Exploring Load Balancing in Parallel Processing of Recursive Queries[1]

Sérgio Lifschitz[2]
lifschit@inf.puc-rio.br

Alexandre Plastino
Departamento de Ciência da Computação
Universidade Federal Fluminense, Niterói RJ
plastino@dcc.uff.br

Celso C. Ribeiro[3]
celso@inf.puc-rio.br

PUC-RioInf.MCC37/96 November, 1996

Abstract

Recent work on load balancing has confirmed its importance when one wants to achieve good performances during the actual evaluation of parallel database queries. Existing work mostly focuses on the join processing for parallel relational databases. We are interested here in more complex queries, such as recursive ones. The main difference is that, in the latter case, the work due to a task cannot be previously determined and, consequently, no method can define at the outset the tasks to be executed in parallel in order to balance the workload at each processor. We propose a *dose-driven* dynamic strategy that aims at obtaining an improved workload balance and better use of the available resources. We examine the applicability of our strategy with its specialization to the case of the transitive closure query. Preliminary computational results on randomly generated test problems illustrate the efficiency of the proposed method.

Keywords: Load Balancing, Dynamic Assignment, Recursive Queries.

Resumo

Trabalhos recentes têm confirmado a importância do balanceamento de carga na tentativa de se obter desempenhos satisfatórios na avaliação de consultas em bancos de dados paralelos. A maioria dos trabalhos existentes se concentram na avaliação paralela da junção em bancos de dados relacionais. Neste trabalho, estamos interessados em cálculos mais complexos, como é o caso das consultas recursivas. A principal diferença é que, no último caso, a quantidade de trabalho associada a uma tarefa não pode ser prevista e, conseqüentemente, nenhum método consegue definir previamente as tarefas a serem executadas em paralelo balanceando a carga em cada processador. Propomos, então, uma estratégia dinâmica de distribuição de tarefas, chamada *dose-driven*, que procura obter um balanceamento de carga adequado e uma melhor utilização dos recursos computacionais disponíveis. Examinamos os resultados da nossa estratégia através da sua especialização para o caso do cálculo do fecho transitivo. Resultados computacionais preliminares, obtidos a partir de testes gerados aleatoriamente, ilustram a eficiência do método proposto.

Palavras-Chave: Balanceamento de Carga, Alocação Dinâmica, Consultas Recursivas.

---

[1] Trabalho parcialmente desenvolvido no Laboratório Nacional de Computação Científica (LNCC)
[2] Parcialmente apoiado por bolsa de produtividade em pesquisa do CNPq 300048/94-7
[3] Parcialmente apoiado por bolsa de produtividade em pesquisa do CNPq 302281/85-1

1 Introduction

In many different computer applications, parallel processing has been motivated by the continuously decreasing cost of parallel architectures and the increasing availability of parallel machines. Among the main issues that have to be managed in order to make these multiprocessor environments practical, the even distribution of work to each processor is a fundamental one. Therefore, we can make full use of this new computational power and obtain corresponding expected performances. Thus, many load balancing techniques have been proposed in recent years.

When database systems are considered, the expected and actual increase in data size and complexity of queries has motivated a great research interest in parallel processing. In this case, load balancing strategies have focused mostly on relational systems and intra-operator parallelism, usually (equi)join processing (*e.g* [Omi91, DNS+92, LT92, WDY+94]). The need for load balancing, in these cases, is due to different kinds of data skew, which are usually classified in *intrinsic skew* - related directly to data, normally when attribute values are not uniformly distributed - and *partition skew*, related to the data distribution and selectivity of joins and selections at each processing site [WDJ91].

Existing parallel join algorithms usually implement a task-oriented method, where these tasks are part of the total amount of work to be executed [LT94]. It is of main importance to determine the number of tasks into which the join operation should be divided. Also, the distribution of tasks might be static, when all tasks are assigned to sites at the outset, or adaptive, when tasks are sent to the sites dynamically, after the parallel execution has started. In both cases the workload may not be balanced among the sites during the join processing and load redistribution is usually employed.

In this paper we are mainly concerned with the parallel processing and load balancing of recursive queries. These appear typically in the context of deductive databases and many important applications have already come to date, such as exploratory data analysis, enterprise modeling and scientific databases [Tsu91, Ram95]. While many algorithms have been proposed for processing (datalog) recursive queries in parallel [WO93, GST90, ZWC95, DSH+94, LV95], in particular for transitive closure queries [CCH93], only a few have considered load balancing issues.

The main difference with respect to join queries is that, in general, the work due to a task is known only during the evaluation process and so, a static approach for task generation may fail to achieve an even workload. Existing load balancing techniques [WO93, DSH+94] for recursive queries are based on an initial static distribution of work to each site and further dynamic redistribution when an uneven workload is detected. It is then a matter of efficiency, as the workload redistribution is usually expensive and even if there is a way to make it cheap, it is not guaranteed that it will achieve a better performance than with the previous situation.

For both recursive and non recursive queries, we believe that a set of tasks should be assigned to each site dynamically during the query evaluation process and workload imbalance should be controlled, avoided if possible.

Also, any workload balancing approach must take into consideration a multi-user (multi-transaction) or even an heterogeneous environment when processing the query. Further care must be taken in order to balance the load and keep all sites as active as possible.

A task-oriented demand-driven strategy with some of these ideas in mind has been suggested in [LT92, LT94] for balancing the load during join processing. There, the task generation phase is guided by the particular join algorithm (*e.g.* hash join) used and whenever there is an uneven workload at the end of execution, a task reorganization strategy is employed to correct it.

