opération s'exécute en parallèle et il détermine le degré de parallélisme (appelé DOP, généralement limité au double du nombre de processeurs) pour chaque opération. Ce dernier représente le nombre de "*parallel execution server*" associé à une seule opération.

Le DOP est spécifié de la manière suivante:

- au niveau d'une expression SQL:

    - avec des *hints,*
    - avec la clause *PARALLEL*,
- au niveau de la session avec la commande *ALTER SESSION FORCE PARALLEL*,
- au niveau de la définition de la table,
- au niveau de la définition de l'index.

Si le nombre d'opérations en parallèles augmente, Oracle crée de nouveaux "*Parallel execution servers*" afin de supporter la nouvelle demande. La limite est fixée avec le paramètre `PARALLEL_MAX_SERVERS`. Si vous fixez ce paramètre avec une valeur trop basse, certaines requêtes ne profiteront pas de l'exécution en parallèle. A l'opposé, si la valeur est trop haute, les performances peuvent s'effondrer.

4 Types de parallélisme

Voici une liste des différents types de parallélismes que l'on peut trouver :

- requêtes en parallèles,
- DDL en parallèle,
- DML en parallèle,
- exécution de fonctions en parallèles,
- autres types de parallélisme…

4.1 Requêtes en parallèles

Vous avez la possibilité de paralléliser des requêtes et sous-requêtes avec l'instruction *SELECT*. Vous avez également la possibilité de le faire pour des instructions DDL ou DML (*INSERT, UPDATE* et *DELETE*). Néanmoins il n'est pas possible de le faire pour des objets distants (*remote*).

Requêtes en parallèles sur des *Index-Organized Tables*

Voici la liste des méthodes de *scannage* en parallèles:

- *Parallel fast full scan* sur des *index-organized table* non-partitionnés,
- *Parallel fast full scan* sur des *index-organized table* partitionnés,
- *Parallel index range scan s*ur des *index-organized table* partitionnés.

4.2 DDL en parallèle

Vous pouvez paralléliser des instructions DDL pour des tables et indexes qui sont partitionnés et non-partitionnés.

Liste des instructions DDL pour des tables et indexes non-partitionnés:

- *CREATE INDEX*

- *CREATE TABLE … AS SELECT*
- *ALTER INDEX … REBUILD*

Liste des instructions DDL pour des tables et indexes partitionnés:

- *CREATE INDEX*
- *CREATE TABLE … AS SELECT*
- *ALTER TABLE … MOVE PARTITION*
- *ALTER TABLE … SPLIT PARTITION*
- *ALTER TABLE … COALESCE PARTITION*
- *ALTER INDEX … REBUILD PARTITION*
- *ALTER INDEX … SPLIT PARTITION* (cette instruction n'est possible que si la partition de l'index (global) est *usable*).

Pour toutes les opérations, la clause *[NO]LOGGING* peut être utilisée.

CREATE TABLE … AS SELECT en parallèle

L'exécution en parallèle permet de paralléliser une requête et de créer une opération de création d'une table comme une sous-requête à partir d'une ou plusieurs tables.

Exemple:

```
CREATE TABLE resume (C1, AVGC2, SUMC3) PARALLEL (5)
AS
SELECT C1, AVG(C2), SUM(C3)
FROM vente_journaliere
GROUP BY (C1) ;
```

4.3 DML en parallèle

Les opérations DML en parallèles (*PARALLEL, INSERT, UPDATE* et *DELETE*) utilisent un mécanisme d'exécution en parallèle qui augmente la vitesse à travers de grosses opérations DML sur de grosses tables et index.

Note: pour utiliser les opérations DML en parallèles, il est impératif d'utiliser le partitionnement.

4.4 Exécution de fonctions en parallèles

Les expressions SQL peuvent contenir des fonctions en PL/SQL, en Java ou des procédures externes en C (faisant partie d'un *SELECT*, une clause *SET* ou une clause *WHERE*). Toute variable d'un package PL/SQL ou attribut statique en Java qui est utilisé par une fonction est susceptible de générer des erreurs lors d'une exécution en parallèle.

