Concurrent Smart Evaluation of Datalog Queries *

J. F. Aldana, J. M. Troya
Depto. de Lenguajes y Ciencias de la Computación.
Universidad de Málaga.
E.T.S.I Informática. E-29071 Málaga. (SPAIN).
e-mail: {jfam, troya}@lcc.uma.es
Phone: +34 5 2132813 Fax: +34 5 2131397

Abstract

A substantial effort has been made in the development of efficient algorithms for (both sequential and parallel) Datalog program evaluation. In this paper we discuss a Dataflow evaluation model and we show how standard algorithms for the bottom-up evaluation of Datalog queries can be significantly improved by means of enhancing the concurrency degree with a concurrent Dataflow evaluation model. We present a concurrent Dataflow algorithm for Datalog query evaluation and we evaluate a multithreaded implementation of it showing how, indeed in the mono-processor case, improvement is important. This algorithm can be evaluated in a multiprocessor without any change. Only the load balancing problem must be addressed and we think that this parallelisation scheme is more general and flexible than previously proposed ones. We also propose to extend semi-naive evaluation algorithm by performing elimination of duplicate facts at relational algebra operation level instead of making that at intensional predicate level. Furthermore, we think that the Dataflow computational model is quite adequate for Distributed Deductive Databases (DDD), like the one we can find in the WWW, that is because query evaluation in such a distributed database environment should be able to deal with large data flows originated in the various sites which participates in the DDD.
Keywords: concurrent/parallel/distributed dataflow evaluation of Datalog queries.

1 Introduction

As set oriented processing is a very desirable characteristic for a database query language, it is commonly assumed that deductive databases, defined by Datalog [1] [2] [3] programs, are usually evaluated bottom-up. Syntactically, a Datalog program P is a finite set of Horn Clauses that define predicates. The predicate alphabet Pred is assumed to be partitioned into two sets; Extensional Database (EDB) and Intensional Database (IDB). EDB elements denote extensionally defined factual predicates. The elements of the IDB denote intensionally defined predicates that correspond to relations defined by Datalog rules. The semantics of a Datalog program is given by means of its least Herbrand model.

Substantial research has been focused on pure evaluation and optimisation techniques for the sequential evaluation of Datalog programs [1] [2] as well as on parallel evaluation as

*This work was funded by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0930-C02-01.

a way of improving performance, which is particularly important in deductive databases as they require very expensive computational processes.

A basic bottom-up approach that uses the logical rules as production rules in Datalog programs, and a naive fixed point iteration is less efficient than a top-down approach. This is because of considerable redundant information being computed in every iteration. A semi-naive [2] algorithm is assumed in the bottom-up evaluation process. This uses incremental variables, that avoid having to compute redundant information, in order to perform the evaluation efficiently. Computing redundant information is not the sole difference between forward chaining and backward chaining approaches. Top-down approaches use an evaluation mechanism that is guided by the goal, avoiding computing unnecessary information. In order to perform an efficient bottom-up computation, the information that is present in the goal that is to be evaluated by means of its bounded variables, must be used to optimize the evaluation process. Taking into account these two sources of inefficiency, bottom-up is never much worse than top-down and sometimes it can be much better [18].

In this paper we show how standard algorithms for the bottom-up evaluation of Datalog queries can be improved in two ways: avoiding some redundant computation that is still present in semi-naive algorithm and enhancing the concurrency degree in this algorithm. We present a new concurrent algorithm for Datalog queries evaluation. The basis for this algorithm is a data flow computational model (Dataflow Datalog: D^2) which has very interesting features in concurrent, parallel and distributed Datalog queries evaluation.

First, in the monoprocessor case, the algorithm, being concurrent, evaluates queries more efficiently by overlapping synchronous I/O and computation. A multithreaded implementation has been developed and evaluated for these algorithms which shows how, indeed in this monoprocessor case, a concurrent algorithm can lead to significant improvement in the evaluation performance.

