

# An Online Approach for Parallel Join Algorithms Analysis\*

Alberto Lerner

Sérgio Lifschitz

Marcus V. Poggi

Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

Dept de Informática, Rio de Janeiro, RJ 22453-900 - Brasil

Tel.(55)(21)529-9463 Fax.(55)(21)511-5645

e-mail: {lerner,lifschitz,poggi}@inf.puc-rio.br

\* \*

PUC-RioInf.MCC01/99, February, 1999

**Abstract.** In this work, we propose a methodology for parallel join algorithms evaluation based on competitive analysis. Such algorithms may be seen as online ones, as load balancing decisions are often made without precise knowledge of important input parameters. We identify three different instances of online problems in the join operation and propose a metric for quality of solution assessment valid for them. The metric is independent of algorithm, parallel architecture and load balancing technique and is the corner stone of the proposed method. Moreover, we suggest, as part of the competitive analysis work, a taxonomy for adversaries used on worst and average case analysis.

**Keywords.** parallel databases, parallel join, workload balancing, competitive analysis, online algorithms

**Resumo.** Neste trabalho é proposto um método para análise de algoritmos paralelos de junção baseado em análise competitiva. Tais algoritmos podem ser vistos como online, já que tomam decisões relativas ao balanceamento de carga sem necessariamente ter informações precisas para tal. Três diferentes instâncias do problema são apresentadas, bem como uma métrica para a avaliação de qualidade de solução por elas geradas. A métrica é independente de algoritmo, da arquitetura paralela ou da técnica de balanceamento de carga em questão e é a pedra fundamental do método proposto. O trabalho sugere também uma taxonomia de adversários a serem utilizados na análise competitiva destes algoritmos.

**Palavras-Chave.** SGBD paralelos, junção paralela, balanceamento de carga, análise competitiva, algoritmos online

---

\* This work was partially supported by CNPq

## 1 Introduction

Since the introduction of parallel processing architectures, there has been a great deal of effort on the development of Database Management Systems (DBMS) that can benefit from the alleged promises of speed-up and scale-up in processing power [3]. As in monoprocessor systems, the join operator has, again, gained much attention, due to its high cost and extreme importance in the relational model. The understanding of the monoprocessor scenario and how join algorithms directed towards it are deployed, will be of great assistance in introducing the problem of join algorithms analysis on the parallel case.

A number of monoprocessor algorithms are well known for the join operation, and they are mainly variation of three techniques: nested-loops, sort-merge, and hashed-based [8]. Instead of computing the cartesian product and thereafter selecting tuple pairs which meet the join criterion, monoprocessor join algorithms often try to minimize the number of tuple comparisons in order to find the resulting relation. The minimization of comparisons is attained by taking advantage of some properties of the input relation (*e.g.*, sorted or near-sorted order over join attribute(s), uniformity of data distribution, auxiliary search structures existence, etc). Depending on which properties a given input operand relation pair presents, a best suited algorithm can be chosen accordingly.

It is interesting to note that, as far as monoprocessor join algorithms goes, the set of all algorithms derived from the above techniques are accepted to cover the whole range of real data scenarios. Moreover, given any two algorithms, it is possible for a set of input instances, to decide which of them will perform better, based on, for example, tuple comparison ratio and complexity analysis – or even in common analytical models shared by different algorithms.

Unfortunately, this is not the case for parallel join algorithms, where a great number of solutions have also been proposed, but with no consensus evaluation methodology for analyzing and comparing such algorithms. Therefore, we are given a number of algorithms but some controversially answered questions remain, like (a) is algorithm A better the algorithm B under circumstances C ? (b) what are average and worst case response times of algorithm A under a set of circumstances C ? (c) given a "rich" set of circumstances D (*i.e.*, the set that covers a broad number of real case scenarios), do we already know a set of algorithms which, combined, attain reasonable performance in all C elements ?

It is our claim that a methodology that can delivers answers to this questions can be derived if we take parallel join algorithms to be online problems [1]. Indeed, these algorithms make a number of decisions based on incomplete or uncertain input parameters – balancing the load is chiefly the issue here –, and this is exactly what online problems are all about.