Fonctions dans les requêtes en parallèles

Dans une instruction *SELECT* ou une sous-requête de type DML ou DDL, une fonction (utilisateur) peut être exécutée en parallèle si :

elle a été déclarée avec le mot-clé *PARALLEL_ENABLE*; si elle a été déclarée dans un package ou un type et qu'elle ait un *PRAGMA RESTRICT_REFERENCES* qui indique *WNDS, RNPS* et *WNPS*; ou si elle a été déclarée avec *CREATE FUNCTION* et que le système puisse analyser le *body* du code PL/SQL et déterminer que le code n'écrive et ne lise pas dans la base de données sans qu'il ne modifie les variables du package.

- **WNDS** signifie "writes no database state" (ne modifie pas les tables de la base de données).
- **WNPS** signifie "writes no package state" (ne change pas les valeurs des variables du package).
- **RNDS** signifie "reads no database state" (pas de requête sur les tables de la base de données).
- **RNPS** signifie "reads no package state" (ne référence pas les valeurs des variables du package).

Fonctions en parallèles dans des instructions DML et DDL

Dans les instructions DML et DDL en parallèles, c'est idem que pour les fonctions dans les requêtes en parallèles mais avec en plus un RNDS pour le PRAGMA.

Pour les instructions *INSERT … SELECT* ou *CREATE TABLE … AS SELECT*, elles sont parallélisées selon les règles de bases.

4.5 Autres types de parallélisme

En plus d'utiliser des instructions SQL en parallèles, Oracle peut utiliser le parallélisme pour les types suivant:

- *Parallel recovery* (après un crash)
- *Parallel propagation* (réplication)
- *Parallel load* (chargement en parallèle avec *SQL*Loader*)

5 Tuning des paramètres pour le Parallel Execution

Vous avez la possibilité de modifier le paramètre d'initialisation `PARALLEL_AUTOMATIC_TUNING` à `TRUE`. Une fois enclenché, l'ajustage des paramètres (comme par exemple `PARALLEL_THREADS_PER_CPU`) se fait de manière automatique pour l'exécution en parallèle. Oracle détermine et fixe les paramètres (ceux concernant parallélisme) par rapport au nombre de CPU qu'il y a sur la machine. Les valeurs par défaut mises par Oracle pour le *parallel execution* sont généralement optimales pour la plupart des environnements. Néanmoins, il semble que les valeurs affectées à `PARALLEL_MAX_SERVERS` (entre 40 et 160) semblent trop élevée. C'est pourquoi il est recommandé de faire quelques tests.

Vous avez également la possibilité de "tuner" le *parallel execution* manuellement, bien qu'Oracle recommande le tuning automatique. Le tuning manuel requière plus de temps au niveau de l'administration, et est plus facilement sujet aux erreurs de calculs concernant les ressources systèmes.

5.1 Paramétrage manuel et automatique pour le Parallel Execution

Il y a plusieurs manière de "tuner" le *parallel execution*. Un moyen et de mettre l'environnement en mode automatique au moyen du paramètre `PARALLEL_AUTOMATIC_TUNING` à `TRUE`. Il est ensuite possible d'adapter ou de personnaliser le tuning en substituant les valeurs actuelles par de nouvelles.

Vous pouvez également laisser la valeur par défaut pour le paramètre `PARALLEL_AUTOMATIC_TUNING` à `FALSE` et ensuite manuellement fixer les valeurs des paramètres affectant le *parallel execution*. Pour la plupart des environnements OLTP, il n'est pas conseillé de mettre le mode automatique.

5.2 Fixer le degré de parallélisme

Comme il a été vu au point *1.3*, le degré de parallélisme (DOP) peut être spécifié de plusieurs manières.

Formule :

Degré minimal de parallélisme = min. (nb de processeurs x 2, nombre de partitions de la table, nombre de périphériques ou de lecteurs indépendants sur lesquels la table ou les partitions sont enregistrés).

Exemples:

```
ALTER TABLE emp PARALLEL 4 ;
```

```
ALTER INDEX iemp PARALLEL 4 ;
```

```
SELECT /*+ PARALLEL(emp, 4) / COUNT(*) FROM emp ;
```

Hints

Vous pouvez spécifier des *hints* dans une instruction SQL et fixer le DOP pour une table ou un index.

