A Study of Workload Balancing Techniques on Parallel Join Algorithms

A. Lerner S. Lifschitz*
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Departamento de Informática
Rio de Janeiro, RJ, Brasil
e-mail: {lerner,lifschitz}@inf.puc-rio.br

Abstract *When parallel join strategies are considered, good workload balancing methods are important in order to achieve reasonable performances. We study here parallel join algorithms that comprise some kind of load balancing activity. By taking into account different skew handling techniques, we discuss pros and cons of well known existing strategies and define a new taxonomy for this class of algorithms, according to the moment the load balancing technique is applied. The choice of a specific step of the algorithm to balance the load is closely related to the efficacy in handling the effects of different types of skew. Based on this idea, we introduce an alternate approach for parallel join processing, which tries to handle all skew situations by dynamically determining and assigning variable-sized tasks to be executed at each processor node.*

Keywords: parallel databases, parallel join, workload balancing, variable-sized tasks, taxonomy.

1 Introduction

The use of parallel algorithms to evaluate the relational join operator has become a key issue in obtaining acceptable response times on very large databases query processing. In [11] some parallel join algorithms have been analyzed and it has been shown that, in the presence of data skew, their performance suffered great penalties.

The problem is closely related to the data partitioning idea underlying most strategies, which always assigns tuples with the same value for their join attributes to the same buckets. Therefore, on a skewed distribution, some parallel processing nodes (PN) might be assigned heavier buckets then the others, making the split of work among sites uneven.

The concept of data skew is detailed in [13], as a distinction between the intrinsic skew (*i.e.*, Attribute Value Skew - AVS) and the partitioning skew is made. While the first one depends only on the data itself, the latter may vary according to the particular join implementation being used.

In the case of AVS, load balancing might be necessary as uniform distributions of join attribute values are unusual in practice. With respect to partitioning skew, the authors in [13] further divide it into four different classes: the Tuple Placement Skew (TPS), as the initial placement of tuples varies among the PNs; the Selectivity Skew (SS), when selection predicates applied before the actual join generates distinct sized fragments on the PNs; the Redistribution Skew (RS), due to a tuple redistribution method that fails in creating equally sized partitions on each PN; and the Join Product Skew (JPS), that happens when the number of resulting tuples generated at each PN is uneven.

There are many parallel join strategies based on combinations of partitioning and load balancing steps (*e.g.*, [2, 3, 4, 14]). Each strategy

*Supported in part by CNPq under grant 300048/94-7.

applies its load balancing technique in a particular moment, always aiming at reducing the problems caused by some kind of skew, usually AVS, RS and/or JPS.

In this paper, we study some of the most important solutions to the skew handling problem on the parallel execution of the join operator - specially on message-passing (also known as shared-nothing) systems - proposed so far. Besides those suggested on [13], other skew situations are also considered: that of a heterogeneous parallel system or of a strongly concurrent multi-user environment. Based on this study, we classify the selected algorithms according to the moment their load balancing technique is applied. This new taxonomy helps us explain some of the drawbacks of the known parallel join methods and is the main motivation to a new strategy proposal, well adapted to different skew scenarios, that we introduce here.

The paper is organized as follows: in Section 2, selected parallel join algorithms are discussed, emphasis on workload balancing issues and skew handling. Next, in Section 3 a taxonomy for load-balanced parallel join strategies is given and an alternate strategy, oriented to a shared-nothing system, is proposed in Section 4. Final comments, ongoing and future work are found in Section 5.

2 Related Work

In this section, we are going to present some of the most relevant works that address the load balancing issue on the parallel execution of the join. It is worth mentioning that most of these works focus on skew effects related to the join operator itself, such as RS and JPS, defined above. TPS and SS partitioning skews are only considered as more complete queries (involving more operators than just the join) are studied.

