Proc. of 1992 VLDB Conference, pages 339–351

# Production Rules in Parallel and Distributed Database Environments

Stefano Ceri

Dipartimento di Elettronica
Politecnico di Milano
Piazza L. da Vinci, 32
I-20133 Milano, Italy
ceri@cs.stanford.edu

Jennifer Widom

IBM Almaden Research Center
650 Harry Road, K55/801
San Jose, CA 95120
USA
widom@almaden.ibm.com

**Abstract.** In most database systems with production rule facilities, rules respond to operations on centralized data and rule processing is performed in a centralized, sequential fashion. In parallel and distributed database environments, for maximum autonomy it is desirable for rule processing to occur separately at each site (or node), responding to operations on data at that site. However, since rules at one site may read or modify data and interact with rules at other sites, independent rule processing at each site may be impossible or incorrect.

We describe mechanisms that allow rule processing to occur separately at each site and guarantee correctness: parallel or distributed rule processing is provably equivalent to rule processing in the corresponding centralized environment. Our mechanisms include locking schemes, communication protocols, and rule restrictions. Based on a given parallel or distributed environment and desired level of transparency, the mechanisms may be combined or may be used independently.

## 1 Introduction

Many next-generation database systems include facilities for creating and processing *production rules*, e.g. [GJ91, Han89, MD89, SJGP90, WCL91]. These "active" rules specify that certain database operations are to be executed automatically whenever certain events occur or conditions or met. Production rules in database systems can be used for integrity constraint enforcement, derived data maintenance, triggers and alerters, authorization checking, and versioning, as well as providing a platform for large and efficient knowledge-bases and expert systems.

Although a wide variety of semantics and algorithms have been proposed for production rule processing in database systems (see, e.g. [HW93, Sel89]), in all cases rules respond to operations on centralized data, and rule processing usually is performed in a centralized, sequential fashion. (There has been some work on parallelizing rule firings in the context of the OPS5 production rule language, but this work still considers centralized data and does not directly apply to database production rule languages; see Section 1.1.) Given the high interest in parallel and distributed database systems [DGS+90, OV91], it clearly is important for database production rule facilities to be adapted to these environments.

For maximum autonomy in parallel and distributed environments, it is desirable for rule processing to occur separately at each site (or node), responding to operations on data at that site. Rule processing can, in fact, be completely independent at each site, as long as each rule is restricted to reference (i.e. be triggered by, read, and modify) data at one site only. However, this seriously limits the expressible rules, provides no notion of *transparency* (the illusion to the user that the database is centralized), and forces rules to be revised if data is redistributed. Once rules are permitted to reference data at multiple sites, independent rule processing at each site may be impossible or incorrect. Rules triggered at one site may need to operate on data at remote sites; these operations may trigger additional rules at remote sites, etc. Priorities between rules further complicate the problem, particularly when they involve rules at different sites. Additional mechanisms are needed to process rules in parallel and distributed environments, and, for transparency and correctness, these mechanisms should ensure that parallel or distributed rule processing is equivalent to rule processing in the corresponding centralized environment.

We consider a database production rule facility we have developed, the *Starburst Rule System* [WCL91], and describe mechanisms for adapting it to parallel and distributed environments. Although some details of our mechanisms are particular to the semantics of the Starburst rule language, in general the mechanisms are applicable to other database rule systems as well.

For distributed environments, we begin by imposing certain restrictions on the environment and the allowable rules that guarantee correct rule processing. We then describe several orthogonal mechanisms, each of which allows a restriction to be lifted. Based on a given environment and desired level of transparency, the mech-



**Proceedings of the 18th VLDB Conference**
**Vancouver, British Columbia, Canada 1992**

anisms may be combined or may be used independently. For the distributed environment, we describe a locking scheme and *rule-task executor* that allow rules to reference data at multiple sites. These mechanisms assume that coordination is used to initiate rule processing, so we give a second locking scheme that alleviates this requirement. Finally, we describe some additional locking and a communication protocol that support priorities between rules at different sites.

These mechanisms all apply to parallel environments as well. However, because of the *horizontal fragmentation* often used in parallel relational database systems (i.e. tables are partitioned across nodes by row), we introduce an additional approach: we define (semantic) restrictions on rules that allow them to effectively be horizontally fragmented along with the data. When rules satisfy these restrictions, the fragmented rules can be processed independently at each site, with a behavior that is equivalent to rule processing in the corresponding non-fragmented environment.

### 1.1 Related Work

Most previous work on parallel or distributed execution of production rules considers the rule language *OPS5* [BFKM85] in the context of main-memory expert systems, e.g. [Gup86, Pas89, SG90, SM84]; some work has considered OPS5 coupled to database systems [RSD91, SHT90]. Results include parallel algorithms for determining which rules are triggered, methods for detecting, at compile-time or at run-time, that multiple triggered rules (or multiple instantiations of a triggered rule) cannot interfere with each other and consequently can execute in parallel, and run-time locking methods for serializing parallel rule executions. Our initial locking scheme given in Section 3.1.2 is similar to proposals in [RSD91, SHT90]; however, since only centralized data is considered in [RSD91,SHT90], their work does not address the many other problems that arise when rules respond to operations in parallel and distributed database environments. Furthermore, the language and rule processing semantics of OPS5 are quite different from those used in most integrated database production rule systems, including Starburst.

In [BKK87], a distributed database architecture is proposed in which data distribution is tailored for parallel production rule processing. Data is horizontally partitioned based on existing rules; indexing and query optimization techniques along with *RETE*-like pattern matching [SDLT86] are used to determine, in parallel, which rules are triggered. The problem of detecting triggered rules in a distributed environment also is investigated in [HSL92]. They propose a method for algebraically decomposing complex and potentially time-consuming rule conditions into multiple conditions evaluated on distributed database sites; they also present the coordination algorithms to evaluate such conditions. Although condition evaluation is distributed, rule processing itself is still centralized.

### 1.2 Outline of Paper

In Section 2 we describe the production rule language and rule processing semantics of the Starburst Rule System. Section 3 covers distributed environments: Mechanisms whereby rules can reference data at multiple sites are given in Section 3.1; mechanisms whereby each site can independently initiate rule processing are given in Section 3.2; mechanisms for intersite rule priorities are given in Section 3.3. A proof of correctness is included in each of these subsections. Section 4 covers parallel environments: Most mechanisms from Section 3 apply here, but an additional notion of rule *partitionability* is introduced and proven correct. Finally, in Section 5 we draw conclusions and propose future work.

## 2 The Starburst Rule System

We provide a brief overview of our set-oriented, SQL-based production rule language, which has been integrated into the Starburst extensible relational database system at the IBM Almaden Research Center [H$^+$90]. Further details and numerous examples appear in [WCL91, WF90]; some examples appear in Section 4.2 of this paper. Note that the current Starburst prototype is a multi-user, centralized database system.

