An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines

M. Bamha, F. Bentayeb and G. Hains

LIFO, Université d'Orléans, B.P. 6759, 45067 Orléans Cedex 2, France
{bamha,bentayeb,ghains}@lifo.univ-orleans.fr

Abstract. The problem of maintenance of materialized views has been the object 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. Moreover, when views are *materialized*, parallelism can greatly increase processing power as necessary for view maintenance.
In this paper, we present a new parallel join algorithm by partial duplication of data and a new parallel view maintenance algorithm where views can involve multi-joins. The performances of these algorithms are analyzed using the scalable and portable *BSP*[1] cost model which predicts a near-linear speedup.
Key words: *PDBMS*[2], *materialized view maintenance, parallel incremental algorithm, data-warehouse, multi-joins, data skew, dynamic load balancing.*

1 Introduction

Data warehousing has become increasingly visible as a research issue following enormous market activity in the past few years [1]. Similarly, materialized views are finding increased research activity, with applications in decision support, OLAP[3], query optimization and replication, all of which are relevant for data warehousing. A view can be materialized by storing the tuples of the view in the database. A materialized view provides fast access to data. The speed difference is critical in applications where the query rate is high and the views are complex, so that it is not feasible to recompute the view of every query. If some of the base relations are changed, materialized views must be updated to ensure correctness of answers to queries. The problem of finding such changes has come to be known as the *view maintenance problem* and has been studied extensively [6, 5, 8]. Many sequential incremental algorithms are developed to this end. Note that if the view is defined by a relational expression containing join operators, the cost involved when reevaluating the view even incrementally may be unacceptable. Parallelism can greatly increase processing power. However, the effectiveness of parallel processing depends on ability to evenly divide

[1] BSP : Bulk Synchronous Parallelism.
[2] PDBMS : Parallel Database Management Systems.
[3] OLAP : On Line Analytical Processing.

load among processors while minimizing local computation and communication costs inherent to multi-processors machines. *Data skew* can have a disastrous effect on parallel performance [4, 13, 2, 9, 7] and parallelism can only maintain acceptable performance through efficient algorithms realizing complex queries on dynamic, irregular and distributed data. Such algorithms must dynamically reduce or eliminate data skew at the lowest possible cost. Our contributions to materialized view maintenance are:

1. a new parallel join algorithm called *duplicating_join*. This algorithm guarantees a near-linear speed-up and is based on:
   (a) a partial duplication of data allowing to reduce the communications' costs inherent to multi-processor machines.
   (b) the redistribution algorithm of *fa-join* [4] which efficiently avoids the problem of attribute value and join product skews.
2. a parallel incremental algorithm to maintain materialized views. This algorithm is based on a new incremental approach. This model has near-perfect balancing properties and supports flexible control of communications.
3. We use the scalable and portable BSP cost model to predict the performances of these algorithms.

2 PDBMS, join operations and the BSP cost model

2.1 The BSP cost model

Bulk-Synchronous Parallelism (BSP) is a parallel programming model introduced by Valiant [14] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time $g*h$ for any arity h.

A BSP program is executed as a sequence of *supersteps*, each one divided into three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l,$$

where $w_i^{(s)}$ = local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during superstep s.

2.2 Load balancing in parallel join operations

Join is an expensive and frequently used operation and parallelization of this operation is highly desirable. In PDBMS, relations are generally partitioned among processors by *horizontal fragmentation* with respect to the values of a chosen attribute. Parallel join usually proceeds in two phases: a redistribution phase by join attribute hashing and then sequential join of local fragments. Many algorithms have been proposed. The principal ones are: *Sort-merge join*, *Simple-hash join*, *Grace-hash join* and *Hybrid-hash join* [12]. All of them (called hashing algorithms) are based on hashing functions which redistribute relations so that tuples having the same attribute value are forwarded to the same node. Local joins are then computed and their union is the output relation. Their major disadvantage is to be vulnerable to both *attribute value skew* (imbalance of the output of the first phase) and *join product skew* (imbalance of the output of local joins) [13, 11, 10]. The former affects immediate performance and the latter affects the efficiency of output or pipelined operations in the case of a multi-join.

Research has shown that the join is parallelizable with near-linear speed-up on scalable SN architectures but only under ideal balancing conditions. Many algorithms have been proposed to handle data skew for a simple join operation, but little is known for the case of complex queries leading to multi-joins [10, 7, 2, 11]. In particular, the performance of PDBMS has generally been estimated on queries involving one or two join operations only [15]. However the problem of data skew is more acute with multi-joins because the imbalance of intermediate results is unknown during static query optimization.