We propose here also a task-oriented parallel processing strategy but for dealing with recursive queries. We deal with more complex queries than (equi)joins, as the actual task size (*e.g.* a subset of rule instantiations) is generated dynamically during execution. Our focus is on determining and controlling the execution when tasks are being distributed, in order to avoid load imbalance at most rather than enabling it and correcting it later. We claim that static-sized tasks may be avoided so to better tune which and how many tasks are to be run at each site. Our strategy is called *dose-driven*, which stands for a task-oriented demand-driven method that aims at obtaining an even workload distribution with variable-sized tasks corresponding to doses that are assigned to the parallel sites.

When comparing to the existing load balancing strategies for recursive queries (*e.g.*[WO93, DSH+94]), there are a few important distinctions: first, each site is allowed to process any kind of task rather than a fixed version of it; the amount of time a site stays idle can be controlled, so to make better use of the available resources, and the strategy adapts well to heterogeneous conditions, either with respect to the hardware, or when it is run in concurrent (and more realistic) environments.

In order to illustrate the applicability of our proposed strategy, we have adapted it to the case of the transitive closure query, the most referenced case of recursive query. We compare the behavior of our approach to that proposed in [AJ88], which is a static distribution strategy that is well adapted to uniform conditions. The observed behavior and practical results obtained are very stimulating. Some open interesting issues have risen from the practical results that could not be observed by analytical simulations. Particularly, a fixed cost associated to tasks being generated was identified, which can make, as we will see later, the load balancing goal fail to obtain the best parallel performance. It should be noted that most previous works were not actually implemented.

The remainder of the paper is organized as follows. In the next section, we review some of the most recent and distinct techniques for balancing the load in parallel database systems. In Section 3, we motivate our proposed strategy and list the desired properties we would like to achieve. Then, in Section 4, the strategy is explained in detail when applied to the transitive closure query, with experimental results given in Section 5. We further explore some important aspects of the proposed workload balancing method in Section 6 before the concluding remarks in Section 7.

2 Load Balancing Techniques

The work on load balancing techniques in database systems has focused on the parallel evaluation of the relational join operator. The general structure of these algorithms is divided into three phases [LT94]:

- task generation,

- task allocation and
- task execution.

As an example, the relations involved in the join may be first decomposed into subrelations, then grouped into sets of subrelations (or buckets) that are assigned to processing nodes and, finally, each set is processed at each node by a local join algorithm.

The last phase is closely related to the processing in monoprocessor environments and are not of further interest here. However, both the first and second phases must be carefully considered. Indeed, the type of data distribution, the total number and size of tasks and also the way - statically or dynamically - these tasks are assigned to processors are issues that may considerably affect the performance of the whole parallel processing strategy.

Join Processing

When join processing is considered, many methods have been proposed in recent years with some underlying technique which balances the workload. In [Omi91], a strategy well adapted to shared-everything environments is proposed, with an extra scheduling phase during GRACE hash join processing to allocate buckets to processors.

In [DNS+92], a multiple join algorithms strategy, each specialized for a different degree of skew, is considered. A small sample of the relations involved in the join operation determines which algorithm is more appropriate. A dynamic load balancing technique is proposed in [ZJM94], where periodic *checkpoints* are done during the evaluation process to detect that actual hash function output. If needed, some of the tasks assigned to overloaded sites are redistributed to the others. In [HTY95], both previous strategies are implemented in a nCUBE/2 parallel computer and practical results are compared, where the method in [DNS+92] is shown to be superior.

In [WDY+94], some of the previous results for sort-merge and hash join algorithms proposed in previous works are reviewed so to obtain better performances. There is a scheduling phase for the assignment of tasks to processors that gives a different treatment when skew is detected, with the use of a divide-and-conquer technique. Practical behavior is simulated by analitical models. A partition size tuning approach is proposed in [HLH95], which aims at balancing the load at the partition (set of buckets) level, as data skew - at the original relations - may cause bucket skew and combining a few buckets into partitions may balance the workload that would be due to a bucket-oriented processing.

A dynamic *demand-driven* strategy for parallel joins is studied in [LT92, LT94]. The tasks correspond to hash buckets that have to be joined and are further included in a *pool* of tasks. Then, they are allocated to sites dynamically, whenever a site becomes idle. By the end of the parallel execution, a task steal approach for load balancing is proposed to deal with workload not balanced. Indeed, when there are no more tasks and only a few sites still work, the idle sites *steal* subtasks from active sites so to balance the workload among slow and fast processing nodes.

The importance of minimizing the processor idle time while executing some load balancing strategy is discussed in [LC93] for shared-nothing systems. A scheduling technique is proposed, where the order in which data pages are loaded into memory is shown to be important to provide a better use of the multiprocessor machine. In [BK96], a control mechanism is proposed to deal with the join product skew [WDJ91], related to the selectivity of the join relations at each site. As many other strategies, there is a detection phase, that determines whether there is a workload imbalance and a correction phase, that includes reorganization of load distribution according to an estimation model of relations cardinalities.

Recursive and Rule Processing

In the case of recursive queries, only a few results are known which deal with load balancing issues. Existing works appear in the context of (datalog) rule programs. Most works have considered parallelization schemes that apply to the rule program from which the relational expressions to be evaluated are derived rather than a straightforward strategy that parallelizes the computation by partitioning relational algebra operations among the processing nodes. The main drawbacks expected are the strong synchronization needed to complete each operation (*e.g.* joins) and the amount of communication during the evaluation [WO93].

The general framework for processing recursive queries is known as *data reduction* paradigm [WO93]. When considering a (datalog) rule program, these methods are also known as *rule instantiations partitioning* strategies. The main idea is to parallelize the query evaluation by assigning subsets of the rule instantiations among the sites, such that each site evaluates the same program but with less data. In fact, each site is responsible for a *restricted version* of the program, which is obtained by appending some arithmetic predicates (*e.g.* hash functions) to some or all of the program rules.