Starburst production rules are based on the notion of *transitions*. A transition is a database state change resulting from execution of a sequence of data manipulation operations. Rules consider only the *net effect* of transitions, as defined in [WF90]. The syntax for creating a rule is:

> **create rule** *name* **on** *table*
> **when** *triggering operations*
> [ **if** *condition* ]
> **then** *action*
> [ **precedes** *rule-list* ]
> [ **follows** *rule-list* ]

The *triggering operations* are one or more of **inserted**, **deleted**, and **updated**$(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are columns of the rule's *table*. A rule is triggered by a given transition if at least one of its triggering operations occurred in the net effect of the transition. The optional *condition* specifies an SQL predicate. The *action* specifies an arbitrary sequence of SQL data manipulation operations (including **rollback**) to be executed when the rule is triggered and its condition is true. The optional **precedes** and **follows** clauses are used to induce a partial ordering on the set of defined rules. If a rule $r_1$ specifies a rule $r_2$ in its **precedes** list, or if $r_2$ specifies $r_1$ in its **follows** list, then $r_1$ is higher than $r_2$ in the ordering. (We also say that $r_1$ has higher *priority* than $r_2$.) When no direct or transitive ordering is specified between two rules, their order is arbitrary.

A rule's condition and action may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule conditions and actions may refer to *transition tables*, which are logical tables reflecting the changes that have occurred during a rule's triggering transition. At the end of a given transition, transition table **inserted** in a rule refers to those tuples of the rule's table that were inserted by the transition. Transition tables **deleted**, **new-updated**, and **old-updated** are similar.

Rules are processed at the commit point of each transaction.[1] The state change resulting from the transaction creates the first relevant transition, and some set of rules are triggered by this transition. A triggered rule $r$ is chosen from this set for consideration. Rule $r$ must be chosen so that no other triggered rule has higher priority than $r$. If $r$ has a condition, then it is checked. If $r$'s condition is false, then another triggered rule is chosen for consideration. Otherwise, if $r$ has no condition or its condition is true, then $r$'s action is executed. Assume for the moment that $r$'s action does not include **rollback**. After execution of $r$'s action, all rules not yet considered are triggered if a triggering operation occurred in the composite transition created by the initial transition and subsequent execution of $r$'s action. Rules already considered (including $r$) are triggered again only if a triggering operation occurred in the transition created by $r$'s action. From the new set of triggered rules, a rule $r'$ is chosen for consideration such that no other triggered rule has higher priority than $r'$. Rule processing continues in this fashion.

At an arbitrary time in rule processing, a given rule is triggered if a triggering operation occurred in the (composite) transition since the last time it was considered. If it has not yet been considered, it is triggered if a triggering operation occurred in the transition since the start of the transaction. If a rule action specifying **rollback** is executed, then the system rolls back to the start of the transaction and rule processing terminates. Otherwise, rule processing terminates when there are no more triggered rules, and the entire transaction then commits.

A skeleton algorithm for rule processing is shown in Figure 1. It is used in the next section as a basis for describing rule processing in distributed environments. For convenience, we refer to steps 1 and 2 in the algorithm as *rule selection*, and steps 3 and 4 as *rule consideration*.

```
----------------------------------------------------
 repeat until step 1 produces no triggered rules:
    1. determine which rules are triggered (based on
       the net effect of appropriate transitions)
    2. choose a triggered rule r such that no other
       triggered rule has higher priority
    3. evaluate r's condition
    4. if true, execute r's action
----------------------------------------------------
```

Figure 1: Algorithm for centralized rule processing

## 3 Distributed Environments

We consider distributed relational database environments in which the tables comprising the database reside at a number of separate *sites*. We assume that tables are not replicated or fragmented across sites, since many distributed database systems do not support these features [GR92,TTC+90].[2] Based on the distribution of database tables, user transactions are distributed to execute across multiple sites at the same time. The typical requirement for providing (location) transparency is that the effect of distributed transaction execution must be equivalent to the effect of the original transaction on the corresponding centralized database.

In these environments, we would like rule processing to also execute in a distributed fashion. A rule system resides at each site, responding to the database changes at that site only. To provide transparency, the combined effect of distributed transaction execution with distributed rule processing must be equivalent to the combined effect of the original transaction with centralized rule processing on the corresponding centralized database. We refer to this as *correct* distributed rule processing.

Clearly, each rule should reside at the site containing the table whose changes trigger that rule. Suppose the following three conditions also hold:

1. All rules read and modify tables at their local site only.
2. All priorities are between rules at the same site.
3. Rule processing at each site $S$ for a transaction $T$ does not begin until it is known that $T$ will not subsequently read or modify tables at $S$.

Then the algorithm of Figure 1 can be run independently at each site and rule processing will be correct: Since rules do not read or modify data at remote sites, they cannot trigger and need not see the effects of rules running at other sites.[3] Since there are no priorities between rules at different sites, no information about triggered rules at remote sites is needed for correct rule selection. Finally, since no user changes are made at a site after rule processing begins, the distributed user transaction logically precedes all distributed rule processing. When rule processing has terminated at all sites, the entire distributed transaction commits (using, e.g., a two-phase commit protocol [GR92]).

The first two conditions given above restrict the expressible rules. Although they represent reasonable design principles, for some applications these conditions are just too restrictive. (This may depend on the level of transparency desired for rule definition, how often rules change, and how often data is redistributed; see Section 5.4 for further discussion.) The third condition above may require extra coordination between sites and

[1] We recently have extended the system to support rule processing at arbitrary user-specified points within a transaction. The mechanisms in this paper are easily adaptable to this extension.

[2] We do consider horizontal partitioning for parallel database environments in Section 4. Vertical partitioning and data replication will be addressed in future work; see Section 5.3.

[3] Note that if a rule action specifying **rollback** is executed at some site, then the entire distributed transaction must be rolled back. We assume the existence of a mechanism for distributed rollback.

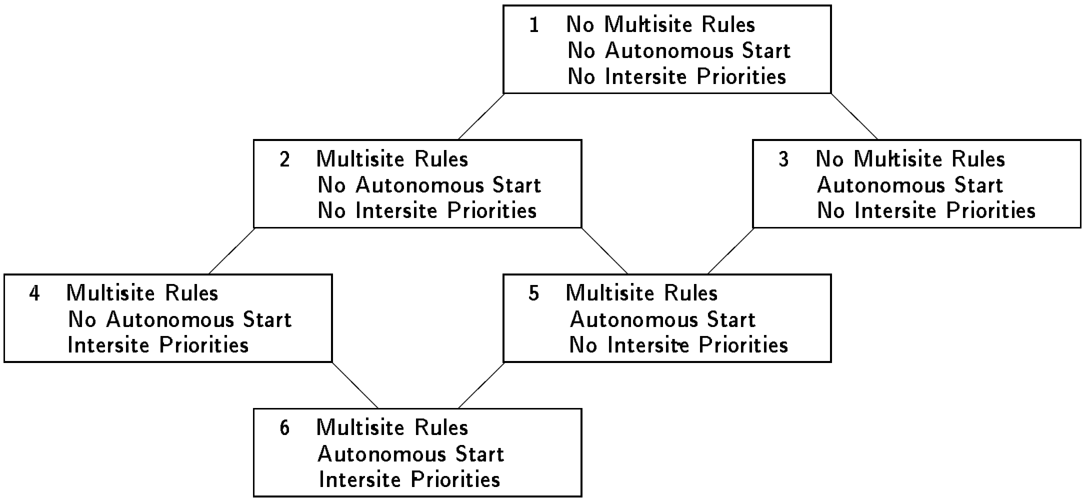


Figure 2: Paradigms for distributed rule processing

may unnecessarily restrict parallelism, particularly in environments where transactions are decomposed into multiple subtransactions (see Section 3.2).