Second, this kind of concurrent (dataflow) algorithm, which also runs on a multiprocessor computer, allows fine grain parallelism to be obtained for which an easy control of the size of the parallel grain is possible and which can achieve more intra-query parallelism than previously proposed parallelisation schemes for Datalog. If we are to evaluate the concurrent algorithm in parallel we must cope with the load balancing problem to efficiently speed-up the query evaluation in a multiprocessor. Furthermore, the data flow model allows an exploitation of parallelism which is transparent to the language. A parallel multithreaded implementation for this algorithm has been developed and some results have been obtained evaluating its properties (speed-up, scale-up, memory consumption, etc.). We have also designed and developed a load balancing algorithm which includes a cost model of the parallel dataflow evaluation algorithm.

Third, if we think of a distributed database environment, a dataflow evaluation algorithm is more flexible regarding database fragmentation as is discussed below. This kind of algorithm is able to deal more naturally than non-dataflow evaluation algorithms with the large and slow data flows that are generated by querying a database distributed over the internet. Furthermore, the incremental nature of the pipeline computation allows an early generation of answers even if the query is not completely computed. All these features could make the dataflow evaluation model a nice candidate for Distributed Deductive Database (DDD) applications as are search engines for WWW.

2 DataFlow Datalog: D^2

Looking at the Semi-Naive (SN) algorithm for the Bottom-Up evaluation of Datalog queries (see figure 1) we can see that it is easy to think of several improvements to make it more efficient. For example it is possible to evaluate the relational algebra equations obtained from the IDB (the rules in the Datalog program) without any order in each algorithm's iteration. We can go farther and evaluate concurrently instead of in sequence all the expression in each algorithm's iteration (see figure 1), if we assign a thread to the evaluation of each one of the algebraic equations in order to overlap computation and synchronous I/O. Such an multithreaded algorithm would also run in a multiprocessor without any change. Only load balancing must be considered and threads should be assigned to processors.

Unfortunately, if we are thinking about a parallel evaluation of the Datalog queries this kind of algorithm imposes very strong synchronisation conditions between the threads: the iteration concept and the termination detection. On the other hand, the whole algebraic equation (the algebraic version of a predicate) doesn't look to be the better functional unit for decomposition in order to obtain the adequate grain size. Therefore a more sophisticated transformation of the algorithm is needed to get an adequate concurrency level to be efficiently put to use in a parallel evaluation.

```
ALGORITHM CONCURRENT GENERAL SEMI-NAIVE
    INPUT: A system of algebraic equations Σ, each linear in its left hand side variable,
           and an extensional database
OUTPUT: The values of the variable relations R1,...,Rn.
METHOD:
        FOR i:= 1 TO n DO Di := Ø;
        FOR i:= 1 TO n DO Ri := Ø;
        REPEAT
                cond:=true;
                FOR i:= 1 TO n DO CONCURRENTLY
                BEGIN
                        Di := Ei[R1,...,Di,...,Rn]-Ri;
                        Ri := Di ∪ Ri;
                        IF Di ≠ Ø THEN cond := false;
                END;
        UNTIL cond;
        FOR i:= 1 TO n DO OUTPUT(Ri).
ENDMETHOD
```

Figure 1: A Concurrent Semi-Naive algorithm for the evaluation of Datalog queries.

With parallel evaluation in mind, the first issue to cope with is how to decompose a Datalog program into functional units (tasks) which will be assigned to processes or threads. All the functional units assigned to the same processor will constitute the only one multithreaded process in that processor which will perform a part of the parallel/distributed evaluation algorithm. Relational algebra's operators don't appear to be a bad option for being the functional units. Firstly, translation from a Datalog program to an algebraic program is well defined [1] [2] [3]. Secondly, there is a significant amount of related work in the context of (parallel) relational technology and it is well known how these operators can be parallelised [19] and therefore the size of the parallel grain can be easily controlled. Thirdly, these operators, being monotonic [1] [3] can be evaluated incrementally allowing pipeline parallelism.

All these characteristics appear to point to a parallel dataflow evaluation model in

which all intra-query, intra-operator and pipeline parallelism can be obtained. Neither parallel evaluation of Datalog programs or dataflow parallelism (in relational databases) are new ideas but parallel dataflow evaluation of Datalog is and we think that it is far better and general than other techniques for parallel evaluation of Datalog.

