UNIVERSITE D'ORLEANS

Faculté des Sciences

LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Un algorithme incrémental parallèle pour la maintenance des vues matérialisées

M. Bamha, F. Bentayeb et G. Hains
LIFO, Université d'Orléans.
{bamha,bentayeb,ghains}@lifo.univ-orleans.fr

Rapport N^o **99-3**

Un algorithme incrémental parallèle pour la maintenance des vues matérialisées

M. Bamha, F. Bentayeb et G. Hains

LIFO - Université d'Orléans 4, rue Léonard De Vinci - BP 6759*

F-45067 Orléans Cedex 02 - France

{bamha,bentayeb,ghains}@lifo.univ-orleans.fr

Résumé : L'étude du problème de la maintenance de vues matérialisées dans les SGBD [1] relationnels a connu un essor considérable ces dernières années, particulièrement à cause de son application dans le domaine du *data-warehousing*. Ce problème consiste à conserver la cohérence des contenus des vues matérialisées par rapport aux contenus des relations de base lorsqu'elles sont modifiées. Par conséquent, les requêtes émanant des utilisateurs peuvent alors être évaluées plus rapidement en utilisant ces vues à la place des relations. Ceci n'est possible que si les vues sont maintenues dans un état cohérent. Des algorithmes incrémentaux ont été proposés dans la littérature pour mettre à jour la vue en n'utilisant que l'information utile; autrement dit en évitant la réévaluation complète de la vue. Notons cependant que lorsque la vue contient des jointures, sa réévaluation même incrémentale peut malheureusement induire un coût important. Ainsi, la parallélisation de l'algorithme de maintenance incrémentale des vues devient une approche très prometteuse pour atteindre des performances acceptables [ADT98].

Dans cet article, nous présentons d'une part un algorithme parallèle pour le calcul de la jointure basé sur la duplication partielle des données et d'autre part un algorithme incrémental parallèle efficace pour la maintenance de vues matérialisées. Les deux algorithmes sont basés sur un modèle de redistribution dynamique des données pour les machines à mémoire distribuée. Ce modèle présente l'avantage d'un fort potentiel d'équilibrage dynamique d'une part, et d'autre part supporte un contrôle flexible des communications induites par le parallélisme intra-transaction.

Les performances de ces deux algorithmes ont été étudiées en utilisant le modèle de coût *BSP* [2]. Ce modèle prévoit une accélération linéaire pour nos algorithmes.

mots clés : *SGBD parallèle, Maintenance des vues matérialisées, Data-warehouse, Multi-jointure, déséquilibre des données, équilibrage dynamique des charges.*

*LIFO : Laboratoire d'Informatique Fondamentale d'Orléans.

1. SGBD : Systèmes de gestion de bases de données.
2. BSP : Bulk Synchronous Parallelism.

Abstract : The problem of maintenance of materialized views has been the object of of increased research activity recently mainly because of applications related to data warehousing. Many sequential view maintenance algorithms are developed in the literature. If the view is defined by a relational expression involving join operators, the cost of re-evaluating the view even incrementally may be unacceptable. Indeed, database management systems require ever higher performance due to the size of data sets they manipulate and the increasing complexity of queries. The join operation is generally costly because of the size of results which is generally large with respect to the base relations' sizes. The size of the result for a multi-join can grow exponentially and parallelization of this operation is highly desirable. Moreover, when views are *materialized*, parallelism can greatly increase processing power as necessary for view maintenance.

In this paper, we present a new parallel view maintenance algorithm where views can involve multi-joins. This algorithm is composed of (1) a parallel algorithm for evaluating incrementally views based on a new parallel join algorithms by partial duplication of data and (2) a new parallel algorithm for updating effectively materialized views. These algorithms are based on a dynamic data-redistribution model for SN machines. The performances of these algorithms are analyzed using the scalable and portable *BSP*[3] cost model which predicts a near-linear speedup for our algorithms.

Key words: *PDBMS*[4]*, materialized view maintenance, parallel incremental algorithm, data-warehouse, multi-joins, partial multi-joins, data skew, dynamic load balancing.*

3. BSP : Bulk Synchronous Parallelism.
4. PDBMS : Parallel Data Base Management Systems.

1 Introduction

Le data warehousing a gagné une importance considérable ces dernières années [IK93, ZGMHW95] dans le domaine des bases de données. Un *data-warehouse* (entrepôt de données) est une collection de vues matérialisées dérivées à partir de relations qui ne résident pas dans le data-warehouse. Par conséquent, les requêtes émanant des utilisateurs peuvent alors être évaluées plus rapidement en utilisant ces vues à la place des relations. Cependant, comme les relations de base changent et évoluent à cause des mises à jour, les vues matérialisées doivent elles aussi subir des changements. Il est donc nécessaire de maintenir régulièrement les vues matérialisées dans un état cohérent lorsque les relations de base sont mises à jour. Ce problème est connu dans la littérature sous le nom de maintenance des vues matérialisées.

Une vue matérialisée peut être réévaluée à tout moment en appliquant l'expression relationnelle définissant la vue sur les relations de base. Cependant, une complète réévaluation de la vue est en général inefficace et le coût induit par cette réévaluation peut être inacceptable. D'où la nécessité d'utiliser une technique incrémentale pour mettre à jour les vues matérialisées. Toutes les approches qui traitent de ce problème [BLT86, BCL89, CW91, GL95] tentent de trouver une méthode incrémentale efficace afin de maintenir régulièrement ces vues. Plusieurs algorithmes incrémentaux *séquentiels* pour la maintenance de vues matérialisées ont été développés. Notons que lorsque la vue est définie par une expression relationnelle complexe, contenant notamment des jointures, la réévaluation même incrémentale de la vue peut malheureusement induire un coût important. En effet, les performances des systèmes de gestion de bases de données ont tendance à devenir critiques en raison du flot important des données et des requêtes qui deviennent de plus en plus complexes [TB91]. La jointure est l'une des opérations les plus fréquentes et les plus coûteuses dans systèmes de gestion de bases de données relationnelles. Le parallélisme est devenu une approche très prometteuse pour atteindre des performances acceptables pour de telles opérations.

Dans le cas où les vues sont matérialisées, afin d'améliorer le coût de la maintenance des vues, il est d'autant plus nécessaire d'exploiter le parallélisme tout en utilisant des structures de sauvegarde et de placement des données adaptées.

Dans cet article, en s'appuyant sur une méthode incrémentale proche de celle présentée dans [BLT86], nous proposons un algorithme incrémental parallèle efficace pour la maintenance de vues matérialisées. Cet algorithme est basé sur un modèle de redistribution dynamique des données pour les machines à mémoire distribuée. Ce modèle présente l'avantage d'un fort potentiel d'équilibrage dynamique d'une part, et d'autre part supporte un contrôle flexible des communications induites par le parallélisme intra-transaction. Les performances de cet algorithme ont été étudiées en utilisant le modèle de coût BSP qui prévoit une accélération presque linéaire pour notre algorithme.

Notre approche se situe dans le cadre du modèle relationnel où les vues sont définies par des expressions de la forme SPJ [5] de l'algèbre relationnelle. De plus, nous considérons les mises à jour qui sont des insertions et des suppressions d'ensembles de tuples sur les relations de base.

Nous citons dans ce qui suit, les points essentiels de notre contribution dans le domaine de la maintenance de vues matérialisées :

1. Un nouvel algorithme parallèle appelé *duplicating_join* pour le calcul de la jointure de deux relations. Cet algorithme garantit une accélération presque linéaire et est basé sur :
   (a) une duplication partielle des données permettant de réduire les coûts des communications inhérentes aux systèmes multi-processeurs.
   (b) une variante de l'algorithme de redistribution fa-join [BH98] qui traite de manière efficace le problème de déséquilibre des données et assure un équilibrage dynamique des résultats.

   Dans le cadre des vues matérialisées, l'algorithme *duplicating_join* est appliqué de telle manière que les données dupliquées concernent uniquement les ensembles de tuples à insérer ou à

5. SPJ : Sélection Projection et Jointure

supprimer des relations de base. Ceci minimise le surcoût en mémoire induit par l'algorithme. Cependant, lorsque la jointure concerne deux relations de même ordre de grandeur, nous utilisons l'algorithme *fa_join* qui a montré son efficacité dans [BH98].

2. Un algorithme parallèle pour la maintenance des vues matérialisées. Cet algorithme concerne la répercussion des tuples sur la vue matérialisée après une mise à jour des relations de base.

3. Utilisation du modèle de coût BSP pour la prévision des performances de l'algorithme *duplicating_join* ainsi que celles de l'algorithme incrémental de maintenance de vues matérialisées.

Concernant la maintenance de vues matérialisées, nous utilisons les algorithmes ci-dessus de la manière suivante : lors d'une mise à jour d'une ou de plusieurs relations de base d'une vue matérialisée définie par la jointure de ces relations, et afin de maintenir la vue de façon incrémentale, nous appliquons dans un premier temps l'algorithme *duplicating_join* ou l'algorithme *fa_join* pour le calcul de la jointure incrémentale, et dans un deuxième temps nous répercutons le résultat sur la vue.

Une analyse portable et réaliste des performances des algorithmes ci-dessus a été faite grâce au modèle BSP qui prévoit une accélération presque linéaire pour nos algorithmes.

L'article est organisé de la manière suivante : La section 2 rappelle la notion de maintenance incrémentale des vues matérialisées et notamment l'approche [BLT86]. La section 3 présente des rappels concernant les différentes machines parallèles existantes et le modèle de programmation BSP. Dans la section 4, nous présentons la jointure parallèle dans une architecture SN. La section 5 présente l'algorithme *duplicating_join* et l'étude de sa complexité. La section 6 présente l'algorithme incrémental parallèle pour la maintenance des vues matérialisées définies par des expressions relationnelles de type SPJ. Enfin, la section 7 conclut l'article par quelques suggestions concernant les travaux à venir dans le cadre de cette approche.