To address this problem, we introduced in [4] a data-redistribution algorithm with near-perfect balancing properties. It is adaptable to θ-join and efficient for multi-joins. It computes *exact* frequency histograms to avoid *join product skew* (JPS). A scalable and portable cost analysis was made with the BSP model. The analysis suggests a hybrid *frequency-adaptive* algorithm (*fa_join* algorithm)[4], dynamically combining histogram-based balancing with standard hashing methods. The *fa_join* algorithm avoids the slowdown usually caused by attribute value skew (AVS) and the imbalance of the size of local joins processed by standard algorithms. It avoids AVS and JPS at the cost of extra processing time. We analyzed this overhead both theoretically and experimentally and concluded that it does not penalize overall performance.

Moreover, *fa_join's* performance can be slightly improved when computing the join of two relations R and Δ where $\|\Delta\| \ll \|R\|$ by reducing communication cost. We therefore introduce a new parallel algorithm called *duplicating join* to perform such joins. Its performance is analyzed using the BSP cost model which also predicts a near-linear speedup for this case.

3 Duplicating_join algorithm

In this section we describe our new algorithm for *SN* machines, adapted to the treatment of the join of two relations R and Δ where $\|\Delta\| \ll \|R\|$. This

algorithm is called *duplicating_join* and is based on :(1) a partial duplication of data allowing to reduce the communication costs between different processors and (2) the redistribution algorithm of *fa_join* algorithm which allows to avoid efficiently both attribute value and join product skews.
In the rest of this paper we use the following terminology for each relation T :

- T_i denotes the fragment of relation T placed on processor i,
- $Hist(T)$ denotes the histogram of T with respect to the join attribute,
- $\|T\|$ denotes the size of relation T.

We first assume that relation R (resp. Δ) is partitioned among processors by horizontal fragmentation and the fragments R_i for $i = 1 \ldots p$ are almost of the same size on every processor, i.e. $\|R_i\| \simeq \frac{\|R\|}{p}$ where p is the number of processors.

We will describe the *duplicating_join* algorithm while giving an upper bound on the *BSP* execution of each phase. This algorithm proceeds in 4 phases :

Phase 1 : Partial data duplication

To minimize the communication costs, each processor i broadcasts the fragment Δ_i to the other processors in time: $time_{phase1} = O\left(g * \|\Delta\|\right)$, where g is the time to send/receive a message of one word (cf. 2.1).

At the end of phase 1, each processor has a local access to the whole relation Δ (note that $\|\Delta\| \ll \|R\|$). Due to the identities: $R \bowtie \Delta = (R_1 \cup R_2 \cup \ldots \cup R_p) \bowtie \Delta = \cup_i (R_i \bowtie \Delta)$, broadcasting the fragments Δ_i allows us to perform (in parallel) local joins: $R_i \bowtie \Delta$ without extra communication costs.

Phase 2 : Processing local semi-joins

To reduce the redistribution cost, we compute the local semi-joins : $\widetilde{R_i} = R_i \ltimes \Delta$ by a sequential scan of the hash table, on each processor, in time:

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

We mention that, unlike the hash-based algorithms where both the *build* and the *probe* relations are redistributed, and contrary to the *fa_join* algorithm where only the build relation is redistributed, in the *duplicating_join*, we only redistribute $R \ltimes \Delta$ to perform $R \bowtie \Delta$. Thus we reduce to the minimum, the communication cost.

Phase 3 : Data redistribution

To avoid data skew and thus evenly divide load among processors, the semi-joins $\widetilde{R_i} = R_i \ltimes \Delta$ are redistributed using the redistribution algorithm of the *frequency-adaptive* algorithm [4]: On each processor i, the relation $\widetilde{R_i} = R_i \ltimes \Delta$ is divided into the two sub-relations $\widetilde{R_i'}$ and $\widetilde{R_i''}$ in the following manner: $\widetilde{R_i} = R_i \ltimes \Delta = \widetilde{R_i'} \cup \widetilde{R_i''}$ where:

- All the tuples of relation $\widetilde{R_i'}$ occur more than a threshold frequency f_0,
- All the tuples of relation $\widetilde{R_i''}$ occur less than the threshold frequency f_0[4].

[4] In this paper, we assume $f_0 = p * log(p)$.