We give mechanisms that allow each of these three conditions to be dropped. For presentation, we consider condition 3 before condition 2. Hence, our first mechanism allows rules to read and modify tables at remote sites (*Multisite Rules*), our second mechanism allows rule processing to be initiated independently at each site even when the user transaction may subsequently read or modify tables at that site (*Autonomous Start*), and our third mechanism allows priorities between rules at different sites (*Intersite Priorities*). The mechanisms may be used in any combination, as illustrated by the lattice in Figure 2. The two paradigms omitted from the lattice combine *No Multisite Rules* with *Intersite Priorities*; this combination is allowable but makes little sense since there is no need for priorities between rules that cannot affect each other.

Note that any paradigm combining *Multisite Rules* and *Intersite Priorities* (Paradigms 4 and 6 in our lattice) provides full transparency with respect to rule definition and processing. That is, rules can be created with no knowledge of data distribution. Each rule is installed automatically at the correct site, and (as we will show) distributed rule processing is equivalent to rule processing in the corresponding centralized environment.

### 3.1 Multisite Rules

Recall that each rule is triggered by changes to one table, so each rule resides at the site containing its trigger table. In this section we consider Paradigm 2 in the lattice of Figure 2—we give mechanisms that allow rules to read and modify tables at remote sites. To model this paradigm, we say that each user transaction $T$ is divided into a set of *tasks*, and each task is executed on the tables at a single site. There may be flow-of-control, communication, and parallelism between tasks. The transformation of transactions into tasks (including their coordination) is performed by the distributed query optimizer, and we assume it is correct: the effect of $T$'s distributed tasks always is equivalent to what $T$ would have produced on the corresponding centralized database.

Since we are assuming *No Autonomous Start*, rule processing does not begin for transaction $T$ at a site $S$ until it is known that $T$ will not subsequently read or modify tables at $S$. Depending on the flow-of-control between tasks, this may require that rule processing does not begin at any site until all of $T$'s tasks have completed. We assume this is the case, however with slight modifications our mechanisms allow rule processing to begin at a site $S$ as soon as it is known that $T$ will execute no further tasks at $S$.

When all of $T$'s tasks have completed (but before $T$ commits), the rule processor takes control at each site. Since rule conditions and actions are database queries, evaluating a rule condition or performing a rule action involves executing additional tasks (determined by the query processor), possibly at remote sites. Whereas we did not concern ourselves with the scheduling of tasks at each site during user transaction $T$, during rule processing the tasks to be executed at each site are managed by a *rule-task executor* at that site, described in Section 3.1.1. This is necessary for correct rule processing: otherwise, if tasks are executed concurrently or arbitrarily at a site during rule processing, rules might be triggered by and process inconsistent sets of changes. (This is explained in more detail below.) In addition to the rule-task executor, a locking scheme is used in which each site obtains special locks on data during rule processing that may conflict with locks held by other sites during their rule processing, described in Section 3.1.2. Together with the rule-task executor, the locking scheme ensures that distributed execution of rules is *serializable* [BHG87], similar to [RSD91, SHT90]. Furthermore, any equivalent serial schedule corresponds to some valid execution of rules on the corresponding centralized database; a proof of correctness is given in Section 3.1.3.

### 3.1.1 Rule-Task Executor

Let $S_1, S_2, \ldots, S_n$ denote the $n$ distributed sites. Once rule processing begins for a transaction $T$, the rule-task executor at each site $S_i$ takes full responsibility for scheduling the tasks to be executed on tables at $S_i$. Some of these tasks may be generated by local rule processing at $S_i$, while other tasks may be requested on behalf of rule processing at other sites. If $S_i$ receives no requests for tasks to be executed on behalf of other sites, then rule processing can proceed according to the algorithm of Figure 1. However, since other sites are running the same algorithm, and since rule conditions and actions may read or modify tables at remote sites, during the rule processing loop requests may be received from other sites to execute tasks.

$S_i$ executes tasks on behalf of remote sites at the discretion of its rule-task executor, but remote tasks are executed only at the "top" of the rule processing loop (recall Figure 1). To see why, suppose instead that remote tasks are executed during rule selection (steps 1–2). Then changes made by remote tasks might trigger additional rules, producing an inconsistent set of triggered rules used in rule selection. If remote tasks are executed before or during rule consideration (steps 3–4), then changes made by remote tasks might trigger additional rules with higher priority than the selected rule, invalidating rule selection.

Rule processing at site $S_i$ cannot terminate until all other sites also have completed rule processing: if rules are still being processed at another site $S_j$, then $S_j$ could generate tasks that modify tables at $S_i$ and trigger additional rules. The modified rule processing algorithm used by the rule-task executors is given in Figure 3. Note that distributed termination of rule processing can be coordinated together with transaction commit. Also note that a variety of fairness criteria can influence task selection in step 0, and we do not explore these issues here.

```
-------------------------------------------------
repeat until step 1 produces no triggered rules
and all other sites also have completed:
  0. if there are requested tasks, execute
     zero or more
  1. determine which rules are triggered (based
     on the net effect of appropriate transitions
     on local data)
  2. choose a triggered rule r such that no other
     triggered rule has higher priority
  3. evaluate r's condition
  4. if true, execute r's action
participate in distributed commit of the
transaction
-------------------------------------------------
```

Figure 3: Algorithm for distributed rule processing

### 3.1.2 Locking Scheme

Even when each site uses a rule-task executor as described in the preceding section, distributed rule processing still may be incorrect. As an example, suppose that the action of a rule $r_1$ at site $S_1$ is decomposed into two tasks, one of which reads and modifies a remote table $t_i$ at site $S_i$, and the other of which reads and modifies a remote table $t_j$ at site $S_j$. Suppose the action of a rule $r_2$ at site $S_2$ is similar. If $r_1$ and $r_2$ are concurrently triggered and selected for consideration during distributed rule processing for a transaction $T$, then the rule-task executors at sites $S_i$ and $S_j$ may execute $r_1$ and $r_2$'s tasks such that their rule actions are effectively interleaved: $r_1$ sees some but not all of $r_2$'s modifications, or vice-versa. This behavior is not equivalent to any valid rule processing sequence following $T$ in the corresponding centralized environment. Hence, an additional mechanism must be used to enforce serializability of rule considerations.

We assume that standard two-phase locking is used for transactions [BHG87,GR92] and that locks obtained by a transaction $T$ are shared by all of $T$'s tasks. We call these *standard-grade* (or $L$) locks; they are used for concurrency control with respect to other transactions. We introduce an additional notion of *rule-grade* (or $R$) locks. R locks are obtained by individual sites on local or remote data during rule processing. R locks are not shared across sites for the same transaction; e.g., if site $S_i$ holds an exclusive R lock on a data item during rule processing for transaction $T$, then site $S_j$ cannot concurrently hold a shared or exclusive R lock on that data item for any transaction (including $T$).