Although not very disseminated among DMBS community, a number of related problems have been deeply studied using competitive analysis techniques. One that particularly stand out is the Paging Problem [1], largely utilized in DBMS buffer cache managers. The caching in distributed databases has also been addressed via competitive analysis techniques in [13].

The parallel join operation itself had already been addressed as an online problem [11]. In that work, the problem formulation is similar to one of the three problem instances we present here (specifically, that of section 2.3), although in that work neither a metric nor competitive analysis have been suggested.

The main goal of the present work is to propose this new methodology for assessment of parallel join algorithms that can be used to answer the above questions. In this work we focus on the presentation and analysis of the proposed methodology concepts, laying ground for a forthcoming work, where examples of its application will be presented.

The rest of this paper is organized as follows. In section 2, we formulate the problem of joining two relations in parallel as an online one. Indeed, we propose three different online instances for the operation. In section 3, we propose a metric on quality of solution for the join operation. Section 4 arranges the algorithms adversary on an initial taxonomy, and describe in more detail adversaries based on data distribution non-uniformity. In Section 5 we close this work describing its future unfolding and stating its contributions and conclusions.

## 2 Online Formulation of the Parallel Join

In this section we are going to state the problem of the parallel joining as an online one. We first introduce an intuitive description of it as an online problem. We then introduce an abstract model which captures the ideas underlying every parallel join proposal and uses it to state the three different instances of online problems identified in the join operation.

### 2.1 Tasks Splitting and Combination

One may think of the join operation as consisting of multiple tasks. Each task might correspond to comparing different subsets of tuples from the two relations [11]. This way, the join of each task might be computed independently of one another. The usual way of generating such tasks is by separating tuples according to their join attribute value.

On one hand, if a generated task is too large, it can be further divided into smaller – yet still independent – tasks. For instance, in a message-passing machine, techniques such as fragment-and-replicate can be used; in a shared-memory machine, a hash table sharing and probe load transfer can be used; just to mention a few. On the other hand, a number of small tasks might as well be combined together, still maintaining its independence characteristic.

The split and combination of tasks are closely related to the load balancing activities needed during the execution of the join. The ultimate goal of an algorithm is to generate equally balanced sets of tasks. As far as every split and combining activity is considered an overhead, algorithms try to balance their load with the minimum possible number of such interventions they can.

The "onlineness" of the join operation comes from the fact that the split and combination of tasks might have to be done without perfect knowledge of all

input data needed to generate a balanced set of tasks. Furthermore, this missing information should not be completely computed, for performance sake, so not to make the load-balancing activity unbearably expensive.

As a matter of fact, a great number of works handle the join problem in what can be considered an online fashion. These algorithms first partition the input relations trying to evenly divide the tasks among the processors. This approach often succeed in balancing the cardinality of the allocated tasks (*i.e.*, work associated to the number of comparisons), mainly because cardinality is often known at this time. Such algorithms does not handle join product and concurrency skew factors among tasks effectively (*i.e.*, work associated to the tuple generating), because result selectivity and external load information are often not precisely known by the algorithms when the computation of the sets is made.

At this point, it should be intuitive that this lack of input data might lead to worse online algorithms results than offline *knowing-the-input-data-algorithms* counterparts. It should be clear as well that different strategies used in load balancing decisions might lead to different computational times.

We now introduce an abstract model for the parallel join operation and revisit the join problem as online instances of the model.

## 2.2 An Abstract Model

Let  $P$  be a set of  $p$  processing nodes on a parallel machine. Let  $S$  be the set of independent join tasks, each corresponding to the comparison of distinct sets of tuples of two operand relations,  $R_1$  and  $R_2$ .

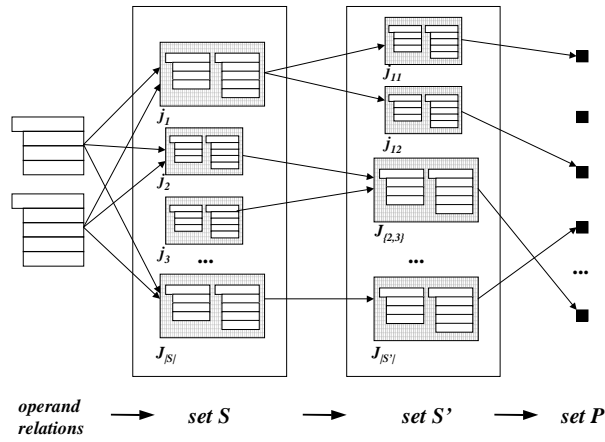
Suppose that a task of  $S$  with a large number of tuples can be split in any number of independent subtasks and suppose also that a number of tasks can be combined together into bigger ones, still maintaining their independence. For simplicity, lets extend the usage of the term task to either cases.