2.1 Dataflow Evaluation Model and Algorithms

Some of the more interesting characteristics of the dataflow model are that the final or intermediate results are passing as data symbols between the operations; the concept of shared memory which is implicit in the traditional notion of variables does not exist; the program sequence is only restricted by data dependencies between operations. The operations can be executed immediately if their operands are (partially) available. This means that many instructions can be processed simultaneously and asynchronously. Dataflow computations are purely functional and sideways effect free. All these characteristics make a dataflow model adequate for parallel and distributed implementation. In D^2 parallelism is completely transparent to the language.

A dataflow graph is a directed graph whose nodes are the dataflow instructions and whose arrows send data symbols between instructions. Each instruction has an operator, one or two operands and one or more destinations to which the results will be sent. The dataflow graph shows the sequencing constraints (data dependencies) between the instructions. We will use these graphs to describe the D^2 programs that we are to evaluate. The D^2 programs can be recursive thus dataflow graphs can be cyclic. More concurrency can be obtained from cyclic graphs (that is from iterative programs) by unfolding the iterations [20]. We are considering an event-driven [22] [23] approach with only two levels of abstraction. The first one is defined at a relational algebra operators level and the second one at a relational algebra intra-operation (parallel implementation) level. The second level is defined to allow parallelization of heavy instructions (like Joins, etc.) and to facilitate load balancing.

Let us start by considering the dataflow parallel evaluation of Datalog queries by introducing a dataflow version of Naive algorithm. Note that, at the moment, we are ignoring problems like optimising by rewritten and load balancing. In a dataflow evaluation we must consider the decomposition of the program into functional units (tasks). We want to obtain a set of concurrent operations, purely functional (without side effects) and amenable for a parallel evaluation with the only restriction of dependencies between data used by the operations. The well known equivalence between logical and algebraical approaches to Datalog evaluation [1] [2] [3] helps in the decomposing of the Datalog program as relational algebra operators are the functional units.

The Datalog program LP is translated to an equivalent algebraic program AP, then AP is compiled onto a dataflow graph DG in which nodes are relational algebra operators and arcs express data dependencies between operations. Finally DG is evaluated concurrently (in a monoprocessor) or in parallel (in a multiprocessor). The nodes in the dataflow graph can be: relational algebra operators, extensional database relations and intensional database relations. Concurrency between operations in the dataflow graph is only limited by data dependencies. Therefore a great amount of concurrency can be obtained. Firstly, all the operations can be evaluated concurrently (in parallel) as soon as they have (a part of) its input arguments, and each operation can be evaluated incrementally. Secondly, the relational algebra operators can also be parallelised. This increases the concurrency degree and helps the load balancing during parallel evaluation. The figure 2 shows a

Datalog program, its equivalent algebraic equations and the dataflow graph associated with it. The topology of DG is not constant. Part of it may be implicated in a recursion cycle and part of it not. As an example the figure 2 shows DG after the first iteration of the algorithm. Note that the difference operator, as it is not monotonic, separates DG in two subgraphs.

Thus the Datalog program LP is compiled onto a dataflow graph DG from which a concurrent executable program E is generated. In this concurrent program each node in DG is assigned to a task. If the program is going to be evaluated in a monoprocessor each of the tasks will be assigned to a thread of the only one process that will perform the query evaluation. If we are in a multiprocessor (parallel or distributed evaluation) a process is placed at each processor which comprises the set of tasks assigned to the processor and each task is assigned to a thread in that process. The threads communicate by message passing but, if they are in the same processor, communication will be as efficient as shared memory.