Beside the ones described in [13], there are two skew situations, not always taken into account. The first one, named Heterogeneity Skew (HS), happens when the parallel processing environment have nodes of different capacities. The second one, named Concurrency Skew (CC), occurs when some PN experience concurrency of resources in a multi-user environment (external processes or multiple transactions). The existing algorithms not always consider both issues, as it is usually assumed (either explicitly or not) that the underlying platform is a homogeneous and single-user environment - clearly, not a valid assumptions in practice.

The work in [4] presents the *partition tuning* technique, shown to be effective on RS skew handling. An example algorithm based on this approach is the Tuple Interleaving Parallel Hash Join (TIJ). The algorithm redistributes the operand relations tuples on an early division step, so that each bucket gets its tuples horizontally interleaved among all PNs. After this step, each of the n available PNs store the n-th part of each bucket. The partition tuning technique is then applied to combine the buckets in equally sized partitions. After the computation of the partitions, they are each assigned to a distinct PN and their corresponding tuples are moved to and processed in that node.

The *incremental* technique is another strategy that has led to load-balanced parallel join algorithms. The Dynamic Balancing Hash-Join Algorithm (DBJ), based on this technique, is proposed in [14]. The DBJ analyzes the distribution of tuples as the partitioning phase goes on and adjusts it dynamically, whenever needed. Thus, even not knowing the complete distribution of data *a priori*, the algorithm uses the partial information gathered that far to estimate future distribution information and to correct uneven workload.

A very interesting technique, called *sampling*, is used in [2]. The authors claim that the overhead to execute a workload balancing technique within a parallel join strategy might be too high in mild skew conditions. It happens when the attribute value distribution of the join operand relations is relatively uniform. The rationale is not to penalize a situation free of skew using heavy skew handling algorithms.

Thus, before making any decision about load balancing (and even partitioning), the relations are sampled (about 5% to 10% at most) in order to detect AVS.

The strategy itself comprises the choice of the best suited algorithm to that particular degree of skew detected at the sampling phase. This method was shown to handle well RS skew and, to some extent, JPS. An interesting result is found in [5], where a variation of this sampling algorithm was shown in practice to be better than a partition-tuning-like algorithm and a variation of DBJ, in many different skew situations.

In [9] a different method is presented. This method is based on two new techniques: the *demand-driven* approach, which assigns a new task to a PN only when it has finished processing its last task; and a load balancing method that consists of a *task stealing* step at the latests stages of execution, when idle PNs may help overloaded PNs to finish the whole join execution by taking them part of their local tasks. In [12] similar ideas with respect to workload balancing during join processing execution are suggested, but quite different implementation mechanisms are used there. The work presented in [10] is an extension of the approach given in [9], though directed to a shared-nothing system and changed to a static task allocation policy, instead of the demand-driven one, used on their earlier work.

The work in [3] focuses specifically on handling the effects of JPS skew during the execution phase of a sampling algorithms, like the one proposed in [2]. The authors claim that the *"don't be idle when there's work left to do"* strategy, presented in [12], is a naïve load migration technique. They introduce their *run-time process migration* instead, which apply a load migration procedure at the precise moment one processor is detected to be overloaded. In this approach, the overload detected is dynamically divided among the processors so that their execution times are re-equalized.

The JPS skew has been one of the most difficult problems to deal with on workload balancing methods, as it is detected only at execution time. Even when the JPS is said to be efficiently managed, as in [3], there is always a considerable overhead added to the strategy that might limit its use in practice.

When studying all these methods, it becomes clear that there is still work to be done in order to handle the skew effects efficiently on the processing of the parallel join. Most works focus on the skew side-effects, when the imbalance is already established, and only a few show effectiveness on their workload balancing techniques.

The common approach found on the best solutions presented so far is to associate a set of techniques (e.g., sampling and run-time process migration) in a "mixed" strategy to evaluate the join operator. None of the known algorithms handle all types of skew on a single load balancing approach. Also, there is a lack of strategies comparison work so that one could choose a specific algorithm, if possible, to achieve the expected parallel performance.

