Predictive Dynamic Load Balancing of Parallel and Distributed Rule and Query Processing *

Hasanat M. Dewan
Mauricio Hernández
Salvatore J. Stolfo
Jae-Jun Hwang

Department of Computer Science
Columbia University, New York, NY 10027
CUCS-025-94
(This paper appeared in the Proceedings of the 1994 ACM SIGMOD Conference.)

Abstract

Expert Databases are environments that support the processing of rule programs against a disk resident database. They occupy a position intermediate between active and deductive databases, with respect to the level of abstraction of the underlying rule language. The operational semantics of the rule language influences the problem solving strategy, while the architecture of the processing environment determines efficiency and scalability.

In this paper, we present elements of the PARADISER architecture and its kernel rule language, PARULEL. The PARADISER environment provides support for parallel and distributed evaluation of rule programs, as well as static and dynamic load balancing protocols that predictively balance a computation at runtime. This combination of features results in a scalable database rule and complex query processing architecture. We validate our claims by analyzing the performance of the system for two realistic test cases. In particular, we show how the performance of a parallel implementation of transitive closure is significantly improved by predictive dynamic load balancing.

1 Introduction

A considerable body of prior work has been done to achieve high performance, complex query processing over large databases by parallel and distributed processing [3, 18, 21, 9]. We study this problem in the context of rule program processing, in which production rules comprising an expert database application are evaluated against a large database of facts stored in an RDBMS. The condition and action parts of the rules typically translate into complex queries over the underlying database. The operational semantics of the underlying rule language favor an evaluation policy in which all processing sites are synchronized at every inference cycle. However, under a synchronous processing policy, an efficient distributed processing ensemble requires that we achieve a "balanced" computation. In other words, we require uniform completion times for matching the rules in a rule program at every processing cycle to minimize the total amount of idle time over all the sites. It is often stated that asynchronous evaluation schemes are more efficient over synchronous schemes for this very reason. However, asynchrony also leads to database and inference concurrency control problems that we seek to entirely avoid [5].

There is a limited amount of work that addresses the workload partitioning problem for knowledge-base systems. Existing work may be broadly characterized as static, compile time analyses that attempt to intelligently guess about runtime behavior, and to schedule a number of relational queries accordingly. What is needed is a dynamic component to account for the actual time evolution of the underlying database during inference and changes in the computing environment during the lifetime of a particular program, as proposed herein. In addition, we aim to remedy the problems associated with tuple-oriented languages by providing an efficient implementation of a set-oriented rule language. The PARULEL language has parallel, set-oriented execution semantics and a programmable conflict resolution capability through the use of programmer specified metarules, as well as a "context" mechanism to group rules into structured modules for control.

The task partitioning problem is addressed as follows. We use the copy-and-constrain [19] technique as the basic partitioning tool for data reduction, and compute a number of restricted versions of the original rule program that are applied in parallel to reduced databases. To ensure balanced operation over an ensemble of processors, we monitor runtime performance and dynamically adjust the constraints or "restrictions" on the program versions to limit the size of the database selected by a program version during rule processing. The dynamic adjustment of restrictions also requires communication among the sites. The communication requirements may be organized into a number of distinct dynamic load balancing protocols.

Our approach is a combination of the following two techniques:

*This work has been supported in part by the New York State Science and Technology Foundation through the Center for Advanced Technology under contract NYSSTFCU01207901, and in part by NSF CISE grant CDA-90-24735.

- statically computed restrictions on the rules of a rule program for initially partitioning the workload of rule evaluation among an arbitrary number of rule program replicas evaluated at distinct processing sites, and

- *predictive dynamic load balancing* (PDLB) protocols that update and reorganize the distribution of workload at runtime by modifying the restrictions on versions of the rule program.

The techniques are predictive meaning we aim to balance the load of distributed processing based upon estimates of future workload of each program replica and available processing resources. Our techniques utilize "meta-data", i.e., statistics gathered on relations (e.g. cardinality, discrete frequency distributions, etc.), in the algorithms that predict future resource requirements. The evaluation of a PARULEL program is carried out within PARADISER (PARA*llel* and DIS*tributed* E*nvironment* for R*ules*). Both the PARULEL language and the PARADISER architecture have been detailed elsewhere [20, 6, 7]. For completeness, we include brief descriptions in subsequent sections.

The contribution of this paper is a description of the implementation and measured performance of two applications written in PARULEL. Specifically, we demonstrate the effectiveness of an initial implementation of PDLB for two test programs consisting of up to 55 rules and databases consisting of up to 50,000 tuples (in both base and derived relations). One of these programs is a variant of transitive closure. For our test databases, the improvement in parallel match time under PDLB over a purely static partitioning ranged from 10-30% for one application, and 23-50% for the other. Furthermore, the measured overhead of PDLB was shown to be under 1% in most cases. Only one PDLB protocol is presently provided by PARADISER. We claim that the software component that implements the dynamic load balancing feature will be widely applicable to general (complex) relational query processing, of which rule processing is one example, and is applicable to much larger databases.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3 and 4, we briefly describe the essentials of the PARULEL rule language and its supporting environment, PARADISER, including its predictive dynamic load balancing component, which establishes the context for the remainder of the paper. Section 5 introduces two expert database applications written in PARULEL, presents performance data, and demonstrates improved performance due to the predictive dynamic load balancing techniques employed. Section 6 outlines the applicability of our results to the case of balanced evaluation of complex queries in parallel. In Section 7, we summarize our conclusions. (Elements of this paper have appeared elsewhere and are included for completeness wherever necessary.)

2 Related Work

In this work, we depart from the traditional thrust of research in high-performance expert systems, i.e., new and improved match algorithms and extracting rule-level parallelism by exhaustive compile-time analysis. We focus instead on disk-based systems coupled with distributed and parallel processing for scalability.

Related work occurs in the general class of database rule processing systems under the deductive, active, or expert database labels. The work in active databases [11, 12, 21, 23] tends to emphasize rule systems that provide the means for accomplishing internal database tasks such as database constraint maintenance, while deductive databases [14, 17, 22] are suited to building knowledge-based applications that have natural expression in logic. Recent work in expert databases [2, 13] addresses the problem of efficiently determining the set of active (or instantiated) rules using various indexing techniques. However, the associated match algorithm, namely "lazy match", uses a rather restrictive semantics for generating instances. Specifically, only a single "best" instance is generated per cycle, where the notion of "best" is hardwired into the match algorithms. Moreover, issues of scaling by parallel processing are not addressed. Thus, while interesting, this work leaves open many of the most pressing concerns of this paradigm, i.e., the need to support a general operational semantics, and the scaling issues associated with increasing database size. Our work provides for a general programmable operational semantics, and scaling opportunities through distributed and parallel processing. Moreover, we develop measures that enable us to test the improvement in performance in a systematic manner.

