

Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems

Lilian Harada
Fujitsu Labs.
lilian@flab.fujitsu.co.jp

Masaru Kitsuregawa
University of Tokyo
kitsure@tkl.iis.u-tokyo.ac.jp

Abstract

When data are uniformly distributed, parallel hash-based join algorithm scales up well. However, the presence of data skew can cause load imbalance among the processors, significantly deteriorating its performance. In this paper we propose a dynamic skew handling algorithm which deals with this load imbalance, by detecting and handling join product skews at run-time. The idea is to monitor the join processing at the join phase and compare the average processing rate of each partition with the rate statically predicted at the scheduling phase. If their difference is detected to be large enough to produce a significant performance degradation, the processor is considered to be overloaded and a workload compensation strategy is dynamically invoked. In this case, based on the measured average processing rate, the amount of overload caused by the unpredicted join product skew is calculated and, the amount of load to be migrated to the non-overloaded processors is determined. We propose two methods - the *result redistribution* and the *processing task migration* - to handle the load migration from the overloaded processor to the non-overloaded processors. Simulation results show that our dynamic skew handling approach can detect and handle load imbalances efficiently, so that the rebalance of load among the processors results in an almost constant join execution time under different join product skews.

1 Introduction

Applying multiprocessor machines to database query processing has been an active area of research. In any parallel environment, in order to get optimal speedup and scaleup, pieces of the processing load must be assigned carefully in order to ensure equal execution times for all processing nodes. Because the join operation is one of the most expensive relational operations, there has been much research in the area of parallel join algorithms. Nowadays, the hash-based join [6], [1] is the most popular algorithm for computing the equi-join in parallel database systems.

Originally, data were assumed to be uniformly distributed and hence it was trivial to generate partitions of equal size and to guarantee equal execution times. However, as parallel hash-join algorithms have become well understood, this data uniformity assumption has been challenged and the nonuniform data distribution, referred to as *data skew*, showed to result in problems of load imbalance and thus in severe performance degradation [7], [10].

The first works on skew handling for parallel hash-joins adopted the simplistic assumption that the execution time of a join operation could be estimated as proportional to the product of the sizes of its two operand relations. Therefore, these preliminary approaches handle the *redistribution skew* and introduce a number of solutions to redistribute the skewed data in equi-sized partitions [4],[5],[9]. However, even when the operand relations are distributed among the processing nodes evenly, each node can still require a different join execution time since the join selectivity may vary among the partitions, due to the *join product skew*. Wolf et al. and DeWitt et al. were the firsts to handle the join product skew for parallel hash-joins in shared-nothing database systems in [11] and [2], respectively. They introduced *static join product skew handling algorithms* which attempt to predict the partitions' join execution behavior, before the actual join phase. The algorithm introduced in [11] adds an extra scanning and scheduling phase to the usual hash-based join algorithm. First, based on statistics of the sizes of fine hash buckets taken in a pre-scan of the relations, these fine hash buckets are grouped into partitions which can fit into memory. Then, using a LPT heuristic scheduling algorithm [3], the partitions are scheduled to the processing nodes so that the total execution time of the processing nodes is equalized. On the other hand, in [2], the relations to be joined are first sampled instead of scanned. Based on the resulting set of samples of the relations, the tuples are divided into small partitions determined by fine subranges of the join attribute value, and are then scheduled to the processing nodes in a round-robin scheme (the algorithm VP-RR), or like in [11], by estimating the fine partitions' join costs and using the LPT to equalize the total time required by the fine partitions allocated to the nodes (the algorithm VP-PS). Then, in the redistribution phase, for the algorithms in both [11] and [2], data are exchanged between the processing nodes according to the static scheduler decision,

and the partitions are locally assembled. Finally, in the join phase, each processor reads its local pair of partitions and joins it on its own site, as in any usual hash-based join algorithm. The evaluation results in [11] and [2] showed that, although the scan/sampling and scheduling phases could guarantee the load balance among the processing nodes in many cases, it was impossible to predict some join product skews where a small number of tuples with a repeated join attribute value caused a blow-up in the number of generated tuples, resulting in poor load balance.

Recently, with the large increase in network speed (with up to 200 MB/sec interconnects!), run-time migration to handle those imbalances caused by unpredicted join product skews becomes feasible. Shatdal & Naughton were the firsts to propose a *dynamic join product skew handling algorithm* for shared-nothing database systems in [8]. In this algorithm, during the join phase, whenever a processor finishes the join of a partition, it checks to see if any other processors are still running; if there are other processors still running, the newly idle processor picks a busy processor and have a portion of that processor's load migrated under the shared-virtual memory paradigm. What they do is to build all the hash-tables in the shared-virtual memory so that only a portion of the probe relation has to be explicitly migrated from a busy processor to a newly idle processor. This dynamic approach that checks the processors and balance their loads after the processing of each partition introduces a synchronization barrier at the end of each partition processing. For the case there is only one partition per processing node, this simple approach seems to be enough to handle the load imbalance among the nodes. Also, it seems to be an efficient approach to handle unexpected join product skews which arise when using static schedulers that guarantee that the processors have the same number of partitions to join, and that those partitions are estimated to have almost the same execution time, like that of VP-RR in [2]. However, it presents some problems when using more general static schedulers like that in [11] or the VP-PS algorithm in [2].

Let us consider, for example, the static scheduler illustrated in Fig. 1(a). In this example there are 4 processors - P_1 is scheduled to join three pairs of partitions, and the other three processors P_2 , P_3 and P_4 are scheduled to join two pairs of partitions each. As illustrated by the different bars' lengths in the figure, all the partitions have different estimated join times; however, the total join time in the 4 processors is estimated to be the same. It is easy to see that in this case, even in the case all partitions' join costs are correctly estimated in the scheduling phase, if a dynamic migration is invoked after the join of each partition, it will result in much unnecessary load migration. So, when should the dynamic migration be invoked? Would it be enough to avoid the synchronization after each partition's join and only perform the load migration when a processor finishes the join of all its partitions? The answer is no. Suppose there is a processor P whose first partition's join selectivity is much higher than the estimated in the static scheduling phase, and also much higher than that of the first partitions in the other processors. In this

case, P takes much more time than the others to finish its first partition's join. Because all the other processors have their own partitions to join yet, there is no load migration from P to the other processors at this point. After some time, P finishes the join of that first partition and begins the join of the second one. At this time, the other processors finish their own partitions' join and so, are available to help P in its partitions' join. However, if the join selectivity of the remaining partitions in P are not high enough, the load migration from P to the other processors is not helpful anyway, since most of the execution time is represented by the time P takes to read off the disk. Therefore, a naive load migration which follows the policy of "don't be idle if there is work left" cannot handle load imbalances efficiently when the processing nodes are scheduled to join different number of partitions, with different execution times. We will introduce an algorithm which also uses the idea of run-time migration. However, our algorithm differs from [8] on that it can handle the unexpected join product skews which arise when applying any of the previously proposed static schedulers in [11] and [2], with no restrictions in the number of partitions scheduled per node, neither the estimated partitions' costs, and also on the fact that our algorithm is not specifically designed for shared-virtual memory.