During rule processing at each site $S_i$, consideration of each selected rule $r$ follows its own two-phase locking protocol using rule-grade locks (as well as standard locking to exclude other transactions). That is, during evaluation of $r$'s condition and execution of $r$'s action, in addition to the usual accumulation of L locks, $S_i$ must obtain corresponding R locks on the same data items. (Note that if an L lock is already held, the corresponding R lock still must be obtained.) When consideration of $r$ is complete, all accumulated R locks are released.[4]

Deadlock based on R lock requests can occur. Such deadlocks can be broken by backing up one site $S_i$ participating in the deadlock to step 0 in its rule processing loop (recall Figure 3), including release of R locks as appropriate. Site $S_i$ can then choose a different task to execute or rule to consider, or it can try the same task or rule again. Backing up and choosing a different task or rule may be a useful strategy when a lock conflict is encountered even if deadlock is not present: with some extra bookkeeping, task and rule selection coupled with backup techniques can try to minimize lock waiting time as well as minimize the possibility of deadlock.

### 3.1.3 Correctness

Recall our assumptions: rules can read or modify remote data, there are no priorities between rules at different sites, and rule processing does not begin for a distributed transaction until all of its tasks have completed. The following theorem shows that, under these

[4] As an optimization, the release of R locks on remote data can sometimes be "piggy-backed" onto remote task execution. Note also that we are assuming the existence of a reliable distributed lock manager.

assumptions, the rule-task executor and locking scheme described above guarantee correct distributed rule processing.

**Theorem 3.1 (Multisite Rules)** Let $T$ be a transaction and assume that the effect of $T$'s distributed tasks is equivalent to what $T$ would have produced on the corresponding centralized database. Any behavior of distributed rule processing following $T$'s tasks is equivalent to some valid behavior of centralized rule processing following $T$ on the corresponding centralized database.

**Proof**: Omitted due to space constraints; see [CW92].

### 3.2 Autonomous Start

In some distributed database environments, each user transaction is decomposed into *subtransactions*, and each site takes responsibility for running one subtransaction. Subtransactions primarily manipulate tables at their local sites, but also may manipulate tables at remote sites. In these environments, it often is reasonable to allow rule processing to begin at each site as soon as its subtransaction has finished. This avoids explicit coordination to initiate rule processing (eliminating message-passing overhead) and may result in more parallelism. However, additional mechanisms are needed in this case to ensure the correctness of distributed rule processing.

In a distributed database environment with sites $S_1, S_2, \ldots, S_n$, each user transaction $T$ is divided into $n$ subtransactions, $T_1, T_2, \ldots, T_n$. Let subtransaction $T_i$ run at site $S_i$, $1 \leq i \leq n$. As previously, we say that each subtransaction $T_i$ is divided into a set of tasks, and each task is executed on the tables at a single site. Let each site begin rule processing as soon as it finishes executing its subtransaction, independent of the other sites. Then subtransaction $T_i$ running at site $S_i$ may execute a remote task on the tables at site $S_j$ even after rule processing has begun at $S_j$. If this task reads tables at $S_j$ then it may incorrectly see the effects of $S_j$'s rule processing; if this task modifies tables at $S_j$ then its modifications may invalidate $S_j$'s rule processing.

To guarantee that distribute rule processing is correct in this environment, we must ensure that each site's subtransaction logically precedes each other site's rule processing. (This is equivalent to ensuring that user transaction $T$ logically precedes rule processing in the centralized environment, an obvious requirement.) Our mechanisms for this are partially optimistic: in certain cases they simply detect that a consistency violation has occurred, and rule processing must be rolled back and restarted.[5] The mechanisms are based on a locking scheme in which each site obtains special locks on data during subtransaction execution that may conflict with special locks held by other sites during rule processing. In addition, each site obtains special table-level locks during rule processing to ensure that rule selection and transition tables are correct. This locking scheme may be used alone (Paradigm 3 in the lattice of Figure 2) or together with the rule-task executors and locking scheme of Section 3.1.2 (Paradigm 5 in the lattice).

[5]Hence, consistency violations can effectively cause *Autonomous Start* to degenerate to *No Autonomous Start*, which probably is the desired behavior in this case.

#### 3.2.1 Locking Scheme

We introduce two new types of locks: *transaction-grade* (or $T$) locks, and *saved-rule-grade* (or *SR*) locks. Intuitively, each site obtains T locks throughout subtransaction execution and SR locks throughout rule processing; T and SR locks on the same data items are incompatible across sites, allowing the detection of inconsistent access to data with respect to subtransactions and rule processing. In more detail, T locks are obtained by individual sites on local or remote data during subtransaction execution (along with the usual accumulation of L locks). Each site releases its T locks when it finishes its subtransaction, before rule processing begins. SR locks are obtained by individual sites on local or remote data during rule processing (along with L and possibly R locks). Unlike R locks, SR locks are accumulated for the duration of rule processing and released at final commit.

T locks do not conflict with each other across sites, nor do SR locks. SR locks do, however, conflict with T locks across sites. That is, site $S_i$ must wait to obtain an SR lock on a data item if another site $S_j$ holds a conflicting T lock on the same item. This correctly forces $S_i$'s rule processing to logically follow $S_j$'s subtransaction. Now suppose site $S_i$ requests a T lock on a data item while another site $S_j$ holds a conflicting SR lock on the same item. Then $S_j$'s rule processing has already logically preceded $S_i$'s subtransaction, and a consistency violation has occurred. In this case, all sites already processing rules must be rolled back to the start of their rule processing (releasing SR locks as appropriate) and rule processing is restarted.[6]

One final mechanism is needed to ensure correct rule processing. Suppose that, during rule processing at site $S_i$, a rule $r$ is selected for consideration, and suppose a subtransaction running at another site $S_j$ subsequently modifies the table at $S_i$ whose changes trigger $r$. Our scheme does not guarantee a lock conflict in this case. However, $S_j$'s subtransaction may not logically precede $S_i$'s rule processing: if $S_j$'s subtransaction actually preceded $S_i$'s rule processing, it could affect the value of $r$'s transition tables or even "untrigger" rule $r$.[7] As a second example, again consider rule $r$ selected for consideration at site $S_i$, and suppose a subtransaction running at site $S_j$ subsequently modifies a table at $S_i$ whose changes trigger a rule $r'$ with higher priority than $r$. Again, our

[6]As an optimization in some cases, such as when there are *No Multisite Rules*, rule processing need not be rolled back at every site but only at the site holding the conflicting SR lock.

[7]A rule is "untriggered" if it is triggered at some point during rule processing but not chosen for consideration, then subsequently no longer triggered because all triggering changes were undone. (Recall from Section 2 that rules consider the net effect of multiple operations.)

| *If conflict* | L at $S_j$ | T at $S_j$ | R at $S_j$ | SR at $S_j$ | Any at another transaction |
|---|---|---|---|---|---|
| L at $S_i$ | OK | OK | OK | OK | (1) |
| T at $S_i$ | OK | OK | OK | (2) | OK |
| R at $S_i$ | OK | OK | (3) | OK | OK |
| SR at $S_i$ | OK | (4) | OK | OK | OK |

(1) *$S_i$ waits for lock to be released by other transaction*
(2) *Consistency violation—roll back rule processing*
(3) *$S_i$ waits for lock to be released by other site or backs up and tries a different task or rule*
(4) *$S_i$ waits for lock to be released by other site or backs up and tries a different rule*

Figure 4: Summary of special lock types

scheme does not guarantee a lock conflict, but $S_j$'s subtransaction may not logically precede $S_i$'s rule processing: if $S_j$'s subtransaction actually preceded $S_i$'s rule processing, rule $r$ may not be eligible for consideration.