- Le *hint PARALLEL* est utilisé seulement pour les opérations sur les tables. Vous pouvez les utiliser pour les requêtes et instructions DML (*INSERT*, *UPDATE* et *DELETE*).
- Le *hint PARALLEL_INDEX* est utilisé pour les *Index Range Scan* des index partitionés. Lors d'une opération sur un index, le *hint PARALLEL* n'est pas valide et est ignoré.

5.3 Equilibrer la charge de travail (Workload)

Pour une instruction SQL parallélisée par partition, si la charge de travail est distribuée sur les partitions, vous pouvez optimiser les performances en faisant correspondre le nombre de "*parallel execution servers*" avec le nombre de partitions ou en choisissant un DOP qui correspond au nombre de partitions qui est un multiple du nombre de processus.

Exemple: avec une table ayant 10 partitions, on peut utiliser 10 "*parallel execution servers*" (DOP égal à 10) pour faire le travail en 1/10 de temps par rapport à l'utilisation d'un seul processus.

5.4 Règles concernant les requêtes parallélisées

Une instruction *SELECT* peut être parallélisée seulement si elle est conforme aux conditions suivantes:

1. la requête inclus un *hint* (*PARALLEL* ou *PARALLEL_INDEX*) ou que les objets du schéma ont une déclaration *PARALLEL* associée.
2. qu'il y ait au moins une table participant à la requête, soit:
    a. un *Full Table Scan*
    b. soit un *Index Range Scan* s'étalant sur plusieurs partitions

Exemple:

```
INSERT /*+ PARALLEL(tbl_ins,2) */ INTO tbl_ins
SELECT /*+ PARALLEL(tbl_sel,4) */ * FROM tbl_sel ;
```

5.5 Forcer le Parallel Execution pour une session

Si vous désirez exécuter une requête en parallèle et que vous ne désirez pas fixer un DOP pour une table ou une requête (*hint*), vous pouvez forcer le parallélisme avec la commande suivante:

```
ALTER SESSION FORCE PARALLEL QUERY ;
```

Après avoir lancé la commande ci-dessus, il est toujours possible de modifier le DOP avec un *hint*.

6 Tuning général des paramètres pour le Parallel Execution

Ce chapitre met en avant certains paramètres importants intervenant dans le *parallel execution*.

PARALLEL_MAX_SERVERS

Oracle recommande la valeur suivante pour ce paramètre:

`2 x DOP x NOMBRE_DE_USERS_CONCURRENTS`

Si vous fixez le paramètre `PARALLEL_AUTOMATIC_TUNING` à `FALSE`, vous avez besoin de spécifier une valeur pour `PARALLEL_MAX_SERVERS`. La valeur de ce dernier est fixée à 5 par défaut.
Oracle recommande de fixer cette valeur à 16 fois le nombre de CPU. Cette une valeur raisonnable qui permettra d'utiliser quatre requêtes en parallèles simultanément, assumant que chaque requête utilise un DOP de huit.

PARALLEL_MIN_SERVERS

La valeur recommandée pour le paramètre `PARALLEL_MIN_SERVERS` est 0 (zéro), qui est la valeur par défaut.

LARGE_POOL_SIZE ou SHARED_POOL_SIZE

Il n'y a pas de recommandation pour la valeur du paramètre `LARGE_POOL_SIZE`. Oracle recommande de ne pas fixer de valeur pour ce paramètre et d'utiliser le paramètre `PARALLEL_AUTOMATIC_TUNING` à `TRUE`.

Note: quand le paramètre `PARALLEL_AUTOMATIC_TUNING` est à `TRUE`, Oracle utilise la *large pool*. Si ce même paramètre est à `FALSE`, Oracle utilise la *shared pool*.