In this paper we present our dynamic join product skew handling algorithm which detects and handles the join product skew at run-time, by only applying the load migration when a processor is detected to be overloaded during its join processing. This overload detection is done by monitoring the join at run-time and comparing the average processing rate of each partition measured during the join phase, to the rate statically predicted in the scheduling phase. If their difference is large enough to produce a significant performance degradation, the processor is considered to be overloaded and thus, a workload compensation strategy is dynamically invoked. In this case, based on the measured average processing rate, the amount of overload caused by the unpredicted join product skew is calculated. Then, the amount of load to be migrated to the non-overloaded processors is determined. We propose two methods - the *result redistribution* and the *processing task migration* - to handle the load migration from the overloaded processor to the non-overloaded processors. In the *result redistribution*, an overloaded processor only migrates the join result to the other non-overloaded processors, which concurrently continue their own join. If the network and CPU were ideal resources of infinite speed, the result redistribution would be enough to handle any overload. However, in fact, these resources have a limited speed and thus, we introduce the *processing task migration*. The processing task migration takes place when a non-overloaded processor finishes its own partition's join and so, its memory is free so that it can alleviate the overloaded processor's CPU and the network utilization by receiving some portion of the overloaded processor's partitions and join them. We consider two types of processing task migration: the *hash-table migration* where a portion of the hash table of the overloaded processor is replicated and

the probing partition is broadcast to the non-overloaded processor; and the *probe migration* where the hash-table is completely replicated and a portion of the probing partition is migrated to the non-overloaded processor. Note that our probe migration mechanism is similar to that in [8], although we don't employ the shared-virtual memory paradigm. Evaluation of our dynamic algorithm and comparison with a static algorithm shows that, even when join product skews cannot be properly predicted in the static algorithm, our dynamic algorithm which checks and re-balances the load among the processors at run-time can efficiently handle them and, despite the overhead incurred in this run-time migration, it results in an almost constant join execution time under different join product skews.

Like the previous approaches to handle the join product skew, as a first step, in this paper we focus on the single join operator only. However, the load migration methods introduced here seem to be promising to handle the join product skew in the case of multi-way joins also. In such a case, the result of one join operator has to be redistributed among the other processors according to the value of the next join attribute and so, a balancing using the *result redistribution* is not possible; in this case, the *processing task migration* is employed. Of course, at the last join operator, both result redistribution and processing task migration can be employed when appropriate. We intend to analyze this case of multi-way join in details in a further work.

The remainder of this paper is organized as follows. Section 2 presents an overview of our dynamic join product skew handling algorithm using an example, and Section 3 details its fundamentals. In Section 4 we present our algorithm description. In Section 5 we present some simulation results. Finally, we conclude in Section 6.

2 An Overview of Our Dynamic Skew Handling Algorithm

In the following, we present an example to illustrate our dynamic join product skew handling algorithm. In this example, we assume a shared-nothing database system composed of four processors P_1 , P_2 , P_3 and P_4 . As in the conventional static join product skew handling algorithms [11],[2], first, in a scan/sampling phase, statistical profile of the join attribute values in the relations are obtained by scanning or sampling the operand relations. Then, in the scheduling phase, based on these statistics, the partitions are scheduled so that the total join execution times for the four processors are almost equal. In this example, as illustrated in Fig. 1(a), P_1 is scheduled to join three pairs of partitions, and the other three processors P_2 , P_3 and P_4 , two pairs of partitions. In the redistribution phase, data are exchanged and assembled in the processors, as determined in the scheduling phase.

In the join phase, first, each processor P_i ($i = 1, \dots, 4$) begins the processing of its first pair of partitions, i.e., each processor P_i reads the first building partition from its local disk, builds the corresponding hash-table in its local memory, and for each tuple of the probing partition read from the disk, P_i probes the hash-table for matches and

when such a match is found, writes it into its local disk. To ensure that these partitions' joins are performed as estimated in the scheduling phase, in our approach these joins are monitored by gathering information from their partial processing statistics. Thus, after an interval of time (Δt_i), the amount of probing partition that was processed during this interval of time (Δt_i) is examined, in order to check if processor P_i is consuming the probing partition at the rate that was estimated in the scheduling phase. In the example in Fig. 1(a), P_1 , P_2 and P_3 are not detected to be overloaded. However, it is detected that the amount of probing partition of P_4 that was processed in (Δt_4) is much smaller than the estimated, which means that P_4 will take a much longer time than the estimated to process the whole probing partition. Note that the estimation of the total partition's join cost from its performance during a partial processing must pay close attention to the order in which tuples are processed, that is, the partial behavior of a join processing can be misleading if the distribution of the values of attributes is correlated to the order in which tuples are processed. Here we consider that the tuples in the probing partition are randomly distributed so that the past processing rate is a good predictor of future work.

As illustrated in Fig. 1(b), the difference between the re-estimated time to process all the partition and the previously mis-estimated time in the scheduling phase represents how much processor P_4 is overloaded. In our approach, this overload is dynamically divided among the four processors so that the execution time of the processors is re-equalized. We use two methods - *result redistribution* or *processing task migration* - to handle the load migration from the overloaded processor P_4 to the non-overloaded processors P_1 , P_2 , and P_3 . *Result redistribution* is applied when the non-overloaded processors are still processing their local pair of partitions and thus, their memory are filled by their own hash tables. In this case, the overloaded processor distributes the result matching tuples to the non-overloaded processors, so that the heavy task of writing the matching tuples is divided among all processors. Note that this approach allows that the reading of the operand partitions from the disk in the overloaded processor can be performed without constant intervention of the heavy task of writing the matching tuples. Concerning the non-overloaded processors, besides writing the results received from the overloaded processor in their local disks, they can concurrently continue their own join processing. On the other hand, *processing task migration* is applied when the non-overloaded processors have already processed their own pair of partitions and thus, their memory are free to load some portion of the building or probing partitions from the overloaded processor, and locally produce and write the corresponding matching tuples.