2 Maintenance incrémentale des vues

Dans le modèle relationnel classique, une vue est définie comme étant une relation non matérialisée calculée à partir des relations de la base par le biais d'une requête. Une vue est donc virtuelle et n'a pas d'existence physique. Une vue virtuelle peut être interrogée comme une relation de la base après sa création mais seulement durant le temps d'une transaction. Il est cependant possible d'associer une relation matérialisée à une vue : on parle alors d'une vue concrète ou vue matérialisée. Dans ce cas, la vue matérialisée peut être interrogée à tout moment contrairement à une vue virtuelle.

La matérialisation des vues consiste à stocker explicitement les tuples de la vue. Dans ce cadre, lorsqu'une relation de base est mise à jour, elle peut modifier le contenu des vues. Ainsi, il est nécessaire de mettre à jour ces vues. Le processus de réponse aux requêtes peut alors être accéléré en maintenant régulièrement les vues. Cependant, l'accès aux relations de base peut être évité uniquement si la vue matérialisée est maintenue dans un état cohérent.

Une méthode incrémentale de réévaluation d'une vue consiste à identifier le nouvel ensemble de tuples à ajouter à la vue dans le cas d'une insertion ou le sous-ensemble de tuples à retirer de la vue dans le cas d'une suppression, sans pour autant réévaluer complètement la vue [BLT86, BCL89, CW91].

Dans le cadre des bases de données relationnelles classiques, plusieurs approches traitent du problème de la maintenance de vues matérialisées de façon incrémentale, citons par exemple : (1) [BLT86, BCL89] qui définissent une vue par une expression de l'algèbre relationnelle, et (2) [CW91, GL95] qui définissent une vue par une requête SQL.

Dans cet article, nous nous intéressons aux vues *SPJ*. Et comme dans [BLT86], nous supposons (lorsque la vue matérialisée contient une projection sur un certain ensemble d'attributs) que les doublons sont gardés dans la vue soit explicitement, soit avec l'aide d'un compteur. Nous rappelons que les doublons sont utiles pour l'opération de suppression [BLT86].

Nous présentons ici la méthode incrémentale selon [BLT86] puisque dans notre approche de la parallélisation de la maintenance incrémentale des vues matérialisées, nous utilisons une méthode incrémentale proche de celle de [BLT86] permettant de réduire considérablement le nombre de jointures à évaluer ce qui réduit le coût de la maintenance de la vue matérialisée.

Par souci de clarté, nous présentons l'algorithme incrémental appliqué aux vues jointures. Nous verrons que cet algorithme s'étend naturellement aux vues de type *SPJ*.

Soit $v = R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n$ où R^i sont des relations. Supposons que l'on insère les ensembles de tuples $Ins(R^i)$ dans les relations R^i, $i = 1, \ldots, n$, respectivement. À la fin de l'insertion, le nouvel état de la vue est $v' = (R^1 \cup Ins(R^1)) \bowtie \ldots \bowtie (R^n \cup Ins(R^n))$.

En appliquant la propriété de la distributivité de la jointure sur l'union, la vue v mise à jour notée v' est égale à l'union de 2^n multi-jointures. Nous appelons ces multi-jointures des multi-jointures *partielles*.

Dans [BLT86], il a été montré que pour maintenir une vue $v = R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n$ de façon incrémentale, il est facile d'obtenir toutes les multi-jointures partielles de $v' = (R^1 \cup Ins(R^1)) \bowtie \ldots \bowtie (R^n \cup Ins(R^n))$ grâce à une table de vérité dont le nombre de colonnes est égal au nombre de relations présente dans la jointure de la manière suivante : Soit B_i un booléen associé au schéma de relation R^i, $i = 1, \ldots, n$. La valeur 0 de B_i correspond aux anciens tuples, i.e. R^i, et la valeur 1 de B_i correspond aux nouveaux tuples, i.e. $Ins(R^i)$. Le nouvel état de la vue v noté v', par exemple pour $n = 3$, est l'union de toutes les multi-jointures partielles contenues dans le tableau ci-dessous.

B_1	B_2	B_3	multi-jointure partielle
0	0	0	$R^1 \bowtie R^2 \bowtie R^3$
0	0	1	$R^1 \bowtie R^2 \bowtie Ins(R^3)$
0	1	0	$R^1 \bowtie Ins(R^2) \bowtie R^3$
0	1	1	$R^1 \bowtie Ins(R^2) \bowtie Ins(R^3)$
1	0	0	$Ins(R^1) \bowtie R^2 \bowtie R^3$
1	0	1	$Ins(R^1) \bowtie R^2 \bowtie Ins(R^3)$
1	1	0	$Ins(R^1) \bowtie Ins(R^2) \bowtie Ins(R^3)$
1	1	1	$Ins(R^1) \bowtie Ins(R^2) \bowtie Ins(R^3)$

Notons que la première expression relationnelle de v', i.e. la multi-jointure partielle $(R^1 \bowtie R^2 \bowtie R^3)$, correspond à la vue matérialisée courante. De plus, si on suppose par exemple que la relation R^3 n'est pas mise à jour, le nouvel ensemble $Ins(v)$ qu'il faut ajouter dans v est défini par l'expression relationnelle suivante : $(R^1 \bowtie Ins(R^2) \bowtie R^3) \cup (Ins(R^1) \bowtie R^2 \bowtie R^3) \cup (Ins(R^1) \bowtie Ins(R^2) \bowtie R^3)$.

En ce qui concerne la suppression de tuples, nous utilisons le même procédé que celui de l'insertion en respectant la propriété de la distributivité de la jointure par rapport à la différence.

3 Programmation parallèle : Machines et Modèles

Les performances des systèmes de bases de données ont tendance à devenir critiques en raison du flot important des données et de la complexité des requêtes. Les limites inhérentes aux machines séquentielles suggèrent qu'il ne sera pas possible d'améliorer indéfiniment leurs performances en raison du ralentissement et l'embouteillage au niveau des entrées et sorties (*Input/Output Bottleneck*). L'exploitation du parallélisme apparaît comme étant la seule façon d'améliorer les performances pour obtenir des temps de calcul raisonnables dans le traitement des requêtes complexes [BH98]. Comme une vue est définie par une requête, il est donc possible d'exploiter efficacement le parallélisme dans le cadre de la maintenance des vues matérialisées.

Notre approche de la maintenance des vues matérialisées se base sur un algorithme parallèle pour des machines SN. Ces architectures sont les plus générales car elles ne supposent aucune structure physiquement partagée.

Nous rappelons dans cette section trois aspects importants liés au parallélisme, à savoir les différentes machines parallèles, les différents modèles de programmation parallèle, et enfin le modèle de programmation BSP.

3.1 Machines parallèles

Trois principaux modèles de machines parallèles sont utilisées par les SGBD parallèles :

a- La machine à mémoire partagée (*Shared Memory*): Elle consiste en un nombre de processeurs connectés à un réseau d'interconnexion qui permet l'accès direct des processeurs à une mémoire globale et à tous les disques.

b- La machine à disques partagés (*Shared Disks*): Chaque processeur ne dispose que de sa mémoire privée (locale) mais tous les processeurs ont accès aux mêmes disques. Les processeurs sont reliés entre eux par un réseau d'interconnexion.

c- La machine à disques répartis (*Shared Nothing* (SN)): Chaque processeur ne dispose que de sa mémoire privée et de ses propres disques. Les processeurs sont reliés entre eux par un réseau d'interconnexion. Les communications dans les machines à disques répartis comme pour les machines à disques partagés, se font par passage de messages à travers le réseau d'interconnexion.

3.2 Modèles de programmation parallèle

Dans les systèmes de gestion de bases de données parallèles, deux types de parallélisme peuvent être cités : le parallélisme de données (*Data Parallelism*) et le parallélisme de programme appelé généralement parallélisme de tâches (*Task Parallelism*) ou parallélisme de contrôle (*Control Parallelism*).

3.2.1 Parallélisme de Programme (*Task Parallelism*):

Le parallélisme de programme, appelé généralement parallélisme pipeliné ou parallélisme de tâches est obtenu, dans le cas des bases de données, par décomposition des requêtes (*query decomposition*). La requête est divisée en sous requêtes qui seront exécutées en parallèle. L'effet de l'exécution du flot de données sur le temps de réponse des requêtes a été étudié par [Wil93]. Cependant l'élaboration d'un modèle général de coût pouvant prédire exactement le temps de réponse pour des requêtes pipelinées reste toujours un domaine de recherche. De plus, la quantité de parallélisme ainsi obtenu est limitée.

L'allocation des sous-requêtes aux processeurs a, de façon similaire au placement des données, un impact majeur sur la performance de la requête initiale. L'allocation idéale est non seulement, déterminée par le placement, mais aussi par le coût de traitement des sous-requêtes et par la charge réelle des divers processeurs.

3.2.2 Parallélisme de données (*Data Parallelism*):

Le parallélisme de données est obtenu par partitionnement des données. Dans ce type de parallélisme, les données sont divisées en sous-ensembles ou fragments. Le nombre de fragments est appelé degré de partitionnement. Par allocation des différents fragments aux différents processeurs, ils peuvent être accédés en parallèle. L'idée de base dans le parallélisme de données est que le temps de réponse à une opération (requête) est réduit au temps d'exécution de la requête sur les fragments en parallèle. Deux types de partitionnement peuvent être envisagés: d'une part le partitionnement statique qui consiste à adopter un modèle fixe de partitionnement pour les différentes opérations et d'autre part, le partitionnement dynamique (*Dynamic partitioning*) dans lequel les données sont partitionnées dynamiquement en fonction de la requête induisant le parallélisme et des résultats intermédiaires; cette dernière technique est mieux adaptée dans le traitement des opérations coûteuses telles que la jointure etc.