To solve these problems, during rule processing at each site $S_i$, before selecting a rule $r$ for consideration, $S_i$ obtains a table-level shared SR lock on the table whose changes trigger $r$ and on all tables whose changes trigger rules with higher priority than $r$ (recall that we are still assuming *No Intersite Priorities*). This ensures that if other sites' subtransactions subsequently try to modify the table whose changes triggered $r$, or subsequently try to modify a table whose changes trigger a rule with higher priority than $r$, then a consistency violation is detected. This approach is somewhat conservative, since a table-level shared lock prevents all modification operations on the table (inserts, deletes, and updates), not just the operations that trigger relevant rules. A variation of *predicate locking* [EGLT76] might be used here to refine our approach; see Section 5.1.

In summary, each site $S_i$ must:

1. Obtain T locks on data during subtransaction execution. If a conflicting SR lock is already held by another site, then there is a consistency violation and sites performing rule processing must be rolled back.

2. Obtain SR locks on data during rule processing. If a conflicting T lock is already held, $S_i$ can wait for its release or can back up and select a different rule to consider.

3. Obtain table-level shared SR locks on the trigger tables for all rules selected for consideration and for all rules with higher priority than rules selected for consideration.

Note that deadlock based on T and SR lock requests is not possible. A site requesting an SR lock may wait for another site to release a conflicting T lock, but a site requesting a T lock will not wait for another site to release a conflicting SR lock; rather, a consistency violation will occur. Hence, a cycle of waiting sites is not possible. Also note that, as in Section 3.1.2, with some extra bookkeeping rule selection can try to minimize lock waiting time. This may be particularly useful with respect to table-level SR locks.

### 3.2.2 Summary and Discussion of Special Locks

Before proving the correctness of our locking mechanisms for *Autonomous Start*, we summarize the special lock types introduced in the previous section and in Section 3.1.2. The behavior of these lock types with respect to each other is given in Figure 4. The rows in the table represent requests for locks by a site $S_i$ on behalf of a transaction $T$. The columns represent locks held by another site $S_j$ on behalf of the same transaction $T$, or in the case of the last column, any lock held on behalf of another transaction. An "OK" entry indicates that lock conflict is not possible, while numbered entries indicate that conflict is possible with the resulting behavior described below.

Since the Starburst database system is extensible at all levels [H$^+$90], it is fairly straightforward in Starburst to introduce new special lock types such as those suggested here [Ric91]. Notice, however, that whenever a site obtains a T lock on a data item, it also obtains an L lock on that item. In Starburst, and perhaps in other systems, it is possible and may be more efficient to implement T locks as a special *mode* of L lock: instead of obtaining T locks, a site can obtain L locks in "transaction" mode during execution of its subtransaction, then downgrade the locks to "normal" mode before rule processing. Similarly, if both *Multisite Rules* and *Autonomous Start* are used, then whenever a site obtains an R lock on a data item, it also obtains an SR lock on that item. Hence R locks might be implemented as a special mode of SR locks. This approach reduces locking to at most two types, each with at most two modes.

### 3.2.3 Correctness

The following theorem shows that the locking scheme of Section 3.2.1 ensures correct distributed rule processing even if some sites begin processing rules while other sites are still executing subtransactions.

**Theorem 3.2 (Autonomous Start)** Let $T$ be a transaction and $T_1, \ldots, T_n$ its distributed subtransactions. Assume that the effect of $T_1, \ldots, T_n$ is equivalent to what $T$ would have produced on the corresponding centralized database. Then $T_1, \ldots, T_n$ along with any behavior of distributed rule processing is equivalent to $T$ along with some valid behavior of centralized rule processing on the corresponding centralized database.

**Proof**: Omitted due to space constraints; see [CW92].

### 3.3 Intersite Priorities

So far we have assumed that priorities exist only between rules at the same site. This clearly is a desirable situation, since it gives maximum autonomy and parallelism, but for some applications intersite priorities may be useful or necessary. Our mechanisms for allowing intersite priorities combine additional locking (using the special lock types already introduced) with some communication between sites. We first consider Paradigm 4 in the lattice of Figure 2, in which there are *Multisite Rules* but *No Autonomous Start*. We then slightly modify the mechanisms to allow *Autonomous Start* (Paradigm 6).

#### 3.3.1 Without Autonomous Start

In a distributed database environment with sites $S_1, S_2, \ldots, S_n$, consider rule processing at a site $S_i$. Suppose there is a triggered rule $r_1$ at $S_i$ such that a rule $r_2$ at another site $S_j$ has higher priority than $r_1$. In the corresponding centralized environment, $r_1$ may be selected for consideration only if $r_2$ is not triggered at the same time. In the distributed environment, since there is concurrency in rule processing, if $r_1$ is to be selected for consideration then not only must we ensure that $r_2$ is not triggered at the same time, we also must ensure that $r_2$ cannot become triggered during $r_1$'s consideration: otherwise $r_1$ and $r_2$ may logically be triggered at the same time, and $r_1$'s consideration may incorrectly logically precede $r_2$'s.

Site $S_i$ can ensure that rule $r_2$ is not triggered at site $S_j$ by communicating with $S_j$ and, if necessary, by waiting for $r_2$ to no longer be triggered at $S_j$. To ensure that $r_2$ cannot become triggered during $r_1$'s consideration, site $S_i$ obtains a table-level shared R lock on the table at $S_j$ whose changes trigger $r_2$. (Recall that we are assuming *Multisite Rules* and *No Autonomous Start*, so all sites are processing rules at this point, and all sites are obtaining R locks during their rule considerations.) $S_i$ releases the table-level R lock when it completes consideration of rule $r$. (As in Section 3.2.1, table-level locks are somewhat conservative here; see Section 5.1.)

Hence, before site $S_i$ selects a rule $r$ for consideration, it performs the following protocol:

```
for each rule r' at a remote site Sj such that r'
has higher priority than r:
  A. obtain a table-level shared R lock on the
     table at Sj whose changes trigger r'
  B. ensure that (or wait until) r' is not
     triggered at Sj
```

Step A in this protocol requires communication with site $S_j$, and all sites must be prepared to participate in such communications. This is reflected in two additional steps in the rule-task executor (Figure 3). First, after step 1, each site receives and responds to inquiries about which rules are triggered at that site. Second, when a site completes consideration of a rule $r$, it notifies any sites that are waiting that $r$ is no longer triggered. The correctness of the latter step is worth explaining: Suppose site $S_j$'s rule-task executor receives an inquiry from site $S_i$ asking whether a rule $r$ is triggered, and suppose $r$ is triggered at $S_j$. If $S_i$ waits for $r$ to no longer be triggered, then $r$ cannot trigger itself during its consideration because $S_i$ holds a table-level R lock on $r$'s trigger table. (Deadlock clearly is possible due to this behavior but is not a significant problem, as explained below.) Consequently, after considering rule $r$, $S_j$ knows that $r$ is no longer triggered.