In Fig. 1(c) we show the time chart for the processing of the 4 processors, illustrating how the load are distributed and executed in each processor. We can see that after the detection of the overload of processor P_4 , processors P_1 , P_2 and P_3 continue the join of their own partitions, and concurrently write the *result redistributed* from P_4 . When processor P_1 finishes the join of its first partition, P_1 will

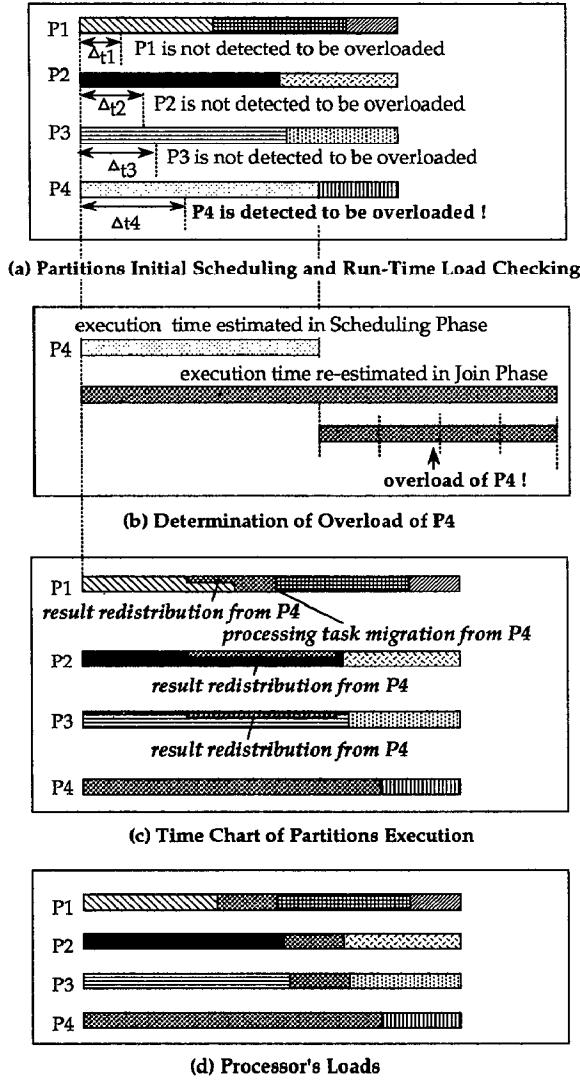


Figure 1: An Illustrative Example

change from *result redistribution* to *processing task migration*. When P_4 finishes migrating 1/4 of the overload to P_1 , P_1 can finally begin the join of its second partition. At this time, processors P_2 and P_3 are still concurrently processing their first partition and the overload from P_4 . When they finally finish to write 1/4 of the overload that is redistributed from P_4 , P_2 and P_3 return to process only their own partitions. As for the first partitions, all the processors are checked to detect any overload in the join of each new partition. However, in this example, no other processor overload is detected until the end of all partitions' joins.

While in Fig. 1(c) we show the processing time chart, in Fig. 1(d) we illustrate the total amount of load processed by each processor. We can contrast it with Fig. 1(a), which shows the initial join estimated times for the partitions, and see that although the overload of P_4 that was not expected has occurred, our dynamic join product skew handler has detected it at run-time and made it pos-

sible to have the load rebalanced equally among the four processors.

3 Fundamentals of Our Dynamic Skew Handling Algorithm

In this section, after introducing some notation, we describe how an overloaded processor is detected, how this overload is estimated, when this overload can be ideally migrated, and how this overload is migrated in our dynamic skew handling algorithm.

3.1 Notation

Let us consider the join of relations R and S in a shared-nothing database system composed of N processors P_i , for ($i = 1, \dots, N$). Let us assume that each processor P_i is allocated the partitions (R_{ij}) and (S_{ij}) , for ($j = 1, \dots, m_i$), where (R_{ij}) is used to build the hash table, and (S_{ij}) is used to probe it. Therefore, there is a total of $\sum_{i=1}^N m_i$ pairs of partitions to be joined by the N processors P_i , and each processor P_i is scheduled to join the pairs of partitions (R_{ij}, S_{ij}) . In the following we introduce some notation adopted throughout this paper.

- $Size(R_{ij})$: size of the j -th. building partition (R_{ij}) ;
- $Size(S_{ij})$: size of the j -th. probing partition (S_{ij}) ;
- $T_{est}(R_{ij})$: estimated time to read (R_{ij}) and build the hash table;
- $T_{est}(S_{ij})$: estimated time only to read (S_{ij}) ;
- $T_{est}(R_{ij}, S_{ij})$: estimated time to read (R_{ij}) and build the hash table, to read (S_{ij}) and probe the hash table, and to write the matching result tuples;
- $T_{meas}(R_{ij}, S_{ij})$: measured time to read (R_{ij}) and build the hash table, to read (S_{ij}) and probe the hash table, and to write the matching result tuples.

The estimation times above are the ones used in the scheduling phase. The static scheduler estimates that $\sum_{j=1}^{m_i} T_{est}(R_{ij}, S_{ij})$ is almost equal for all i ($i = 1, \dots, N$). In order to simplify the notation, we also define:

- $T_{est}^*(R_{ij}, S_{ij}) = T_{est}(R_{ij}, S_{ij}) - T_{est}(R_{ij})$: estimated time to read (S_{ij}) and probe the hash table, and to write the matching result tuples.

By assuming that the estimated and measured times to read (R_{ij}) and build the hash table are the same, that is, $T_{est}(R_{ij}) = T_{meas}(R_{ij})$, we also define:

- $T_{meas}^*(R_{ij}, S_{ij}) = T_{meas}(R_{ij}, S_{ij}) - T_{est}(R_{ij})$: measured time to read (S_{ij}) and probe the hash table, and to write the matching result tuples.

3.2 Overload Detection

In order to allow a dynamic balance during the join phase, an overloaded processor has to be properly checked and detected at run-time. Let us suppose that a processor P_u began the processing of a pair of partitions (R_{uv}, S_{uv}) at time t_{1u} , and at time t_{2u} , the portion of (S_{uv}) that has been read and finished processing is (ΔS_{uv}) . Assuming that $(\Delta t_u) = t_{2u} - t_{1u}$, the measured time to join

(R_{uv}) and (ΔS_{uv}) , that is, to build the hash table of (R_{uv}) , process the probing of (ΔS_{uv}) and write back the result tuples is $T_{meas}(R_{uv}, \Delta S_{uv}) = (\Delta t_u)$. It was estimated that it would take $T_{est}(R_{uv}, S_{uv})$ time to build the hash table of (R_{uv}) , probe all tuples of (S_{uv}) and write the results. In other words, we could say that the time to probe one tuple of (S_{uv}) and write the result tuples it generates was estimated as $T_{est}^*(R_{uv}, S_{uv})/Size(S_{uv})$. However, from $T_{meas}(R_{uv}, \Delta S_{uv})$, this time was in fact $T_{meas}^*(R_{uv}, \Delta S_{uv})/Size(\Delta S_{uv})$. Therefore, we can say that, for the processing of (R_{uv}, S_{uv}) , processor P_u is overloaded if:

$$\frac{T_{meas}^*(R_{uv}, \Delta S_{uv})}{Size(\Delta S_{uv})} - \frac{T_{est}^*(R_{uv}, S_{uv})}{Size(S_{uv})} > \alpha \quad (1)$$

The value α for this parameter depends upon the system configuration and how little skew we are willing to tolerate. In fact, any dynamic load balance will incur some overhead, as will be presented later and thus, α should be determined by considering this overhead cost.