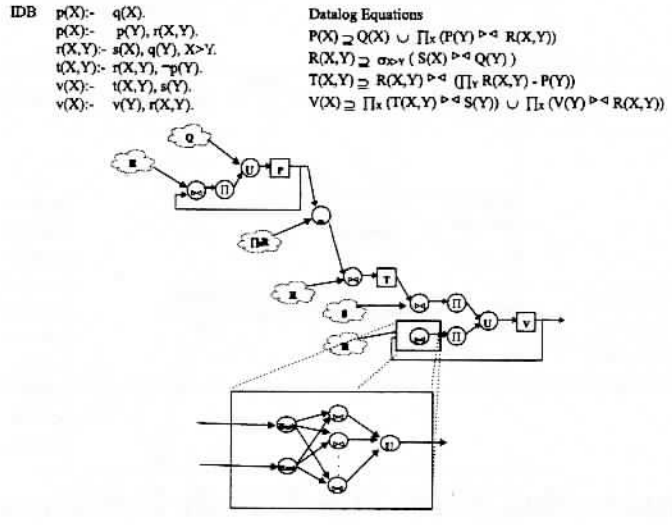

Figure 2: A Datalog program, its Datalog Equations and the Dataflow Graph associated with it (after first iteration).

It is easy to realise that such an algorithm has a concurrency degree much greater than the Naive one. Anyway, this algorithm still has many problems: it computes redundant information like Naive does and it still imposes very strong synchronisation conditions between the threads as a consequence of the iteration concept and the termination detection. The first problem can be solved by performing a dataflow Semi-Naive evaluation. This evaluation is identical to the previously described but the incremental relation concept is introduced and thus the translation from LP to AL is slightly different. This fairly solves the problem of redundant computation but much more redundancy can be removed from computation. furthermore, more concurrency can still be enhanced in the algorithm because as D^2 programs can be recursive and thus dataflow graphs can be cyclic maintaining the iteration concept in the dataflow evaluation could seriously shrink concurrency. Therefore, the concept of iteration should be eliminated.

More redundancy can be removed by performing duplicate tuple elimination at relational operator level instead at intensional predicate level as semi-naive algorithm does. Our implementation of relational algebra operators keeps in the memory all the facts produced by the operation to perform the duplicate elimination (this algorithm is call Smart in Fig.3). Smart algorithm performs strictly less redundant computation than semi-naive one outperforming it even when evaluating in a monoprocessor (see Fig.4 and Fig.5).

In addition, as operations keeps memory of produced facts it is possible to evaluate the Datalog program completely asynchronously (in just one iteration). This is what algorithm ASN does and it is a consequence of the monotonicity property of relational algebra operators. In [25] there is a more in depth discussion of algorithm ASN. This algorithm has high concurrency degree and a good performance. There is no need to use incremental relations in algorithm ASN because each process keeps all the facts produced by it in memory and only produces really new facts. Therefore the dataflow graph for ASN evaluation is the same used for Naive one.

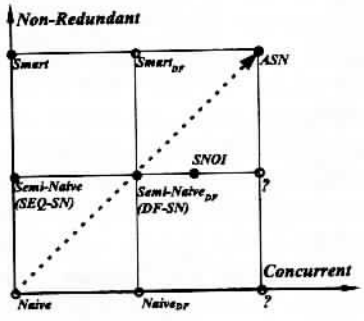

Figure 3: Improvements in Datalog evaluation made by ASN algorithm.

3 Discussion and some Results on Performance

The algorithm previously described had been implemented in C++. A threads library complying Pthreads standard was used and inter-process communication was implemented by means of the PVM library. Some preliminary results have been obtained on it evaluation.

3.1 Concurrent Dataflow Evaluation of Datalog Queries

Even in the monoprocessor case, the algorithm ASN performs quite differently from SN one. This is, in part, because of its different concurrency degree and therefore because of its capabilities of overlapping synchronous I/O and computations; and, in part, because of their different intrinsic efficiency.

The firts analysis shows that ASN algorithm can be up to three times faster than SN algorithm in a monoprocessor evaluation. In Figures 4 and 5 we show monoprocessor performance and memory consumption for these algorithms. It is easy to see that ASN (Smart because we are not considering concurrency) algorithm is not only consistently faster than the other ones but also less memory consuming. As we discussed before, in

algorithm ASN duplicate tuple elimination is performed at relational operator level. This reduces the generation of duplicate tuples (see Fig.5), increasing algorithm's performance. Note that when evaluating in a monoprocessor performance results for algorithms in the same horizontal line in Fig.3 are very similar. Also note that Fig.4 shows only CPU cost and that we are not considering the effect of (overlapped) I/O when comparing with semi-naïve algorithm. Graphs used in evaluation are described in Table 1.