Although much work has been done in order to develop efficient parallel strategies to process recursive queries, load balancing issues are not usually taken into account. It is recognized in [WO93] that even the best restricting functions might fail to balance the workload and there is a need to include a load balancing step into the complete strategy. A method is then proposed where a list of alternative parallelization strategies (a set of different restricting predicates) could be used to change the strategy dynamically whenever a processing node knows it is active for a long time and some other nodes are idle. So, each site would replace its local program version by a new one if load imbalance is detected. This approach has many drawbacks. It is clearly not efficient since a complete strategy change may lose all the work already done. Also, there is nothing that can guarantee a better performance with the new strategy and, finally, there is no simple way to implement it.

In [DSH+94], the task generation and distribution to sites is static at the outset and a more sophisticated parallel strategy is proposed, where there is a *predictive* protocol for detecting potential uneven processing at each site and a *correction* algorithm that balances the load. The problem here is that they make some considerations about load imbalance that may not be true in practice. Indeed, it is considered that a larger intensional and extensional local database in a given site implies more work in the future and this is not always the case, as it happens in the join product skew. Furthermore, in practical situations, there are some other issues to be considered when forecasting future processing status, such as some external load, from other users and transactions, in one or more sites.

3 A Dynamic Workload Balancing Strategy

We present here a parallelization strategy for processing recursive queries that takes into consideration intrinsic and partition skew conditions, while trying to keep all processing sites permanently activated. It is a dynamic method in the sense that the workload is assigned to each site during the evaluation process and is demand-driven because new tasks are allocated to a site when this site becomes idle and asks for new tasks to process. We call our strategy *dose-driven*, which allows variable-sized tasks so to better tune the processing and avoid any redistribution of tasks.

A task, in our case, is not guided by a specific relational operator algorithm but is rather any amount of work that can be defined from the total work to be done to process a query. There is no step in the proposed strategy that aims at correcting any load imbalance that may occur. We claim that these dynamic workload redistributions are not efficient in general, particularly for recursive queries, when any inference work being done may be lost due to a change in the evaluation strategy after it has started.

The basic idea for processing the recursive query (or a datalog program) in parallel is the rule instantiations paradigm. There are two types of sites participating in the execution: a *coordinating* site, which monitors the execution and controls task distribution and the *processing* sites, which make the actual query processing.

The coordinating site is mainly responsible for two processes:

- *task derivation*: in the case of recursive queries, there are many ways to define the granularity of tasks. The most common are *tuple-based* tasks, which are strongly related to horizontal data fragmentation in distributed database systems and *domain-based* tasks, where the set of attribute constants relevant to the query define the work that needs to be executed; and
- *task assignment*: this determine the dynamic distribution policy which controls the whole evaluation process, being responsible for keeping the workload balanced.

It should be noted that tasks can be determined by the actual work that has to be done, therefore both tuple-based and domain-based task derivation could be employed in a specific situation. Also, variable-sized tasks could be generated in the task derivation step due to a fine tuning load balancing technique employed by the task assignment step. One possibility could be to leave smaller tasks to the end of the evaluation process, as there is less derivation of new facts.

The processing sites are responsible for:

- *task processing*: the actual task processing, which consists in the general case of the instantiation of program rules. It is important to note that, instead of being responsible for a fixed set of instantiations, all sites may fire rule instantiations corresponding to any restricted version of the program; and
- *task request*: as soon as they become idle, the processing sites send control messages to the coordinating site indicating that there are no more tasks to be processed and that new tasks should be transmitted.

When shared-memory or shared-disk architectures are considered, the shared resource is used as a global memory that makes it possible for the coordinating site to generate tasks from newly derived facts and instantiations that have not yet been performed. In the case of shared-nothing, it is, as expected, harder to keep the load balanced and there will be a need for some data transmissions among sites. Anyhow, completeness of the evaluation is guaranteed as long as all instantiations can be executed. This can be controlled from the set of attribute constants present in the database. As mentioned in [LV95], some further optimization can be made when considering *potential productive* instantiations, where only those instantiations that can derive facts not yet produced would generate tasks to be assigned to sites. The proposed strategy may also work well under uniform and balanced conditions, which guarantees its applicability to any situation.

An important set of positive features can be seen when we apply our ideas to the simple, yet very important, transitive closure query. We have chosen to illustrate our ideas with the linear transitive closure query, not only because of an easier intuition of what is expected from the implementations but also in order to be able to make some comparisons to previous works that do not include load balancing techniques. This is the case of the algorithm proposed in [AJ88] - that we will call here AJ - which implements the data reduction strategy for the transitive closure.

4 Case Study: Transitive Closure

In this section, we give the specialization of our strategy for the evaluation of the transitive closure of a binary relation, say R, usually defined as follows:

$$\begin{array}{lll} r_1: & Tc(x,y):- & R(x,z), Tc(z,y). \\ r_2: & Tc(x,y):- & R(x,y). \end{array}$$

The evaluation of the Tc relation may be understood as the computation of all successors of all nodes in the relation corresponding direct graph. Thus, we may define the tasks to be executed as the computation of all successors of a subset of constants in R. In its linear definition, the transitive closure evaluation can be executed in parallel with no communication during the evaluation process (known as *pure parallelization*), as long as R is replicated through all sites. This property of recursive programs is called *decomposability* [WO93].

If we want to apply our strategy to the transitive closure query processing, the number of tasks that will be generated must be bigger than the number of processing sites available. If there are p sites and t tasks, where $t > p$, we could determine each of the tasks as follows: considering an order in the constants set taken into account by the query, a task will be the pair (i, j), where i is the i-th constant in the domain and j represents the number of constants belonging to the task itself. Without loss of generality, let us consider that a relation attribute domain is defined by the range [1..n]. Then any of the tasks to be generated will be the pair (i, j), which corresponds to the subrange of constants [i, i+j]. Consequently, sending and receiving tasks are reduced to the transmission of this given pair.