Now, let us define $dev(R_{ij}, S_{ij})$ as the deviation in the measured and estimated times to join (R_{ij}, S_{ij}) , that is,

$$dev(R_{ij}, S_{ij}) = T_{meas}(R_{ij}, S_{ij}) - T_{est}(R_{ij}, S_{ij})$$

If this deviation is too small to be detected in (1), processor P_u is not considered to be overloaded by (1). However, by accumulating many small deviations, there is a point beyond which their summation becomes large enough so that the join processing of a pair of partitions starts much delayed than it was estimated by the scheduling phase. Thus, we can say that the processor P_u that is processing the v -th. partition (R_{uv}, S_{uv}) is overloaded if:

$$\sum_{j=1}^{v-1} dev(R_{uj}, S_{uj}) > \beta \quad (2)$$

Like α , the value β for this parameter depends upon the system configuration and how little skew we are willing to tolerate.

When (1) or (2) occurs, the processor P_u that is joining (R_{uv}, S_{uv}) is determined to be overloaded and should have its overloaded portion migrated to other processors in order to equalize the total processing times among all processors.

3.3 Estimation of Overload

As shown by conditions (1) or (2) in Subsection 3.2, in the join phase, a processor P_u that is joining (R_{uv}, S_{uv}) is detected to have a heavier load than estimated in the scheduling phase. What happens for P_u is that the number of matching tuples is much larger than the estimated and thus, the high cost of writing these matching tuples was unexpected. In our approach, what we do is to dynamically re-estimate the overload of handling the unexpected matching tuples for P_u , and distribute this overload to the lightly loaded processors. In contrast to the ABJ approach in [4] which physically relocates the excess partitions from the overloaded processor to the other processors attempting to balance the data load prior to the join operations, we

do not relocate (R_{uv}) nor (S_{uv}) to other processors. This is because if the join product skew is not large enough, the high cost of moving the partitions to other processors might be higher than the cost of processing the skewed partition in the overloaded processor. Therefore, we avoid the retransfer of any partition, which has already been transferred once in the redistribution phase. In our approach, (R_{uv}) and (S_{uv}) are stuck at the original overloaded processor P_u , and only this processor continues to read them, in order to avoid any unnecessary and costly disk write and read of (R_{uv}) and (S_{uv}) . The details will be clarified below.

In the following, we will describe how to estimate the overload of the unexpected matching tuples in P_u . As shown in the previous subsection, it was estimated that it would take $T_{est}^*(R_{uv}, S_{uv})$ time to read and probe all the tuples of (S_{uv}) and write the matching result tuples. However, based on $T_{meas}^*(R_{uv}, \Delta S_{uv})$, for the overloaded processor P_u , we know that this time is mis-estimated and it will take more time. Let us take:

- $T_{dyn}(R_{uv}, S_{uv})$: new re-estimated time to read (R_{uv}) and build the hash table, to read (S_{uv}) and probe the hash table, and to write the matching result tuples, based on $T_{meas}^*(R_{uv}, \Delta S_{uv})$;
- $T_{dyn}^*(R_{uv}, S_{uv}) = T_{dyn}(R_{uv}, S_{uv}) - T_{est}(R_{uv}, S_{uv})$: new re-estimated time to read (S_{uv}) and probe the hash table, and to write the matching result tuples.

Therefore, we can calculate:

$$T_{dyn}^*(R_{uv}, S_{uv}) = T_{meas}^*(R_{uv}, \Delta S_{uv}) \cdot \frac{Size(S_{uv})}{Size(\Delta S_{uv})}$$

It was estimated that it would take $T_{est}^*(R_{uv}, S_{uv}) - T_{est}(S_{uv})$ to probe the hash table and write the matching tuples, but in fact it will take $T_{dyn}^*(R_{uv}, S_{uv}) - T_{est}(S_{uv})$, because the number of matching tuples is larger than the estimated. In order to denote the average number of matching tuples generated per tuple of the probing partition (S_{uv}) , we define the blow-up ratio in the join of a pair of partitions (R_{uv}, S_{uv}) , $B(R_{uv}, S_{uv})$, as the ratio of the time spent to probe the hash table and to write the matching result tuples of (R_{uv}, S_{uv}) , to the time to read the probing partition (S_{uv}) , that is:

$$B(R_{uv}, S_{uv}) = \frac{T_{dyn}^*(R_{uv}, S_{uv}) - T_{est}(S_{uv})}{T_{est}(S_{uv})}$$

Because the blow-up ratio is larger than the estimation, it will take $(T_{dyn}(R_{uv}, S_{uv}) - T_{est}(R_{uv}, S_{uv}))$ more time than it was estimated by the scheduling phase to join the pair of partitions (R_{uv}, S_{uv}) . In fact, considering the many accumulated deviations until the processing of (R_{uv}, S_{uv}) , it becomes:

$$T_{dyn}(R_{uv}, S_{uv}) - T_{est}(R_{uv}, S_{uv}) + \sum_{j=1}^{v-1} dev(R_{uj}, S_{uj})$$

We define the matching tuples that are produced and written in this interval of time as the overload to be migrated in the processing of (R_{uv}, S_{uv}) , $M(R_{uv}, S_{uv})$, whose size is given by:

$$\begin{aligned} \text{Size}(M(R_{uv}, S_{uv})) &= \text{Size}(\Delta S_{uv}) \cdot B(R_{uv}, S_{uv}) \cdot \\ \frac{T_{\text{dyn}}(R_{uv}, S_{uv}) - T_{\text{est}}(R_{uv}, S_{uv}) + \sum_{j=1}^{v-1} \text{dev}(R_{uj}, S_{uj})}{T_{\text{meas}}^*(R_{uv}, \Delta S_{uv}) - T_{\text{est}}(\Delta S_{uv})} \end{aligned}$$

Therefore, the load to migrate $M(R_{uv}, S_{uv})$ denotes the matching tuples that were not expected to be generated, but because of a blow-up ratio $B(R_{uv}, S_{uv})$ higher than the estimation, are in fact produced. In the following we will define the condition necessary for this unexpected overload to be equally migrated among all processors.

3.4 Condition for Ideal Migration

We estimated the size of the overload to migrate $M(R_{uv}, S_{uv})$ above. If we can migrate $(N-1)/N$ of this overload to the non-overloaded processors ideally, when will the processing of the mis-estimated partition finish? Before discussing how to migrate the load ideally (which will be described in the following subsection), here we will first estimate the time it takes for an overloaded processor P_u to execute (R_{uv}, S_{uv}) , for the ideal case its overload can be equally balanced among all processors.