This protocol guarantees that, for the duration of rule $r$'s selection and consideration, no higher priority rules at other sites are or can become triggered. The cost to guarantee this is the possible waiting time to obtain locks, communication with other sites to determine whether rules are triggered, and the possible waiting time for rules to no longer be triggered at other sites.

This protocol is only necessary for considering rules that have lower priority than rules at other sites; all other rules can be considered as usual.[8] Furthermore, when several rules are eligible for consideration, the rule-task executor can select a rule based on the existence of intersite priorities, the availability of locks, and the information it receives about triggered rules at other sites. One further optimization is that when a site $S$ receives an inquiry about whether a given rule $r$ is triggered, if $r$ is indeed triggered then $S$ can select it for consideration as soon as possible.

Deadlock based on the R lock requests in this protocol is possible, but such a deadlock can always be broken by one site choosing a different rule to select—no rolling back is necessary.[9] A "true deadlock", in which there are no alternative rules to select at each site participating in the deadlock, is only possible in the presence of cyclic rule priorities, which are not allowed.

#### 3.3.2 With Autonomous Start

Now suppose *Autonomous Start* is used, so some sites may be processing rules while other sites are still executing subtransactions. In this case, during step 1 of the protocol in the preceding section, a table-level shared SR lock must be obtained along with the R lock. This ensures that, with respect to triggering remote rules that have higher priority than rules at $S_i$, all other sites' subtransactions logically precede $S_i$'s rule processing (recall Section 3.2.1). Also note in this paradigm that sites may receive inquiries about whether rules are triggered while they are still executing their subtransactions. Until rule

[8] Recall from Section 2, however, that rule priorities are transitive. So, for example, if a rule $r_1$ at site $S_1$ is specified to have lower priority than a rule $r_2$ at site $S_2$, then $r_1$ also has lower priority than all other rules at $S_2$ or elsewhere with higher priority than $r_2$. Consequently, intersite priorities, as well as local priorities in the presence of intersite priorities, must be specified with care.

[9] In fact, rule selection algorithms based on static properties of rule ordering can prevent deadlock entirely.

processing begins, these inquiries can be answered negatively; any inconsistency due to a subtransaction triggering higher priority rules will be detected by the locking mechanisms.

### 3.3.3 Correctness

To prove the correctness of our mechanisms for intersite rule priorities, we modify the proofs for Multisite Rules Theorem 3.1 and Autonomous Start Theorem 3.2, eliminating the assumption in these proofs that there are no priorities between rules at different sites.

**Theorem 3.3 (Intersite Priorities without Autonomous Start)** When there are *Multisite Rules* but *No Autonomous Start*, the protocol in Section 3.3.1 guarantees correct distributed rule processing even in the presence of *Intersite Priorities.*

**Proof**: Omitted due to space constraints; see [CW92].

**Theorem 3.4 (Intersite Priorities with Autonomous Start)** When there are *Multisite Rules* and *Autonomous Start*, the modified protocol in Section 3.3.2 guarantees correct distributed rule processing even in the presence of *Intersite Priorities.*

**Proof**: Omitted due to space constraints; see [CW92].

## 4 Parallel Environments

We consider parallel relational database environments in which the data resides at some number of separate *nodes.* Each table in the database may be *horizontally partitioned* (or *fragmented*) across multiple nodes, so that some rows of the table reside at one node while other rows reside at other nodes [DGS+90, OV91]; we assume that data is not replicated across nodes. Horizontal partitioning usually is defined by a set of mutually exclusive covering predicates. Based on the partitioning, user transactions are parallelized to execute across multiple nodes. The typical requirement to provide (fragmentation) transparency is that the effect of parallel transaction execution must be equivalent to the effect of the original transaction on the corresponding non-fragmented database.

As in distributed environments, we would like rule processing to occur separately at each node, reponding to the database changes at that node only. If rules are triggered by changes to table fragments, then the mechanisms given in Section 3 for distributed environments can be directly adapted for parallel environments. However, this provides no notion of transparency in rule definition (since the rule definer must be aware of the fragmentation), and it forces rules to be revised if data is repartitioned. To provide some transparency, we allow rules to be defined that are triggered by changes to entire tables, but these rules must satisfy certain criteria for *partitionability.* Based on the fragmentation, each rule is "partitioned" automatically into multiple rules that reside at the appropriate nodes. Each partitioned rule is triggered by, reads, and modifies only table fragments at its own node. Based on the criteria for partitionability, parallel rule processing using the partitioned rules is provably equivalent to centralized rule processing using the original rules on the corresponding non-fragmented database.

Since partitioned rules do not read or modify remote data, parallel rule processing for partitioned rules corresponds to *No Multisite Rules* and *No Intersite Priorities* in our distributed environments; *Autonomous Start* may be used if desired. (That is, we are considering parallel versions of Paradigms 1 and 3 in the lattice of Figure 2.) We note, however, that not all useful rules satisfy our criteria for partitionability. For maximum flexibility but at the expense of transparency, it may be preferable to specify rules directly on table fragments, then apply the appropriate mechanisms for any paradigm from Figure 2.

### 4.1 Partitionability

Let $N_1, N_2, \ldots, N_n$ denote the $n$ nodes in a parallel database environment. Consider a rule $r$ that is triggered by, reads, and modifies tables $t_1, t_2, \ldots, t_j$. As an initial requirement for partitionability, there must be some group of nodes, call them $N_1, N_2, \ldots, N_k$, over which each table $t_1, t_2, \ldots, t_j$ is fragmented. (That is, $t_1, t_2, \ldots, t_j$ are fragmented over the same nodes.) In practice, this is common for most rules and fragmentations. Rule $r$ is partitioned as follows: For each node $N_i$, $1 \leq i \leq k$, create a rule $r_i$ from $r$ by changing each table reference $t_m$ in $r$ to instead specify the fragment of table $t_m$ that resides at node $N_i$. If the original rule $r$ satisfies our criteria for partitionability given below, then each partitioned rule $r_i$ can be installed at its node $N_i$, and parallel rule processing will be correct.

To specify the criteria for partitionability, we first introduce some notation. Let table $t_m$ be rule $r$'s trigger table, and suppose $r$ is triggered and considered during rule processing in the non-fragmented environment. Let $\Delta$ denote the triggering changes to table $t_m$, let $C$ denote the result of evaluating $r$'s condition, and let $A$ denote the effect of executing $r$'s action. (If $r$'s condition is omitted then $C = true$.) In the fragmented environment, for a set $\Delta$ of changes to table $t_m$, let $\Delta_i$ denote the same set of changes projected onto the fragment of $t_m$ that resides at node $N_i$. If a partitioned rule $r_i$ is considered during rule processing at $N_i$, let $C_i$ denote the result of evaluating $r_i$'s condition and let $A_i$ denote the effect of executing $r_i$'s action. Finally, let $A \mid N_i$ denote the effect of $r$'s action on the table fragments that reside at node $N_i$, and let $\Phi$ denote an action with no effect.