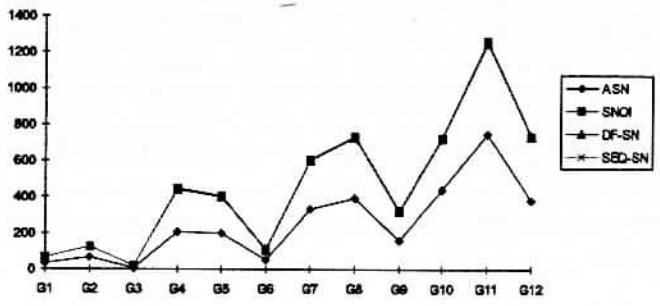

Figure 4: Comparison of the time consumed by the several algorithms during the evaluation of the closure of graphs G1 to G12 (without I/O).

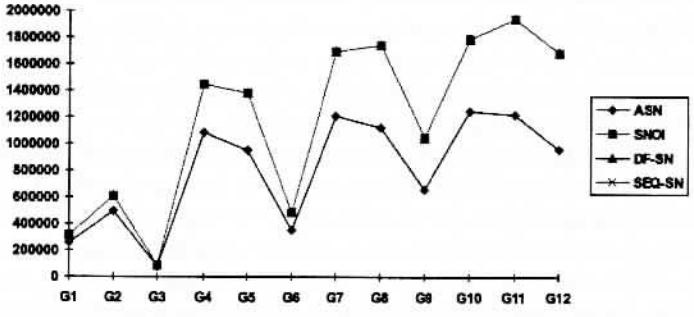

Figure 5: Comparison of the number of tuples generated by the several algorithms during the evaluation of the closure of graphs G1 to G12.

3.2 Parallel Evaluation of Datalog Queries

The exploitation of parallelism in a database system is different from its use in general purpose computer systems. In a database system the use of parallelism is strongly determined by the communication between the nodes of the system. The execution of a query in a parallel database system requires Data messages which are used to transport database data from one node to another and Control messages which are used to control and synchronize the query execution and data transports. The number of Control messages can be reduced in a data flow approach. Distributed control helps in avoiding control bottleneck and thus the system scales up better. Data flow is a natural way to exploit pipelining and therefore offers the opportunity to reduce a communication bottleneck further by spreading the network load over the time. On the other hand data

Graph Name	Out Degree	Generation Locality	Arc Number	Max. Level Node	Graph Height	Graph Width	Closure Size
G1	2	8	1368	109	32	42	64,613
G2	2	80	1590	46	18	88	122,151
G3	2	800	1494	20	6	249	19,356
G4	4	8	2400	353	152	15	269,675
G5	4	80	3136	84	38	82	235,639
G6	4	800	3255	35	19	171	86,565
G7	8	8	3684	497	227	16	298,984
G8	8	80	5917	144	73	81	276,163
G9	8	800	6273	61	40	156	160,104
G10	16	8	4866	629	303	16	309,520
G11	16	80	11093	210	105	105	297,945
G12	16	800	11983	97	64	187	231,836

Table 1: Description of the Graphs used in Performance Evaluation.

flow means more asynchronous activity, more competition for a processor and a higher protocol complexity.

3.2.1 Related Work on Parallel Evaluation of Datalog Queries

Dataflow parallelism has been studied and applied of relational databases [19] [24]. Relational algebra is a closed algebra of set oriented, monotone operators. This fact makes it an ideal candidate for parallel query execution. One way to obtain a high degree of parallelism in query evaluation consists of partitioning data so that a relational operation could be decomposed into several other independent ones, each one working in a different part of the input data. On the other hand relational queries can be executed in a dataflow way using the output of an operation as another one's input and thus obtaining what is called pipeline parallelism or program parallelism.

The exploitation of dataflow parallelism in relational databases is constrained by three facts: first, the dataflow graph for relational queries hardly allows a large sequence of operations; second, some operators in relational databases, like sort and aggregation, don't produce any tuple until they have consumed its entire input; and third, usually there is an operation which is dominant in cost in a relational query.