Although not exhaustive, the list of papers discussed here cover a broad variety of load balancing techniques published so far. The next section tries to organize those techniques according to the exact moment their load balancing procedures take place and to analyze their success in handling different types of skew.

3 A New Taxonomy

In the previous section we have reviewed briefly some of the most important works on parallel join strategies that consider the workload balancing problem. It is quite difficult to group these strategies in classes, much more than in the monoprocessor situation, where any work lies in one of the three known classes, nested-loops, sort-merge and hash-based.

To the best of our knowledge, there is only one such effort in the sense of a classification [10]. In that work, the authors claim that it is always possible to identify, in a given strategy, three distinct phases: a data partitioning, a tasks allocation (to PNs) and a local join (at each PN).

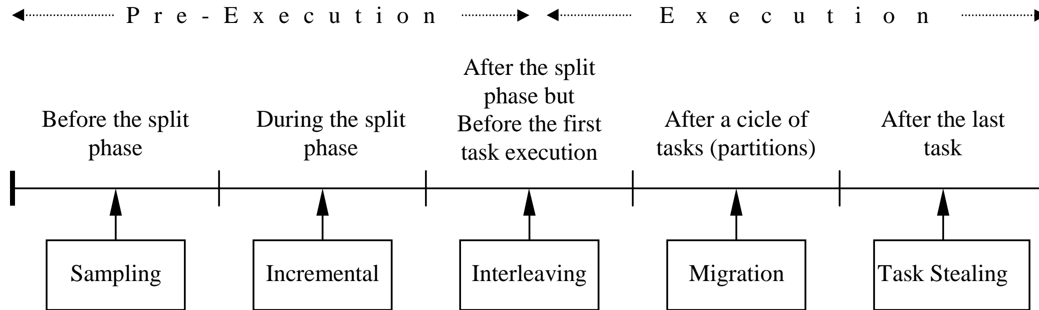

Figure 1: Algorithms and their balancing procedure time table.

In [10], the load balancing techniques are divided in two categories: static and dynamic. The first one refers to those strategies where the *tasks-to-PNs* mapping is made before the local join phase, while the latter comprises strategies which allow this mapping to be changed during the local join phase. However, we can see that workload balancing procedures occur in different phases of known algorithms (*e.g.*, before data partitioning, at later stages of execution, et cetera) and sometimes even more than just in one phase.

Actually, we can identify at least five different moments used by the known algorithms to execute balancing procedures. Therefore, we believe that a more specific classification of load balancing techniques might capture in detail those differences.

The load balancing algorithms are described in Figure 1 in terms of the moment their corresponding balancing techniques take place.

As odd as it may sound, we claim that the correct choice of that moment might be as important as the efficiency of the balancing technique itself to the success in handling a specific type of skew. Let's take for example the handling of JPS effects: no technique is known to be effective when applied in any pre-execution moment - although RS are very well handled by some techniques applied on that phase.

The question that we state here is whether one can pick the best moment for handling skewed distribution effects in a preventive way. The search for a promising moment to apply preventive load balancing techniques is strongly motivated by an intrinsic characteristic of shared-nothing systems: the cost of task movement around the PNs. In the *reactive* load balancing procedures, a tuple (or a whole set of them) might be moved around several PNs before they get actually processed. In a shared-nothing system, this movement could be expensive, as not seldom involves disk transfers (and not just message-passing). In a *preventive* scheme, the tuple should be moved only once (if so) to its destination PN.

A parallel join strategy based on this idea could oppose the techniques described so far, which focus on fixing the skew effects, whenever detected. The overhead due to partition adjustment or task stealing phases on all these algorithms is considerable. In the next section, we will discuss a proposal that takes this question in consideration and, at the same time, deals well with all kinds of skew.