The set  $S'$  is the set derived from  $S$  by applying a split or a combining technique on any number of tasks. Note that the processing of  $S$  and  $S'$  yields the same resulting set of tuples – although, probably, with different computational times. The relationship between  $R_1$ ,  $R_2$ ,  $S$ ,  $S'$  and  $P$  is depicted in figure 1.

The idea behind the sets is that  $S$  represents the initial task mapping devised by an algorithm, given the input relations and all available information and estimations it has. On its turn, the set  $S'$  represents the set of tasks which were actually processed. The difference between the two sets reflects online load balance decisions taken by an algorithm.

An online load-balanced parallel join algorithm will try to build  $S'$  and the *tasks-to-processor assignment* using all available information about input relations it can gather and will take decisions about load-balancing based on this partial information. An offline load-balanced parallel join algorithm is the one who generates  $S'$  knowing exactly every necessary parameter regarding input data.

Indeed, depending on a number of assumptions, different online scenarios might be addressed. In the next section, we introduce formally three instances



**Fig. 1.** Operand relations are initially partitioned into tasks of set  $S$ . As load-balancing is taking place, tasks are further divided or combined. The actual set of tasks processed is the  $S'$ , as the result of the assignment to processor of the set P

of online parallel join problems.

### 2.3 Online Instances of the Problem

The major lack of information for the computation of the set  $S'$  arise from (a) the fact that the problem of estimation of selectivity to an acceptable degree of accuracy remains open even though it has been studied in some detail in the past [8]; (b) the incomplete knowledge of the exact sequence of tasks of  $S$ ; and (c) the existence of external application load on the PN assigned for the computation. In all cases, an online algorithm faces difficulties in making the split or combination of tasks decisions, leading to non-optimal solutions. Let us examine each case in more detail.

**Lack of Selectivity Factor Knowledge** Calculating the exact selectivity factor is almost as difficult as resolving the whole join. Thus, it makes it difficult to estimate the work taken to join a task. An online algorithm would try to balance the load counting on whatever partial information available.

Let  $sf(j)$  be the selectivity factor of a task  $j$ , which is not known until its actual processing. Let  $t(j)$  be the time taken to process a task  $j$ , also not known until its actual processing due to its selectivity factor uncertainty. Let  $mint(j)$  and  $maxt(j)$  be the previously known upper and lower bounds for  $t(j)$ , corresponding to  $sf(j) = 0$  and  $sf(j) = 1$ , respectively. Let  $maxsf(S)$  be an upper bound for the selectivity factor of the tasks of  $S$ .

An online task selection algorithm must generate  $S'$  and assign every task  $j$  to a processor knowing only  $maxsf(S)$ ,  $mint(j)$ ,  $maxt(j)$  and the results of past joined tasks.

We should remark that a number of recent works have made good progresses in selectivity factor estimation [5, 6, 10]. Each of these works relaxes some of

the previously restricting assumptions but, nevertheless, some of them still remain. Results of these works can be used in estimating more realistic values for  $maxsf(S)$  (*i.e.*, less than 1) and for  $mint(j)$  and  $maxt(j)$ , surely bringing more accuracy to online decision taken by algorithms.

**Lack of Task Sequence Knowledge** In order to make the optimum split and combination of tasks, an algorithm has to know the complete set of tasks to be joined *a priori*. Another instance of online problem is that of the join of two relations which input data are presented to the algorithm in a partial fashion. This is the case if we consider that operators on a Query Execution Plan (QEP) are often part of a data pipeline. A join algorithm must be confronted to data coming from previously started operators that, even though producing output tuples, are not finished yet.

Let  $\sigma$  be the sequence of task  $j_1, j_2, \dots, j_n$  that are defined in an online fashion<sup>2</sup>. Each task has a weight value that corresponds to an execution time  $t(j)$ . The arriving tasks are not necessarily served immediately, *i.e.*, the scheduler might keep some tasks on hold in order to further combine or split them.