The first and second assertions are no longer true in the case of Datalog queries. The third one points to a problem which is addressed in D^2 in two ways. In the first way, a hierarchical decomposition of the Datalog program is made and relational operations are parallelised according to its computational costs. Therefore nodes in the dataflow graph are at intra-operation level. In the second way, a load balancing algorithm has been developed which is based on a predictive cost model and which makes process to processor mapping.

On the other hand, almost all approaches to parallel evaluation of Datalog programs, and related effort in transitive closure paradigm in the algebraic context, are aimed at obtaining coarse-grain parallelism [5-16]. These can be classified as: Program Partition, Data Partition, and mixed approaches.

Program partition schemes consider the fragmentation of the IDB into loosely coupled predicate subsets. These somewhat independent predicate sets will be assigned to the different processors. This approach also implies EDB replication. Computational load is distributed between the processors because each of them evaluates a smaller program. The need to replicate EDB not only makes this technique have the problems associated with program partitioning but also makes it generate, usually, a much greater amount of communication because what we are really partitioning is the logic program.

Data partition makes a partitioning (maybe a replication) of EDB relations and replicates rules which are slightly altered to avoid redundant processing and to include communication primitives. This kind of technique allows the Datalog program parallelization to have a reduced (for some kind of programs null) communication cost but on the other hand requires a high data replication.

Mixed approaches combine both data and program partitioning to perform a more flexible fragmentation. In any case, as we'll discuss later, in practice, EDB may end up being replicated in each node and this is not a very good option for database taking into account many considerations such as data size, redundancy, mutual consistency, updates, etc. In all three cases the evaluation algorithm is the same, perhaps Semi-Naive one, with communication capabilities to send tuples generated in each iteration to other nodes in which identical algorithms are running. All these algorithms are cooperating to evaluate the query. The original logic program may be partitioned (both EDB and IDB) and perhaps modified with some hash function to avoid computing the same results in several nodes.

3.2.2 Parallel Dataflow Evaluation of Datalog queries

If we take a look at the previous effort in parallelizing Datalog queries evaluation we can see that all these methods are based on three ideas: algorithm replication, database partition or replication and communication. The first difference that we can find when we compare D^2 and other Datalog parallelization schemes is that while this last one replicates a somewhat altered version of the Semi-Naive algorithm, in D^2 we have a really distributed evaluation algorithm.

The second big difference is about the data partition (data declustering) that is necessary for the parallel evaluation. Traditional Datalog parallelization schemes make data partition according to syntactic properties of the program. The method which partitions the EDB does so by means of an homogeneous hash function generating equal size fragments. This is to equally divide computational load between processors making them evaluate the same logic program, slightly transformed with hashing functions and communication primitives, but with less data.

This produces several problems related to how data is partitioned. As data declustering is very important because it determinates how the load is distributed between processors and how much communication is generated, this aspect will greatly influence the performance. In very simple cases it is possible to make the partition without any problem thus obtaining very efficient transformed programs with a reduced (maybe null) redundancy degree and a reasonable communication amount. But a clear partition scheme is only always possible for sirups (single rule programs). If we try to partition a multipredicate Datalog program we'll find that partitioning is not so clearly defined as we thought. We can think about making some kind of derived fragmentation of the EDB predicates according to the dependency graph of the Datalog program and some other relations (EDB predicates) primarily fragmented but the same predicate can be used in the definition of several other ones.

As a consequence of this, to set up a partition of the EDB relations in order to perform parallel query processing which takes into account all the predicate definitions in which any relation participates could be almost impossible. The practical consequence is that we'll need to replicate the database in every node. This, which is not very realistic in a database, doesn't imply negative consequences from the computational cost point of view because it is possible to select which part of each relation we want to process in each

Load Balancing	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
Heuristic	372	196	181	154	148	133	166	117
Round Robin	372	337	249	262	187	201	169	178
Random	372	377	285	220	193	275	191	149

Table 2: Time for the parallel evaluation of a complex query over graph G3 with one to eight processors.

node. From the communication point of view this doesn't increase the communication cost.