4 An Alternate Strategy

We claim that the answer to the question stated in section 3 relies upon a combination of two techniques: the demand driven approach for allocating tasks, as seen in [9][1], and the dose-driven technique for sizing tasks, proposed in [8]. The main rationale of this combined approach is

- to assign a task to a PN when (and only when) it is ready to execute it; and

[1] Although [9] is based on a shared-disks environment and we remain here in the shared-nothing systems.

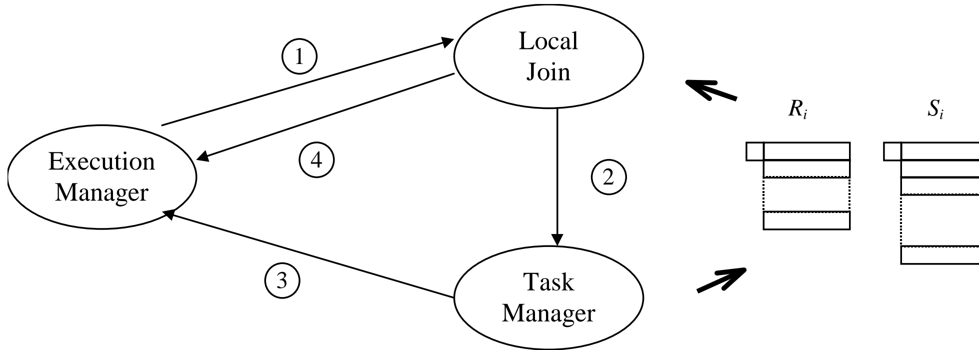

Figure 2: The VAST-PJ Architecture.

- to define the task size to be processed in a conservative *taking-the-skew-into-account* way.

We expect better parallel times, as a result of keeping processors busy (as long as there is work left to do) and sizing tasks in such a way that processors are not overloaded.

We introduce here the VAriable-Sized Tasks Parallel Join Architecture (VAST-PJ), based on this approach [6]. It distinguishes itself from related works in a number of ways. First, it is presented as an architecture, rather than an algorithm, thus allowing for different implementations of its components. Second, it doesn't fit in the *partition/allocation/execution* phases model [10], as its phases actually happen concurrently.

The VAST-PJ comprises the interaction of three components: an Execution Manager, a Task Manager and a Local Sequential Join component. Each of them have their own role in the parallel processing of the join operator.

Figure 2 illustrates the way in which the three components interact, as described next.

1. The EM initiates the process by sending a join command to LJ in a PN. The size of the task to be processed is part of this command;

2. LJ asks TM to transfer to its disk a task sized just the way EM commanded;

3. TM transfers the task tuples (R_i-S_i) to the PN that asked for it and gathers statistics information that might help EM in determining the next tasks sizes;

4. The join process takes place and a status report is sent to EM. Whenever a LJ finishes running in a PN, the whole processes starts again.

As a matter of fact, the VAST-PJ suggests a sixth moment for the execution of balancing procedures in Figure 1. This moment stands between those of Interleaving and Migration techniques. The VAST-PJ architecture will try to balance the workload just before the execution of a task, by simply sizing it in way not to overload the PN.

It is of main importance to notice how appropriate this moment is for preventive load-balancing. It is neither late, because the next task has not yet been assigned to the demanding processor, nor early, because the algorithm have already collected enough information to make an allocation decision - information that might have been collected on its initial stages or on previously executed tasks. By making a *lazy* allocation decision, the execution statistics might be of great influence. The reader may refer to [7] for a more complete discussion on effectiveness of moments for balancing procedures.

Simply stated, the greatest advantage of a preventive scheme is that no wasted tuple movements are made.

In the next three sub-sections, each of the components expected behavior is analyzed and candidate applicable algorithms are suggested.

4.1 The Execution Manager

The execution manager (EM) is responsible for task sizes determination. It should carefully consider execution and pre-execution statistics information in order to prevent a task to overload a processor.