Un autre problème important dans les systèmes de gestions de bases de données parallèles liée au partitionnement des données, est le placement des données (*Data placement*) qui a un grand impact dans le chargement de la distribution du système (*System Load Distribution*). Il est préférable d'apporter les opérations aux données que d'apporter les données aux opérations puisque les premières peuvent être chargées après la compilation, étant statiques et moins volumineuses. Une technique assez répandue consiste à placer les fragments fréquemment utilisés ensemble dans une mémoire commune.

Le placement des données dépend généralement du traitement en cours d'exécution, cependant pour des raisons d'efficacité, il est préférable que le placement des données soit ajustable au moment de l'exécution des futures opérations sans interrompre l'exécution des applications en cours de traitement. Ceci impose d'augmenter la disponibilité des données par rapport aux processeurs afin d'éviter les erreurs et les échecs des exécutions concurrentes des requêtes induisant les même données. Cependant, la duplication des données (*Data Duplication*) apporte le problème supplémentaire de maintenir la consistance et la cohérence des données dupliquées.

L'influence des ces trois techniques : Le partitionnement des données (*Data Partitioning*), le placement des données (*Data Placement*) et la duplication des données (*Data Duplication*) sur les performances des systèmes est en général difficile à prévoir et dépend de ces trois principaux facteurs: le degré de partitionnement, le traitement et le degré de duplication.

3.3 Le modèle de programmation BSP

Le modèle *BSP* (*Bulk Synchronous Parallelism*) est un modèle de programmation parallèle proposé par *Leslie Valiant* [Val90] dont l'objectif est de fournir un niveau d'abstraction facilitant d'une part, la portabilité des programmes sur une grande variété d'architectures parallèles et d'autre part de prévoir les performances d'un même programme sur différentes machines réelles. BSP fournit une machine abstraite dont les opérations sont de plus haut niveau que celles de la majorité des modèles de programmation parallèle.

3.4 La machine BSP

La machine BSP est une abstraction d'une architecture parallèle. Elle peut être décomposée en trois composantes principales:

1. Un ensemble de couples processeur/mémoire. Généralement des processeurs séquentiels avec des blocs de mémoire locale.

2. Un réseau d'interconnexion permettant le passage des messages entre les processeurs.

3. Une composante permettant la synchronisation globale de tous les processeurs.

La performance d'une machine BSP est caractérisée par les 4 paramètres suivants :

p : le nombre de couples processeur/mémoire.

l : la latence du réseau, qui définit le temps nécessaire pour réaliser une synchronisation globale des processeurs.

g : le nombre d'opérations locales requises pour transférer un mot d'un processeur à un autre au moment au moment où les autres processeurs communiquent simultanément des messages de 1 mot. Le paramètre g est défini formellement comme étant le coût pour qu'une *h-relation* puisse se réaliser en $h * g$ unités de temps. Une *h-relation* est une communication globale où chaque processeur peut envoyer ou recevoir au maximum h unités de données (les unités sont, en pratique, des mots).

s : la vitesse des processeurs, c-à-d. le nombre d'opérations locales effectuées par seconde. En général, on exprime g, p et l comme des multiples de s, ce qui ramène le modèle à 3 paramètres.

3.4.1 La programmation sous BSP

Le modèle BSP définit la manière selon laquelle les programmes sont exécutés sur la machines BSP. Un programme est une séquence d'étapes appelées *supersteps*. Chacune est divisée en trois phases successives et logiquement disjointes :

1. Une phase de traitement local dans chaque processeur en utilisant les valeurs de la mémoire locale.

2. Une phase de communication induisant le transfert des données entre les processeurs.

3. Une barrière de synchronisation globale qui met fin à la *superstep* et qui rend disponible les données transférées aux mémoires locales des processeurs destinations pour la prochaine étape (*superstep*).

Le temps d'exécution d'une étape s (*superstep* à ne pas confondre avec le paramètre s) est donc la somme du temps maximum des traitements locaux des processeurs, du temps d'acheminement des communications et du temps de réalisation d'une barrière de synchronisation :

$$Temps(s) = \max_{p_i:processeur} W_{p_i}^{(s)} + \max_{p_i:processeur} h_{p_i}^{(s)} * g + l$$

où $W_{p_i}^{(s)}$: le temps d'exécution du traitement local du processeur p_i à l'étape s.
$h_{p_i}^{(s)}$: désigne le nombre d'unités de données transmises ou reçues par le processeur p_i à l'étape s, g est une mesure de la bande passante globale : le temps requis pour acheminer une relation de communication où chaque processeur émet et reçoit au plus un message.
l est le temps d'une barrière de synchronisation globale.

Le temps d'exécution d'un programme BSP composé de S étapes est donc:

$$Temps = \sum_{s:superstep} W^{(s)} + \sum_{s:superstep} h^{(s)} * g + S * l$$

où $W^{(s)} = \max_{p_i:processeur} W_{p_i}^{(s)}$ est le maximum des traitement locaux des différents processeurs à l'étape s.
$h^{(s)} = \max_{p_i:processeur} h_{p_i}^{(s)}$ le nombre maximum d'unités de données transmises ou reçues par les différents processeurs à l'étape s.

En général les valeurs W, h et S dépendent du nombre de processeurs p, de la taille n des données et/ou de paramètres plus complexes comme la répartition des données.

Afin de minimiser le temps d'exécution d'un programme BSP, le programmeur doit donc minimiser conjointement le nombre de *superstep* (et donc le nombre d'étapes de synchronisation), le nombre maximum de paquets de données envoyés ou reçus par chaque processeur au cours de chaque *superstep* et enfin équilibrer les charges des processeurs lors de la phase des traitements locaux de chaque *superstep*.

Dans la section suivante nous présentons la jointure dans les systèmes de gestion de bases de données parallèles.

4 La jointure parallèle

Dans les systèmes de gestion de bases de données parallèles ($PDBMS$[6]), le partitionnement des relations entre les processeurs se fait via une *fragmentation horizontale* selon un certain attribut. Trois principaux types de fragmentations horizontales peuvent être cités [DG92]: La fragmentation par hachage (*hash partitioning*), la fragmentation par blocs ou par rang (*Range partitioning*) et enfin la fragmentation cyclique (*Round-Robin partitioning*). Les indices des relations sont aussi partitionnés.
Nous supposons que le partitionnement des relations est effectué de manière équitable. Il est à noter que la fragmentation est physique dans le cas d'une machine SN (*Shared Nothing multi-processors machine*) et logique pour une machine SD (*Shared Disk*) [SR95].

La *θ-jointure* de deux relations R et S sur l'attribut A de R et l'attribut B de S (les attributs A et B sont de même domaine) est la relation T, notée $R \bowtie S$, obtenue par la concaténation des tuples de R avec ceux de S vérifiant la condition $R.A\ \theta\ S.B$ où $\theta \in \{=, \neq, <, >, \leq, \geq\}$. Quand θ est $=$, on parle d'*équi-jointure*.

La *semi-jointure* de S par R est la relation, notée $S \ltimes R$, composée des tuples de S qui se joignent avec R (c-à-d qui apparaissent dans la jointure de *R et S*). L'opération de *semi-jointure* réduit la taille de S, et on a la relation suivante: $R \bowtie S = R \bowtie (S \ltimes R)$.

L'opération de jointure parallèle se fait généralement en deux phases: une phase de redistribution des données basée généralement sur une fonction de hachage de l'attribut de jointure et une phase de jointures locales.

Plusieurs algorithmes de jointure parallèle ont été proposés dans la littérature, les principaux sont : le *Sort-merge join*, le *Simple-hash join*, le *Grace-hash join* et le *Hybrid-hash join* [SD89]. Tous ces algorithmes (dits *algorithmes de hachage*) sont basés sur des fonctions de hachage qui permettent de redistribuer les tuples des relations, R et S, sur les nœuds (il est à noter que tous les tuples ayant la même valeur pour l'attribut de jointure sont hachés vers le même nœud). Ensuite une étape de jointure locale est effectuée. La jointure de R et S est alors la réunion des résultats des jointures locales des nœuds.

L'inconvénient majeur de ces algorithmes est qu'ils sont vulnérables d'une part au déséquilibre des valeurs de l'attribut de jointure des relations (*Attribute Value Skew ou AVS*) et d'autre part au déséquilibre des résultats des jointures locales, qui affecte le traitement post-jointure (sauvegarde des données, jointures pipelinées etc....) appelé généralement le *Join Product Skew ou JPS* [SY90, PI96, LOT94].

- **Attribute Value Skew (*AVS*)**: survient quand la distribution des valeurs de l'attribut de jointure n'est pas uniforme (des valeurs apparaissent fréquemment plus que d'autres), on parle de *Data skew* simple (*Simple data skew*) quand seulement l'une des relations présente un AVS et de *Data skew* double (*Double data skew*) quand les deux relations présentent un AVS sur le même attribut de jointure.

- **Join Product Skew (*JPS*)**: affecte le traitement post jointure (sauvegarde des données, jointures pipelinées etc ....). Le JPS apparaît quand les jointures locales sont de tailles inégales. Il est important de remarquer que le JPS peut avoir lieu sans que les relations n'aient d'AVS.

Pour remédier au problème de déséquilibre des données dans le cadre des réponses aux requêtes complexes, nous avons introduit dans [BH98] l'algorithme fa_join (*frequency adaptive join algorithm*) basé sur deux nouveaux algorithmes de redistribution des données. Ces algorithmes de redistribution

6. PDBMS : Parallel Data Base Management Systems.

sont bien adaptés pour le traitement de la *θ-jointure* (cas général où θ n'est pas nécessairement l'égalité) pour des relations telles que $\|Hist(R)\| \leq \frac{\|R\|}{p}$ ($Hist(R)$ désigne l'histograme de R c-à-d. la liste des (d,n_d) où d est une valeur de l'attribut de jointure et n_d est le nombre (non nul) de tuples de R ayant cette valeur pour cet attribut et $\|R\|$ est la taille de R). Comme les relations ne satisfaisant pas cette relation sont relativement mieux équilibrées pour l'attribut de jointure, une méthode hybride permet de traiter efficacement le cas général.