To avoid *JPS*, the relations $\widetilde{R'_i}$ are redistributed using the redistribution algorithm of the *fa_join* [4] in time:

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

where $\|Hist(\widetilde{R'})\|$ denotes the size of the histogram, of $\widetilde{R'}$, based on the join attribute and l the time to achieve a synchronization barrier (cf. 2.1). Due to $\|Hist(\widetilde{R'})\| \leq \frac{\|\widetilde{R'}\|}{f_0} = \frac{\|\widetilde{R'}\|}{p*\log(p)}$, we have :

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

Relations $\widetilde{R''_i}$ are also redistributed using a hash function in time: $time_{a2} = O\Big(g * \max_{i=1\ldots p} \|\widetilde{R''_i}\|\Big)$. Hence, the global redistribution of the relations $\widetilde{R_i}$ has therefore taken the sum of the above two redistribution times:

$$time_{phase3} = time_{a1} + time_{a2} \leq O\Big(g * \max_{i=1\ldots p} \|\widetilde{R_i}\| + (g+l) * \log(p)\Big).$$

Redistribution of $\widetilde{R_i}$ is performed to avoid JPS because the values which could lead to AVS (those having high frequencies) are those which often cause the join product skew. It avoids thus, the slowdown usually caused by attribute value skew and the imbalance of the size of local joins processed by the standard algorithms. Local joins $\widetilde{R_i} \bowtie \Delta$ will have therefore almost the same size on each processor (i.e. $\|\widetilde{R_i} \bowtie \Delta\| \simeq \frac{\|R \bowtie \Delta\|}{p}$, for $i = 1 \ldots p$).

Phase 4 : Processing local joins

Once the redistribution of $\widetilde{R_i}$ is achieved, local joins could be computed in time:

$$time_{phase4} = O\Big(\max_{i=1\ldots p} (\|\widetilde{R_i}\| + \|\Delta\| + \|\widetilde{R_i} \bowtie \Delta\|)\Big) \leq O\Big(\frac{\|R\|}{p} + \|\Delta\| + \frac{\|R \bowtie \Delta\|}{p}\Big).$$

The global cost of the *duplicating_join* algorithm has therefore taken the sum of the costs of the 4 phases above:

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

Remark : Observe that sequential join processing of two relations R and S would require at least time : $bound_1 = \Omega(\|R\| + \|S\| + \|R \bowtie S\|)$. Parallel processing with p processors requires therefore $bound_p = \frac{1}{p} * bound_1$, and the *duplicating_join* algorithm has optimal asymptotic complexity when :

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

4 Incremental view maintenance : A new approach

In relational database systems, a derived relation or *view* is defined by a relational expression (i.e. a query evaluated over the base relations). A view may be virtual, which corresponds to the traditional concept of a view; or materialized, which means that the resulting relation is actually stored. As the database changes because of updates applied to the base relations, the materialized views involving the updated relations may also require change. The incremental method to evaluate the materialized view consists to identify the set of tuples to be inserted or deleted in the views avoiding the whole evaluation of the view [6, 5, 8].

In this paper, we are interested in SPJ[5] views. As in [6], we suppose that the duplicates are kept in the view, either explicitly, or using a counter (necessary to the delete operations when the materialized view involves project operators [6]). Let $v = R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n$ where R^i are base relations. Assume that we insert the sets of tuples $Ins(R^i)$ in relations R^i, $i = 1, \ldots, n$, respectively. Then the new state of the view is $v' = (R^1 \cup Ins(R^1)) \bowtie \ldots \bowtie (R^n \cup Ins(R^n))$.

By distributivity of join over union, the updated view v denoted by v' is equal to the union of 2^n multi-joins called *partial multi-joins*. For $n = 3$, the [6]'s (resp. the following new approach's) partial multi-joins are shown in table Tab. 1.(a) (resp. Tab. 1.(b)). The multi-join $R^1 \bowtie R^2 \bowtie R^3$ corresponds to the materialized view v.

(a)	(b)
$R^1 \bowtie R^2 \bowtie R^3$	$R^1 \bowtie R^2 \bowtie R^3$
$R^1 \bowtie R^2 \bowtie Ins(R^3)$	$Ins(R^1) \bowtie R^2 \bowtie R^3$
$R^1 \bowtie Ins(R^2) \bowtie R^3$	$(R^1 \cup Ins(R^1)) \bowtie Ins(R^2) \bowtie R^3$
$R^1 \bowtie Ins(R^2) \bowtie Ins(R^3)$	$(R^1 \cup Ins(R^1)) \bowtie (R^2 \cup Ins(R^2)) \bowtie Ins(R^3)$
$Ins(R^1) \bowtie R^2 \bowtie R^3$	
$Ins(R^1) \bowtie R^2 \bowtie Ins(R^3)$	
$Ins(R^1) \bowtie Ins(R^2) \bowtie R^3$	
$Ins(R^1) \bowtie Ins(R^2) \bowtie Ins(R^3)$	

Tab.1 : [6]'s and new approach's partial multi-joins for n=3.