Basically, the EM has to estimate two values: the execution finished time and next task size. The estimated finished time is the amount of time taken by the whole algorithm to completely finish execution. On one hand, the EM is not able to precisely calculate this time, despite having operand relations cardinality information, as there are unforeseen execution behavior as a result of JPS or CS skew. But, on the other hand, the EM may count on previous task executions information in order to, for example, estimate the selectivity factor of the operand relations. As more and more tasks are being executed, more information might be gathered, so at later stages of execution values such as the selectivity factor might be almost completely known.

In order to correctly size the next task, the EM considers the estimated finished time described above. It will select a size so that the execution of such a task do not take longer than the whole amount of tuples left to process. The heuristic should consider basically the possibility of having JPS skew, but is not limited to that. Actually, it may take into account any phenomena that might affect the execution time, as HS and CS skews.

The working conditions of the EM resemble those found on online scheduling algorithms [1]. On an usual scheduling problem, an assignment algorithm maps tasks (jobs) to processors (machines) knowing *a priori* all needed information about them. An online algorithm might make the assignment decisions without complete information and still maintain a good *competitiveness*, compared to off-line ones. The EM online scheduler should instead select a task, given an idle processor trying to make as good as its off-line counterpart.

4.2 The Task Manager

The task manager (TM) deals with task generation. It should be able to transfer the tuples corresponding to a specifically sized task to the PN where they will be processed, as the EM itself only determines the size, not the actual task itself.

As with all VAST-PJ components, there is no single algorithm to the TM. The candidates must balance between accuracy and overhead. The most accurate algorithms would scan through the operand relations before any actual execution in order to correctly map tuples cardinality and distribution at each PN. This would probably yield the best task selection algorithm. Needless to say, accuracy comes hand-in-hand with overhead. Sampling techniques can dramatically decrease the overhead, not completely sacrificing the accuracy of the TM algorithm.

A task generation algorithm should choose an existing task, given the EM defined size. The TM could use a best-fit heuristic to pick a task. If it does not find a good candidate task, the algorithm might try to combine smaller buckets into a bigger one in a bucket tunning approach. As a last resource, the fragment-and-replicate technique should be used to break heavy buckets into lighter ones. The application of the three techniques should make it possible to always make a good decision.

Once chosen, the tuples corresponding to a task should be transferred to the PN where they will be processed. This is not an easy task to accomplish in a shared-nothing environment, given the tuples might be horizontally spread all over the PNs of the parallel machine. Besides, this should be done just before the task execution, as moving tuples around not knowing its actual destination PN would risk moving tuples more than once - a great overhead penalty.

The suggested solution is to instantiate a

copy of the TM at each PN and to coordinate their activity in order to consistently distribute tuples to processors as the algorithm executes.

4.3 The Local Join

The local join component (LJ) actually performs the join. It executes a local (sequential) algorithm in order to join the tuples of its assigned task.

Any variation of the three known classes of sequential algorithm might be used. An optimization might be coordinated between the three components: the generation of tasks in which one of the relation tuples fit in the PN memory. Any sequential algorithm might benefit from that.

5 Final Comments, Ongoing and Future Work

The contribution of this work is twofold. First, it has organized previous related works in a new taxonomy. We have identified and analyzed different moments where the load balancing steps of these algorithms take place and have suggested a strategic one in preventing (instead of fixing) load imbalance caused by different types of skew.

Second, the VAST-PJ architecture was introduced, motivated by the evidences of the above mentioned analysis. Besides being preventive, this approach is the first, to the best of our knowledge, to suggest the use of (a) a demand-driven strategy on a shared-nothing system and (b) a dose-driven technique to size parallel join tasks.

The accommodation of different algorithms under the same solution is not completely unknown in the area. By shaping the solution as an architecture, we allow for specific configuration of algorithms to address specific scenarios.

This work is currently being extended in a number of ways. The taxonomy itself is being validated as a result of the evaluation of a more representative number of works. We plan to review not only shared-nothing (and alike) works, but also shared-memory, shared-disk and hybrid (hierarchical) systems.