L'algorithme *fa_join* permet de supporter de manière efficace le problème de déséquilibre de l'attribut de jointure et du déséquilibre des résultats des jointures locales. Cela réduit considérablement, les coûts de redistribution des résultats intermédiaires des multi-jointures et donc, augmente les performances de ces multi-jointures qui sont généralement vulnérables au déséquilibre des données. L'algorithme *fa_join* a montré son efficacité dans le cadre des réponses aux requêtes, cependant dans le cas des vues matérialisées la performance de cet algorithme peut être améliorée pour traiter des jointures de la forme $R \bowtie \Delta$ lorsque $\|\Delta\| \ll \|R\|$. Nous proposons donc un nouvel algorithme parallèle appelé *duplicating_join* pour traiter de telles jointures. Cet algorithme est basé sur la duplication partielle des données d'une part, et sur l'algorithme de redistribution *fa_join* d'autre part.

Nous montrons dans cet article que dans le cas de la parallélisation des jointures de type $R \bowtie \Delta$ où $\|\Delta\| \ll \|R\|$, l'algorithme *duplicating_join* est plus performant que l'algorithme *fa_join* tout en traitant de manière efficace le problème de déséquilibre des données. Sa complexité est analysée en utilisant le modèle de coût *BSP* .

Nous présentons dans ce qui suit, l'algorithme parallèle de jointure par duplication ainsi que sa complexité selon le modèle BSP.

5 Algorithme de jointure par duplication des données (*duplicating_join algorithm*)

Dans cette section, nous allons décrire un algorithme, pour les machines *SN*, adapté au traitement de la jointure de deux relations R et Δ lorsque $\|\Delta\| \ll \|R\|$. Cet algorithme appelé *duplicating_join* est basé sur :

- une duplication partielle des données permettant de réduire les coûts de communication entre les différents processeurs et,
- l'algorithme de redistribution de l'algorithme *fa_join* qui permet de traiter de manière efficace d'une part le problème du déséquilibre de l'attribut de jointure et d'autre part, le problème du déséquilibre du résultat de la jointure. Contrairement à *fa_join* où l'algorithme de redistribution est appliqué à toute la relation R, dans notre approche, l'algorithme de redistribution n'est appliqué qu'au résultat de la semi-jointure, $R \ltimes \Delta$, de R et de Δ.

Dans ce qui suit, pour une relation $T \in \{R, \Delta\}$,

- T_i désigne le fragment de la relation T associé au processeur i,
- $Hist(T)$ désigne l'histograme de la relation T selon un certain attribut de jointure,
- $\|T\|$ désigne la taille de la relation T.

Préalablement, nous supposons que la relation R (resp. Δ) est répartie sur les différents processeurs via une fragmentation horizontale selon un certain attribut.

Nous supposons en outre, que la fragmentation de la relation R est équilibrée, c-à-d que les fragments R_i pour $i = 1 \ldots p$ sont à peu près de même taille (i.e. $\|R_i\| \simeq \frac{\|R\|}{p}$, où p désigne le nombre de processeurs).

Nous allons décrire notre algorithme en donnant, selon le modèle *BSP*, une borne supérieure sur le temps d'exécution de chaque étape de l'algorithme. L'algorithme de jointure par duplication (*duplicating_join algorithm*) est composé de quatre phases:

Phase 1 : Phase de duplication partielle des données

Afin de minimiser les coûts de communication entre les différents processeurs, on procède tout d'abord par une étape de diffusion des fragments des relations Δ_i sur tous les processeurs. Lors de cette phase, chaque processeur i envoie, en parallèle, le fragment Δ_i dont il dispose à tous les autres processeurs. Le coût global de cette phase est de l'ordre de :

$$temps_{phase1} = O\left(g * \|\Delta\|\right)$$

où g est le temps nécessaire pour envoyer un message de longueur 1 d'un processeur à un autre (cf. 3.4). En effet chaque processeur i reçoit $\Sigma_{j \neq i} \|\Delta_j\|$ mots, soit environ $\|\Delta\|$ mots.
Après la phase 1, chaque processeur dispose en local de la totalité de la relation Δ. Rappelons que par hypothèse $\|\Delta\| \ll \|R\|$. Grâce à la propriété suivante :

$$\begin{aligned} R \bowtie \Delta &= (R_1 \cup R_2 \cup \ldots \cup R_p) \bowtie \Delta \\ &= (R_1 \bowtie \Delta) \cup (R_2 \bowtie \Delta) \cup \ldots \cup (R_p \bowtie \Delta), \end{aligned}$$

la duplication des fragments Δ_i nous permet d'effectuer, en parallèle, les jointures locales $R_i \bowtie \Delta$ pour $i = 1, \ldots, p$.

Phase 2 : Phase de calcul des semi-jointures locales

Afin de minimiser le coût de redistribution et donc le temps de communication entre les différents processeurs, on procède ensuite au calcul des semi-jointures locales suivantes : $\widetilde{R_i} = R_i \ltimes \Delta$. Ce calcul s'effectue par un parcours séquentiel de la table de hachage sur chaque processeur en temps de l'ordre de :

$$temps_{phase2} = O\left(\max_{i=1\ldots p} \|R_i\| + \|\Delta\|\right) = O\left(\frac{\|R\|}{p} + \|\Delta\|\right).$$

Nous rappelons que, contrairement aux algorithmes parallèles basés sur le hachage où les relations d'une jointure de type $R \bowtie \Delta$ sont toutes les deux redistribuées[7] sur les différents processeurs, et contrairement à l'algorithme *fa_join* où seule la relation R est redistribuée sur les processeurs, dans l'algorithme *duplicating_join* seul le résultat de la semi-jointure de R avec Δ est redistribué. Ainsi, le coût de la redistribution est réduit au minimum. L'étape de redistribution fera l'objet de la phase suivante.

Phase 3: Phase de redistribution des données

Afin d'assurer un équilibrage des résultats des jointures locales (et donc des charges des processeurs), une redistribution des semi-jointures $\widetilde{R_i} = R_i \ltimes \Delta$ est nécessaire. Pour cela, la relation $\widetilde{R_i} = R_i \ltimes \Delta$ est répartie en deux sous relations $\widetilde{R'_i}$ et $\widetilde{R''_i}$ de la manière suivante : $\widetilde{R_i} = R_i \ltimes \Delta = \widetilde{R'_i} \cup \widetilde{R''_i}$ pour $i = 1 \ldots p$, où :

- Tous les tuples de la relation $\widetilde{R'_i}$ sont associés à des fréquences élevées de valeurs de l'attribut de jointure, i.e. celles qui apparaissent plus souvent dans R qu'une certaine fréquence seuil f_0,
- Tous les tuples de la relation $\widetilde{R''_i}$ sont associés à des fréquences faibles, i.e des fréquences inférieures à la fréquence seuil f_0.

7. On entend ici par "redistribution" une phase de communication servant à rééquilibrer les fragments d'une relation.

Nous utilisons la même fréquence seuil que dans l'algorithme de redistribution *fa_join* [BH98], i.e. $f_0 = p*log(p)$. Les relations $\widetilde{R'_i}$ peuvent être calculées par échantillonnage ce qui permet de se passer du calcul de l'histograme global de $\widetilde{R}$ (nous rappelons que le calcul et la redistribution à tous les processeurs de l'histograme de $\widetilde{R}$ nécessite un temps de l'ordre de $O(\max_{i=1...p} \|\widetilde{R_i}\| + (1+g) * \|Hist(\widetilde{R})\| + l))$. Les relations $\widetilde{R'_i}$ sont ensuite redistribuées sur les différents processeurs en utilisant l'algorithme de redistribution *fa_join*, le tout en un temps de l'ordre de :

$$temps_{a1} \quad = \quad O\Big(g * (\max_{i=1...p} \|\widetilde{R'_i}\| + \|Hist(\widetilde{R'})\|) + \|Hist(\widetilde{R'})\| * \log(p) + (g+l) * \log(p)\Big).$$

où $\|Hist(\widetilde{R'})\|$ désigne la taille de l'histograme de $\widetilde{R'}$ selon l'attribut de jointure et l désigne le temps d'une barrière de synchronisation (cf. 3.4).
Comme $\|Hist(\widetilde{R'})\| \leq \frac{\|R'\|}{f_0} = \frac{\|R'\|}{p*\log(p)}$, on a alors:

$$temps_{a1} \quad \leq \quad O\Big(g * \max_{i=1...p} \|\widetilde{R'_i}\| + (g+l) * \log(p)\Big).$$

Les relations $\widetilde{R''_i}$ sont aussi redistribuées sur les différents processeurs en utilisant une fonction de hachage en un temps maximum de l'ordre de :

$$temps_{a2} = O\Big(g * \max_{i=1...p} \|\widetilde{R''_i}\|\Big).$$

Ainsi le coût total de la redistribution des relations $\widetilde{R_i}$ (où $\widetilde{R_i} = R_i \ltimes \Delta$) est :

$$\begin{aligned} temps_{phase3} &= temps_{a1} + temps_{a2} \\ &\leq O\Big(g * \max_{i=1...p} \|\widetilde{R'_i}\| + (g+l) * \log(p) + g * \max_{i=1...p} \|\widetilde{R''_i}\|\Big). \\ &= O\Big(g * \max_{i=1...p} \|\widetilde{R_i}\| + (g+l) * \log(p)\Big). \\ &\leq O\Big(g * \frac{\|R\|}{p} + (g+l) * \log(p)\Big). \end{aligned}$$