Using the [6] approach, For a given view $v = R^1 \bowtie \ldots \bowtie R^n$, we must perform $2^k - 1$ multi-joins where k is the number of the updated relations involved in v.

In this paper, to reduce the number of the necessary partial multi-joins to be performed to maintain v incrementally, we use a variant of the [6] approach using only k partial multi-joins. To capture these k partial multi-joins, we associate to $v = R^1 \bowtie \ldots \bowtie R^n$ a matrix T with n rows and n columns in such away that every row of T corresponds to one partial multi-join PMJ_i of v :

$$PMJ_i = T_{i,1} \bowtie T_{i,2} \ldots \bowtie T_{i,n} \quad \text{where :} \quad T_{i,j} = \begin{cases} R^j \cup Ins(R^j) & \text{if } j < i \\ Ins(R^j) & \text{if } i = j \\ R^j & \text{if } j > i \end{cases}$$

Note that, if a relation R^j is not updated, the correspending partial multi-join PMJ_j is a null relation. We generalize this result to SPJ views and show that for every materialized view, $v = \sigma_{\mathcal{C}} \pi_{\mathcal{X}}(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n)$, if we insert the sets of tuples $Ins(R^i)$ in the relations R^i, $i = 1, \ldots, n$, respectively, the set of tuples v_{Ins} to be inserted in the view v is :

[5] SPJ : Select, Project and Join.

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

Tab. 2 : Insertion into base relations.

The same idea may be applied when we delete the sets of tuples $Del(R^i)$ from the relations R^i [3]. Thus, the set of tuples v_{Del} to be deleted from the view v is the set v_{Ins} defined before in which $Ins(R^i)$ are replaced by $Del(R^i)$ and the union symbol $\cup$ is replaced by the set difference symbol $-$.

5 Parallel incremental evaluation

An update of a base relation of a materialized view v can be decomposed into two phases: insertion phase in base relations followed by deletion phase from base relations. After insertion phase (or deletion phase) of sets of tuples in base relations, we evaluate the set of tuples v_{Ins} (or v_{Del}) to be inserted (or deleted) in (from) v using the appropriate parallel incremental algorithm, then we use the appropriate parallel maintenance algorithm to perform the view update.

We present our parallel algorithm in the case of the insertion into base relations, the deletion update is detailed in [3].

5.1 Case of simple join

Let $R \bowtie S$ be the join expression defining the view v. Assume we insert the sets of tuples $Ins(R)$ and $Ins(S)$ in relations R and S respectively. To maintain v incrementally, according to the [6] approach, it is sufficient to compute the set v_{Ins} of tuples to be inserted in v :

$$v_{Ins} = (Ins(R) \bowtie S) \cup (R \bowtie Ins(S)) \cup (Ins(R) \bowtie Ins(S)). \tag{1}$$

Now we present a parallel incremental evaluation algorithm of v based on the [6] approach. The sequential maintenance of the view v (cf. equation 1) requires the following lower bound :

$$
\begin{aligned}
bound_{inf_1} = \Omega\Big(& \|Ins(R)\| + \|S\| + \|Ins(R) \bowtie S\| + \|R\| + \|Ins(S)\| + \\
& \|R \bowtie Ins(S)\| + \|Ins(R)\| + \|Ins(S)\| + \|Ins(R) \bowtie Ins(S)\|\Big).
\end{aligned} \tag{2}
$$

Therefore, the processing time using p processors requires at least :

$$bound_{inf_p} = \frac{1}{p} * bound_{inf_1}. \tag{3}$$

We first assume that relations R (resp. S) is evenly partitioned among different processors using horizontal fragmentation. And when the base relations are updated, each processor holds a fragment of relations $Ins(R)$ and $Ins(S)$.