The heuristic used in the next task size calculation of VAST-PJ is drawing special attention. As a matter of fact, one might think of it as a peculiar scheduling algorithm. Instead of assigning tasks (jobs) to PNs (machines), this scheduler must pick a task whenever a demanding PN asks for it. The task selection must be accomplished in a way not to compromise the whole system response time in the case of skewed distribution, therefore it sizes the task, although not knowing its exact selectivity factor (i.e., its running time). This restriction resembles those met by online scheduling algorithms and we are currently evaluating the applicability of a number of them.

Finally, an implementation of some VAST-PJ derived algorithms, proposed in [6] is being built. Our goal is to study the VAST-PJ behavior under real join scenarios (not only simulations) of a experimental parallel database management system (DBMS). As a DBMS addresses the issues of data storage and memory management in very peculiar ways, we think this kind of experiment is extremely important in order to analyze the suggested algorithms. Besides, an effective measure of the overhead, that of the whole algorithm, and particularly the one involved on the demand-driven generation of tasks, could only be met by such an experiment.

This paper is the first one to introduce the VAST-PJ, but there is still work to be done. In the near future, we intend to devote a specific paper to discuss the detailed architecture and to present its practical results.

As mentioned earlier, the area lacks comparison reports between the different suggested strategies. Such a report would help gathering all the research effort together and would be of great assistance in establishing use profiles for each algorithm.

References

[1] S. Albers. Competitive Online Algorithms. *Optima Mathematical Programming Society Newsletter*, 54:1–8, 1997.

[2] D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Procs. 18th Intl Conf. on VLDB*, pages 27–40, 1992.

[3] L. Harada and M. Kisturegawa. Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems. In *Procs. 4th Intl. Conf. on Database Systems for Advanced Applications*, pages 246–255, 1995.

[4] K.A. Hua, C. Lee, and C.M. Hua. Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tunning. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):968–983, 1995.

[5] K.A. Hua, W. Tavanapong, and H.C. Young. A Performance Evaluation of Load Balancing Techniques for Join Operations on Multicomputer Database Systems. In *Procs. Intl Conf. on Data Engineering*, pages 44–51, 1995.

[6] A. Lerner. An Architecture for Handling Load Balanced Parallel Joins in Shared Nothing Systems. Master's thesis, PUC-Rio Departamento de Informática, 1998. (in portuguese).

[7] A. Lerner and S. Lifschitz. Towards a taxonomy for Load Balancing Techniques on Shared-Nothing Parallel Join Algorithms. Technical report, Pontifícia Universisdade Católica do Rio de Janeiro, Computer Science Department, MCC no. 28, 1998.

[8] S. Lifschitz, A. Plastino, and C.C. Ribeiro. Exploring Load Balancing in Parallel Processing of Recursive Queries. In *Procs. 3rd Intl Euro-Par Conf*, pages 1125–1129, 1997.

[9] H. Lu and K-L. Tan. Dynamic and Load-Balanced Task Oriented Database Query Processing in Parallel Systems. In *Procs. 3rd Intl Conf. EDBT*, pages 357–372, 1992.

[10] H. Lu and K-L. Tan. Load-Balanced Join Processing in Shared-Nothing Systems. *Journal of Parallel and Distributed Computing*, 23:382–398, 1994.

[11] D.A. Schneider and D.J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. In *Procs ACM SIGMOD Intl Conf*, pages 110–121, 1989.

[12] A. Shatdal and J.F. Naughton. Using Shared Virtual Memory for Parallel Join Processing. In *Procs ACM SIGMOD Intl Conf*, pages 119–128, 1993.

[13] C.B. Walton, A.G. Dale, and R.M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *Procs. 17th Intl Conf on VLDB*, pages 537–548, 1991.

[14] X. Zhao, R.G. Johnson, and N.J. Martin. DBJ - A Dynamic Balancing Hash Join Algorithm in Multiprocessor Database Systems. *Information Systems*, 19(1):89–100, 1994.