Most of the existing work that addresses the dynamic workload partitioning problem for distributed rule or query processing can be characterized as static, compile time analyses that attempt to intelligently guess about runtime behavior. There also exists work on purely runtime scheduling of asynchronous parallel rule firing systems [15], where load balancing is essentially unnecessary. In the latter, asynchronous parallel execution of rules is made possible by a data locking protocol suited to rule processing. There is a centralized lock manager for this purpose. While this allows for asynchronous parallel execution of rules, it does not guarantee serializable programs in the presence of negated condition elements. That burden is shifted to the programmer who must be careful in developing "correct" programs using special constructs that the language provides. This approach differs from ours in that we support a synchronous rule execution policy, where conflicting rule instantiations are filtered using a metarule facility, and the potential imbalance among the processing sites due to synchronous operation is corrected through the predictive dynamic load balancing facility. This last point is the focus of this paper.

3 The PARULEL Language

Here, we review the main features of PARULEL to lay the groundwork for our subsequent discussion.

3.1 Syntax

PARULEL syntax is derived from OPS5, while its operational semantics resemble those of Datalog$^{\neg *}$ [1, 5]. (As in Datalog$^{\neg *}$, PARULEL is capable of set-oriented updates involving all rule actions that do not conflict according to certain criteria. The relationship between PARULEL and Datalog$^{\neg *}$ is discussed at length in [5]).

PARULEL programs consist of "object-level" production rules, "metarules" for a programmable operational semantics, and a database of facts. Object-level rules consist of a conjunctive *left-hand side* (LHS) of conditions which are matched against the database of facts forming *rule instances*, and a *right-hand side* (RHS) of actions to be performed on the database. The LHS corresponds to rule bodies of Datalog-style languages, and the RHS corresponds to rule heads. The object-level rules encode the basic problem solving knowledge. The language also supports arbitrary arithmetic predicates on the left hand side of rules, increasing expressiveness. For managing large applications, PARULEL provides a "context" construct whereby all object level rules relevant to a particular stage of problem solving are grouped together by annotating rule names by the name of the context they belong to. This familiar notion allows rule programs to be structured in a natural way. The user specifies a directed "context switch graph" (CSG) (possibly cyclic), which delineates the sequencing of contexts. The runtime system processes the rules in a given context until fixpoint, and then moves on to the next context as determined by the CSG (explicit jumps between contexts is possible under program control). Global fixpoint is detected when, during a full traversal of the CSG, no context produces rule instances. The metarules are written separately for each context and only applicable metarules are used at every stage when filtering the instances produced by matching rules in a particular context. The metarules are also written in production rule style, with rule names forming the condition classes.

3.2 Operational Semantics

The "programmable" conflict resolution strategy is realized via meta-level rules, that express domain-dependent relationships among the rule instantiations in the conflict set at any given cycle. These metarules specify what specific types of interactions among rule instances indicate a conflict. The action of these metarules is to remove, or *redact*, one or more of the conflicting rule instances from the conflict set. The post-redaction conflict set is considered to be conflict-free, and can be fired concurrently, realizing set-oriented database updates. The metarules provide a mechanism for resolving inconsistencies implicit in the original rule set as execution progresses. The operational semantics or conflict resolution strategies can themselves be declaratively specified, separating the logic of a program from its control. The generalized operational semantics of PARULEL is displayed in Figure 1.

PARULEL Operational Semantics

MARK the root node of the CSG as **ACTIVE**.
WHILE there are **ACTIVE** contexts DO *in parallel*
BEGIN
  $\Lambda \leftarrow$ set of **ACTIVE** contexts.
  For every $\mathcal{C} \in \Lambda$ DO *in parallel*
  BEGIN
    1. MATCH all rules in $\mathcal{C}$ against the database *in parallel.*

    2. IF there are no instances, MARK $\mathcal{C}$ as **INACTIVE**,
       MARK all of its successors in the CSG as **ACTIVE**,
       and PLACE these in the set of **ACTIVE** contexts, Λ.

       OTHERWISE, FILTER the generated instances with
       the appropriate metarules for this context.

    3. EXECUTE the RHS operations of the filtered set of
       instances from all **ACTIVE** contexts, *in parallel.*
  END
END

Figure 1: Operational Semantics of PARULEL

4 PARADISER Components

The operational semantics of PARULEL removes the inherent sequentiality of traditional rule languages by adopting set oriented database updates. Moreover, for efficient operation of any distributed processing scheme, issues of data (load) partitioning, both static and dynamic, must be addressed in a comprehensive manner. The PARADISER environment provides these facilities for PARULEL programs. The task partitioning and load balancing techniques used in PARADISER has been the subject of other reports [8, 4, 6]. Here we provide an outline appropriate for the present paper.

4.1 Memory Model

The database support for PARULEL programs is provided by an RDBMS, rather than the LISP environment of typical AI rule languages such as OPS5. Within the PARADISER environment, PARULEL rules are compiled into an intermediate form consisting of relational expressions which are processed by a runtime evaluation system that is loosely coupled with the database management system via client server interactions. Thus, PARULEL rules may be processed against a very large store of persistent data that is managed by DBMS storage and access managers. Since both the LHS and the RHS of PARULEL rules may be expressed as relational Select-Project-Join (SPJ) queries, rule matching and rule firing translates into a series of SPJ queries to form rule instances, followed by a series of updates on the database, respectively.