The view update of v according to the [6] approach, is performed by computing the view v_{Ins} which corresponds to the three joins $Ins(R) \bowtie S$, $R \bowtie Ins(S)$ and $Ins(R) \bowtie Ins(S)$ in the following manner:

a: when $\|Ins(R)\| \ll \|S\|$, the (*duplicating_join algorithm*) can be used efficiently to process the join $Ins(R) \bowtie S$ in time:

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

b: when $\|Ins(S)\| \ll \|R\|$, the (*duplicating_join algorithm*) can be used efficiently to process the join $R \bowtie Ins(S)$ is in time:

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

c: To compute the join $Ins(R) \bowtie Ins(S)$, we compute one of the following local joins: $Ins(R) \bowtie Ins(S)_i$ or $Ins(R)_i \bowtie Ins(S)$. If we choose to compute $Ins(R)_i \bowtie Ins(S)$, the join $Ins(R) \bowtie Ins(S)$ is processed at most in time:

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

Hence, the global view maintenance cost is:
$time_{v_{Ins}} = time_{step_a} + time_{step_b} + time_{step_c}$.

$$= O\Big((1+g)*(\|Ins(R)\|+\|Ins(S)\|+\max_{i=1...p}\|R_i\|+\max_{i=1...p}\|S_i\|)+\frac{\|Ins(R) \bowtie S\|}{p}$$
$$+\frac{\|R \bowtie Ins(S)\|}{p}+\max_{i=1...p}\|Ins(R)_i \bowtie Ins(S)\|+(g+l)*\log(p)\Big). \quad (4)$$

The view maintenance algorithm is optimal when:

$$\max\Big(\|Ins(R)\|, \|Ins(S)\|, \max_{i=1...p}\|Ins(R)_i \bowtie Ins(S)\|\Big)$$
$$\leq \frac{1}{p}*\max\Big(\|R\|, \|S\|, \|R \bowtie Ins(S)\|, \|Ins(R) \bowtie Ins(S)\|\Big), \quad (5)$$

this is due to the fact that all the terms in $time_{v_{Ins}}$ are bounded by those of $bound_{inf_p}$(cf. équation : 3).
Observe that unlike the [6] approach, in our incremental algorithm, the two joins $R \bowtie Ins(S)$ and $Ins(R) \bowtie Ins(S)$ are implicitly evaluated by computing the join $(R \cup Ins(R)) \bowtie Ins(S)$ (cf. section 4). Using the *duplicating_join* algorithm, the computation of $(R \cup Ins(R)) \bowtie Ins(S)$ is done in time: $O\Big((1+g)*(\max_{i=1...p}(\|R_i\|+\|Ins(R)_i\|)+\|Ins(S)\|)+\frac{\|Ins(R)\bowtie Ins(S)\|}{p}+\frac{\|R \bowtie Ins(S)\|}{p}+(g+l)*\log(p)\Big)$, which is slightly better than the sum of the times of the steps b and c, i.e. $time_{step_b} + time_{step_c}$. This reduces to two the number of the joins to be evaluated. In general (cf. section 4), the number of the joins to be evaluated is reduced from 2^k-1 to k (where k is the number of the updated relations). Therefore, the incremental evaluation's cost for multi-joins views is greatly reduced.

The equation 5 is generally verified when $\|Ins(R)\| \ll \|S\|$ and $\|Ins(S)\| \ll \|R\|$. However, when $\|Ins(R)\| \simeq \|S\|$ (resp. $\|Ins(S)\| \simeq \|R\|$), the join $Ins(R) \bowtie S$ (resp. $R \bowtie Ins(S)$) can be efficiently evaluated using the *fa_join* [4] algorithm with near-linear speedup. We recall that the two algorithms *fa_join* and *duplicating_join* guaranty a near-perfect balance of join results. Hence, these algorithms are well adapted for multi-joins operations. Due to the fact that all the multi-joins are evaluated with near-linear speedups, the incremental view maintenance is then evaluated in a near-linear speedup too.

5.2 Case of SPJ views

We consider in this section materialized views defined by relational SPJ expressions of the form $v = \sigma_{\mathcal{C}}\pi_{\mathcal{X}}(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n)$ where R^i are relations, $\mathcal{C}$ a select condition and $\pi_{\mathcal{X}}$ is the projection over the set of attributes $\mathcal{X}$. Assume we insert the sets of tuples $Ins(R^i)$ in the base relations R^i, $i = 1, \ldots, n$, respectively.