In this instance, an online task assignment algorithm must allocate an arriving task to a processor  $p$  (or a group of them) knowing only  $t(j)$  of the current task  $j$  and the result of past joined tasks. This case has been addressed before on [11], although competitive analysis measures have not been established.

**External Application Load** Join algorithms are eventually executed on parallel machines under concurrency of external applications. A third instance of an online join is that of an algorithm which should make balancing decisions without perfect knowledge of the external load.

Let  $t(j)$  be the exact time taken to process a task  $j$  exclusively on a PN  $p$ . Let  $el(p, s)$  be a measure of an external load to which node  $p$  is faced in instant  $s$ . The  $el$  function can only be computed for the present time, and can as well be preserved as an historical function of the past.

An online task selection algorithm must generate a set  $S'$  and assign every task  $j$  to a processor  $p$  knowing only  $t(j)$ , the historical function  $el(p, s)$  and the results of past joined tasks.

This might be the most difficult instance of the problem, as the instantaneous processor capacity available for the join algorithm might not follow a precise pattern. Works on the predictability of process resource usage, as [4], for example, might be of great assistance in handling such situations.

It is interesting to note that the three above cases are orthogonal. They might very well happen on the same instance of the problem, posing additional complexity to the analysis of an algorithm that handles the combined case.

Addressing the problem simply as an online does not completely attain our goal of establishing a comparison methodology – another important *piece* is

---

<sup>2</sup> Note that, in order to be called tasks, which we consider to be independently joinable, the algorithm must care for the operation completeness

missing in this puzzle: a specific metric for quality of solutions is needed, so to make competitive analysis work. In other words, online and optimal offline algorithms results scores could then be confronted and *competitiveness ratios* could then be established. We address this issue in the next section.

### 3 A Metric for Quality of Solutions

Competitive analysis is based on the difference in the quality of solutions generated by online and optimal offline algorithms [1]. Thus, one of the most important aspects in this kind of analysis is the definition of such a metric. In this section, we discuss how to obtain more precise measures of output quality and present the metric we suggest for this purpose.

#### 3.1 On the Quality Estimation of a Parallel Join Algorithm

One interesting thing about parallel join algorithms is that, as far as correctness goes, the resulting relation itself is unchanged, no matter what algorithm had been used to produce it. We should, therefore, concentrate in evaluating not the solution itself, but the means used by an algorithm to produce it.

Quality of output in parallel join algorithms have to do with workload differences assigned to distinct PN versus the overhead price paid to maintain such difference within a small threshold. The optimal solution would be that which minimizes the maximum time taken to the most overloaded processor to evaluate its assigned tasks, while trying to avoid unnecessary split and combinations of tasks.

It should be noted that achieving a perfectly balanced solution, *i.e.*, an optimal  $S'$ , does not always guarantees the best response times. It is of little help getting the optimal  $S'$  and having (a) to transfer a task a number of times before it is actually processed; (b) to split a task unnecessarily, for the extra processing effort needed by such a technique as fragment-and-replicate; (c) to combine a task in the first place and further split it due to overload, among other issues.

Based on this reasoning, we propose the combined use of three parameters, namely, Tuple Comparison Ratio, Workload Difference, and Mean Tuple Transfer, as a metric for quality of solution assessment for load-balanced parallel join algorithms. We now turn our attention to each parameter and show that they can be used by every parallel join algorithm, regardless of targeted architecture or load balancing strategy, to characterize its solution quality.

#### 3.2 Tuple Comparison Ratio

The first parameter, named Tuple Comparison Ratio, is similar to that used in the monoprocessor algorithms analysis. The less comparisons an algorithm makes to generate the output, the better it is – which is a valid assumption for the parallel case either.

The best possible value for this parameter is 100%, which means that every single tuple comparison made by an algorithm did meet the join condition of the operation being evaluated. In other words, there were no waste of computational time in comparison of tuples which were not meant to be joined.

It is easy to see that this parameter is independent of parallel architecture and load balancing strategy. It has only to do with the algorithm's underlying data partitioning strategy.