6.1 Ajuster la mémoire après le début du traitement

Les requêtes SQL dans cette section ne sont qu'un point de départ pour la surveillance de la mémoire. Que vous utilisiez le mode de tuning automatique ou manuel, vous devez surveiller l'usage de la mémoire. Pour se faire, vous pouvez utiliser diverses requêtes :

```
SELECT POOL, NAME, SUM(BYTES)
FROM V$SGASTAT
WHERE POOL LIKE '%pool%'
  GROUP BY ROLLUP (POOL, NAME) ;
```

Le résultat devrait ressembler à ceci:

```
POOL         NAME                            SUM(BYTES)
------------ ------------------------------ ----------
java pool    free memory                      64204800
java pool    memory in use                     2904064
java pool                                     67108864
shared pool  1M buffer                         1049088
shared pool  Checkpoint queue                   141152
…
```

Pour obtenir les informations concernant l'usage de la mémoire pour une session exécutant un *parallel execution*, exécutez la requête suivante:

```
SELECT * FROM V$PX_PROCESS_SYSSTAT WHERE STATISTIC LIKE
'Buffer%' ;
```

Le résultat devrait ressembler à ceci:

```
STATISTIC                       VALUE
------------------------------ ----------
Buffers Allocated                     134
Buffers Freed                         128
Buffers Current                         6
Buffers HWM                            28
```

Nombre de *buffers* (*message*) actuellement utilisés.

Nombre max. de *buffers* (*message*) alloués coucuremment.

Le montant de mémoire utilisé apparaît dans les colonnes `Buffers Current` et `Buffers HWM`.

Exemple de calcul: la valeur pour la *large pool* (`java pool memory in use`) est de 2'904'064 ou 3 Mo. Le `Buffers HWM` à une valeur de 28, ce dernier multiplié par le paramètre d'initialisation `PARALLEL_EXECUTION_MESSAGE_SIZE` (2'148) donne un résultat de 60'144, ou 60 Ko. Dans cet exemple, le "high water mark" a atteint approximativement 2% (60144/2904064) de sa capacité. Ce qui signifie qu'il est très peu utilisé dans notre exemple.

Les vues *V$PQ_SYSSTAT* et *V$PQ_SESSTAT*

Oracle dispose de vues de performances dynamiques V$ permettant de collecter des statistiques pour des opérations de type PQ (*Parallel Query*). Ces statistiques sont disponibles au niveau session ou au niveau système. La vue *V$PQ_SYSSTAT* fournit des informations très intéressantes permettant de déterminer les valeurs des paramètres d'initialisation `PARALLEL_MIN_SERVERS` et `PARALLEL_MAX_SERVERS`.

Exemple d'informations fournies par la table V$PQ_SYSSTAT :

```
SQL> select Statistic, Value from V$PQ_SYSSTAT ;

STATISTIC                           VALUE
------------------------------ ----------
Servers Busy                            0
Servers Idle                            8
Servers Highwater                      30
Server Sessions                        80
Servers Started                       164
Servers Shutdown                      156
Servers Cleaned Up                      0
Queries Initiated                     109
DML Initiated                          19
DFO Trees                             132
Sessions Active                         0

STATISTIC                           VALUE
------------------------------ ----------
Local Msgs Sent                     17886
Distr Msgs Sent                         0
Local Msgs Recv'd                  423102
Distr Msgs Recv'd                       0
```

Si le paramètre *Servers Busy* reste à un niveau proche de la valeur de `PARALLEL_MAX_SERVERS`, il faut éventuellement augmenter celle-ci. A l'opposé, si la valeur de *Servers Busy* reste proche de 0 la plupart du temps, cela signifie peut-être qu'un grand nombre de *servers* PQ (*Parallel Query*) n'est pas nécessaire, ce qui fait que l'on peut réduire la valeur de `PARALLEL_MAX_SERVERS`.

Si la valeur de *Server Busy* dépasse régulièrement la valeur du paramètre `PARALLEL_MIN_SERVERS`, il serait peut-être judicieux d'augmenter cette valeur pour se rapprocher de celle présentée par les statistiques.

Les valeurs présentées par *Server Shutdown* et *Servers Started* peuvent indiquer des demandes de *servers* PQ supplémentaires. La valeur pour *Servers Highwater* indique le nombre maximal de *servers* PQ démarrés à un instant donné.