One of the sites, the coordinator, controls the distribution of tasks to all p sites. There are two phases: in the first one, the coordinator distributes p tasks, exactly one allocated to each site. Each processing node computes its task and at the end sends an *end of task* *execution* control message to the coordinator and waits for a new task. All local evaluations, that is, task processing, correspond here to a seminaive evaluation. The initial set of tuples for the base relation are those in R whose first attribute equals some constant included in the task. Therefore, all successors of the constants in the task will be determined when that task evaluation is done. The seminaive algorithm has also been used in the implementation of the AJ strategy, so that behavior comparisons are allowed.

In the second and last phase, it is time for the dynamic distribution of the remaining $t-p$ tasks. As long as there are still tasks to be distributed, the coordinator site sends a new task to a site upon request as soon as it becomes idle. When all tasks have been assigned to the processing sites, the coordinator waits until every site sends its end of execution message and broadcasts a message, indicating that the evaluation has reached its end. It is important to note that the second phase can start before the end of the computation of all initial tasks. In fact, right after the first site sends its end of processing of the initial task to the coordinator, the dynamic and adaptive task assignment strategy begins.

We claim that this strategy avoids any load imbalance. Each one of the p sites gets a number of tasks that it is able to execute in that moment. Considering a multi-user or multi-transactions (non exclusive use of available resources) parallel environment, if any site is overloaded, it will get less tasks than others as its processing speed is low, while sites less charged will be responsible for a higher number of tasks. Even in a single-user situations, the dose-driven idea can successfully balance the load in algorithms like AJ, where there is no way to forecast the amount of work to be done by a given task. If long duration tasks were assigned to the processing sites, other sites take care of the smaller ones. It is then possible in extreme situations that a site gets its initial task to execute and no more tasks in the second phase.

It should be clear by now that our algorithm is equivalent to algorithm AJ if the number of tasks is equal to number of processing sites.

5 Experimental Results

We have implemented both the strategy presented in the previous section and the AJ algorithm, so to investigate their behavior in different skew situations.

We have done all implementations on the IBM 9076 SP/2 machine. It is a parallel machine supporting SPMD program applications, with its basic configuration of up to 16 nodes, each composed by a RISC/6000 processor, private disk and memory, interconnected by a high speed switch.

At each node, a complete Open Ingres DBMS is available, being responsible for all relational operations and storage of permanent and temporary relations. Strategies and the transitive closure query were all coded in C with database access made through SQL. Each node runs a local seminaive algorithm on the portion of data designed for it by the coordinating site. To make full use of the parallel environment, MPI (Message Passing Interface) [MPI95] was the chosen interface.

Binary relations (corresponding to cyclic and acyclic graphs) were randomly generated defined by two parameters: the number of attribute constants (graph nodes) and the edge probability, that determines whether two nodes are directly connected or not. This probability could be fixed for all nodes or also randomly generated, so to simulate a non-uniform data distribution.

First, we have tried out a few skew effects and observed the behavior of the AJ algorithm, in a single-user and multi-user situations. Then, we have run our Dose Driven (DD) algorithm so to better understand its behavior in practice.

5.1 Skew effects

In what follows and in order to better illustrate our experiments, we have chosen a few input relations (Table 1) that will be used to further comment on practical results. Many other experiments, even on inputs generated on similar parameters, were realized but with no additional observations. It should be noted that input relations R1 and R2 correspond to acyclic graphs while R3 and R4 are cyclic ones. In particular, for R3 and R4 we obtain the corresponding complete graph as the transitive closure.

Input	Base Relation	Answer Relation	Constants	Edge Probability
R1	5025 tuples	153713 tuples	1000	1%
R2	7238 tuples	258266 tuples	1200	1%
R3	20520 tuples	40000 tuples	200	random
R4	10158 tuples	1000000 tuples	1000	1%

Table 1: Sampled Input Data

To illustrate the effects of partition skew, here related to the unknown workload associated to each task assigned to a processing node, we show in Figure 1 bar coded graphics representing the evaluation of the transitive closure query by the AJ strategy in a single-user (exclusive) parallel environment. As there are no external load neither database concurrency, the results show the exact processing time each site has taken to evaluate its tasks. Times are given in seconds and N01, N02, ... N10 are the 10 sites used.

As we can see, there is a strong workload imbalance in Figure 1(a). Although it was an equal distribution of attribute constants to each site through a hash function (*e.g. mod*) on uniformly distributed random data, site N01 has taken twice as much the time N04 has processed its job. That is, both sites were supposed to determine all successors of 100 (1000 constants equally distributed to 10 sites) nodes and N01 had much more work to do. Indeed, 17326 tuples were derived at N01 and only 12838 at N04. A similar situation occurs in Figure 1(b). However, for both relations R3 and R4, there was almost a perfect workload balance, which is due to the fact that the final answer is the complete graph and every node has the same number of successors, thus, an even tuple production work.