### 3.3 Workload Difference

The second proposed parameter is the Workload Difference. It accounts for the worst workload difference between any two PN an algorithm might allow for. One important aspect of this parameter is that workload measure is itself composite.

For instance, take two identical capacity PN and assign them two exactly equal tasks – same cardinality, same selectivity. Their workload difference is zero, the best one can get. Now, take the same PN and assign them two different tasks. Give one a task with a huge cardinality (a great amount of comparisons to make) but low selectivity (few tuples to generate) and give the second a task with less cardinality than the first but with a greater selectivity. The first processor spends most of its time making memory comparisons, while the second one is probably writing tuples to disk. Even so, they might very well finish their tasks on exactly the same time. They also get zero workload difference.

In order to be independent of platform, this parameter should be quantified as a percentage of the total amount of work. In other words, if a workload difference exists, the extra work assigned to the most overloaded processor should be expressed as a percentage of the total work.

For illustration purposes, suppose one overloaded processor had to compare 10000 tuples more than the most underloaded one. Suppose also that among these 10000 comparisons, 500 resulting tuples were found. The total amount of tuples compared by all PN were 100000, among those 1000 resulting tuples were found. Thus, the overload processor handled 10% more comparisons and 50% more resulting tuples than the underloaded one.

One should note that a workload difference percentage in a platform A might correspond to different processing time than that of a platform B. Nevertheless, the quantification of the difference this way gives a good quality measure of the imbalance allowed by an algorithms.

### 3.4 Mean Tuple Transfer

The third parameter of the metric is the one accounting for the overheads. Without it, one could easily propose a "good" algorithm based on the two earlier parameters: one with a very fine grained hashed function generating buckets of tuples with the same join attribute value and dynamically adjusting workload among the PN, for example.

Mean tuple transfers is responsible for the qualitative representation of the extra effort made by an algorithm to obtain good results on the previous parameters. The latter algorithm would have probably spent a great effort in (a) storing back to disk the fine grained tasks and (b) dynamically moving these small tasks around in order to obtain a small workload difference. Any such tuple movements – disk to memory, memory to disk, PN to PN, etc – would reflect in the referred parameter.

As its name suggests, the value of Mean Tuple Movement, is a mean of tuple movement around the parallel machine. Ideally, every tuple should be read from disk into memory once and should be processed by the PN who commanded the read. This would be represented by the value of 1 for the parameter. If, on the other hand, every tuple is read more than once, and is reassigned in order to balance the load, algorithms could have values greater than that.

Suppose an algorithm solved the join in the following way. From the total amount of tuples, 40% were read from disk straight to its allocated PN (*i.e.*, one movement), 35% were read by a PN and further redirected to its processing PN (*i.e.*, two movements), the 25% remaining tuples had to be redirected twice, for dynamic balancing purposes (*i.e.*, three movements). The Mean Tuple Movement in that case was  $0.4 * 1 + 0.35 * 2 + 0.25 * 3 = 1.85$ .

Although many other useful parameters exist, we claim that the combination of these three is able to depict a faithful picture of an algorithm's effectiveness. In every algorithm, their values would be influenced by the way we configure the problem, like range and frequency of values on the attribute join, initial disk tuple placement, etc. In this sense, competitive analysis can also be seen as a request-answer game: an adversary generates requests, and an algorithm has to serve them online [1]. In the next section, we propose a taxonomy for such adversaries of online parallel join algorithms.

## 4 A Taxonomy of Adversaries

Competitive analysis can often be seen as a two players game. On the one side, there is a player who is actually running the algorithm under analysis. On the other side, there is an adversary, who tries to confront its opponent to conditions which generate the worst algorithm output. In a way, the adversary always gets the advantage, as it always knows which algorithm its opponent is running.

Thus, competitive analysis is the confrontation of the result of running an algorithm against its worst opponent to the results of the best offline algorithm for that particular input. Competitive analysis based on adversaries can also be used for online join algorithms, as we are going to show in this section.