Par construction, la redistribution des relations $\widetilde{R_i}$ garantit que les jointures locales $\widetilde{R_i} \bowtie \Delta$ soient à peu près de même taille (i.e. $\|\widetilde{R_i} \bowtie \Delta\| \simeq \frac{\|R \bowtie \Delta\|}{p}$, pour $i = 1 \ldots p$).

phase 4 : Phase de calcul des jointures locales

Après redistribution des $\widetilde{R_i}$, les jointures locales $R_i \bowtie \Delta = \widetilde{R_i} \bowtie \Delta$ peuvent être calculées en temps maximum de l'ordre de :

$$\begin{aligned} temps_{phase4} &= O\Big(\max_{i=1...p} (\|\widetilde{R_i}\| + \|\Delta\| + \|\widetilde{R_i} \bowtie \Delta\|)\Big). \\ &\leq O\Big(\frac{\|R\|}{p} + \|\Delta\| + \frac{\|R \bowtie \Delta\|}{p}\Big). \end{aligned}$$

Le coût global de l'algorithme de jointure par duplication (*duplicating_join algorithm*) n'est autre que la somme des coûts des quatre phases ci-dessus :

$$\begin{aligned} temps_{duplicating-join} &= temps_{phase1} + temps_{phase2} + temps_{phase3} + temps_{phase4} \\ &= O\Big((1+g) * (\frac{\|R\|}{p} + \|\Delta\|) + \frac{\|R \bowtie \Delta\|}{p} + (g+l) * \log(p)\Big). \end{aligned}$$

Remarque : En séquentiel le temps d'exécution de la θ-jointure pour $\theta \in \{=, \neq, <, >, \leq, \geq\}$ de deux relations R et S vérifie la borne inférieure suivante :

$$temps_{inf_1} = \Omega\left(\|R\| + \|S\| + \|R \bowtie_\theta S\|\right). \tag{1}$$

Dans un environnement multi-processeurs, la borne inférieure de la θ-jointure de R et S en utilisant p processeurs devient :

$$temps_{inf_p} = \frac{1}{p} * \Omega \left(\|R\| + \|S\| + \|R \bowtie_\theta S\| \right). \tag{2}$$

Par conséquent, l'algorithme de jointure par duplication (*duplicating_join algorithm*) est optimal avec une constante multiplicative de l'ordre de g, lorsque :

$$\|\Delta\| \leq \frac{1}{p} * \max \Big(\|R\|, \|R \bowtie \Delta\| \Big).$$

Nous présentons dans ce qui suit, notre algorithme incrémental parallèle de maintenance des vues matérialisées. Cet algorithme est basé d'une part, sur l'algorithme *fa_join* et d'autre part sur l'algorithme *duplicating_join* ci-dessus.

6 Algorithme incrémental parallèle

6.1 Introduction

Une accélération linéaire est l'objectif principal des algorithmes parallèles, cependant dans les applications réelles, une accélération linéaire est difficilement atteignable : ceci est dû généralement d'une part aux coûts de communications inhérents aux systèmes multi-processeurs et d'autre part au déséquilibre des charges des différents processeurs; ces deux facteurs peuvent avoir un effet dévastateur sur la performance [BH98, SY90, AL91, HTY95, DNSS92].

Bien que plusieurs algorithmes parallèles aient été proposés pour le traitement du déséquilibre dans la jointure simple, ces algorithmes restent inefficaces dans le cas de la multi-jointure et de la θ-jointure. Dans ce contexte, l'algorithme *fa_join* [BH98] a été introduit pour le traitement de la θ-jointure dans le but de supporter efficacement le déséquilibre de l'attribut de jointure tout en garantissant un équilibrage dynamique des résultats des jointures, ce qui permet de réduire le coût de redistribution des résultats intermédiaires et donc d'augmenter les performances des multi-jointures. De plus, nous avons introduit, dans cet article, l'algorithme optimal *duplicating_join* basé sur la duplication partielle des données pour le traitement de la jointure de deux relations.

L'émergence des applications de bases de données dans les domaines tels que le data warehousing, le data mining et l'aide à la décision qui font généralement appel à de très grands volumes de données stockées sous forme de vues matérialisées rend la parallélisation des algorithmes de maintenance des vues nécessaire pour avoir un temps de réponse acceptable. Dans ce cadre, nous proposons un algorithme incrémental parallèle pour la maintenance des vues matérialisées. Cet algorithme est basé d'une part sur l'algorithme *duplicating_join* et d'autre part sur l'algorithme *fa_join*. Nous montrons que ces deux algorithmes peuvent être applicables directement et efficacement à n'importe quelle expression relationnelle de type *SPJ* (Sélection, projection et jointure) qui définit une vue matérialisée. Une analyse de sa complexité est faite en utilisant le modèle de coût BSP. Notre algorithme incrémental est composé de deux étapes à savoir : l'évaluation incrémentale et la mise à jour des vues matérialisées.

6.2 Algorithme incrémental: variante de [BLT86]

Nous avons vu dans la section 2 que pour maintenir une vue matérialisée de façon incrémentale, il est possible de capturer toutes les multi-jointures partielles à effectuer grâce à une table de vérité [BLT86]. Ainsi, pour une vue $v = R^1 \bowtie \ldots \bowtie R^n$, il faut effectuer $2^k - 1$ multi-jointures partielles où k est le nombre de relations de v mises à jour. Dans le but de réduire le nombre de multi-jointures partielles nécessaires pour la maintenance incrémentale de v, nous utilisons un algorithme incrémental proche de celui de [BLT86] mais nécessitant seulement k multi-jointures partielles. Pour capturer ces

k multi-jointures partielles, nous associons à $v = R^1 \bowtie \ldots \bowtie R^n$ un tableau J de dimension $[1..n, 1..n]$ de telle façon que chaque ligne de J représente une multi-jointure partielle Mjp_i de v telle que :

$$Mjp_i = J_{i,1} \bowtie J_{i,2} \ldots J_{i,n} \text{ où pour tout } j = 1, \ldots, n:$$

$$J_{i,j} = \begin{cases} R^j \cup \Delta(R^j) & \text{si } j < i \\ \Delta(R^j) & \text{si } i = j \\ R^j & \text{si } j > i \end{cases}$$

À titre d'exemple, supposons que l'on insère les ensembles de tuples $Ins(R^i)$ dans les relations R^i, $i = 1, \ldots, 3$ respectivement. Il faut donc effectuer 3 Mjp pour calculer la partie incrémentale v_{Ins} de v :

$$\begin{aligned} v_{Ins} \quad = \quad & \Big(Ins(R^1) \bowtie R^2 \bowtie R^3\Big) \cup \\ & \Big((Ins(R^1) \cup R^1) \bowtie Ins(R^2) \bowtie R^3\Big) \cup \\ & \Big((Ins(R^1) \cup R^1) \bowtie (Ins(R^2) \cup R^2) \bowtie Ins(R^3)\Big) \end{aligned}$$

Nous généralisons ce résultat pour les vues SPJ et nous montrons que pour toute vue matérialisée $v = \sigma_{\mathcal{C}} \pi_{\mathcal{X}} \Big(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n\Big)$ où les R^i sont des relations, $\mathcal{C}$ une condition de sélection et $\pi_{\mathcal{X}}$ la projection sur l'ensemble des attributs $\mathcal{X}$, si :

- on insère les ensembles de tuples $Ins(R^i)$ dans les relations R^i, $i = 1, \ldots, n$, respectivement, l'ensemble de tuples v_{Ins} à ajouter à v est :

$$\begin{aligned} v_{Ins} \quad = \quad & \sigma_{\mathcal{C}} \pi_{\mathcal{X}} \Big(Ins(R^1) \bowtie R^2 \bowtie R^3 \bowtie \ldots \bowtie R^n\Big) \quad \cup \\ & \sigma_{\mathcal{C}} \pi_{\mathcal{X}} \Big((R^1 \cup Ins(R^1)) \bowtie Ins(R^2) \bowtie R^3 \bowtie \ldots \bowtie \mathcal{R}^n\Big) \quad \cup \\ & \sigma_{\mathcal{C}} \pi_{\mathcal{X}} \Big((R^1 \cup Ins(R^1)) \bowtie (R^2 \cup Ins(R^2)) \bowtie Ins(R^3) \bowtie \mathcal{R}^4 \bowtie \ldots \bowtie R^n\Big) \quad \cup \\ & \qquad \vdots \\ & \sigma_{\mathcal{C}} \pi_{\mathcal{X}} \Big((R^1 \cup Ins(R^1)) \bowtie (R^2 \cup Ins(R^2)) \bowtie \ldots \bowtie (R^{n-1} \cup Ins(R^{n-1})) \bowtie Ins(R^n)\Big). \end{aligned}$$

- on supprime les ensembles de tuples $Del(R^i)$ des relations R^i, $i = 1, \ldots, n$, respectivement, l'ensemble de tuples v_{Del} à retirer de v est l'ensemble v_{Ins} ci-dessus où l'on remplace $Ins(R^i)$ par $Del(R^i)$ et l'opérateur $\cup$ par l'opérateur de différence $-$.

6.3 Mises à jour des relations de base et maintenance des vues matérialisées

Trois approches principales ont été proposées pour la maintenance des vues matérialisées :

- une première approche consiste à maintenir la vue matérialisée immédiatement après chaque insertion ou suppression dans les relations de base,
- une seconde approche consiste à différer la mise à jour de la vue jusqu'à la soumission d'une requête mettant en œuvre les données de la vue matérialisée,
- une troisième approche consiste à maintenir la vue matérialisée de façon périodique.

Une mise à jour des relations de base d'une vue matérialisée v peut être décomposée en deux étapes : une étape d'insertion dans les relations de base suivie d'une étape de suppression dans les relations de base. Lors de l'étape d'insertion (ou de suppression) de tuples dans les relations de base, nous évaluons l'ensemble de tuples v_{Ins} (ou v_{Del}) à ajouter (ou retirer) dans v en utilisant l'algorithme d'évaluation incrémental parallèle approprié, ensuite nous effectuons la mise à jour sur la vue en utilisant l'algorithme parallèle de maintenance des vues approprié.