We consider now a non-exclusive environment (a more realistic situation) where there are other processes - accessing the database or not - running concurrently on the same sites. To better understand what may happen, we have run AJ algorithm with relations R3 and R4 in this multi-user environment. As expected, an uneven processing time has occurred, although the work executed at each site was equivalent. This can be better observed in

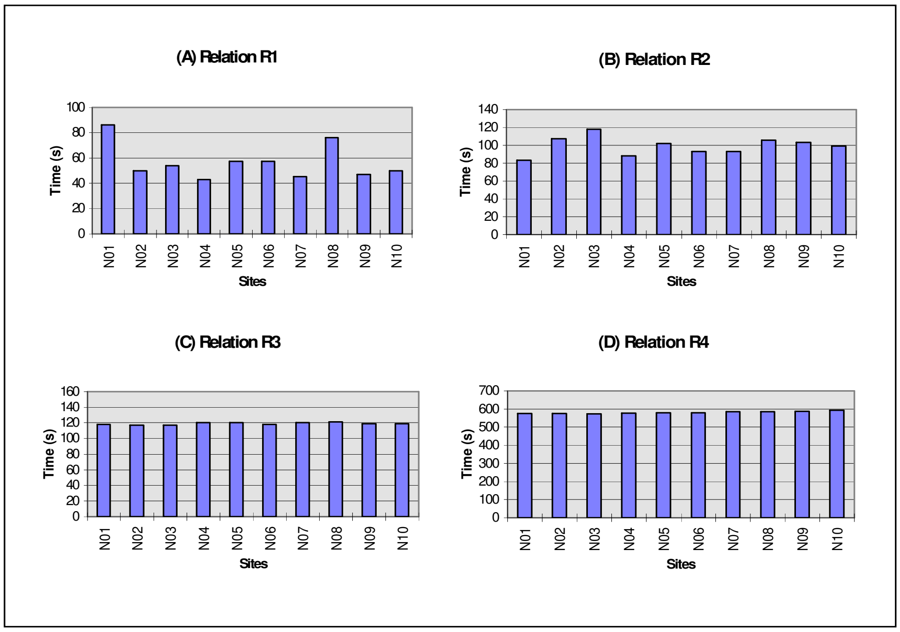

Figure 1: *Strategy Skew*

Figure 2. There, we show the total processing time at each site in both situations, multi-user mode (not exclusive) on the left and single-user (exclusive) on the right.

It can be observed that when concurrency for multi-processors resources exists, the time needed for the slowest site[4] to complete its task is almost 4 times bigger than the fastest one for both cases. Even if we believe that this difference was due to the particular moment when the executions were made, there is a clear need for workload balancing here. So, the best parallel strategy is not always the one that partitions the work to be done in equally sized parts but rather, the one that achieves an equal assignment of tasks considering the availability of resources.

Next, we will compare the results obtained with our proposed strategy with the previous results and some other situations.

5.2 Behavior Comparison

We will show here the applicability of our dose-driven strategy to the case of recursive queries. Actually, as we wanted to explore different possibilities in the dynamic assignment of tasks to sites, we have tried out the DD strategy with distinct total number of tasks, ranging from 20 up to 1000 (limit situation where one task correspond to exactly one attribute constant) tasks. It should be noted that a total number of 10 tasks in a 10 sites environment corresponds to the AJ strategy[5].

In Figure 2(b), it is seen that the parallel time of the AJ strategy for relation R4 is 2240 seconds. As it can be seen in Figure 3, when there are a total of 20 tasks to be dynamically allocated to the sites, algorithm DD has obtained a parallel processing time of 1676 seconds

[4]equivalent to the parallel time

[5]the only difference is in the way tasks are determined

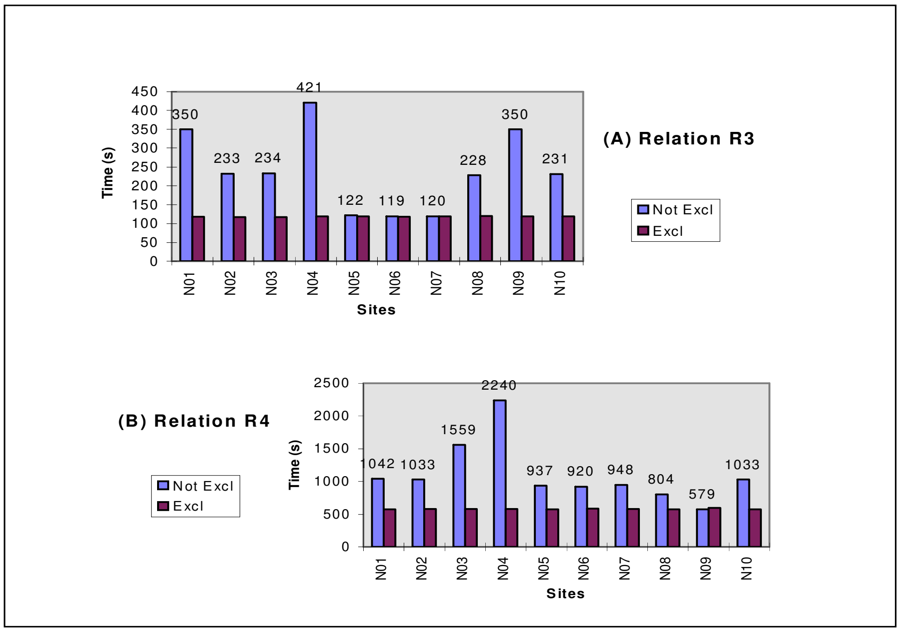

Figure 2: *Concurrency Skew*

and when the number of tasks is doubled, the total execution time is even smaller, equal to 1263 seconds.

One could think that a continuous increase on the number of tasks would imply even better results, as the strategy can better tune the assignment of tasks with respect to the actual load of processing sites. However, with a total of 60 tasks to be executed, the parallel time is worse than the 40 tasks option and, as shown in Figure 3, as the number of tasks increase, the parallel time keep its ascending curve, up to 1000 tasks (one constant per task), where it gets even worse than the AJ algorithm.