4.2 Rule Matching in Parallel

We use the copy-and-constrain paradigm [19, 25] as the basic mechanism for matching rules in parallel. In this paradigm, rules are replicated with additional constraints attached to each copy. Such restricted rules can be matched in parallel, thus providing a speedup [16, 6]. To illustrate, let rule r be:

$$r: \ C_1 \wedge C_2 \wedge \cdots \wedge C_n \rightarrow A_1, \cdots, A_m \qquad n, m \geq 1$$

where the $C_i, i = 1 \cdots n$ are conditions *selecting* a subset of the database tuples, and $A_j, j = 1 \cdots m$ comprise a set of m actions (updates to the database). Let $R(C_i)$ denote the tuples selected by C_i. The work, $W(r)$, to process r is bounded by the size of the cross product of the tuples selected on the LHS of r:

$$W(r) \leq \ \mid R(C_1) \mid \times \mid R(C_2) \mid \times \cdots \times \mid R(C_n) \mid$$

Suppose we choose to copy-and-constrain rule r on condition C_i to produce k new conditions $\{C_i^1, C_i^2, \cdots, C_i^k\}$, and k new replicas of r, $\{r^1, r^2, \cdots, r^k\}$, where each replica has C_i replaced by C_i^l, $l = 1 \cdots k$, on the LHS. If the new conditions are chosen such that $\mid R(C_i^l) \mid \approx \frac{|R(C_i)|}{k}$, $l = 1 \cdots k$, $\bigcap_{l=1}^k R(C_i^l) = \emptyset$, and $\bigcup_{l=1}^k R(C_i^l) = R(C_i)$, then

$$W(r^1) \approx \cdots \approx W(r^k) \approx \mid R(C_1) \mid \times \cdots \times (\tfrac{|R(C_i)|}{k}) \times \cdots \times \mid R(C_n) \mid \approx \tfrac{W(r)}{k}$$

For appropriately constructed restriction conditions on a uniformly distributed *restriction attribute* (RA), each of the k replicas require $\frac{1}{k}th$ the amount of work as the original rule r to process, forming the same set of instantiations. If the replicas are processed in parallel, the evaluation of r will be sped up by a factor of k. As an intuitive example, consider a relation AUTO, describing cars with only five colors (`black, white, red, blue, green`). If the colors are uniformly distributed, then copying the relation AUTO to five processors and constraining the color attribute to a distinct color will reduce the effort of processing queries involving AUTO to about a fifth of the original effort involving the unconstrained relation. Note that the restriction predicates defined in terms of the RA produce a logical view of the restricted relation that is similar to a relation fragment in a horizontally fragmented database.

It is possible to estimate workload accurately at compile time from statistics on extensional or input databases. Clearly, this may not be true for derived or intensional databases formed during inference. It is this workload that we wish to balance at runtime in the face of differing processor characteristics, i.e. to adjust (and possibly readjust as necessary) the chosen conditions on the RA that determines equal workload at every site.

It is important to note that we study this problem in the context of a replicated database configuration. Since we replicate the database at every site, the workload at every site to update the database is uniform at every inference cycle. Thus, we can factor out database update in our cost models and in the load balancing algorithms. The use of a replicated database configuration allows us to simplify one dimension of the problem (i.e., dynamic load balancing) in order to study fundamental techniques via the simplicity of copy-and-constraining. For example, redistribution of workload at runtime is implemented solely by passing messages to various sites that simply contain new bounds on the ranges of the constraints on the RA's. On a fully distributed database configuration, the match operation of distributed rule programs must be redesigned entirely, and database update likewise enters into the load balancing techniques significantly. For example, database tuple communication costs must be accounted for as part of the workload partitioning process. We address these issues briefly in Section 6.

4.3 Workload Partitioning by Copy-and-Constrain

In a synchronous parallel rule firing environment, disparities in the processing time among the sites can reduce the efficiency of the ensemble by reducing the overall performance to that of the slowest processor at every cycle. A dynamic component is necessary to account for the actual time evolution of the underlying database during processing, and to account for any changes in the computing environment during the lifetime of a particular program. To summarize, the processing of a rule program may become unbalanced at some cycle due to any of the following reasons.

- heterogeneity among the computing elements of the ensemble,
- data characteristics, such as a skewed data distribution which results in some sites generating larger fragments of the intensional database than others, or
- poor initial static partitions, i.e., initial load optimizers guessed incorrectly.

We thus provide in PARADISER both initial (static) and runtime (dynamic) workload partitioning via the RA and dynamic load balancing methods. If the RA is not uniformly distributed, the effects of data skew will render simple range partitioning ineffective. Moreover, the available processing sites may not have identical resources (especially in heterogeneous or wide-area environments). We represent the processing potential of the available processing sites by a vector of "loading weights". To handle skewed distributions we generalize the copy-and-constrain scheme as follows.

For each rule, a restricted relation and an RA that selects that relation are chosen heuristically [4] at compile time. The database environment is configured automatically by the runtime system to maintain statistics on the RA of each rule by means of appropriately defined triggers. Discrete frequency distributions (DFDs) are maintained by the triggers that are installed for each RA. The *weighted range splitting* (WRS) algorithm, initially reported in [6], embodies a technique for both static and dynamic workload assignment. It is designed to handle skewed distribution of the restriction attribute, as well as providing a means to construct restrictions that take into account the processing potential of each processing site at any given time based upon the vector of loading weights corresponding to the sites. WRS computes the bounds on the value of an RA according to a *weight* that models the desired level of loading due to some rule at some site. WRS utilizes the DFD of an RA to compute the restrictions on any given rule for each site of the ensemble.

The static workload assignment algorithm is called *isoweight copy-and-constrain* (ICC). It uses the WRS technique to compute the initial restrictions by using an initial set of loading weights (which may be either uniform, or determined by an initial probe of the distributed sites). In contrast, the dynamic workload adjustment algorithm, called *dynamic restriction adjustment* (DRA), attempts to adjust the rule restrictions at runtime if the initial processing allocation fails to attain a balanced load. It uses the WRS algorithm and computes the bounds on rule restrictions using a loading schedule that is computed from observed performance of the sites during previous cycles, as well as statistical meta-data on the database.

4.4 Balancing the Distributed Computation by Predictive Dynamic Load Balancing (PDLB) Protocols

The DRA algorithm is applied as part of a predictive dynamic load balancing (PDLB) protocol for detecting and correcting an unbalanced system. The particular PDLB protocol for detecting imbalance and the DRA algorithm for correcting it constitute overhead not paid in the sequential case. The former of these sources deserves attention in that, as the number of sites is scaled up, the communication costs for determining imbalance of the ensemble can display fast growth, and may dominate the overhead costs. Thus, the choice of PDLB protocol is crucial for minimizing this overhead.

In previous work [6], we have shown, through isoefficiency analysis, that the family of PDLB protocols we consider have some members with slow-growing isoefficiency functions. An isoefficiency function shows how the total task size for a distributed computation must grow with the number of processing sites in order to keep the efficiency of the parallel ensemble at a constant value. It is a measure of scalability for parallel algorithms whose overhead increases with the number of processors. A perfect isoefficiency function is linear in the number of processors, which is not attainable in practice. One protocol we have studied is the Global Coordinator (GC) protocol, which has been implemented in the current PARADISER system as a base case, and is outlined below.