In D^2 computational efficiency of the parallel evaluation is not so strongly dependent on the Datalog program topology as data partitioning based algorithms are. Indeed, the relations of the EDB can be arbitrarily distributed and/or fragmented and this fact can be solved in D^2 by means of the load balancing algorithm. On the other hand, load balancing is a main issue concerning query evaluation in D^2 because the scheduling/load balancing of the logic tasks over the available resources in an event-driven computation has been shown to be an NP-complete problem. Thus, it is necessary to use heuristics operation scheduling and load balancing. Operation to processor mapping is based on the estimation of operation complexity. If these estimations are accurate enough heuristics will result in a near optimal scheduling.

The load balancing algorithm performs a smart clustering of the processes in the dataflow graph in order to assign equal size clusters to each processor. Clusters are built regarding inter process dependencies and interferences. Also some graph restructuration primitives like hashing and semijoin are used in order to facilitate load balancing. A more detailed discussion of the load balancing algorithm can be found in [25].

Table 2 shows some very preliminary results of the algorithm's performance. It shows the time (in seconds) taken by the ASN algorithm to evaluate the same query with one to eight processors. Times are taken in a cluster of SuperSparc-Solaris workstations. No data declustering or partitioning is considered in this performance analysis and all the EDB relations are assigned to the same processor (this is a very limitating constraint). No optimisation algorithm is applied and three load balancing algorithm are considered: heuristic, round robin and random assignment of processes. These are very preliminary results and we must still further improve load balancing algorithm and evaluate the implications of the EDB location in the algorithms performace.

3.3 Distributed Dataflow Evaluation of Datalog Queries

If we consider the distributed database case, syntactic EDB partitioning techniques based on program topology don't appear to be very adequate. In a distributed database information is fragmented considering the user's information requirements, locality and frequency of data accesses, hardware parameters, etc. It is easy to realise that syntactic partitioning (data declustering) is not compatible with distributed query processing.

Dataflow evaluation, on the other hand, does not have such a strong dependence on data partitioning. In D^2 the properties of the program w.r.t. parallelization are not so dependent on the kind of the program we are considering: depency graph topology or the number of predicates. Furthermore, parallelism is transparent to the language. Therefore D^2 can be used for evaluating Datalog queries over an existing distributed database or over multiple (maybe distributed) databases.

We think that this kind of algorithm based on a Dataflow computational model allows an arbitrary distribution of EDBs and therefore is more flexible regarding database frag-

mentation as is discussed below. Furthermore Dataflow algorithms are able to deal more naturally than non-dataflow evaluation algorithms with the large and slow data flows that are generated by querying a database distributed over the internet. The incremental nature of the computational model allows an early generation of answers even if the query is not completely computed. All these features could make the dataflow evaluation model a nice candidate for Distributed Deductive Databases and for some kinds of applications like they could be search engines for WWW with sophisticated query capabilities.

4 Conclusions and Future Work

We have presented and discussed a new concurrent dataflow algorithm for the parallel/distributed bottom-up evaluation of Datalog queries: Asynchronous Semi-Naive (ASN). In this algorithm a high concurrency degree is achieved.

ASN algorithm perform strictly less redundant computation than semi-naive one. This lead to a significant improvement in evaluation outperforming traditional algorithms even when evaluating in a monoprocessor.

The Dataflow computational model allows more intra-query parallelism in the Datalog program evaluation process to be obtained. Using this model, a concurrent (multithreaded) implementation of the algorithm has been developed and evaluated. This algorithm can be evaluated in multiprocessors obtaining both intra-query and intra-operation parallelism in Datalog query evaluation. Currently, a parallel version of the system is being evaluated both in loosely and tightly coupled distributed memory multiprocessors. Furthermore, we think that the dataflow computational model is well suited for query evaluation in Distributed Deductive Databases.

Some algorithms have been developed, and are being evaluated, to cope with the load balancing problem. These include a predictive cost model for the whole query evaluation system. We have developed an specialised algorithm to optimise the data flow evaluation process by filtering data flows according to the query bindings. It allows many kinds of integrity constraints to be propagated. A comparison of the optimisation algorithm with other ones is being developed.

References

[1] Ullman, "Principles of Database and Knowledge-Base Systems". Vol. 2. Computer Science Press, New York. 1989.