What happens in practice is that the query processing work, when partitioned in a set of tasks, has a fixed cost per task that is intrinsically sequential and that every task carries with it. The sum of this fixed cost is minimized when there is only one task but increases when more tasks are created. It is not the case here to determine the *optimal* number of tasks to be chosen but it becomes clear now that this number cannot be close to one task per site neither to the maximum of tasks that can be generated, in our case, one constant per task. It is worth saying that before obtaining these experimental results, we were expecting the best behavior exactly for the one constant per task situation, which we know now that must not be considered.

In Figure 4, we observe the actual distribution of tasks that occurred for algorithm DD with 40 tasks, which has obtained the best parallel time before. In the horizontal axis, *I[J]* indicates that site *NI* has performed *J* tasks. So, we see that N04 has executed only 2 tasks, as its external load was high, while site N09 was responsible for 7 tasks, almost 20% of the total number of tasks. It should be noted that this same site was the fastest one when processing the AJ algorithm.

Not only there is a gain in the efficiency of the parallel query processing but also it is shown that a good workload balancing was obtained. However, if the minimization of the total parallel time is the goal to be achieved, it should be clear that not always

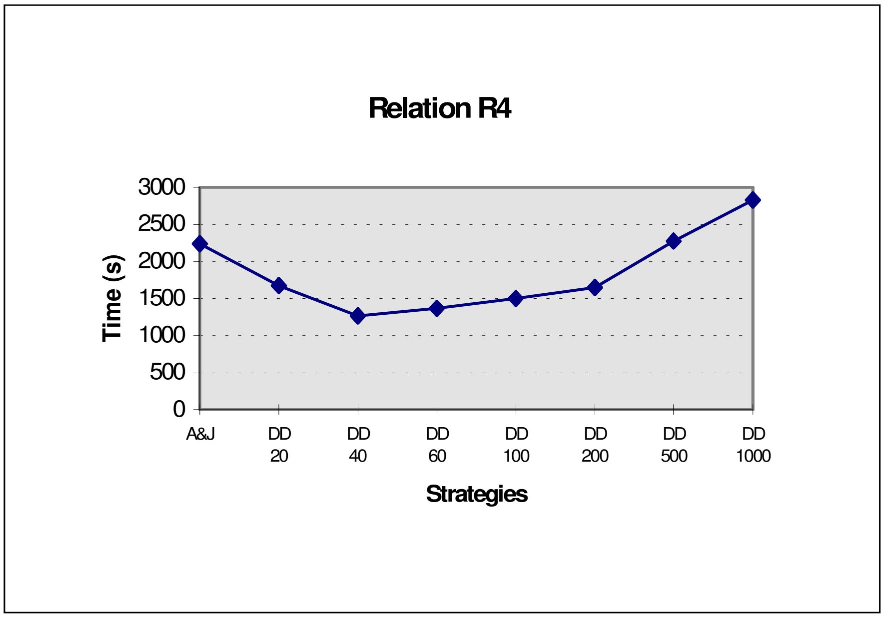

Figure 3: *AJ versus DD Strategy*

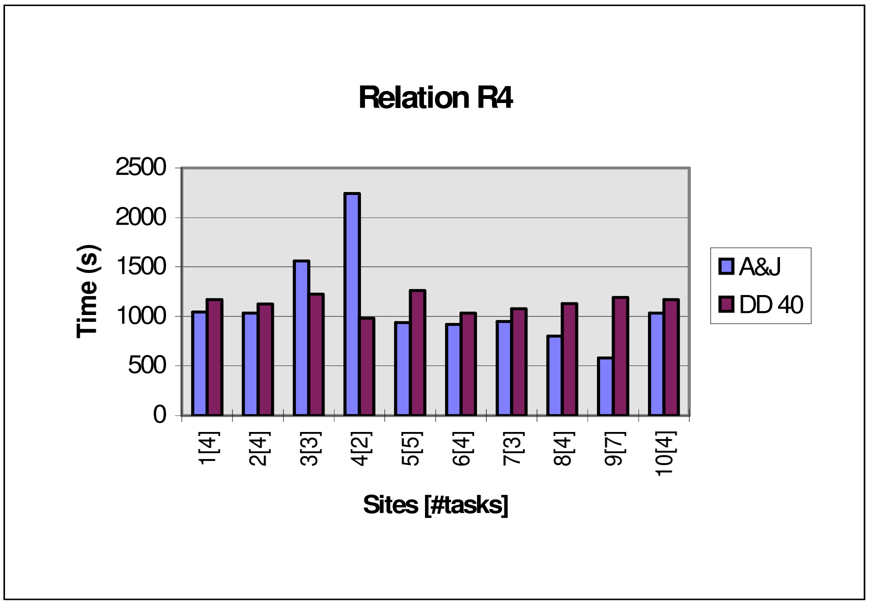

Figure 4: *Workload Balancing Comparison*

the best workload balance corresponds to the best parallel strategy. Indeed, as seen in Figure 5, we compare the workload balance obtained by three distinct evaluations with the DD algorithm: the one with 40 tasks and two others, with 100 and 200 tasks. In the latter case, the workload is quite similar in every site but all processing times obtained are superior than the biggest time obtained for 100 tasks. The same occurs with the 100 tasks execution with respect to the 40 tasks one. These situation motivates the following discussion.