GC PDLB Protocol:

1. At every processing cycle, each of the P sites keeps track of its local program and rule completion times. At most $\log(P)$ steps after the last site completes, a distinguished "coordinator" site receives the values of the maximum and minimum completion times (MAX and MIN, respectively) among all the sites using a tournament selection method. The imbalance is defined to be $\mathcal{I} = \mid MAX - MIN \mid$.

2. If $\mathcal{I} > C$ (system-wide threshold), the global coordinator broadcasts a BALANCE message. All sites then send rule and program completion times to the coordinator. For each rule at every site, the coordinator computes adjustments to the restrictions using the DRA algorithm. The DRA algorithm computes the percentage deviation of the completion time for each rule relative to the system average. If the absolute value of the relative deviation exceeds a threshold, then the current loading weight for the site with respect to the rule is adjusted (either increased or decreased, based upon the sign and magnitude of the deviation). The adjusted weights are predictors of future performance of the sites over the short term, and are used to dynamically recompute the restrictions on the rule for each site. The new restrictions are then incorporated into the local program replicas. We mention that initially, all restrictions are computed assuming equal weights for every site. After incorporation of the new restrictions, the coordinator broadcasts a CYCLE message.

   Otherwise, if there is no imbalance, the coordinator simply broadcasts a CYCLE message.

3. On receiving a broadcast CYCLE message, each site begins a new processing cycle, beginning with the match phase.
4. Termination is detected by the coordinator when every site reaches a fixpoint.

The GC protocol has an isoefficiency function of the form:

$$W = f(P) = \lambda P \log(P) + \mu P^2$$

where W and P are total work load and the number of processors, respectively, and λ, μ are constants. Intuitively, this expression indicates that as the number of processors P is scaled up, the efficiency of the distributed processing system can be kept at a fixed value if the total work load grows as the function $f(P)$. The function f is also a measure of the cost of the protocol. To see this, assume that the system runs with no work to be performed in rule matching or database updates. Then the incurred costs arise only from the PDLB protocol, and in the case of the GC protocol, takes the form shown above. If load balancing occurs infrequently, then $\mu \approx 0$, and we get $f(P) \approx \lambda P \log(P)$. In simulations, we have verified that the cost of GC is indeed quite low and its growth rate is approximately $cP \log(P)$, for some constant c. For fast workstation clusters such as the HP9000 cluster, the protocol cost incurred by GC for 64 sites is less than a fraction of a second. This is acceptable when inference cycle times tend to be large by comparison, as is the case when large databases are involved.

A dynamic load balancing protocol is predictive if it attempts to predict unbalanced behavior at runtime and takes preventive remedial action, rather than simply responding to system imbalance as it occurs. In PARADISER, the dynamic load balancing protocols initiate load balancing when at any cycle, completion time disparity between any two sites exceed some threshold. The typical approach to load balancing in distributed computing is based upon demand-driven techniques that use information about the current state of the system: what resources are available, and what resources are requested. They then schedule the resources accordingly to reduce heavy loads (high demands) at certain sites. In PARADISER, if the current system state indicates that an overloaded processing site exists, then we attempt to reduce the load at that site on the next processing cycle by reallocating the entire computation over all sites. During every cycle, statistics on the extensional and intensional relations are updated with the aid of triggers whenever any database table is modified. These statistics are used to estimate the amount of workload for rule processing in future cycles, i.e. how many tuples should be allocated from the restricted relation for matching a given rule at each site, and thus how to compute the restrictions on the RA of each rule for each program replica.

If the dynamic load balancing protocol determines at any cycle that load balancing should be initiated, then the loading weights are adjusted by observing the past performance of each site. New weights are computed as a function of the weights in the k previous cycles, where k is a system tuning parameter. The exact algorithm for weight adjustment has been reported elsewhere [6, 8]. Here we mention that the adjusted loading weights are used as a measure of *predicted future resource requirements* at each site. The actual load balancing phase, embodied in the DRA (Dynamic Restriction Adjustment) algorithm, uses these loading weights to construct new restrictions on the rules of the rule program versions at each site. This dynamic load balancing method attempts to predict future loads based upon the current state and the assumption that the most recently computed loading weights will be stable and will reflect the processing activity in the near future (this assumption is similar to locality principles used in paging algorithms).

This assumption is justified by the following two observations. First, if the load imbalance was due to external events causing processing sites to be overloaded, and if the average duration of such external loads is greater than the basic cycle time, then we presume that they will continue to be loaded again on the next cycle. Second, if the load imbalance at a site was due instead to having spent significantly more than the average completion time at some site in rule processing, then we may presume that the same site is likely to have produced more updates to its logical partition of the restricted relation than the average, and thus will continue to be heavily loaded again. If either or both situations occur, the assumptions will hold, and dynamic load balancing will be effective. If we guess wrong, and the prediction fails, then we rebalance again on the next cycle, repairing the faulty prediction.

In order to improve the predictive properties of the dynamic load balancing protocols, we can additionally use meta-data to predict imminent system imbalance, as opposed to actually detecting imbalance at some cycle and then attempt to correct it beforehand. For this purpose, statistical meta-data on all relevant relations are maintained. If update activity results in a skewed distribution of values in some database table, the protocols attempt to predict the cost of evaluating the rule program at all sites using a cost model of the rule program version at each site. Given a reasonably accurate model, this method would allow imbalance detection before it actually occurs. This last technique has not yet been fully implemented, and forms a major part of our future work.

5 Applications and Performance Analysis

There is a large range of applications to which the distributed rule processing and load balancing technique may be applied. In this section, we detail two applications to demonstrate the efficacy of the method. In the following analysis, we distinguish between performance measured under "one-time" static load allocation, utilizing only the ICC algorithm, and performance under predictive dynamic load balancing, utilizing both the ICC and DRA algorithms. These two modes are referred to as SLB (static load balancing) and PDLB (predictive dynamic load balancing) modes, respectively. The experimental data reported here pertain to an actual implementation of PARADISER, running on Sun SPARCstations. The underlying DBMS is Sybase.