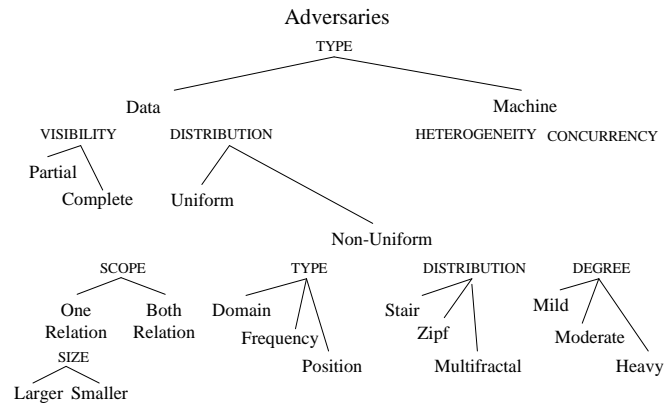
Basically, the idea behind join algorithms' adversaries is to generate such conditions that challenge their ability to efficiently balance the load. Adversaries can be seen as *backstage entities* that create and place operand relations tuples across parallel machine disks, and manipulate platform parameters – either before or during an algorithm run. Therefore, they are able to reproduce a great number

of non-uniformity real DBMS data scenarios that algorithms should be prepared to deal with.

Different non-uniformity conditions can be created by varying three main characteristics, namely, (a) amount of data input that can be seen at once; (b) platform characteristics the algorithm is supposed to adapt to; and, (c) data distribution uniformity. Let us look at them, each on its turn.

By amount of data, we refer to the likelihood of an algorithm to evaluate all data input before making any decision. As the join operator is essentially pipelineable on a query execution plan, algorithms are eventually confronted to partial results coming from previously started operators on the plan – and, are expected to produce output as soon as possible, so to stream up pipelined data. On the other hand, in order to provide the best possible results, algorithms are eventually allowed to access all input data before taking any decision.

By platform characteristics, we mean the ability of an adversary to pose challenges to an algorithm based on platform parameters, mainly heterogeneity and concurrency ones. Heterogeneity might be used by an adversary to configure a parallel machine with different capacities PN. Concurrency might be used to reduce a PN computational capacity instantaneously, as if external application load were using part of PN availability.



**Fig. 2.** Taxonomy of adversaries for online join algorithms

In figure 2, we depict such classes of adversaries along with a third one: data distribution uniformity. In the rest of this section, we concentrate on data distribution adversaries, leaving the further analysis of the two previously mentioned categories for future works.

**Non-Uniform Data Distribution Adversaries** Non-uniform data distribution adversaries are mainly concerned in producing the different skew types described in [12]. This type of adversaries are able to reproduce non-uniformity characteristics based on the variation of a number of aspects, described next.

- **Scope** – Non-uniformity can occur isolated in one of the operand relations or simultaneously in both of them. In the case it occurs only in one of the relations, an adversary can further choose which of them – either the larger or the smaller – to attack. This would explore the fact that some algorithms choose inner and outer relations based only on their size, regardless of scope of skew;
- **Type** – Non-uniformity can be produced in three different orthogonal types. First, on the domain level, when the full range of the domain values are not present on join attribute values. Second, on a frequency level, when repetition of certain values are not as frequent as repetitions of others. Lastly, on the initial positioning level, when tuples are placed on disk so to make PN access more expensive (notably in shared-nothing machines);
- **Distribution** – Non-uniformity can be modeled by differently shaped functions. Walton et. al suggest stair functions, although it uses them just in one of the parallel PN [12]. Zipf distributions has also been largely used in join algorithms evaluation [14]. Recently, new distributions representations appear to reflect more realistically skewed data scenarios [5]. Adversaries might explore certain algorithms assumption about distributions, such as two (or more) PN stair functions, or even mixed distributions;
- **Degree** – Lastly, the degree of non-uniformity applied to data input can be varied. It can be so mild that balancing benefits might be outperformed by its costs. It can also be moderated or heavy, when every balancing effort worth it. Adversaries can test algorithms sensitiveness to the degree of skew, so to check weather they apply heavy load-balancing procedures to every case or they balance effort vs. results.

## 5 Conclusions, Ongoing and Future Works

In this work, we have described a new approach for analyzing parallel join algorithms. The approach is based on the observation that join algorithms are themselves online, as they are supposed to take decisions in the absence of important information. This work had focused on the presentation of such an idea, and had also laid ground to a forthcoming revisit of the analysis of known parallel join algorithms under the new approach.