If the overload of P_u is equally distributed among all the processors, that is, in the ideal case each processor P_i ($i \neq u$) consumes $1/N$ of the overload $M(R_{uv}, S_{uv})$, the time for the overloaded processor P_u to terminate the processing of (R_{uv}, S_{uv}) is $T_{\text{term}}(R_{uv}, S_{uv})$, which can be expressed as:

$$\begin{aligned} T_{\text{term}}(R_{uv}, S_{uv}) &= T_{\text{est}}(R_{uv}, S_{uv}) + \\ \frac{1}{N} \cdot (T_{\text{dyn}}(R_{uv}, S_{uv}) - T_{\text{est}}(R_{uv}, S_{uv}) + \sum_{j=1}^{v-1} \text{dev}(R_{uj}, S_{uj})) \end{aligned}$$

Now, is this ideal case of equal distribution of the overload $M(R_{uv}, S_{uv})$ among the N processors always possible? In the following, we express the condition necessary to attain this ideal migration.

$$\Delta t_u + \frac{N-1}{N} \cdot T_{\text{est}}\left(\frac{M(R_{uv}, S_{uv})}{B(R_{uv}, S_{uv})}\right) \leq T_{\text{term}}(R_{uv}, S_{uv}) \quad (3)$$

This means that from time t_{2u} , that is, from the time the processing of (R_{uv}, S_{uv}) is detected to be overloaded, P_u needs at least the period of time to generate $(N-1)/N$ of the overload $M(R_{uv}, S_{uv})$, that is, the time to read pages of (S_{uv}) whose number of matching tuples is $(N-1)/N$ of the overload $M(R_{uv}, S_{uv})$. If the overload detection is too late, P_u cannot migrate its overload $M(R_{uv}, S_{uv})$ successfully among all the processors P_i . In such a case, the difference in time between the left-hand side and right-hand side of equation (3) represents the unresolved load imbalance, and is counted in $\text{dev}(R_{uv}, S_{uv})$. If this deviation $\text{dev}(R_{uv}, S_{uv})$ is large enough, it is detected by the equation (2) described in Subsection 3.2.

The second term at the left-hand side in (3) is usually small when the blow-up ratio $B(R_{uv}, S_{uv})$ is relatively high. So if the blow-up ratio $B(R_{uv}, S_{uv})$ is estimated to be very small, we should pay a bit more attention in the determination of (Δt_u) so that we can avoid such a failure

for most of the cases. In order to simplify the problem, hereafter we assume the condition (3) is attained and the overload can be migrated ideally among all processors.

3.5 How to Migrate the Overload

We can consider two ways a processor P_i ($i \neq u$) has the overload, that is, $1/N$ of $M(R_{uv}, S_{uv})$ migrated from the overloaded processor P_u . Details of both cases, called *result redistribution* and *processing task migration*, respectively, are presented below.

3.5.1 Result Redistribution

Let us assume the case processor P_u is overloaded, and any of the $(N-1)$ processors P_i ($i \neq u$) that is processing a pair of partitions (R_{ik_i}, S_{ik_i}) has its memory filled by the hash table of (R_{ik_i}) and thus, has no more memory space to load any data from the overloaded processor P_u . In this case, P_i only receives the result matching tuples of $1/N$ of $M(R_{uv}, S_{uv})$ from P_u and thus, contributes by only writing them in its own disk. While there is no result data sent by P_u , P_i continues its own join of (R_{ik_i}, S_{ik_i}) . However, when some result data are received from P_u , they are written to disk with higher priority than its own (R_{ik_i}, S_{ik_i}) results.

In the case the blow-up ratio $B(R_{uv}, S_{uv})$ is high, since the join result size is usually much larger than the partition sizes $\text{Size}(R_{uv})$ and $\text{Size}(S_{uv})$, one idea is to save transfer time by compressing the result data to be transferred. For example, let us assume that the hash table of (R_{uv}) contains 3 tuples $r1, r2, r3$, and a page of (S_{uv}) contains 5 tuples $s1, s2, s3, s4, s5$ with the same join attribute value. In this case, instead of sending the 15 tuples $r1s1, r1s2, \dots, r3s5$, resulted from the join, the result could be sent in a compact form like $[(r1, r2, r3), (s1, s2, s3, s4, s5)]$, which contains only 8 tuples (whose length are usually shorter than that of the join result tuples). Of course it requires some compaction and decompaction time, which may still save some transfer time.

3.5.2 Processing Task Migration

Now, let us assume that there is processor P_u that is overloaded, and a processor P_i ($i \neq u$) that has already finished the join of its pair of partitions (R_{ik_i}, S_{ik_i}) . In this case, the memory of P_i becomes empty. Therefore, if blow-up ratio $B(R_{uv}, S_{uv}) \geq 1$, in order to alleviate the overloaded processor's CPU and the network utilization, processor P_i can receive some part of (R_{uv}) and (S_{uv}) of the overloaded processor P_u and actually compute their join locally, instead of only receiving and writing the matching tuples as is done in the result redistribution case. The maximum number of such P_i 's to have some processing task migrated from the overloaded processor P_u simultaneously is $\lfloor B(R_{uv}, S_{uv}) \rfloor$. Here, we call this migration the *processing task migration*.

Two reciprocal types of *processing task migration* from P_u to P_i are possible: the *hash table migration* that restricts the hash table size, while (S_{uv}) is broadcast to P_i ; and the *probe migration* that restricts the data flow of (S_{uv}) , while the hash table is replicated in P_i .

- (a) **Hash Table Migration:** In this case, $1/B(R_{uv}, S_{uv})$ of the hash table of the building partition (R_{uv}) in the overloaded processor P_u is migrated to the non-overloaded processor P_i , while pages of the probing partition (S_{uv}) are broadcast to P_i . This broadcast of (S_{uv}) is done until $1/N$ of $M(R_{uv}, S_{uv})$ is migrated from P_u to P_i . The restriction on the size of the hash table received by P_i to $1/B(R_{uv}, S_{uv})$ of $Size(R_{uv})$ means that the disk bandwidth of P_i is fully utilized when (S_{uv}) is broadcast.

Concerning the memory utilization, this case of *hash table migration* has the great advantage that the memory of P_u and P_i are not fully utilized and thus, it can be used to buffer output result tuples.

- (b) **Probe Migration:** In this case, the hash table of the building partition (R_{uv}) assigned to the overloaded processor P_u is simply replicated in the non-overloaded processor P_i , while only $1/B(R_{uv}, S_{uv})$ of the pages of the probing partition (S_{uv}) read in P_u are sent to P_i . This migration of (S_{uv}) is done until $1/N$ of $M(R_{uv}, S_{uv})$ is migrated from P_u to P_i . This restriction on the data stream of (S_{uv}) received by P_i to $1/B(R_{uv}, S_{uv})$ of (S_{uv}) read in P_u means that the disk bandwidth of P_i is fully utilized.

Concerning the memory utilization, compared to the case of *hash table migration*, it has the disadvantage that the memory is fully utilized because the whole hash table of partition (R_{uv}) must be copied in P_i . Therefore, there is no space left to buffer output results in processor P_u neither in P_i .