5.1 ROUTE: a variant of transitive closure

The first application is a variant of the well-studied transitive closure problem. Our version, called "ROUTE", finds a route between two nodes in a directed graph with "colored" arcs whenever a path may be constructed by conjoining adjacent arcs with the same color. Every arc has a value for its color attribute drawn from a finite set. This slight variation on transitive closure allows us to control the total number of routes generated by adjusting the distribution of the color attribute on the arcs, thus providing the opportunity to evaluate performance for different databases with varying characteristics (the databases are generated by a parametrized data generator). The input database consists of a directed acyclic graph with colored arcs. ROUTE finds all the routes in such a graph. It uses a stratification on the length of the routes to limit search. The PARULEL implementation of ROUTE is shown below. This example also serves to illustrate the syntax and structure of PARULEL programs.

```
(scheme arc
  (int ep1) (int ep2) (varchar color))
(scheme route
  (int ep1) (int ep2) (int len) (varchar color))
(scheme length (int len))
(scheme active_contexts (varchar name))

(defcontext FIRST (SECOND))
(defcontext SECOND)
(defcontext FINISH)

(p ::FIRST::create-routes
    (arc *^ep1 <x> ^ep2 <y> ^color <c>)
   -->
    (make route ^ep1 <x> ^ep2 <y> ^color <c> ^len 1))

(p ::FIRST::change-context
    ()
   -->
    (gocontext SECOND))

(p ::SECOND::closure
    (length ^len <l>)
    (route ^ep1 <x>  ^ep2 <y> ^color <c> ^len <l>)
    (arc *^ep1 <y>  ^ep2 <z> ^color <c>)
   -->
    (make route ^ep1 <x> ^ep2 <z> ^color <c>
                        ^len (compute <l> + 1)))

(p ::SECOND::increment-len
    (length ^len <l>)
   -->
    (modify 1 ^len (compute <l> + 1)))

(p ::SECOND::finish
    (length ^len <n>)
   -(route ^len <n>)
   -->
    (gocontext FINISH))
```

Some syntactic similarities with OPS5 are obvious, e.g., production rule names are written in list form and begin with a `p`, while attribute names of relations are preceded by a caret. There are also several differences, e.g., the `*` symbol before an attribute name indicates an RA explicitly. The `scheme` declarations define the relational schema: The `arc` relation has two endpoints and a color attribute, while the `route` relation has the same attributes as well as a length attribute. The `defcontext` declarations define the three contexts `FIRST`, `SECOND` and `FINISH`, and also implicitly the structure of the CSG (the declaration `defcontext FIRST (SECOND)` says that control goes to `SECOND` after `FIRST` has been processed to fixpoint).

The rule `::FIRST::create-routes` creates routes of length 1 in context `FIRST` for every arc in the database, and assigns to them the same color as the corresponding arcs. The rule `::FIRST::change-context` forces a context change to `SECOND` after `FIRST` has been processed. All instances of `::FIRST::create-routes` and a single instance of `::FIRST::change-context` are fired at the end of processing context `FIRST`, in the first inference cycle.

The rule `::SECOND::closure` is the transitive closure rule. It extends routes of length n to routes of length $n+1$ whenever an adjoining arc exists having the same color. The extended route preserves its former color. The rule `::SECOND::increment-len` increments the current length in the database table `length`, and thus sets up for the next round of processing. Rule `::SECOND::finish` forces a switch to the special, globally recognized finishing context, `FINISH`, when no route exists in the database with length equal to the last updated value in the `length` relation. When in the `FINISH` context, the system terminates program evaluation. We mention that this PARULEL program does not require any metarules, and all matched instances at every cycle are fired concurrently to update the database.

The compiled *template* form (SQL) of the LHS of the rule `::SECOND::closure` is as follows:

```
select  *
from    length length#0,
        route route#1,
        arc arc#2,
        active_contexts _ac
where   %d <= arc#2.ep1 and arc#2.ep1 < %d
        and route#1.len = length#0.len
        and arc#2.ep1 = route#1.ep2
        and arc#2.color = route#1.color
        and _ac.name = ``SECOND''
```

Notice that the `WHERE` clause contains a constraint or restriction, namely the selection condition:

```
%d <= arc#2.ep1 and arc#2.ep1 < %d
```

on the `ep1` attribute of the `arc` relation, which is the RA specified in the PARULEL source for this rule. The occurrence of the string "`%d`" in this template indicate *placeholders*, which specify that the bounds of the restriction are *dynamically filled in* at each site by the runtime system *during* program evaluation.

As mentioned previously, the ICC algorithm computes the restrictions for initial partitioning (assuming equal weights), and the DRA algorithm computes the restrictions dynamically using weights computed from observing runtime performance. Both algorithms use the basic WRS method to divide the DFD (discrete frequency distribution) for the purpose of computing the restrictions. A stylized and abbreviated version of the database trigger defined on `ARC.EP1` is shown below. It updates the DFD that is used by WRS to compute the restrictions. The actual code is too involved to be reproduced here.

```
create trigger arc_ep1_trigger on arc
for insert, delete, update as
begin
   (actual code to maintain DFD)
end
```

In Figure 2, we display the DFD for a 10000 arc database. Suppose the loading weights at some cycle are computed by PDLB to be {.18, .25, .12, .30, .15} for 5 sites (P=5), $\{p_1, p_2, p_3, p_4, p_5\}$, respectively. There are 50 bins (heuristically determined), hence the bin size is 200. The bins are defined on the `ep1` attribute of the `arc` relation (`ARC.EP1`).

The DRA algorithm computes the following ranges of bins for the 5 sites respectively: [0-9, 10-22, 23-28, 29-43, 44-49]. The total number of arcs in each of these ranges are 1840, 2603, 1186, 3002, and 1368 respectively, which can be verified by summing the frequencies for each range. These are in the proportion {0.18, 0.26, 0.12, 0.30, 0.14}, which closely matches the weight vector {.18, .25, .12, .30, .15}. The translation from bounds on the bin numbers to bounds on the value of