6.4 Évaluation incrémentale parallèle : Cas de la jointure simple

Nous considérons dans cette section des vues matérialisées définies par des expressions relationnelles de la forme $R \bowtie S$ où R et S sont deux relations de base.

6.4.1 Les insertions dans les relations de base

Soit $R \bowtie S$ la jointure définissant la vue v où R et S sont deux relations de base. Supposons que l'on insère les ensembles de tuples $Ins(R)$ et $Ins(S)$ dans les relations R et S respectivement. Après l'insertion, le nouvel état de la vue est :

$$\begin{aligned} v' &= \Big[R \cup Ins(R)\Big] \bowtie \Big[S \cup Ins(S)\Big] \\ &= \underbrace{(R \bowtie S)}_{v} \bigcup \underbrace{\Big[Ins(R) \bowtie S\Big] \cup \Big[R \bowtie Ins(S)\Big] \cup \Big[Ins(R) \bowtie Ins(S)\Big]}_{v_{Ins}} \\ &= v \bigcup v_{Ins}. \end{aligned}$$

Pour maintenir la vue v de façon incrémentale, selon l'approche [BLT86], il suffit de calculer l'expression :

$$v_{Ins} = (Ins(R) \bowtie S) \cup (R \bowtie Ins(S)) \cup (Ins(R) \bowtie Ins(S)).$$

Nous présentons à présent notre algorithme d'évaluation incrémentale de v basé sur l'approche [BLT86].

En séquentiel (cf. équation 1), la maintenance de la vue v nécessite un temps de calcul au moins égal à :

$$\begin{aligned} borne_{inf_1} = \Omega\Big(\quad \|Ins(R)\| \quad &+ \|S\| + \|Ins(R) \bowtie S\| + \|R\| + \|Ins(S)\| + \|R \bowtie Ins(S)\| \\ + \quad &\|Ins(R)\| + \|Ins(S)\| + \|Ins(R) \bowtie Ins(S)\|\Big). \end{aligned} \tag{3}$$

Par conséquent, en utilisant p processeurs, le temps parallèle vérifie la borne inférieure suivante :

$$borne_{inf_p} = \frac{1}{p} * borne_{inf_1}. \tag{4}$$

Nous supposons que la relation R (resp. S) est préalablement répartie sur les différents processeurs via une fragmentation horizontale selon un certain attribut. Lors de la mise à jour des relations R et S, on suppose que chaque processeur dispose d'un fragment des relations $Ins(R)$ et $Ins(S)$ respectivement.

Dans ce qui suit, pour une relation $T \in \{R, Ins(R), S, Ins(S)\}$, T_i désigne le fragment de la relation T associé au processeur i, et $\|T\|$ désigne la taille de la relation T.
Nous supposons que :

- le partitionnement de la relation R est basé sur une fragmentation horizontale équilibrée, c-à-d que les fragments R_i pour $i = 1 \ldots p$ sont à peu près de même taille (i.e. $\|R_i\| \simeq \frac{\|R\|}{p}$, p désigne le nombre de processeurs).
- de même pour la relation S.

Du fait que la partie incrémentale de la vue v vérifie la relation suivante :

$$\begin{aligned} v_{Ins} &= (Ins(R) \bowtie S) \quad \cup \quad (R \bowtie Ins(S)) \quad \cup \quad (Ins(R) \bowtie Ins(S)), \\ &= \bigcup_i (Ins(R) \bowtie S_i) \quad \cup \quad \bigcup_i (R_i \bowtie Ins(S)) \quad \cup \quad \bigcup_i (Ins(R)_i \bowtie Ins(S)), \end{aligned}$$

la mise à jour de la vue v selon l'approche [BLT86], revient à calculer la vue v_{Ins} en calculant les trois jointures suivantes $Ins(R) \bowtie S$, $R \bowtie Ins(S)$ et $Ins(R) \bowtie Ins(S)$. Pour cela,

a: lorsque $\|Ins(R)\| \ll \|S\|$, l'algorithme par duplication (*duplicating_join algorithm*) peut être utilisé de manière efficace pour le traitement de la jointure $Ins(R) \bowtie S$. Le coût global de cette jointure est de l'ordre de :

$$temps_{etape_a} = O\Big((1+g) * (\max_{i=1...p} \|S_i\| + \|Ins(R)\|) + \frac{\|Ins(R) \bowtie S\|}{p} + (g+l) * \log(p)\Big).$$

b: lorsque $\|Ins(S)\| \ll \|R\|$, l'algorithme par duplication (*duplicating_join algorithm*) peut être utilisé de manière efficace pour le traitement de la jointure, $R \bowtie Ins(S)$, de R avec $Ins(S)$. Le coût global de cette jointure est de l'ordre de :

$$temps_{etape_b} = O\Big((1+g) * (\max_{i=1...p} \|R_i\| + \|Ins(S)\|) + \frac{\|R \bowtie Ins(S)\|}{p} + (g+l) * \log(p)\Big),$$

c: pour le calcul de la jointure de $Ins(R) \bowtie Ins(S)$, on calcule l'une des jointures locales : $Ins(R) \bowtie Ins(S)_i$ ou $Ins(R)_i \bowtie Ins(S)$. En choisissant $Ins(R)_i \bowtie Ins(S)$, la jointure $Ins(R) \bowtie Ins(S)$ est évaluée en temps maximum de l'ordre de :

$$temps_{etape_c} = O\Big(\max_{i=1...p} \|Ins(R)_i\| + \|Ins(S)\| + \max_{i=1...p} \|Ins(R)_i \bowtie Ins(S)\|\Big).$$

Ce temps ne mesure que du calcul parallèle, sans communication. En effet, l'étape b: ci-dessus aura déjà recopié $Ins(S)$ sur chaque processeur (ou sinon, l'étape a: aura déjà recopié $Ins(R)$ sur chaque processeur).

Le coût global de la maintenance de la vue v n'est autre que la somme des coûts des trois étapes ci-dessus :

$$\begin{aligned} temps_{v_{Ins}} &= temps_{etape_a} + temps_{etape_b} + temps_{etape_c}. \\ &= O\Big((1+g) * (\|Ins(R)\| + \|Ins(S)\| + \max_{i=1...p} \|R_i\| + \max_{i=1...p} \|S_i\|) \\ &+ \frac{\|Ins(R) \bowtie S\|}{p} + \frac{\|R \bowtie Ins(S)\|}{p} + \max_{i=1...p} \|Ins(R)_i \bowtie Ins(S)\| + (g+l) * \log(p)\Big). \end{aligned} \quad (5)$$

L'algorithme de maintenance de la vue est optimal, lorsque :

$$\begin{aligned} &\max\Big(\|Ins(R)\|, \|Ins(S)\|, \max_{i=1..p} \|Ins(R)_i \bowtie Ins(S)\|\Big) \\ &\qquad \le \frac{1}{p} * \max\Big(\|R\|, \|S\|, \|R \bowtie Ins(S)\|, \|Ins(R) \bowtie Ins(S)\|\Big), \end{aligned} \quad (6)$$

ceci est dû au fait que tous les termes de $temps_{v_{Ins}}$ sont dominés (à une constante près) par ceux de $borne_{inf_p}$(cf. équation : 4).

Notons que contrairement à l'approche [BLT86], les jointures $R \bowtie Ins(S)$ et $Ins(R) \bowtie Ins(S)$ sont implicitement évaluées en calculant la jointure de $(R \cup Ins(R)) \bowtie Ins(S)$ (cf. 6.2). En utilisant l'algorithme *duplicating_join*, le calcul de $(R \cup Ins(R)) \bowtie Ins(S)$ se fait en temps de l'ordre de :

$$\begin{aligned} &O\Big((1+g) * (\max_{i=1..p} \|(R_i \cup Ins(R)_i)\| + \|Ins(S)\|) + \frac{\|(R \cup Ins(R)) \bowtie Ins(S)\|}{p} + (g+l) * \log(p)\Big) \\ &= O\Big((1+g) * (\max_{i=1..p} (\|R_i\| + \|Ins(R)_i\|) + \|Ins(S)\|) + \frac{\|Ins(R) \bowtie Ins(S)\|}{p} \\ &\qquad + \frac{\|R \bowtie Ins(S)\|}{p} + (g+l) * \log(p)\Big). \end{aligned}$$

ce qui est légèrement meilleur que le temps des étapes b et c additionnées : $temps_{etape_b} + temps_{etape_c}$. Ce qui réduit à deux le nombre de jointures à évaluer. Et comme dans le cas général (cf. 6.2) le nombre de multi-jointures partielles à effectuer passe de $2^k - 1$ à k (k étant le nombre de relations mises à jour), il est clair que le temps d'évaluation incrémentale d'une vue définie par une multi-jointure se verra réduit lui aussi.

L'équation 6 est généralement vérifiée lorsque $\|Ins(R)\| \ll \|S\|$ et $\|Ins(S)\| \ll \|R\|$. Cependant lorsque $\|Ins(R)\| \simeq \|S\|$ (resp. $\|Ins(S)\| \simeq \|R\|$), la jointure $Ins(R) \bowtie S$ (resp. $R \bowtie Ins(S)$) peut être évaluée de manière efficace en utilisant l'algorithme *fa_join* [BH98] avec une accélération presque linéaire. Nous rappelons que les deux algorithmes *fa_join* et *duplicating_join* assurent un équilibrage presque parfait des résultats des jointures et par conséquent, ils sont bien adaptés au cas des multi-jointures. Ainsi toutes les multi-jointures sont évaluées avec une accélération presque linéaire et par conséquent la maintenance incrémentale est elle aussi évaluée avec une accélération presque linéaire.