We have not only formulated the join problem as a *classical* online problem (*i.e.*, that in which tasks arrive in a non predicted sequence and have to be served online) but also identified two additional online problem instances. In order to make competitive analysis possible, we have also introduced a metric for quality of solutions assessment. The metric utilizes three different parameters, that combined, provide a good estimation of workload balancing efficiency, at

the same time that reflect overhead costs for achieving so. Finally, we have put together a taxonomy of adversaries for this class of algorithms.

In conclusion, this work helps answering the initially posed questions in the following ways.

1. Is algorithm A better than algorithm B under circumstances C ?  
A set of circumstances may be modeled as an adversary (or a group thereof) using the proposed taxonomy. An algorithm will be considered better than another should better competitive ratios in the presence of the chosen adversaries be achieved.
2. What are average and worst case response times of algorithm A under a set of circumstances C ?  
Worst case analysis can be obtained via competitive analysis, i.e., by confronting the algorithm to the most challenging adversary – that which produces the worst competitive ratios. Average case analysis is a matter of, again, modeling average case scenarios via adversaries. Although actual response times could not be obtained, comparison of online algorithms to their optimal offline counterparts should provide reasonable estimates.
3. Given a "rich" set of circumstances D (i.e., the set that covers a broad number of real case scenarios), do we already know a set of algorithms which, combined, attain reasonable performance in all C elements ?  
Given a set of adversaries, lower bounds for competitive ratios for algorithms performance might be calculated. The search for new algorithms should be inversely proportional to actual algorithms lower bounds distance.

The present work is part of a major effort of the authors in establishing techniques for analyzing parallel join algorithms. The reader may refer to [2] for a correlated work.

## References

1. Albers, S.: Competitive Online Algorithms. *Optima Math. Prog. Society News.*, **54** (1997) 1–8
2. Carvalho, S.E.R., Lerner, A., Lifschitz, S.: An Application Framework for the Parallel Join Operation in Database Systems. Technical Report n.40/98, Depto. de Informática, PUC-Rio (1998)
3. DeWitt, D.J., Gray, J.: Parallel Database Systems – The Future of High Performance Database Systems. *Comm. of the ACM* **35(6)** (1992) 85–98
4. Devarakonda, M.V., Iyer, R.K.: Predictability of Process Resource Usage: A Measure-Based Study on UNIX. *IEEE Trans. on Software Eng.*, **15(12)** (1989) 1579–1586
5. Faloutsos, C., Matias, Y., Silberschatz, A.: Modeling Skewed Distributions Using Multifractals and the '80-20 Law'. *Procs. Intl. Conf. On VLDB*, (1996) 307–317
6. Ganguly, S., Gibbons, P.B., Matias, Y., Silberschatz, A.: Bifocal Sampling for Skew-Resistant Join Size Estimation. *Procs. ACM SIGMOD Intl. Conf.*, (1996) 271–281
7. Lakshmi, M.S., Yu, P.S.: Effectiveness of Parallel Joins. *IEEE Trans. on Knowledge and Data Engineering*, **2(4)** (1990) 410–424

8. Mishra, P., Eich, M.H.: Join Processing in Relational Databases. *ACM Comp. Surveys*, **24**(1) (1992) 63–113
9. Schneider, D.A., DeWitt, D.J.: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Procs. ACM SIGMOD Intl. Conf.*, (1989) 110–121
10. Swami, A., Schiefer, K.B.: On the Estimation of Join Result Sizes. *Procs. of Intl Conf. on Advances in Database Technologies*, (1994) 287–300
11. Swami, A., Young, H.C.: Online Algorithms for Handling Skew in Parallel Joins. *Procs. of Intl. Conf. on Parallel Processing*, (1993) 253–257
12. Walton, C.B., Dale, A.G., Jenevein, R.M.: A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. *Procs. Intl. Conf. On VLDB*, (1991) 537–548
13. Wolfson, O., Huang, Y.: Competitive Analysis of Caching in Distributed Databases. *IEEE Trans. on Par. and Dist. Syst.*, **9**(4) (1998) 391–409
14. Zipf, G.K.: *Human Behavior and Principle of Least Effort: an Introduction to Human Ecology*. Addison Wesley, Cambridge, Massachusetts (1949)