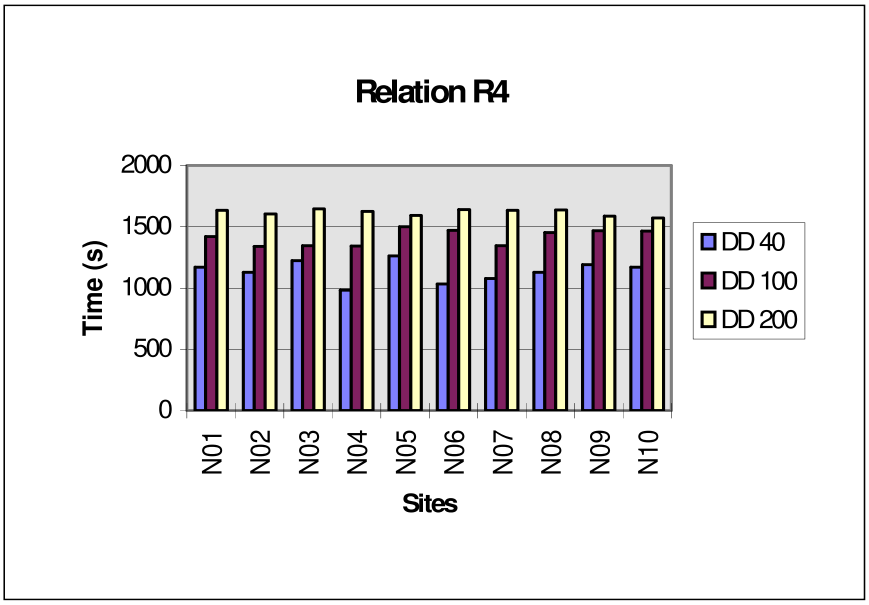

Figure 5: *Load Balancing versus Performance*

6 Discussion

In this section, we investigate further and formalize the previous discussion on optimization, where we have noticed that the minimization of the total parallel processing time and the minimization of load imbalance, with respect to the number and size of tasks, are problems which optimal solutions may not coincide.

We denote by $T = \{t_1, \cdots, t_n\}$ the set of tasks to be executed and by $P = \{p_1, \cdots, p_m\}$ the set of processors where they will be processed. Let c_j be the execution time of task $t_j \in T, \forall j = 1, \cdots, n$. Moreover, let $\mathcal{S}$ denote the set of feasible schedules of tasks to processors and $A_k(s)$ denote the set of tasks assigned to processor $p_k, \forall k = 1, \cdots, m$, according to schedule $s \in \mathcal{S}$. Then, a feasible schedule $s \in \mathcal{S}$ may be defined by a vector $(A_1(s), \cdots, A_m(s))$ such that $\bigcup_{k=1}^{k=m} A_k(s) = T$ and $A_k(s) \bigcap A_\ell(s) = \emptyset, \forall k \neq \ell \in \{1, \cdots, m\}$. The following optimization problems are associated with load balance optimization:

(i) *Minimization of the load of the most charged processor*, i.e., minimizing the processing time:

P_{time} :

$$min_{\ (A_1,\cdots,A_m)\in\mathcal{S}} \ \{ \ max_{\ k=1,\cdots,m} \ \{ \sum_{j\in A_k} c_j \} \ \}$$

(ii) *Minimization of load imbalance*, i.e., optimizing load distribution among processors:

P_{load} :

$$min_{\ (A_1,\cdots,A_m)\in\mathcal{S}} \ \mid \ max_{\ k=1,\cdots,m} \ \{\sum_{j\in A_k} c_j\} - min_{\ k=1,\cdots,m} \ \{\sum_{j\in A_k} c_j\} \ \mid$$

We notice that both problems lead to very close solutions in most cases and that, very often, they can be interchangeably used. However, the situation is slightly different in the case of the problem studied in the current work. Here, in fact, we show that there is an atomic unit of work whose size is q, on which both the execution time and the number of tasks depend.

We now denote by $n(q)$ the number of tasks to be solved when the problem is decomposed into atomic tasks whose size is q. Accordingly, let $c_j(q)$ be the associated execution time of task $t_j \in T, \forall j = 1, \cdots, n(q)$. Then, the above problems can be reformulated as problems $P_{time}(q)$ and $P_{load}(q)$ below, in which we look for the optimal atomic unit of work optimizing, respectively, processing time and load distribution:

$P_{time}(q)$:

$$min_{\ q} \ \{ \ min_{\ (A_1,\cdots,A_m)\in\mathcal{S}} \ \{ \ max_{\ k=1,\cdots,m} \ \{\sum_{j\in A_k} c_j(q)\} \ \} \ \}$$

$P_{load}(q)$:

$$min_{\ q} \ \{ \ min_{\ (A_1,\cdots,A_m)\in\mathcal{S}} \ \mid \ max_{\ k=1,\cdots,m} \ \{\sum_{j\in A_k} c_j(q)\} - min_{\ k=1,\cdots,m} \ \{\sum_{j\in A_k} c_j\} \ \mid \}$$

We notice that, as far as problems $P_{time}(q)$ and $P_{load}(q)$ above are very sensitive to the size q of the basic unit of work, their optimal solutions can be quite different and lead to opposing results in terms of the criteria they optimize. In fact, we can see from the Figure 5 that the strategy DD with 40 tasks, corresponding to the smaller processing time (defined by site N05), has a larger load imbalance than that of DD with 100 tasks, which, in turn, is better in terms of load balance despite of showing a larger processing time (again, observed at processor N05). Analogous comments can be done on the 100-tasks with respect to the 200-tasks DD algorithm.

7 Final Comments

There are many interesting points to discuss and further explore. First, we believe that an increasing number of tasks is valid while the sum of fixed costs related to every task does not offset the gain in performance obtained by the DD strategy. It is still an open question if there is any fixed cost variation with respect to the size of the tasks and this must be better investigated.

An important issue we will investigate refers to data fragmentation. We have so far considered simple hash and range partitioning for determining the tasks to be executed. We would like to see the performance of our strategy when schemes proposed specially for recursive queries [HAS93, ZZO94] are taken into account.