3.6 Calibration of Measured Time

Returning to equation (1) in Subsection 3.2, it contains the term $T_{meas}(R_{uv}, S_{uv})$. Let us suppose a non-overloaded processor P_u is executing its own join processing of (R_{uv}, S_{uv}) and also receiving the overload $M(R_{xz}, S_{xz})$ from an overloaded processor P_x . Therefore, $T_{meas}(R_{uv}, S_{uv})$ must be calibrated since P_u spent some time for the migrated load $M(R_{xz}, S_{xz})$, besides the time for its own processing of (R_{uv}, S_{uv}). This calibration can be done by excluding the time spent for $(\Delta M(R_{xz}, S_{xz}))$, that is, the amount of $M(R_{xz}, S_{xz})$ that P_u has already processed concurrently with the processing of (R_{uv}, S_{uv}). Moreover, $T_{meas}(R_{uv}, S_{uv})$ is calibrated by excluding the time given by:

$$\frac{1}{N} \cdot \frac{Size(\Delta M(R_{xz}, S_{xz}))}{Size(M(R_{xz}, S_{xz}))} \cdot (T_{dyn}(R_{xz}, S_{xz}) - T_{est}(R_{xz}, S_{xz}) + \sum_{j=1}^{z-1} dev(R_{xj}, S_{xj}))$$

Although we omit details here, the calibration of $T_{meas}(R_{uv}, \Delta S_{uv})$ can be done in the same way.

4 Description of Our Dynamic Skew Handling Algorithm

Our dynamic join product skew handling algorithm uses the fundamentals introduced in the previous section to de-

tect and migrate overloads. In the following we describe how the balance is dealt by the coordinator processor and the processors' load is balanced at run-time.

4.1 Basic Description

As we have illustrated in Section 2, first, in a scan/sampling phase, like in the static join product skew handling algorithms [11],[2], a coordinator processor gathers the statistical profile of the join attribute values in the operand relations. Then, in the scheduling phase, based on these statistics, the coordinator estimates the partitions' join execution times and schedules the partitions to the processors, so that the execution times are almost equal for all processors. This scheduling information is sent from the coordinator to all the processors and thus, in the redistribution phase, data are exchanged between the processors and the partitions are locally assembled. Finally, in the join phase, each processor reads its local pairs of partitions and joins them on its own site.

In our approach, during the join phase, the coordinator monitors the partitions' join by gathering information from partial processing statistics, to ensure that their costs match the predictions in the scheduling phase. Here, we assume that most of the estimations used in the scheduling phase are correct, and only in a few peculiar cases they are detected to be mismatching. When the coordinator wants to check the processors' load, it sends a signal to them. Upon receiving the signal, each processor P_i ($i = 1, \dots, N$), sends back the current amount of the probing partition (S_{ij}) processed, that is, (ΔS_{ij}) . Receiving the (ΔS_{ij}) from the processors, the coordinator calculates their average processing rate and compare it to the estimated in the scheduling phase, as given by equations (1) and (2) in Subsection 3.2. In case a processor P_u is detected to be overloaded, the coordinator schedules the other more lightly loaded processors to work on the task of P_u , so that its overload $M(R_{uv}, S_{uv})$ is equally migrated to all the lightly loaded processors. First, the overloaded processor P_u is scheduled to do *result redistribution* to the other processors P_i ($i \neq u$), which will write the received results in their own disks, besides concurrently continuing their own join processing. Whenever any of these processors P_i finish their own join, they immediately inform the coordinator of their memory availability. If blow-up ratio $B(R_{uv}, S_{uv}) \geq 1$ is satisfied, the *processing task migration* from P_u to P_i will alleviate the overloaded processor's CPU and network resources and thus, the coordinator inform P_u and P_i to change to the *processing task migration*. Otherwise, P_i continues to receive *result redistributed* from P_u and concurrently begins the join of its next pair of partitions. The maximum number of P_i 's that can activate the *processing task migration* simultaneously is given by $\lfloor B(R_{uv}, S_{uv}) \rfloor$. When the migration of $1/N$ of $M(R_{uv}, S_{uv})$ is finished for processor P_i , it is reported to the coordinator and so, P_i is notified to return to the join of its own partitions only. Finally, when a processor finishes the join of all its own partitions, it reports to the coordinator of its availability, and it is assigned to process a portion of the load of any other processor still working.

In the following, we present a simplified algorithmic description of this dynamic approach.

4.2 Algorithmic Description

Let us consider the join of relations R and S in a shared-nothing database system composed of processors P_i , for $(i = 1, \dots, N)$. Based on the statistics gathered in the scan/sampling phase, in the scheduling phase, to each P_i , the coordinator allocates the pairs of partitions (R_{ij}, S_{ij}) , for $(j = 1, \dots, m_i)$. After assembling the partitions in the redistribution phase, in the join phase the processors P_i begin the join of their pair of partitions (R_{ij}, S_{ij}) . In the description below, we adopt the notation P_i^j to denote that processor P_i is (or has just finished) joining the pair of partitions (R_{ij}, S_{ij}) .

During the join phase, the coordinator keeps the status of the processors and holds them in three sets of processors, P_{proc} , P_{end} and P_{endall} , where:

- P_{proc} is the set of processors P_i^j that are joining (R_{ij}, S_{ij}) ;
- P_{end} is the set of processors P_i^j that have just notified the coordinator they have finished the join of (R_{ij}, S_{ij}) ; and
- P_{endall} is the set of processors $P_i^{m_i}$ that have finished the join of the last (R_{im_i}, S_{im_i}) .

For each processor P_i^j , the coordinator has a $timer_i$, which indicates it is time to check the load of P_i^j . Therefore, whenever a processor P_i^j begins the join of a new pair of partitions (R_{ij}, S_{ij}) , the coordinator keeps this time in t_{1i} and also sets the $timer_i$ to send an interruption after an appropriate interval of time (Δt_i) . In the description below, we use the following notation:

- $\text{set}(t_{1i})$ registers the time P_i^j begins the join of (R_{ij}, S_{ij}) ;
- $\text{set}(timer_i)$ sets $timer_i$ to $t_{1i} + (\Delta t_i)$; and
- $\text{reset}(timer_i)$ resets it.

The processors interrupt the coordinator in two cases, that is, there are two types of interruption from the processors to the coordinator:

- when a processor P_x^z finishes the join of (R_{xz}, S_{xz}) ; and
- when a processor P_x^z finishes the receiving of $1/N$ of $M(R_u^v, S_u^v)$ from an overloaded processor P_u^v .

In the following we present the algorithmic description of how, during the join phase, the coordinator deals the load balancing at run-time.