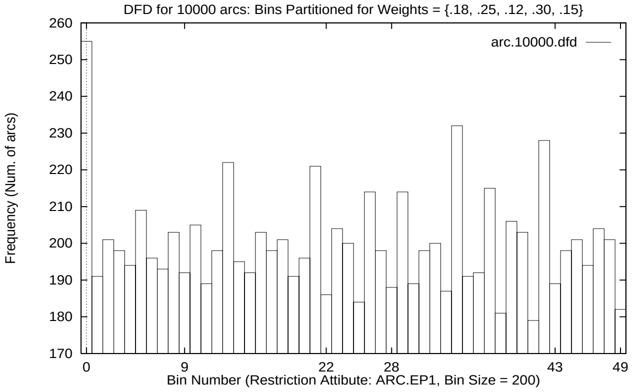

Figure 2: Range of Bins for Each Site

the `ep1` attribute of the `arc` relation is straightforward. Knowing bin size (200), we get the actual restrictions by multiplying the corresponding bin ranges by 200. Thus, the bounds on `ep1` corresponding to the bin range for site p_4 (29-43) are `ep1 >= 5800` and `ep1 < 8600`. The runtime version of the compiled template for the rule `::SECOND::closure` shown above would then have the placeholders filled in as follows:

```
5800 <= arc#2.ep1 and arc#2.ep1 < 8600
```

In order to measure the performance of the PDLB protocol, we executed several runs of ROUTE with varying database size on an ensemble of 5 (P=5) sites. The databases were generated randomly. We plot the normalized standard deviation of the match times (cumulative over all inference cycles) over all the sites, in both SLB and PDLB modes, as a function of database size in Figure 3.

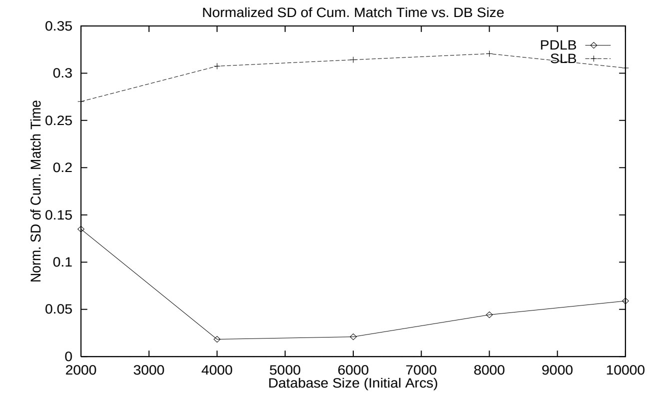

Figure 3: Normalized SD of Match Time vs. DB Size

We notice that the normalized standard deviation of match times is consistently lower for PDLB mode relative to SLB mode, indicating that PDLB successfully reduces the completion time disparities among the sites, resulting in overall efficiency. To see how this translates into improved performance, we plot the actual match time for an ensemble of 5 sites (P=5) as the database is scaled up in Figure 4. The graph shows that match time in PDLB mode is significantly lower (up to 30% less) relative to SLB mode, for all database sizes in our experimental suite. In all cases, the cost of the protocol itself is a very small fraction of the total match time (less than 1% in most cases), and this fraction decreases as the database is scaled up.

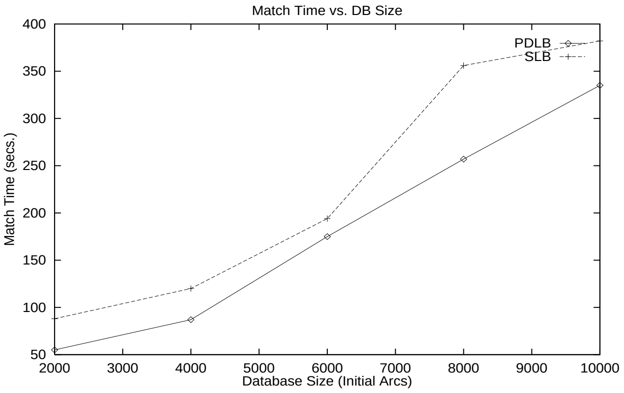

Figure 4: Match Time vs. DB Size

5.2 Merge-Purge

The second application is called *merge-purge* (abbreviated MP henceforth). Merging and coalescing multiple sources of information into one unified database requires more than structurally integrating diverse database access methods and data models. In applications where the data is corrupted (incorrect or ambiguous), the problem of integrating multiple databases is particularly challenging. We call this the merge-purge problem. The key to successful merging of information in this case depends on semantic integration which requires a means of identifying similar data from diverse sources. The determination that two pieces of information are similar, and that they represent some aspect of the same domain entity, depends on sophisticated inference techniques and knowledge of the domain. We implemented a PARULEL rule program that declaratively specifies an equational domain theory for this purpose. This application finds a natural fit with the features of PARADISER, since it involves inference over large databases. Furthermore, the size of databases involved may be very large (as is the case in commercial systems designed to perform this task, where hundreds of millions of records are common). Thus, efficient implementations of MP are highly desirable.

The PARULEL program that implements the equational domain theory to solve the MP problem contains 28 rules and 6 metarules, divided over 8 contexts. For brevity, we will only mention the organization of the contexts and their intended function without describing

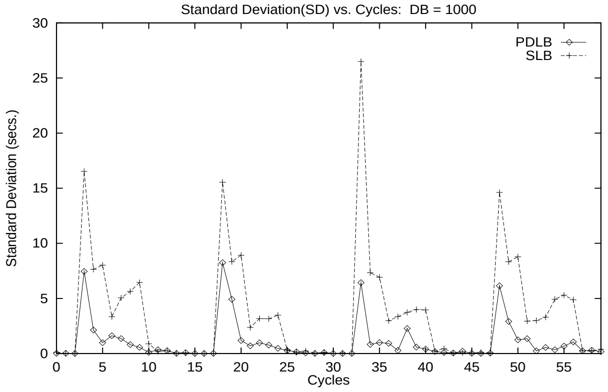

Figure 5: Standard Deviation of SLB vs. PDLB

the rules contained therein. The database consists of a single extensional relation, PERSON, whose scheme consists of several attributes, such as a unique identifier, social security number, first, last, and middle names, street address, city, state, and zipcode. The database contains different records which are semantically the same, but structurally and syntactically different. For instance, two records may be identical except for a large variation in the last name field. We assume that a preprocessing sorting phase clusters the records so that semantically "close" records are in close proximity to each other.

To reduce the computational complexity, the MP application is coded in PARULEL so that the computation is based on a sliding window over the extensional database (the PERSON relation), which defines the "virtual" database over which the merged relation is computed. The rules compute the cross product of this virtual database, and filters them with the constraints specified by the equational theory. Parallelism is achieved by allocating a portion of the virtual database to each processing site by means of restrictions on the RA, while load balancing is achieved by adjusting the size of the allocations at runtime. The program terminates when a complete pass over the PERSON relation has been completed by the sliding window. This approach is justified since potentially mergeable records are expected to appear in the same neighborhood as a result of the preprocessing. The PARADISER environment distributes the window-based computation of the PARULEL program over a number of sites using the workload partitioning and load balancing techniques detailed earlier. The performance is measured while running the system in SLB and PDLB modes.