Another question to be studied is whether to have initial tasks with different sizes compared to all other tasks. We may expect that if the degree of concurrency at all sites is about the same, it could be interesting to determine initially large tasks, with a small part left for the second phase, for better tuning the possible uneven workload. Nevertheless, when the existing external load at each site varies drastically, it might be better to start with smaller tasks so to identify which sites could slow down the whole process and, consequently, avoid sending big tasks to it.

It is also a good alternative to permit variable-sized tasks during the evaluation process. A possibility is to keep the size of tasks decreasing until we get close to the end of the evaluation and, in this way, enabling a fine tuning of the processing times. As another example, this could be defined when more information on the current status of the processing nodes is available and it is possible to know whether to generate and assign smaller or larger tasks to a given idle site. In this case, we must take into consideration the fixed cost of very small tasks, that can be harmful to the whole query evaluation process.

It is also important to note that some of the ideas discussed here can be applied to database queries in general, not only recursive ones. We are planning to investigate this issue further looking forward to develop a framework for parallel strategies that achieve a workload balance.

References

[AJ88] R. Agrawal and H.V. Jagadish "Multiprocessor Transitive Closure Algorithms" *Procs. Intl. Symp. on Database in Parallel and Distributed Systems*, 1988, pp 56–66.

[BK96] L. Brunie and H. Kosch "Control Strategies for Complex Relational Query Processing in Shared-Nothing Systems" *SIGMOD Record* 25(3), 1996, pp 34–39.

[CCH93] F. Cacace, S. Ceri and M. Houtsma, "A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs", *Distributed and Parallel Databases* 1(4), 1993, pp 337–382.

[DSH+94] H.M. Dewan, S.J. Stolfo, M.A. Hernandez and J-J. Hwang, "Predictive Dynamic Load Balancing of Parallel and Distributed Rule and Query Processing", *Procs. of the ACM-SIGMOD Intl. Conf. on Management of Data*, 1994, pp 277-288.

[DNS+92] D.J. DeWitt, J. Naughton and D.A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins" *Procs. Intl. Conf. Very Large Data Bases*, 1992, pp 27–40.

[GST90] S. Ganguly, A. Silberschatz and S. Tsur, "A Framework for the Parallel Processing of Datalog Queries", *Procs. of the ACM-SIGMOD Intl. Conf. on Management of Data*, 1990, pp 143–152.

[HAS93] M.A.W Houtsma, P.M.G. Apers and G.L.V. Schipper, "Data Fragmentation for Parallel Transitive Closure Strategies", *Procs. IEEE Intl. Conf. Data Engineering*, 1993, pp 447–456.

[HLH95] K.A. Hua, C. Lee and C.M. Hua, "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning", *IEEE Transactions on Knowledge and Data Engineering* 7(6), 1995, pp 968–983.

[HTY95] K.A. Hua, W. Tavanapong and H.C. Young, "A Performance Evaluation of Load Balancing Techniques for Join Operations on Multicomputer Database Systems", *Procs. IEEE Intl. Conf. Data Engineering*, 1995, pp 44–51.

[LC93] C. Lee and Z-A. Chang, "Workload Balance and Page Access Scheduling for Parallel Joins in Shared-Nothing Systems", *Procs. IEEE Intl. Conf. Data Engineering*, 1993, pp 411–418.

[LT92] H. Lu and K-L. Tan, "Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems", *Procs. Intl. Conf. on Extending Data Base Technology*, 1992, pp 357–372.

[LT94] H. Lu and K-L. Tan, "Load-Balanced Join Processing in Shared-Nothing Systems", *Journal of Parallel and Distributed Computing* 23, 1994, pp 382–398.

[LV95] S. Lifschitz and V. Vianu, "A Probabilistic View of Datalog Parallelization", *Procs. Intl. Conf. on Database Theory*, 1995, pp 294–307. (extended version to appear in *Theoretical Computer Science*)

[MPI95] Message-Passing Interface Forum "MPI: A Message-Passing Interface Standard", University of Tennessee, 1995.

[Omi91] E. Omiecinski, "Performance Analysis of a Load-balancing Relational Hash Join ALgorithm for a Shared-memory Multiprocessor" *Procs. Intl. Conf. Very Large Data Bases*, 1991, pp 375–385.

[Ram95] R. Ramakrishnan, editor, *Applications of Logic Databases*, Kluwer Academic Publishers, 1995.

[Tsu91] S. Tsur, "Deductive Databases in Action", *Procs. ACM Symp. on Principles of Database Systems*, 1991, pp 142–153.

[WDJ91] C.B. Walton, A.G. Dale and R.M. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", *Procs. Intl. Conf. Very Large Data Bases*, 1991, pp 537–548.

[WDY+94] J.L. Wolf, D.M. Dias, P.S. Yu and J. Turek, "New Algorithms for Parallelizing Relational Database Joins in the Presence of Data Skew", *IEEE Transactions on Knowledge and Data Engineering* 6(6), 1994, pp 990–997.

[WO93] O. Wolfson and A. Ozeri, "Parallel and distributed processing of rules by data-reduction", *IEEE Transactions on Knowledge and Data Engineering* 5(3), 1993, pp 523–530.

[ZJM94] X. Zhao, R.G. Johnson and N.J. Martin, "DBJ - A Dynamic Balancing Hash Join Algorithm in Multiprocessor Database Systems", *Information Systems* 19(1), 1994, pp 89–100.

[ZWC95] W. Zhang, K. Wang and S-C. Chau, "Data Partition and parallel evaluation of datalog programs", *IEEE Transactions on Knowledge and Data Engineering* 7(1), 1995, pp 163–176.

[ZZO94] X. Zhou, Y. Zhang and M.E. Orlowska, "A New Fragmentation Scheme for Recursive Query Processing", *Data and Knowledge Engineering* 13, 1994, pp 177–192.