[2] Ceri, Gottlob and Tanca "Logic Programming and Databases". Surveys in Computer Science. Springer Verlag. 1990.

[3] Abiteboul, Hull and Vianu , "Fundations of Databases". Addison-Wesley Publishing Company, 1995.

[4] Bancilhon, "Magic Sets and Other Strange Ways to Implement Logic Programs". In Proceedings of the 5th ACM Symposium on PODS. 1986.

[5] Hulin, "Parallel Processing of Recursive Queries in Distributed Architectures". In Proc. 15th Int. Conf. on Very Large Databases, Amsterdam, the Netherlands, 1989, pp. 87-96.

[6] Van Gelder, "A Message Passing Framework for Logic Query Evaluation". In Proc. of ACM SIGMOD Conf., pp. 155-165, 1986.

[7] Cheiney, "A Parallel Strategy for Transitive Closure Using Double Hash-Based Clustering". In Proc 16th Int, Conf. on Very Large Databases, Brisbane, Australia, Aug. 1990, pp. 347-358.

[8] Valduriez and Khoshafian, "Parallel Evaluation of the Transitive Closure of a Relation". In Int. Journal of Parallel Programming, 17:1 Feb. 1988.

[9] Houstma, "Parallel Hierarchical Evaluation of Transitive Closure Queries". In Proc. 1st Int. Conf. on Parallel and Distributed Information Systems, IEEE Computer Society Press, 1991, pp. 130-137.

[10] Dong, "A Framework for the Parallel Processing of Datalog Queries" Proc. of ACM SIGMOD Conf., pp. 26-35, 1989.

[11] Wolfson and Silberschatz, "Distributed Processing of Logic Programs". In Proc. of ACM SIGMOD Conf., pp. 329-336, 1988.

[12] Wolfson, "Sharing the Load of Logic Programs Evaluation". In Proc. of the 1st Int. Symp. on Databases in Parallel and Distributed Systems, pp. 45-55, 1988.

[13] Cohen and Wolfson, "Why a Single Parallelization Strategy is not Enough in Knowledge Bases". Proc. of the 8th Symp. on Principles of Database Systems, pp. 200-216, 1989.

[14] Ganguly, "On Distributed Processability of Datalog Queries by Decomposing Databases" Proc. of ACM SIGMOD Conf., pp. 143-152, 1990.

[15] Wolfson and Ozeri, "A New Paradigm for Parallel and Distributed Rule-Processing". In Proc. of ACM SIGMOD Conf., pp. 133-142, 1990.

[16] Shao Bell and Hull, "Combining Rule Decomposition and Data Partition in Parallel Datalog Program Processing". In Proc. 1st Int. Conf. on Parallel and Distributed Information Systems, IEEE Computer Society Press, 1991, pp. 106-115.

[17] Apt, Blair and Walker "Towards a theory of declarative knowledge" IBM res. Report RC 11681, April 1986.

[18] Ramakrishnan "Top-Down vs, Bottom-Up Revisited" In Proc. of the Intern. Conference of Logic Progremming, 1991, pp. 321-336.

[19] Hongjun Lu, Beng-Chin Ooi and Kian-Lee tan "Query Processing in parallel Relational Database Systems" IEEE Computer Society Press. 1994

[20] Arvind and Gostelow "The U-Interpreter" IEEE Comp., vol. 15, num. 2, Feb 1982, pp.42-50

[21] Gajski et alt. Dependence Driven Computation" Proc. COMCON, Spring., Feb. 1981, pp. 168-172.

[22] Hwang and Su "Priority Scheduling in Event-Driven Data flow Computers" TR-EE 83-86, Prudue Univ., Ind., Dic 1983.

[23] Hwang and Su "Multitask Scheduling in Vector Supercomputers" TR-EE 83-52, Prudue Univ., Ind., Dic 1983.

[24] DeWitt et alt. "The Gamma Databse Machine Project" IEEE Trans. Knowledge and Data Engineering. Vol2, No1, March 1990, pp 44-62.

[25] Aldana "Multithreading, Dataflow and Datalog: D^{2}" Tech. Rep. Dpt Lenguajes y Ciencias de la Computacin. Univ. of Mlaga, Feb. 1996.