To demonstrate the efficacy of the load balancing features of PARADISER, we plot the standard deviation of match times at each cycle among the sites of an ensemble of processors (P=5) for a run of MP on a 1000 person database in Figure 5. Two superposed graphs, one for SLB and the other for PDLB mode operation, are displayed.

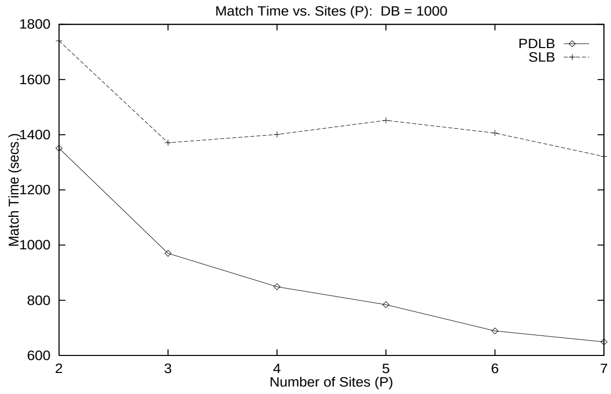

Figure 6: Match Times in SLB mode vs. PDLB mode

We notice that the graph is periodic with respect to the processing cycle, with the SLB standard deviation being consistently higher than that of PDLB. The periodic spikes and subsequent reduction in standard deviation are explained as follows. The spikes occur every time control enters a context which computes "initial matches" on the sliding window. This context generates the largest set of possibly mergeable records among all the contexts. The intensional relation thus produced is reduced by the rules in the successive contexts. Since this context is computationally intensive, the inherent disparities among the sites become prominent. In PDLB mode, an attempt is made at runtime to redistribute the load so that the standard deviation is lower than in SLB mode, where no such attempt is made. Notice also that as the computation becomes less intensive (control goes to other contexts), the loads are again spread out more or less equally among the sites, so that when the system enters the initial match context again, it again takes a large performance hit. But PDLB reduces this effect significantly over all cycles of the inference process.

The graph in Figure 6 shows the cumulative match times over all cycles for SLB and PDLB modes, for a 1000 person database. We notice that in PDLB mode, cumulative match time decreases in a virtually linear fashion as the number of sites (P) is increased. However, in SLB mode, this is not the case. In fact, cumulative match time remains essentially the same, and is always significantly higher than the corresponding values for PDLB mode. We conclude that PDLB mode offers much higher linearity in speedup performance, and that speedup in SLB mode is negligible. The overhead of the PDLB protocol is observed to be a small percentage of the total match, and match time in PDLB mode is up to 50% less than in SLB mode.

In Figure 7, we plot the normalized standard deviation of the cumulative match times over all cycles as a function of the number of processing sites, P. The normalized SD for PDLB mode is found to be well below

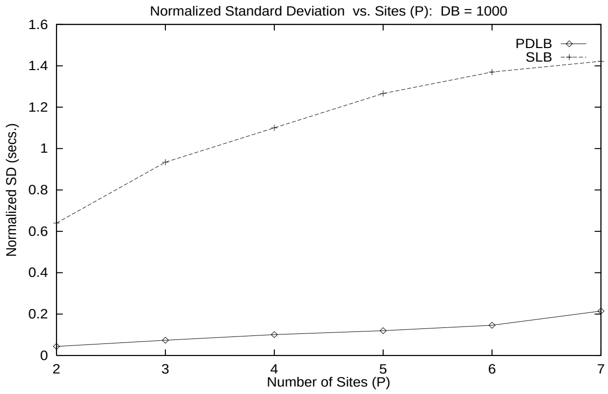

Figure 7: Normalized SD vs. Number of Sites (P)

that for SLB mode, for values of P in the range [2-7]. Moreover, the normalized SD curve for SLB mode generally has a steeper slope than for PDLB. This indicates that PDLB mode is scalable in the number of processors, i.e., as we increase P, we can expect PDLB to consistently reduce runtime variance among the sites, while for SLB mode, the disparities grow with the value of P.

6 Application to Balanced Parallel Query Processing

In Section 4.3, we mentioned that the fundamental principles established in the simpler context of replicated data and the copy-and-constrain technique may be extended to the case of a fully distributed database. In this Section, we outline the process which achieves that end.

In the last decade, many researchers have focused on developing database machine architectures for fast execution of complex SPJ queries. Many of these efforts have resulted in the development of efficient parallel join algorithms for shared-nothing multiprocessor environments [3, 10, 18]. These algorithms are parallel versions of sort-merge or hash-based joins previously developed for centralized uniprocessor database machines. While there are many subtle differences, they all assume a homogeneous ensemble of processors, which do not exhibit performance variations over time. Another class of parallel join algorithms have been described in the literature to address the problems introduced when data skew is present [24]. However, in this case as in the previous ones, the underlying processing resources are assumed to be homogeneous and time-invariant. Because of these assumptions, work to date on parallel joins have not been concerned with dynamic load balancing, since a good initial allocation of tasks to processors suffices under those conditions.

In contrast, we consider parallel and distributed processing of rules and SPJ queries over heterogeneous resources, where any site of the ensemble may deviate from its nominal rated performance for any period of time due to external loads that are not related to the processing of the query in question. In this section, we briefly show how the PDLB protocols of PARADISER may be utilized to achieve this goal.

The relations to be joined, say R and S, are fragmented in round-robin fashion over all available disks, where they reside. When a join is computed, any selection predicates are applied to local fragments of R and S at each site, and the filtered tuples are hashed into bucket files. Given large enough base relations, the number of hash buckets, N, is heuristically chosen to be large relative to the number of sites, P. Portions of each bucket for either relation may reside over all the sites.