La vue *V$PQ_SESSTAT* fournit une vue globale des opérations PQ exécutées au cours d'une session. Ces informations ne sont valides que lorsqu'elles sont interrogées à l'intérieur de la même session. Voici un exemple :

```
SQL> select Statistic, Last_Query, Session_Total from
V$PQ_SESSTAT ;

STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    1             2
DML Parallelized                        0             0
DFO Trees                               1             2
Server Threads                          8             0
Allocation Height                       8             0
Allocation Width                        1             0
Local Msgs Sent                       232           464
Distr Msgs Sent                         0             0
Local Msgs Recv'd                     232           464
Distr Msgs Recv'd                       0             0
```

Cet exemple indique certainement que la dernière requête a été exécutée en parallèle, la valeur *Query Parallelized* étant non nulle. La valeur *Allocation Width* renvoie le nombre de fois où la requête a été exécutée, *Allocation Height* le nombre de *servers* PQ demandés par instance, et *Server Threads* le nombre de *servers* PQ utilisés.

Une autre vue, *V$PQ_SLAVE*, fournit des informations sur chaque *server* PQ. Si aucune ligne n'est sélectionnée, cela signifie qu'il n'y a aucun processus esclave.

Exemple :

```
SQL> SELECT slave_name,status, cpu_secs_total
     FROM v$pq_slave;

SLAV STAT CPU_SECS_TOTAL
---- ---- --------------
P000 IDLE               4
P001 IDLE               4
P002 IDLE               4
P003 IDLE               4
P004 IDLE               3
P005 IDLE               3
P006 IDLE               4
P007 IDLE               0
P008 IDLE               0
P009 IDLE               0
P010 IDLE               2
```

DB_FILE_MULTIBLOCK_READ_COUNT

La valeur recommandée par Oracle pour un système OLTP est 8 pour un db_block_size de 8 Ko, ou quatre pour un db_block_size de 16 Ko. Par défaut c'est la valeur 8.

Pour un data warehouse, les valeurs sont supérieures. De plus si on récolte des statistiques (à l'aide du package *DBMSSTAT*) il est conseillé d'augmenter la taille, par exemple à 64 Ko jusqu'à une taille maximale de 1 Mo.

6.2 Plan d'exécution pour les opérations en parallèles avec EXPLAIN PLAN

EXPLAIN PLAN permet d'avoir un plan d'exécution pour des requêtes en parallèles. Des informations sont données concernant les colonnes: *COST*, *BYTES* et *CARDINALITY*. Vous pouvez utiliser le script *utlxplp.sql* (situé dans *$ORACLE_HOME/rdbms/admin/*) qui donne une représentation du résultat d'un EXPLAIN PLAN avec des informations sur le parallélisme.

6.3 Limitation du degré de parallélisme

Si vous utilisez le parallélisme durant une opération de type *UPDATE, MERGE* ou *DELETE*, le DOP doit être égal ou inférieur au nombre de partitions de la table.

6.4 La clause [NO]LOGGING

La clause *[NO]LOGGING* s'applique aux tables, partitions, tablespaces et index. Lorsque la clause *NOLOGGING* est fixée pour une table ou un index, il n'y a pas de génération d'*undo* ou de *redo logs*. Les processus vont plus vite avec cette option puisque rien n'est "logué". Mais en contre-partie, c'est que s'il y a un disfonctionnement au niveau physique (crash disque), tables, partitions et index peuvent être corrompus.

Au niveau du *tablespace*, la clause *[NO]LOGGING* spécifie l'attribut par défaut pour toutes les tables, index et partitions créées dans le tablespace (*LOGGING* par défaut).

Note: il est important de toujours analyser (*ANALYZE*) régulièrement les tables et index afin de s'assurer que l'optimiseur (*optimizer*) puisse utiliser au mieux les statistiques.

7 Conclusion

Dans ce document nous avons vu que le parallélisme apporte de nombreux avantages. Mais il paraît important de préciser que le mode automatique simplifie grandement la maintenance de la base de données. Le mode manuel permet d'ajuster plus finement l'utilisation du degré de parallélisme, néanmoins il faut savoir exactement ce que l'on fait et il faut absolument être rigoureux et séquentiel dans le réglage des paramètres.

8 Bibliographie

Documentation Oracle

Oracle9i Data Warehousing Guide
Using Parallel Execution

Livres

Piloter l'entreprise grâce au data warehouse
Sandrine de Lignerolles et Jean-Michel Franco
Editions Eyrolles