____coordinator_main()____

Inform each P_i^1 to begin its 1st. partition join (for $i = 1, \dots, N$);
 $P_{proc} = P_i^1$, $\text{set}(t_{1i})$, and $\text{set}(timer_i)$ (for $i = 1, \dots, N$);

In case of interruption from :

- {case 1:} $timer_x$, call $\text{check_load}(timer_x)$;
- {case 2:} processor P_x^z which finished the join of its own partition (R_{xz}, S_{xz}) , call $\text{end}(P_x^z)$;
- {case 3:} processor P_x^z which finished the receiving of $1/N$ of $M(R_u^v, S_u^v)$ from an overloaded processor P_u^v , call $\text{endadj}(P_x^z, P_u^v)$;

____check_load(timer_x)____

$\text{reset}(timer_x)$;

Asks to P_x^z for $\text{Size}(\Delta S_{xz})$ (and $\text{Size}(\Delta M(R_u^v, S_u^v))$, if there is any) processed during Δt_x ;

Based on received information, check the load of P_x^z (by conditions (1) or (2) in Subsection 3.2);

If P_x^z is detected to be overloaded

Estimate the overload $M(R_{xz}, S_{xz})$;

$maxtpm = \lfloor B(R_{uv}, S_{uv}) \rfloor$;

Inform P_x^z to do *result redistribution* to each P_i^j in P_{proc} ;

____end(P_x^z)____

If $(j < m_x)$ /* there is next partition to join */

Then $P_{proc} = P_{proc} - \{P_x^z\}$;

If (there is an overloaded processor P_u^v making *result redistribution* to P_x^z) and $(maxtpm \geq 1)$

Then $P_{end} = P_{end} + \{P_x^z\}$;

Change *result redistribution* to *processing task migration* from P_u^v to P_x^z ;

$maxtpm - -$;

Else $z + +$;

Inform P_x^z to begin its z -th. partition join;

$P_{proc} = P_{proc} + \{P_x^z\}$, $\text{set}(t_{1x})$ and

$\text{set}(timer_x)$;

Else $P_{end} = P_{end} - \{P_x^z\}$;

$P_{endall} = P_{endall} + \{P_x^z\}$; /* it was the last partition to join */

If (there is an overloaded processor P_u^v making *result redistribution* to P_x^z) and $(maxtpm \geq 1)$

Then Change *result redistribution* to *processing task migration* from P_u^v to P_x^z ;

$maxtpm - -$;

Else If $P_{proc} \neq \emptyset$

Then Inform one processor in P_{proc} to do *result redistribution* to P_x^z ;

Else **END**($R \bowtie S$);

____endadj(P_x^z, P_u^v)____

{case 1: $P_x^z \in P_{proc}$ }

/* P_x^z was having *result redistribution* from P_u^v , and was joining its pair of partition (R_{xz}, S_{xz}) concurrently */

Inform P_u^v to stop doing *result redistribution* to P_x^z ;

{case 2: $P_x^z \in P_{end}$ }

/* P_x^z was having *processing task migration* from P_u^v , and it has its own next partition to join */

$maxtpm + +$;

$P_{end} = P_{end} - \{P_x^z\}$;

$z + +$;

Inform P_x^z to begin its z -th. partition join;

$P_{proc} = P_{proc} + \{P_x^z\}$, $\text{set}(t_{1x})$ and $\text{set}(timer_x)$;

{case 3: $P_x^z \in P_{endall}$ }

/* P_x^z was having *processing task migration* (or *result redistribution*) from P_u^v , and it has no more partition to join */

Inform P_u^v to remain doing *processing task migration* (or *result redistribution*) to P_x^z , until P_u^v finishes the join of (R_{uv}, S_{uv}) ;

5 Evaluation Results

In this section we examine some preliminary evaluation results of our dynamic join product skew handling algorithm. In our evaluation, we adopted the data skew model used in [9] and [2] where, for a relation of size $|R|$, in each attribute the value 1 appears in some fixed number

of tuples, while the remaining tuples contain values uniformly distributed between 2 and $|R|$. For example, the x10 attribute has the value 1 appearing in exactly ten tuples, while the remaining $|R| - 10$ tuples contain values between 2 and $|R|$. Table 1 describes various operations times used in our evaluation, which are based on the Gamma system configuration and that are also used in [10]; and Table 2 summarizes the workload parameter values, which are the same as those in [2]. Note that these configuration parameters present lower physical resources capacity and database sizes than those expected for the next-generation parallel servers. However, we adopted them here in order to compare our simulation results with those measured in the Gamma prototype system in [2], and to validate our simulator. Note that our purpose here is not to present absolute join execution times but the relative improvement resulted from our dynamic skew handling algorithm when unexpected join product skews occur.

read / write a disk page	20 msec
hash tuple	3 μ sec
send message	5 msec
probe hash table	6 μ sec
receive message	5 msec
join output tuple	40 μ sec
check one processor overload	5 msec

Table 1: System Characteristics

cardinality of R	500,000 tuples
cardinality of S	500,000 tuples
tuple length of R	208 Bytes
tuple length of S	208 Bytes
tuple length of result (R,S)	416 Bytes
memory capacity	8 MBytes
number of nodes	30
page size	8 KBytes
message length	8 KBytes

Table 2: Workload Characteristics

In the following, we present some results for joins on the pair of relations R and S, varying the level of skew in both relations, yet keeping the result size constant. We compare the performance of our dynamic join product skew handling algorithm with VP-RR, the static join product skew handling algorithm which shows the best performance results in [2]. In the VP-RR algorithm, tuples are partitioned in fine virtual partitions, and are redistributed to processors using round-robin allocation. Like in [2], we used 60 virtual processors per processor for the 500,000 tuples relations. In this preliminary evaluation we adopted (Δt_i) as $1/3$ of $T_{est}^*(R_{ij}, S_{ij})$. However, its optimal value has to be carefully determined and requires further extensive empirical analysis. Here we present the execution time results for the redistribution and join phases, which are the representative components of the total execution time; the sampling and scheduling phase's times, which are almost the same for both algorithms, are not included in the presented times.

The three joins j1, j2 and j3 in Table 3 are the same as those used in [2], and their join attributes are (x10K for R

and x10 for S), (x1K for R and x100 for S) and (x100 for R and x1K for S), respectively. All the three join results contain about 600 K tuples, 100K of which are due to joining tuples that contain 1's in the join attribute. In Table 3, besides the simulation results for our dynamic approach and for VP-RR, we also include the implementation results for VP-RR, reproduced from [2]. We can observe that despite the difference between the absolute values of our simulation times and the measured times in the Gamma database machine, they are close enough to validate our simulator.