To optimize the incremental evaluation cost, the partial views are evaluated from the top to the bottom (cf. table Tab.2). Hence, the first partial view to evaluate is $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}(Ins(R^1) \bowtie R^2 \bowtie R^3 \bowtie \ldots \bowtie \mathcal{R}^n)$. As the rest of the partial views to evaluate involve the relation $R^1 \cup Ins(R^1)$, we definitively update the relation R^1 by computing $R^1 \cup Ins(R^1)$, we compute then the partial view $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 \cup Ins(R^1)) \bowtie Ins(R^2) \bowtie R^3 \bowtie \ldots \bowtie R^n)$. After that, we definitively update the relation $\mathcal{R}^2$ by computing $R^2 \cup Ins(R^2)$ in time $\max_{i=1\ldots p} \|(Ins(R^2))_i\|$. Then, we compute the partial view $\sigma_{\mathcal{C}}\pi_{\mathcal{X}}((R^1 \cup Ins(R^1)) \bowtie (R^2 \cup Ins(R^2)) \bowtie Ins(R^3) \bowtie \mathcal{R}^4 \bowtie \ldots \bowtie R^n)$. Note that when we compute the above partial view, we do not evaluate $R^1 \cup Ins(\mathcal{R}^1)$ since it is already done in the previous step. The rest of partial views to be evaluated is computed with the same manner so that, during every partial view evaluation, one and only one base relation is updated.

The *fa_join* and *duplicating_join* algorithms avoid the problem of AVS and JPS. Therefore, the local joins results sizes are almost the same in each processor. The select and project operations could be done with a near-linear speedup.

We have shown that, in the insertion (resp. deletion[3]) updates over base relations, the parallel incremental evaluation algorithm of the materialized view v, has a near-linear speedup since all the partial views of v_{Ins} (resp. v_{Del}) have near-linear speedups. Hence, our parallel algorithm for evaluating views incrementally has a near-linear speedup.

6 Parallelizing materialized view updates

The evaluation algorithm of the view v_{Ins} (resp. v_{Del}) has near-linear speedup. For the incremental view maintenance, we must insert (resp. delete) the tuples of the view v_{Ins} (resp. v_{Del}) in (resp. from) the materialized view.

To this end, local fragments $(v_{Ins})_i$ (resp. $(v_{Del})_i$) are redistributed using the same placement function as for the materialized view. The global redistribution cost (and thus the materialized view update cost) depends generally on the placement function used to store the materialized view on different processors. The average cost of the redistribution is $g * \max_{i=1,\ldots,p}(v_{Ins})_i$ (resp. $g * \max_{i=1,\ldots,p}(v_{Del})_i$). Note that in the worst case, when all the blocs $(v_{Ins})_i$ (resp. $(v_{Del})_i$) are sent to same processor, the redistribution cost would not exceed $g * \|v_{Ins}\|$ (resp. $g * \|v_{Del}\|$). This algorithm is detailed in [3].

7 Conclusion

In this paper we have presented a new parallel join algorithm called *duplicating_join* based on partial duplication of data with near perfect balancing properties. We have shown that this algorithm combined with the fa_join[4] algorithm can greatly increase processing power in the case of materialized views and thus data-warehousing applications by the design of a new parallel view maintenance algorithm for Shared Nothing machines.

The performances of these algorithms are analyzed using the BSP cost model which predicts a linear speedup. The $O(\ldots)$ notation only hides small constant factors: they depend only on the implementation but neither on data nor on the the BSP machine. Our experience with the join operation [4] is evidence that the above theoretical analysis is accurate in practice.

References

1. 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, 1998.
2. 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.
3. M. Bamha, F. Bentayeb, and G. Hains. Un algorithme incrémental parallèle pour la maintenance des vues matérialisées. Technical Report RR99-3, LIFO, Université d'Orléans, 1999.
4. 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.
5. 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.
6. José A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. *ACM SIGMOD*, 1986.
7. 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.
8. Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1995.
9. K. A. Hua, W. Tavanapong, and H. C. Young. A Performance Evaluation of Load Balancing Techniques for Join Operations on Multicomputer Database systems. In *Proc. of the 11th International Conference on Data Engineering*, CA, USA, 1995.
10. Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, California, 1994.
11. 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, Bombay, India*, 1996.
12. 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.
13. 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*, 1990.
14. Leslie Valiant. A Bridging Model for Parallel Computation,. *Communication of the ACM, Vol 33, No. 8.*, August 1990.
15. Annita N. Wilschut, Jan Flokstra, and Peter M.G. Apers. Parallel Evaluation of Multi-join Queries. *In the Proc. Of the ACM-SIGMOD, California*, 1995.