6.4.2 Les suppressions dans les relations de base :

Soit $R \bowtie S$ la jointure définissant la vue v où R et S sont deux relations de base. Supposons que l'on supprime les ensembles de tuples $Del(R)$ et $Del(S)$ des relations R et S respectivement. Après suppression, le nouvel état de la vue v est $v - v_{Del}$ tel que, d'après (cf. 6.2), on a :

$$v_{Del} \quad = \quad \underbrace{\Big[Del(R) \bowtie S\Big] \cup \Big[(R - Del(R)) \bowtie Del(S)\Big]}$$

Pour maintenir la vue v de façon incrémentale, il suffit donc de calculer l'expression :

$$\begin{aligned} v_{Del} &= (Del(R) \bowtie S) \quad \bigcup \quad \Big((R - Del(R)) \bowtie Del(S)\Big). \\ &= \cup_i (Del(R) \bowtie S_i) \quad \bigcup \quad \cup_i \Big((R_i - Del(R)) \bowtie Del(S)\Big). \end{aligned}$$

Comme dans le cas de l'insertion dans les relations de base, lorsque $\|Del(R)\|$ et $\|Del(S)\|$ sont négligeables en comparaison avec les tailles des relations $\|R\|$ et $\|S\|$, le calcul de v_{Del} nécessite (en remplaçant Ins par Del dans l'équation 5) un temps maximum de l'ordre de :

$$\begin{aligned} temps_{v_{Del}} &= O\Big((1+g) * (\max_{i=1...p} \|R_i\| + \max_{i=1...p} \|S_i\| + \|Del(R)\| + \|Del(S)\|) \\ &+ \frac{\|R \bowtie Del(S)\|}{p} + \frac{\|Del(R) \bowtie S\|}{p} + \max_{i=1...p} \|Del(R)_i \bowtie Del(S)\| + (g+l) * \log(p)\Big) \quad (7) \end{aligned}$$

L'algorithme de la maintenance de la vue est optimal, lorsque :

$$\begin{aligned} &\max\Big(\|Del(R)\|, \|Del(S)\|, \max_{i=1..p} \|Del(R)_i \bowtie Del(S)\|\Big) \\ &\qquad \leq \frac{1}{p} * \max\Big(\|R\|, \|S\|, \|R \bowtie S\|, \|Del(R) \bowtie Del(S)\|\Big), \quad (8) \end{aligned}$$

ceci est dû au fait que tous les termes de $temps_{v_{Del}}$ sont dominés par ceux de $borne_{inf_p}$ (cf. équation : 6) en remplaçant Ins par Del.

L'équation 8 est généralement vérifiée lorsque : $\|Del(R)\| \ll \|S\|$ et $\|Del(S)\| \ll \|R\|$. Cependant lorsque $\|Del(R)\| \simeq \|S\|$ (resp. $\|Del(S)\| \simeq \|R\|$), la jointure $Del(R) \bowtie S$ (resp. $R \bowtie Del(S)$) peut être évaluée de manière efficace en utilisant l'algorithme *fa_join* [BH98]. Nous rappelons que les deux algorithmes *fa_join* et *duplicating_join* assurent un équilibrage presque parfait des résultats des jointures et par conséquent, ils sont bien adaptés au cas des multi-jointures. Ainsi les multi-jointures peuvent être évaluées avec une accélération presque linéaire.

6.5 Évaluation incrémentale parallèle : Cas de vues SPJ

Nous considérons dans cette section des vues matérialisées définies par des expressions relations de la forme $v = \sigma_{\mathcal{C}}\pi_{\mathcal{X}}\Big(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n\Big)$ où les R^i sont des relations, $\mathcal{C}$ une condition de sélection et $\pi_{\mathcal{X}}$ la projection sur l'ensemble des attributs $\mathcal{X}$.

Nous rappelons ici que pour maintenir la vue v de façon incrémentale selon la variante de l'algorithme [BLT86] présenté dans la section 6.2, nous devons calculer :

$$
\begin{array}{l}
\sigma_{\mathcal{C}}\pi_{\mathcal{X}}\Big(\delta(R^1) \bowtie R^2 \bowtie R^3 \bowtie \ldots \bowtie R^n\Big) \quad \cup \\
\sigma_{\mathcal{C}}\pi_{\mathcal{X}}\Big((R^1 \ Op \ \delta(R^1)) \bowtie \delta(R^2) \bowtie R^3 \bowtie \ldots \bowtie \mathcal{R}^n\Big) \quad \cup \\
\sigma_{\mathcal{C}}\pi_{\mathcal{X}}\Big((R^1 \ Op \ \delta(R^1)) \bowtie (R^2 \ Op \ \delta(R^2)) \bowtie \delta(R^3) \bowtie \mathcal{R}^4 \bowtie \ldots \bowtie R^n\Big) \quad \cup \\
\qquad\qquad\qquad \vdots \\
\sigma_{\mathcal{C}}\pi_{\mathcal{X}}\Big((R^1 \ Op \ \delta(R^1)) \bowtie (R^2 \ Op \ \delta(R^2)) \bowtie \ldots \bowtie (R^{n-1} \ Op \ \delta(R^{n-1})) \bowtie \delta(R^n)\Big).
\end{array}
$$

tel que $Op \ = \cup \ et \ \delta = Ins$ ou $Op \ = - \ et \ \delta = Del$ selon que la mise à jour est une insertion ou une suppression.

Pour la maintenance de la vue v, on procède par le calcul des jointures mettant en œuvre les relations $\delta(R^i)$ en utilisant l'algorithme *fa_join* lorsque les opérandes de la jointure sont à peu prés de même tailles alors que l'algorithme par duplication (*duplicating_join algorithm*) n'est appliqué que dans le cas où les tailles des opérandes des jointures ne sont pas de même ordre de grandeur. De plus, si une relation R^j n'est pas mise à jour, toutes les jointures mettant en cause $\delta(R^j)$ ne seront pas évaluées et par conséquent le nombre de multi-jointures à évaluer est égal au nombre de relations mises-à-jour.

Nous présentons dans ce qui suit, les algorithmes relatifs au calcul incrémental dans le cas de l'insertion et dans le cas de la suppression.

6.5.1 Les insertions dans les relations de base

Supposons que l'on insère les ensembles de tuples $Ins(R^i)$ dans les relations R^i, $i = 1, \ldots, n$, respectivement. Pour optimiser le coût de l'évaluation incrémentale, les vues partielles sont exécutées dans l'ordre du haut en bas (cf. section 6.5). Ainsi la première vue partielle à évaluer est $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}(Ins(R^1) \bowtie R^2 \bowtie R^3 \bowtie \ldots \bowtie \mathcal{R}^n)$. Du fait que toutes les vues partielles qui restent à évaluer font appel à la relation $R^1 \cup Ins(R^1)$, nous mettons définitivement à jour la relation R^1 en calculant $R^1 \cup Ins(R^1)$ en temps de l'ordre de $\max_{i=1\ldots p} \|(Ins(R^1))_i\|$, une fois calculée $R^1 \cup Ins(R^1)$, nous calculons ensuite la vue partielle $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 \cup Ins(R^1)) \bowtie Ins(R^2) \bowtie R^3 \bowtie \ldots \bowtie R^n)$ et une fois terminée, nous mettons définitivement à jour la relation $\mathcal{R}^2$ en calculant $R^2 \cup Ins(R^2)$ en temps de l'ordre de $\max_{i=1\ldots p} \|(Ins(R^2))_i\|$, nous calculons ensuite la vue partielle $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 \cup Ins(R^1)) \bowtie (R^2 \cup Ins(R^2)) \bowtie Ins(R^3) \bowtie \mathcal{R}^4 \bowtie \ldots \bowtie R^n)$, notez bien que dans l'évaluation de cette vue partielle on ne re-calcule pas l'expression $R^1 \cup Ins(\mathcal{R}^1)$ puisqu'elle a été déjà calculée dans l'étape d'avant. Nous appliquons le même procédé pour évaluer les vues partielles qui restent. Ainsi lors du calcul de chaque vue partielle, une et une seule relation est mise à jour.

Ceci permet de réduire considérablement d'une part le coût des mises à jour des relations de base de la vue matérialisée et d'autre part le coût de la maintenance de la vue matérialisée en évaluant uniquement k vues partielles où k désigne le nombre de relations mises à jour. Nous rappelons que, dans l'approche [BLT86], le nombre de vues partielles à évaluer est égal à $2^k - 1$ où k désigne le nombre de relations mises-à-jour.

6.5.2 Les suppressions dans les relations de base

Supposons que l'on supprime les ensembles de tuples $Del(R^i)$ dans les relations R^i, $i = 1, \ldots, n$, respectivement. Comme dans le cas de l'insertion, pour optimiser le coût de l'évaluation incrémentale, les vues partielles sont exécutées dans l'ordre du haut en bas (cf. section 6.5). Ainsi la première vue partielle à évaluer est $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}(Del(R^1) \bowtie R^2 \bowtie \mathcal{R}^3 \bowtie \ldots \bowtie R^n)$. Du fait que toute les vues partielles qui restent à évaluer font appel à la relation $R^1 - Del(\mathcal{R}^1)$, nous mettons définitivement à jour la relation R^1 en calculant $R^1 - Del(R^1)$ en temps de l'ordre de $max(\max_{i=1\ldots p} \|(R^1)_i\|, \|Del(R^1)\|)$, une fois calculée $R^1 - Del(R^1)$, nous calculons ensuite la vue partielle $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 - Del(R^1)) \bowtie Del(R^2) \bowtie R^3 \bowtie \ldots \bowtie R^n)$ et une fois terminée, nous mettons définitivement à jour la relation R^2 en calculant $R^2 - Del(R^2)$ en temps de l'ordre de $max(\max_{i=1\ldots p} \|(R^2)_i\|, \|Del(R^2)\|)$. Nous calculons ensuite la vue partielle $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 - Del(R^1)) \bowtie (R^2 - Del(R^2)) \bowtie Del(R^3) \bowtie R^4 \bowtie \ldots \bowtie R^n)$, notez bien que dans l'évaluation de cette vue partielle on ne recalcule pas l'expression $R^1 - Del(R^1)$ puisqu'elle a été déjà calculée dans l'étape d'avant. Nous appliquons le même procédé pour évaluer les vues partielles qui restent. Ainsi lors du calcul de chaque vue partielle, une et une seule relation est mise à jour.