Balanced operation over a heterogeneous ensemble may be achieved as follows. A small fraction of the N buckets from both relations are first distributed among the P sites so that every site has the same number of bucket pairs (equal loading weights for all sites). This is the "sampling phase". Corresponding buckets are joined at each site to which they have been allocated by either a nested-loop or hash-probe method. The PDLB protocol being used (say the GC protocol) then determines the degree of imbalance, if any, by examining the performance for the sampling phase and computes new loading weights. (Thus, the allocated buckets correspond to the weighted range partitioning specified by the RA's in our previous formalism.) The next batch of bucket pairs (another small fraction of the remaining unprocessed buckets) are then allocated among the sites according to the computed weights. The joins are computed for this new batch of bucket pairs, and the protocol measures imbalance and again recomputes loading weights. This process is repeated until all buckets have been processed. The fraction of N in each batch is a tuning parameter and may depend on the size of the relations, the physical memory available at each site, and P. This method makes it possible to detect "overpopulated" buckets, indicating skewed data. Skewed buckets are handled separately using a simple extension of the basic scheme.

This algorithm is suitable for parallel rule languages by simply extending single query processing to parallel processing of a "join queue" that represents all the rules in a rule program. For this purpose, a high level scheduler is needed to schedule different join tasks over the processors concurrently, whenever possible. This algorithm is the subject of a forthcoming report [4] that analyzes the performance and design issues in the parallel processing of rule languages over a fully distributed database architecture.

7 Conclusion

We believe this architecture provides a sufficiently rich model for general balanced processing of rules or queries in a parallel and distributed environment. Indeed, PARADISER compiles rules into complex queries for subsequent parallel and distributed processing under the operational semantics of PARULEL. Thus, any optimizations and load balancing protocols tested in this environment are generally applicable to the case of processing complex queries as well.

We have shown, using transitive closure and merge-purge as examples, that PDLB, with a low-cost protocol, produces substantial performance improvements over naive parallel evaluation. In particular, we have demonstrated the efficacy and utility of predictive dynamic load balancing.

Previous published work established the isoefficiency of a suite of possible load balancing protocols. To date, only the Global Coordinator protocol is available in PARADISER as a base case, and its isoefficiency function is considerably worse than several others we have analyzed. Even so, the implementation reported here establishes that its overall overhead costs are not substantially high in a parallel or distributed processing environment with a relatively modest number of processors. However, to scale to very large numbers of processors, future work is aimed at reducing PDLB overhead even further without sacrificing the quality of the achieved load balance by utilizing the other protocols we have developed.

We have recently constructed a simulator to test PDLB under the various protocols and varying database sizes, data distribution schemes, and data skew conditions for large numbers of (up to 128) processors. The results will be reported in a forthcoming paper on the performance of PDLB on arbitrarily complex queries.

References

[1] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of datalog*. *Journal of Theoretical Computer Science*, 1991.

[2] D.A. Brant and D.P. Miranker. Index Support for Rule Activation. In *To appear in the proceedings of ACM-SIGMOD, Intl. Conf. on the Management of Data*, 1993.

[3] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of the ACM SIGMOD 1988, Intl. Conf. on the Management of Data*. ACM Press, 1988.

[4] H. M. Dewan. Runtime Reorganization of Parallel and Distributed Expert Database Systems.

Technical report, Department of Computer Science, Columbia University, April 1994. Ph.D. Thesis.

[5] H. M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, and S. DaSilva. Incremental Database Rule Processing in PARADISER. *Journal of Intelligent Information Systems*, 1:2, October 1992.

[6] H.M. Dewan and S.J. Stolfo. System Reorganization and Load Balancing of Parallel Database Rule Processing. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, pages 186–197, Trondheim, Norway, June 1993.

[7] H.M. Dewan and S.J. Stolfo. The Distributed Evaluation of Rules in PARADISER. Technical Report In Preparation, Department of Computer Science, Columbia University, May (expected) 1994.

[8] H.M. Dewan, S.J. Stolfo, and L. Woodbury. Scalable Parallel and Distributed Expert Database Systems with Predictive Load Balancing. *J. Parallel and Distrib. Computing, special issue on scalable systems*, 1994. Submitted.

[9] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. In *Communications of the ACM*. ACM, June 1992.

[10] Tandem Performance Group. A Benchmark of Non-Stop SQL on the Debit Credit Transactin. In *Proceedings of the ACM SIGMOD 1988, Intl. Conf. on the Management of Data*. ACM Press, 1988.

[11] E.N. Hanson. Rule condition testing and action execution in Ariel. In *In proceedings of ACM-SIGMOD, Intl. Conf. on the Management of Data*, June 1992.

[12] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *In proceedings of ACM-SIGMOD, Intl. Conf. on the Management of Data*, June 1989.

[13] D.P. Miranker, D. Brant, B. Lofaso, and D. Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 685–692, 1990.

[14] K. Morris, J.D. Ullman, and A. Van Gelder. Design Overview of the NAIL! System. In *Proceedings of the third Intl. Conference on Logic Programming*, pages 554–568, 1986.

[15] D. Neiman. *Issues in the Design and Control of Parallel Rule-Firing Production Systems*. PhD thesis, University of Massachusetts, Amherst, Computer Science Dept., September 1992.

[16] A. Pasik. Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules. Technical Report CUCS-313-87, Department of Computer Science, Columbia University, 1987.

[17] R. Ramakrishnan, P. Seshadri, D. Srivastave, and S. Sudarshan. An Overview of CORAL. Technical report, Department of Computer Science, University of Wisconsin-Madison, 1989.

[18] D.A. Schneider and D.J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the ACM SIGMOD 1989, Intl. Conf. on the Management of Data*. ACM Press, 1989.

[19] S. Stolfo, D.P. Miranker, and R. Mills. A Simple Processing Scheme to Extract and Load Balance Implicit Parallelism in the Concurrent Match of Production Rules. In *Proc. of the AFIPS Symposium on Fifth Generation Computing*, 1985.

[20] S. J. Stolfo, H.M. Dewan, and O. Wolfson. The PARULEL parallel rule language. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages II:36–45. IEEE, 1991.

[21] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation DBMS. In *Communications of the ACM*, Oct. 1991.

[22] S. Tsur and C. Zaniolo. LDL: A logic-based data-language. pages 33–41, 1986. In proc. Intl. Conference on Very Large Databases.

[23] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. 1991. In proc. 17th Intl. Conference on Very Large Databases.

[24] J.L. Wolf, D.M. Dias, P.S. Yu, and J. Turek. Comparative Performance of Parallel Join Algorithms. In *First Intl. Conference on Parallel and Distributed Systems*, pages 78–88. IEEE, 1991.

[25] O. Wolfson and A. Ozeri. A New Paradigm for Parallel and Distributed Rule-processing. In *Proc. ACM-SIGMOD*, 1990.