**Definition 4.1 (Criteria for Partitionability)** Given any database state, any non-fragmented set of changes $\Delta$ that trigger rule $r$, and the corresponding sets of fragmented changes $\Delta_1, \ldots, \Delta_k$ (that may trigger $r_1, \ldots, r_k$), the following four conditions must hold:

1. *Partitionability of r's action*:

   For all $i$, $1 \leq i \leq k$, $A \mid N_i = A_i$

   That is, the effect of $r$'s action on the table fragments that reside at node $N_i$ is equivalent to the effect of $r_i$'s action at $N_i$.

2. *Partitionability of r 's condition*:

   For all $i$, $1 \leq i \leq k$, $C_i \Rightarrow C$

   That is, if $r_i$'s condition is true at node $N_i$, then $r$'s condition is true on the non-fragmented tables. (Conversely, if $r$'s condition is false on the non-fragmented tables, then $r_i$'s condition is false at $N_i$.)
3. *Not triggered implies no action*:

   For all $i$, $1 \leq i \leq k$, $(\Delta_i = \emptyset) \Rightarrow (A_i = \Phi)$

   That is, if $r_i$ has no triggering changes then its action would have no effect.
4. *False condition implies no action*:

   For all $i$, $1 \leq i \leq k$, $(C_i = \mathit{false}) \Rightarrow (A_i = \Phi)$

   That is, if $r_i$'s condition is false then its action would have no effect.

□

Further intuition for these criteria is deferred until the next section, where we give examples of rules that do and do not satisfy the four conditions. In practice, we expect the criteria to be satisfied by many, but not all, fragmentations and rule applications.

The criteria in Definition 4.1 are dynamic properties of rule behavior based on arbitrary database states and triggering changes; hence, they can be quite difficult to verify for complex rules. Many rules obviously do satisfy the criteria, however, as illustrated by our simple examples below. Not included in this paper, but as very important future work, we plan to identify static properties of rules that guarantee the criteria. This will allow automatic analysis of rule partitionability, freeing the rule definer from any knowledge of the database fragmentation.

Suppose now that the partitionability criteria are satisfied for a set of rules $r^1, r^2, \ldots, r^k$. Each rule is partitioned into multiple rules, with one partitioned rule installed at each relevant node. Any user-defined priorities between original rules are reflected between partitioned rules at the same node. That is, if $r^x$ has higher priority than $r^y$, then at each node $N_i$ with both a partitioned rule $r^x_i$ and a partitioned rule $r^y_i$, rule $r^x_i$ has higher priority than $r^y_i$.

For correctness in this paradigm, it is necessary that non-prioritized rules are selected for consideration in the same order across sites. That is, suppose rules $r^x$ and $r^y$ have no relative priority, and let $r^x$ and $r^y$ both have partitioned rules at nodes $N_i$ and $N_j$. Then $r^x$'s partitioned rule should be selected for consideration before $r^y$'s partitioned rule when both are triggered at node $N_i$ if and only if the same is true at node $N_j$. This can be achieved by using any valid total ordering of the rules consistently across sites. Although this may appear to be a strong requirement, many current database rule systems perform rule selection in this way [ACL91].

Finally, observe that we are assuming a scenario in which all desired rules satisfy the partitionability criteria, hence the rules can be partitioned as described and will behave correctly (as proven in Section 4.3). Some applications, however, may require rules that are not partitionable in this way. If so, such rules can be partitioned "by hand" (by a knowledgeable designer) so they specify table fragments rather than entire tables, but this is at the loss of guaranteed equivalence between parallel rule processing using the partitioned rules and centralized rule processing using the original rules.

### 4.2 Examples

In this section, we give eight examples, one rule that satisfies and one that does not satisfy each of the four criteria in Definition 4.1 of partitionability. We assume some familiarity with SQL [IBM88], and we ask the reader to recall the syntax for Starburst production rules from Section 2. We consider a standard database of employee information stored in a table called **emp**; we also assume there is a copy of table **emp** called **emp-copy**, and a table **high-paid** of highly paid employees. For each of the four rules used to illustrate non-partitionability, notice that the rule meanwhile does satisfy the other three criteria for partitionability.

**Example 4.1** The following rule satisfies Criterion 1, $A \mid N_i = A_i$, assuming tables **emp** and **emp-copy** are partitioned based on the same predicate:

```
create rule r on emp
when inserted
then insert into emp-copy
       (select * from inserted)
```

**Example 4.2** The rule in Example 4.1 does not satisfy Criterion 1 if tables **emp** and **emp-copy** are partitioned based on different predicates, since an employee inserted into a fragment of **emp** at one node might belong to a fragment of **emp-copy** at a different node.

**Example 4.3** The following rule satisfies Criterion 2, $C_i \Rightarrow C$:

```
create rule r on emp
when inserted
if exists
     (select * from inserted where salary > 50)
then ...
```

**Example 4.4** The following rule does not satisfy Criterion 2 since the **not exists** condition may be true for a fragment of **emp** at one node but false in the non-fragmented database.

```
create rule r on emp
when inserted
if not exists
     (select * from inserted where salary > 50)
then ...
```

**Example 4.5** The rule in Example 4.1 satisfies Criterion 3, $(\Delta_i = \emptyset) \Rightarrow (A_i = \Phi)$.

**Example 4.6** The following rule does not satisfy Criterion 3 since salaries should still be increased in fragments of **emp** with no deletions:

```
create rule r on emp
when deleted
then update emp set sal = sal + 10
```

**Example 4.7** The following rule satisfies Criterion 4, $(C_i = \text{false}) \Rightarrow (A_i = \Phi)$:

```
create rule r on emp
when inserted
if exists
     (select * from inserted where salary > 50)
then insert into high-paid
       (select * from inserted where salary > 50)
```

**Example 4.8** The following rule does not satisfy Criterion 4 since there may be inserted employees with salary $> 25$ in fragments of **emp** for which there are no inserted employees with salary $> 50$:

```
create rule r on emp
when inserted
if exists
     (select * from inserted where salary > 50)
then update emp set sal = sal + 10
     where emp.id in
       (select id from inserted where salary > 25)
```

In these examples, and in general, we believe that rules satisfying the partitionability criteria are considerably more intuitive and realistic than rules not satisfying these criteria.

### 4.3 Correctness

For brevity, the proof of correctness assumes *No Autonomous Start*, i.e. it uses Paradigm 1 in the lattice of Figure 2. From the proof, it is clear that our mechanisms for *Autonomous Start* (Section 3.2) can be added without compromising correctness.

**Theorem 4.9 (Parallel Rule Processing)** Consider any set of rules such that all rules satisfy the partitionability criteria in Definition 4.1, and let each rule be partitioned as described above. Let $T$ be a transaction and assume that the effect of $T$'s parallelized execution is equivalent to what $T$ would have produced on the corresponding non-fragmented database. Any behavior of parallel rule processing following $T$'s parallelized execution is equivalent to some valid behavior of centralized rule processing following $T$ on the corresponding non-fragmented database.