Ceci permet de réduire considérablement d'une part le coût des mises à jour des relations de base de la vue matérialisée et d'autre part le coût de maintenance de la vue matérialisée en évaluant uniquement k vues partielles où k désigne le nombre de relations mises à jour.

6.5.3 Insertion et suppression : Résultats

Les algorithmes *fa_join* et *duplicating_join* ont été introduits pour traiter de manière efficace le problème du déséquilibre de l'attribut de jointure et du déséquilibre du résultat des jointures. Ainsi les résultats des jointures locales seront à peu prés de même tailles, les étapes de sélection et de projection peuvent alors être effectuées avec une accélération presque linéaire.

Aussi bien dans le cas de l'insertion que de la suppression dans les relations de base, l'algorithme parallèle de l'évaluation incrémentale de la vue matérialisée v a une accélération presque linéaire du fait que toutes les vues de la partie incrémentale de la vue v_{Ins} (resp. v_{Del}) ont des accélérations presque linéaires. Et par conséquent l'algorithme de l'évaluation de la vue v_{Ins} (resp. v_{Del}) a une accélération presque linéaire.

Les étapes d'insertion et de la suppression de la vue v_{Ins} (resp. v_{Del}) dans la vue matérialisée v fera l'objet des sections 6.6.1 (resp. 6.6.2).

6.6 Mise à jour parallèle des vues matérialisées

6.6.1 Cas de l'insertion

Dans le cas de l'insertion dans les relations de base, l'algorithme de l'évaluation de la vue v_{Ins}, a une accélération presque linéaire. Pour la maintenance incrémentale de la vue matérialisée, il reste à insérer l'ensemble des tuples de v_{Ins} dans la vue matérialisée. Pour cela deux cas sont possibles :

> **Cas 1:** la fragmentation de la vue matérialisée n'est basée sur aucune fonction de placement
>
> Lorsque la fragmentation de la vue matérialisée n'est basée sur aucune fonction de placement, la vue matérialisée est donc répartie sur les différents processeurs de manière aléatoire. Dans ce cas, les fragments de la vue v_{Ins} sont immédiatement stockés sur les disques locaux avec un coût de l'ordre de $\max_{i=1,\ldots,p} \|(v_{Ins})_i\|$ et aucun traitement supplémentaire n'est effectué. Et par conséquent l'algorithme incrémental parallèle de la maintenance de la vue matérialisée a donc une accélération linéaire du fait que l'évaluation et la mise sont réalisés avec une accélération linéaire.
>
> **Cas 2:** fragmentation de la vue basée sur une fonction de placement
>
> Une fois terminée l'étape de calcul de v_{Ins}, les fragments locaux, $(v_{Ins})_i$ sont redistribués sur différents processeurs en utilisant la fonction de placement utilisée dans la fragmentation de la vue matérialisée initiale. Le coût global de la redistribution de la vue v_{Ins} dépend généralement de la fonction de placement utilisée.
>
> Notons que dans le pire des cas où tous les blocs $(v_{Ins})_i$ sont envoyés au même processeur, l'étape de redistribution n'excède pas $g * \|v_{Ins}\|$.

6.6.2 Cas de la suppression

Dans le cas de la suppression dans les relations de base, l'algorithme de l'évaluation de la vue v_{Del} a une accélération presque linéaire. Pour la maintenance incrémentale de la vue matérialisée, il reste à supprimer l'ensemble des tuples de v_{Del} de la vue matérialisée. Pour cela deux cas sont possible :

Cas 1: la fragmentation la vue matérialisée n'est basée sur aucune fonction de placement

Lorsque la fragmentation de la vue matérialisée n'est basée sur aucune fonction de placement, la vue matérialisée est donc répartie sur les différents processeurs de manière aléatoire. Dans ce cas, nous appliquons l'algorithme suppression suivant qui permet de supprimer les tuples de v_{Del} de la vue matérialisée :

Pour $j = 1$ **à** p **faire**

début

- En parallèle, chaque processeur i, supprime du fragment v_i les tuples de $(v_{Del})_i$.
- En parallèle, on calcule $Del_i := (v_{Del})_i - [(v_{Del})_i \cap v_i]$ (i.e. Del_i est l'ensemble des tuples de $(v_{Del})_i$ non supprimés de v_i).
- En parallèle, chaque processeur i, envoie Del_i au processeur $i+1$.
- $(v_{Del})_i := Del_{(i-1)}$.

fin.

Notons que l'algorithme de suppression a une complexité maximale de l'ordre de $g * \|v_{Del}\|$.

Cas 2: fragmentation de la vue basée sur une fonction de placement

Une fois terminée l'étape de calcul de v_{Del}, les fragments locaux, $(v_{Del})_i$ sont redistribués sur différents processeurs en utilisant la fonction de placement utilisée dans la fragmentation de la vue matérialisée initiale. Le coût globale de la redistribution de la vue v_{Del} dépend généralement de la fonction de placement utilisée. le coût moyen de la redistribution des fragments $(v_{Del})_i$ est de l'ordre de $g * \max_{i=1,\ldots,p}(v_{Del})_i$. Notons que dans le pire des cas, l'étape de redistribution n'excède pas $g * \|v_{Del}\|$.

7 Conclusion

Dans cet article, nous avons présenté un nouvel algorithme parallèle (*duplicating_join*) basé sur la duplication partielle des données pour le traitement de la jointure de deux relations lorsque les opérandes de la jointure n'ont pas des tailles de même ordre de grandeur. Nous avons montré que cet algorithme traite de manière efficace le problème de déséquilibre des données qui apparaît généralement lors du traitement de la jointure et de la multi-jointure.

En s'appuyant sur l'algorithme incrémental présenté dans [BLT86], nous avons proposé un algorithme incrémental parallèle efficace pour la maintenance de vues matérialisées pour des machine *SN*. Cet algorithme est basé d'une part sur l'algorithme *fa_join* (frequency adaptive algorithm) introduit dans [BH98] et d'autre part sur l'algorithme *duplicating_join* introduit dans cet article.

Les performances de cet algorithme ont été étudiées en utilisant le modèle BSP qui prévoit une accélération linéaire pour notre algorithme. Il est à remarquer que les constantes cachées dans nos analyses de coûts sont faibles. L'étude expérimentale de *fa_join* [BH98] a montré qu'ils ne nuisent pas à l'accélération parallèle.

Références

[ADT98] B. Moon A. Datta and H. Thomas. A case for Parallelism in Datawarehousing and OLAP. In *Ninth International Workshop on Database and Expert Systems Applications, DEXA 98*, IEEE Computer Society, Vienna, Austria, 1998. Roland R. Wagner.

[AL91] Kien A.Hua and Chieng Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. *in Proc. 17th international conf. on very Large Data Bases, pp 525-535,*, 1991.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, September 1989.

[BH98] M. Bamha and G. Hains. A self-balancing join algorithm for Shared Nothing machines. *In the Proc of the 10th International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada*, October 1998.

[BLT86] José A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. *ACM SIGMOD*, 1986.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th International conference on Very Large Data Bases*, Barcelona, September 1991.

[DG92] David J. Dewitt and J. Gray. Parallel Database Systems : The future of High Performance Database Systems. *Communications of the ACM / Vol 35 (6)*, June 1992.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshrdri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, 1992.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 328–339, San Jose, California, 22–25 May 1995.

[HTY95] K. A. Hua, W. Tavanapong, and H. C. Young. A Performance Evaluation of Load Balancing Techniques for Join Operations on Multicomputer Database systems. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering*, pages 44–51, Los Alamitos, CA, USA, March 1995. IEEE Computer Society Press.

[IK93] W. H. Inmon and C. Kelly. Rdb/VMS : Developing the Data Warehouse. In , Boston Massachusetts, 1993. Publishing Group.

[LOT94] Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, Los Alamitos, California, 1994.

[PI96] Viswanath Poosala and Yannis E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. *In: Proc. 22th Int. Conf. on Very Large Database Systems, VLDB'96: 448-459, Bombay, India*, sept 1996.

[SD89] Donovan A. Schneider and David J. DeWitt. A performance of four parallel join algorithms in a shared-nothing multiprocessor environment. *in the Proc ACM SIGMOD, pp. 110-121*, 1989.

[SR95] T. Stöhr and E. Rahm. Analysis of parallel scan processing in shared disk database systems. *Proc. EURO-PAR, Stockholm, Springer-Verlag, LNCS*, Aug. 1995.

[SY90] M. Seetha and Philip S. Yu. Effectiveness of Parallel Joins. *published in the IEEE, Trans. Knowledge and Data Enginneerings, Vol. 2, No 4, pp 410-424, Institute of Electrical and Electronics Engineers.*, Dec. 1990.

[TB91] Wouter B. Teeuv and Henk M. Blanken. Complex objet join in a distributed database. *University of Twente, Departement of Computer Science, The Netherlands*, 1991.

[Val90] Leslie Valiant. A Bridging Model for Parallel Computation,. *Communication of the ACM, Vol 33, No. 8.*, August 1990.

[Wil93] A.N. Wilschut. *Parallel Query Execution in a Main Memory Database System.* PhD thesis, Twente University, 1993.

[ZGMHW95] Yue Zhuge, Hector García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, 22–25 May 1995.