In Table 3, for (x10K $\bowtie$ x10) join, the VP-RR algorithm generates enough virtual partitions so that the 1's are mapped to all processors. For the (x1K $\bowtie$ x100) join, because the number of 1's decreases for the building relation, only 4 of the 30 processors receive the virtual partitions containing 1's and we can see that the performance is degraded compared to the previous join. Finally, for (x100 $\bowtie$ x1K) join, the VP-RR algorithm fails to distribute the 1's and they are concentrated in only 1 of the 30 processors. On the other hand, we can see that our dynamic skew handling approach is successful in detecting and redistributing the overload among all the 30 processors. For (x10K $\bowtie$ x10) join, because all processors receive 1's, no detectable overload is found. However, because the number of 1's in 24 of the 30 processors is half of the number in the other 6 processors, they finish their partitions' joins earlier. Then, these 24 processors have the result of the other 6 processors redistributed to them, so that the writing task of the 6 processors is decreased, which contributes to the performance improvement. For the (x1K $\bowtie$ x100) join, the 4 processors containing the 1's are detected at run-time, and their overload are migrated to the other 26 processors, so that the load is rebalanced dynamically. Finally, for (x100 $\bowtie$ x1K) join, the single overloaded processor migrates its overload to all the other 29 processors so that the balance of the system is achieved. For the three skew cases in Table 3, we can see that the overhead of the dynamic load balance is greatly compensated by the performance improvement achieved by the load-rebalance among the 30 processors. Although these result times are for the system configuration presented in Tables 1 and 2, we believe that our dynamic approach works well even under the recent massively parallel machines which employ high speed RISC processors and high bandwidth RAID system.

time(s)	j1	j2	j3
VP-RR (from [2])	49.7	85.8	151.0
VP-RR	39.9	63.5	139.9
Our Approach	38.9	39.3	40.4

Table 3: Execution Times when Result Size is about 600 Ktuples

For the four joins j4, j5, j6 and j7 in Table 4, the join attributes are (x100K for R and x10 for S), (x10K for R and x100 for S), (x1K for R and x1K for S) and (x100 for R and x10K for S), respectively. The data skew in these experiments is much larger than in the previous ones, and all the four join results contain about 1.5 M tuples, 1 M of which are due to joining tuples that contain 1's in the

join attribute. The table shows that our approach outperforms the VP-RR in all cases by dynamically rebalancing the processors overloads. In the VP-RR, for $(x100K \bowtie x10)$ and $(x10K \bowtie x100)$ joins, the 1's are mapped to all the 30 processors; for the $(x1K \bowtie x1K)$ join, the 1's are concentrated in 4 processors, and for $(x100 \bowtie x1K)$ join, they are concentrated in only 1 of the 30 processors. Because the result relations contain 1.5 M tuples and are much larger than the results for Table 3, we can see that the fail in distributing the 1's equally among the 30 processors has a much worse effect for the VP-RR. Concerning our approach, for $(x100K \bowtie x10)$ join, no detectable overload is found. However, because one of the 30 processors has more 1's than the others, it takes longer time to finish its partitions join. Therefore, it redistributes its results to the other 29 processors as soon as they finish their own processing. For the $(x10K \bowtie x100)$ join, although all the 30 processors receive 1's, 6 of them have more 1's and thus, they are detected to be overloaded and have their overload migrated to the other 24 processors first by result redistribution, and when they finish their own partitions join, by processing task migration. For the $(x1K \bowtie x1K)$ join, the 4 processors containing the 1's are detected at run-time, and their overload are migrated to the other 26 processors by result redistribution, and then by processing task migration. Finally, for $(x100 \bowtie x10K)$ join, the single processor containing all the 1's is detected as overloaded at run-time, and migrates its overload to the other 29 processors first by result redistribution and then, by processing task migration. As shown in Table 4, our dynamic algorithm detects the load imbalance and redistributes the overload among all the 30 processors, resulting in an almost constant time for the processing of any of the four skewed joins.

time(s)	j4	j5	j6	j7
VP-RR	68.8	92.4	325.9	1087.0
Our Approach	66.7	71.7	72.0	71.8

Table 4: Execution Times when Result Size is about 1.5 Mtuples

6 Conclusion

In this paper we propose a dynamic join product skew handling algorithm for parallel hash-joins in shared-nothing database systems, which rebalances the load during the join phase when unpredicted join product skews are detected at run-time. Basically, if the measured processing rate of a partition is too large compared to the one statically predicted at the scheduling phase, the overload caused by the unexpected join product skew is calculated and the corresponding production of skewed output tuples is dynamically migrated from the overloaded processor to other non-overloaded processors, while the reading of the building and probing partitions are stuck at the overloaded processor. Note that this approach differs from those like [4], which attempt to balance the data load prior to the join operation by physically relocating the excess partitions from the overloaded processor to the non-overloaded ones, re-

sulting in a high cost to move the partitions and then, an unnecessary and costly write and read of those partitions in the non-overloaded processors' disks. In order to illustrate the effects of our dynamic approach, we present the results of a preliminary evaluation based on simulation. Although infant and requiring further investigation, our evaluation could show the feasibility of the dynamic load balancing approach in resolving load imbalances due to join product skew, even when the static scheduler fails in eliminating it.

Some interesting open questions remain to be addressed in a future work. Firstly, the threshold value of α and β have to be more analyzed; their optimal value depends on the tradeoff between overhead and skew tolerance, which also depends upon the system configuration. Secondly, in this work we assume that most of the re-estimations are precise since their costs are detected at run-time, and small imprecision are absorbed by the deviations; however, re-estimations might have some larger errors, which we intend to examine in further research. Finally, the proposed algorithm depends on several parameters which are heavily dependent on its implementation details; thus, we plan to verify the proposed dynamic join product skew handling algorithm in a prototype parallel database system.

References

- [1] D. DeWitt et. al, "Implementation Techniques for Main Memory Database Systems", Proc. 1984 ACM-SIGMOD Conf., pp.1-8, 1984
- [2] D. DeWitt, J. F. Naughton, D. A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins", Proc. 18th. Int. Conf. VLDB, pp.27-40, 1992
- [3] R. Graham, "Bound on Multiprocessing Timing Anomalies", SIAM Journal of Computing, vol. 17, pp.416-429, 1969
- [4] K. A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", Proc. 17th. Int. Conf. VLDB, pp.525-535, 1991
- [5] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash : A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", Proc. 16th. Int. Conf. VLDB, pp.210-221, 1990
- [6] M. Kitsuregawa, H. Tanaka and T. Motooka, "Application of Hash to Database Machine and Its Architecture", New Generation Computing, 1, 1, pp.63-74, 1983
- [7] M. S. Lakshimi and P. S. Yu, "Effectiveness of Parallel Joins", IEEE TKDE, 2, 4, pp. 410-424, 1990
- [8] A. Shatdal and J. F. Naughton, "Using Shared Virtual Memory for Parallel Join Processing", Proc. 1993 ACM-SIGMOD Conf., pp.119-128, 1993
- [9] E. Omiecinski, "Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared Memory Multiprocessor", Proc. 17th. Int. Conf. VLDB, pp.375-385, 1991
- [10] C. B. Walton, A. G. Dale and R. M. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", Proc. 17th. Int. Conf. VLDB, pp.537-548, 1991
- [11] J. L. Wolf, D. M. Dias, P. S. Yu and J. Turek, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew", Proc. 7th. Int. Conf. DE, pp.200-209, 1991