**Proof**: Omitted due to space constraints; see [CW92].

## 5 Conclusions and Future Work

We have described a number of mechanisms that allow production rule processing to occur separately at each site or node in a parallel or distributed database environment. Our mechanisms cover a variety of different restrictions, assumptions, and environments; in all cases, parallel or distributed rule processing is provably equivalent to rule processing in the corresponding centralized environment. We believe this work establishes many new ideas and useful initial frameworks for parallel and distributed rule processing. There is, however, substantially more research do be done, both in adapting the frameworks to particular rule languages and database environments, and in improving and extending the mechanisms in this paper. The remainder of this section introduces a number of topics we plan to consider in the future.

### 5.1 Improvements to our mechanisms

*More refined locking*: Several of our mechanisms rely on setting table-level shared locks to ensure that rules cannot become triggered, or that transition tables remain consistent for triggered rules; see Sections 3.2.1 and 3.3. Although these mechanisms do work correctly, they may unnecessarily inhibit parallelism. As an example, suppose a table-level shared lock is set on a table $t$, preventing all modifications to $t$. If the lock was set only to prevent a rule from being triggered by insertions into $t$, then deletes and updates on $t$ could still be allowed. A different notion of locking from the standard shared and exclusive locks might be used here—one in which it is possible to lock specific operations on tables.

*Optimizing selection and backup*: For a number of our mechanisms, we have suggested that rule selection, task selection, and backup techniques might be used to minimize lock waiting time and consequently maximize parallelism. Clearly, there is interesting work in developing concrete algorithms for this. A related issue to be addressed is fairness in task selection, as mentioned in Section 3.1.1.

### 5.2 Extensions to our mechanisms

*Determining partitionability*: We plan to develop a static analysis framework that can take a rule and a description of horizontal partitioning of the tables referenced in the rule (specified, e.g., as a set of predicates), then determine automatically whether the rule satisfies the partitionability criteria of Definition 4.1. Such methods are likely to be conservative or example-based, but even this could prove to be quite useful in practice.

*Other notions of partitionability*: In Section 4.1, we considered a notion of rule partitionability in which partitioned rules are triggered by, read, and modify table fragments at a single node. There is a different possible notion of partitionability, in which partitioned rules are triggered by table fragments at a single node, but in their conditions and actions reference entire tables. (Conditions and actions are then parallelized by the query optimizer.) Rule processing in this case would use our mechanisms for *Multisite Rules*. We plan to develop criteria for this alternative notion of partitionability. It is interesting to note that these criteria are not necessarily weaker than our criteria in Definition 4.1—some rules may be partitionable in our current scheme but not in this scheme, and vice-versa.

*Exploiting static rule properties*: All of our mechanisms in Section 3 are based on run-time behavior, such as locking and communication protocols. Some previous work on parallelizing OPS5 rule processing (recall Section 1.1) has incorporated static properties of rules, e.g.

determining that certain rules cannot interfere so coordination is not required. We similarly might be able to statically identify such rules, then eliminate locking and communication appropriately.

### 5.3 Other environments

*Replication*: We plan to develop mechanisms for rule processing in parallel or distributed environments with replicated tables or table fragments.

*Vertical fragmentation*: We plan to consider rule processing in parallel or distributed environments where tables are partitioned across sites or nodes by columns. This might require additional mechanisms, or it might use a notion of partitionability analogous to that used for horizontal fragmentation in Section 4.

*Other transaction paradigms*: Our mechanisms are based on conventional atomic transactions; we would like to extend our mechanisms to encompass newer distributed transaction paradigms, such as long-lived transactions with save points, compensating actions, etc.

### 5.4 Data organization based on rules

Some database rule applications are particularly rule-intensive—rules are triggered frequently and rule processing may be lengthy and complex, e.g. large-scale inferencing systems. (In other rule applications, such as traditional database applications using rules as monitors, rules are triggered infrequently and rule processing is relatively simple.) For rule-intensive applications in parallel and distributed environments, it may be advisable for data organization to conform to rules (rather than to predicted transactions) as suggested in [BKK87]. (In the ideal situation, of course, it can conform to both.) In the parallel environment, tables might be fragmented to ensure that rules are partitionable; in the distributed environment, tables might be located so that rules need not read or modify tables at remote sites and intersite priorities are not required. We are interested in understanding design criteria and developing methodologies for such environments.

### 5.5 Implementation and experimentation

Finally, to fully develop and accurately assess our mechanisms, it is necessary to implement and experiment with them in a running system prototype.

## Acknowledgements

Thanks to Alex Aiken and Laura Haas for helpful comments on an initial draft and to Mike Olson and Mike Stonebraker for useful observations on rollback.

## References

[ACL91] R. Agrawal, R.J. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, September 1991.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[BKK87] J. Bein, R. King, and N. Kamel. MOBY: An architecture for distributed expert database systems. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 13–20, Brighton, England, September 1987.

[CW92] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. IBM Research Report RJ 8564, IBM Almaden Research Center, January 1992.

[DGS+90] D. Dewitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[GJ91] N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 327–336, Barcelona, Spain, September 1991.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1992.

[Gup86] A. Gupta. *Parallelism in Production Systems.* PhD thesis, College of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1986.

[H+90] L.M. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[Han89] E.N. Hanson. An initial report on the design of Ariel: A DBMS with an integrated production rule system. *SIGMOD Record, Spe-*

*cial Issue on Rule Management and Processing in Expert Database Systems*, 18(3):12–19, September 1989.

[HSL92] I.-M. Hsu, M. Singhal, and M.T. Liu. Distributed rule processing in active databases. In *Proceedings of the Eighth International Conference on Data Engineering*, Phoenix, Arizona, February 1992.

[HW93] E.N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[IBM88] IBM Form Number SC26-4348-1. *IBM Systems Application Architecture, Common Programming Interface: Database Reference*, October 1988.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, May 1989.

[OV91] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Pas89] A.J. Pasik. *A Methodology for Programming Production Systems and its Implications on Parallelism.* PhD thesis, Department of Computer Science, Columbia University, New York, 1989.

[Ric91] J.E. Richardson. LCK: The lock manager. Internal document, IBM Almaden Research Center, San Jose, California, January 1991.

[RSD91] L. Raschid, T. Sellis, and A. Delis. On the concurrent execution of production rules in a database implementation. Technical Report CS-TR-2751, Department of Computer Science, University of Maryland, September 1991.

[SDLT86] M.I. Schor, T.P. Daly, H.S. Lee, and B.R. Tibbitts. Advances in RETE pattern matching. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 226–232, Philadelphia, Pennsylvania, August 1986.

[Sel89] T. Sellis, editor. *Special Issue on Rule Management and Processing in Expert Database Systems,* SIGMOD Record 18(3), September 1989.

[SG90] J.G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Boston, Massachusetts, 1990.

[SHT90] J. Srivastava, K.-W. Hwang, and J.S.E. Tan. Parallelism in database production systems. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 121–128, Los Angeles, California, February 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, Atlantic City, New Jersey, May 1990.

[SM84] S.J. Stolfo and D. Miranker. DADO: A parallel processor for expert systems. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1984.

[TTC+90] G. Thomas, G.R. Thompson, C.-W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22(3):237–266, September 1990.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.