To appear in: *IEEE Transactions on Knowledge and Data Engineering*

The Starburst Active Database Rule System

Jennifer Widom*

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

widom@cs.stanford.edu

Abstract

This paper describes our development of the Starburst Rule System, an active database rules facility integrated into the Starburst extensible relational database system at the IBM Almaden Research Center. The Starburst rule language is based on arbitrary database state transitions rather than tuple- or statement-level changes, yielding a clear and flexible execution semantics. The rule system has been implemented completely. Its rapid implementation was facilitated by the extensibility features of Starburst, and rule management and rule processing is integrated into all aspects of database processing.

Index terms: *active database systems, database production rules, extensible database systems, expert database systems*

1 Introduction

Active database systems allow users to create *rules*—rules specify data manipulation operations to be executed automatically whenever certain events occur or conditions are met. Active database rules provide a general and powerful mechanism for traditional database features such as integrity constraint enforcement, view maintenance, and authorization checking; active database rules also support non-traditional database features such as version management and workflow control. Because active database rules are similar to the forward-chaining *production rules* used by Artificial Intelligence applications, active database systems also provide a convenient and efficient platform for large and efficient knowledge bases and expert systems.

In this paper we describe our development of the *Starburst Rule System*, an active database extension to the Starburst prototype extensible relational database system at the IBM Almaden Research Center. We cover both our design of the rule language and our implementation of rule processing as an extension to Starburst. The Starburst rule language

*This work was performed while the author was at the IBM Almaden Research Center, San Jose, CA.

differs from most other active database rule languages in that it is based on arbitrary database state transitions rather than tuple- or statement-level changes, permitting an execution semantics that is both cleanly-defined and flexible. The implementation of the Starburst Rule System was completed rapidly and relies heavily on the extensibility features of Starburst. The Starburst rule processor differs from most other active database rule systems in that it is completely implemented, and it is fully integrated into all aspects of database processing, including query and transaction processing, concurrency control, rollback recovery, error handling, and authorization.

The paper proceeds as follows. In Section 2 we survey other active database rule systems. In Section 3 we describe the syntax of the Starburst rule language and in Section 4 we specify the semantics of rule execution; examples are given in Section 5. The architecture of the rule system implementation is described in Section 6. Section 7 covers several implementation features in more detail, including transition information maintenance, concurrency control, authorization, and error handling. Section 8 concludes and provides a retrospective discussion of the Starburst Rule System, highlighting what we feel are the successes and the failures of our language design and implementation. Finally, in Section 9 we mention several applications of the Starburst Rule System, and we discuss future directions of this work.

2 Related Work

Numerous other active database systems have been designed and some have been implemented. The three systems closest to the Starburst Rule System are *Ariel* [31], the second version of the *POSTGRES Rule System* [42], and *Chimera* [12,14]. The Ariel system has a rule language and execution semantics based closely on *OPS5* [9], a production rule language originally designed for expert systems. The Ariel project has focused on the design of an OPS5-like rule language for the database setting, and on methods for highly efficient rule condition testing using variations on the *Rete* and *TREAT* algorithms designed for OPS5 [44]. The Ariel rule language is fully implemented using the Exodus database toolkit [31]. The POSTGRES Rule System, sometimes referred to as *PRS2* to distinguish it from an earlier proposal [41], focuses in both its language and its implementation on providing several different classes of rules, each appropriate for a particular suite of applications. There are two implementations of the POSTGRES Rule System, one based on run-time marking of tuples affected by rules, the other based on compile-time rewriting of queries to incorporate the effects of rules [42]. The Chimera system combines object-oriented, deductive, and active database technology. Its active rule language is based on Starburst's, with extensions for object-orientation and for "configurable" rule semantics (see Section 4). A first prototype of Chimera has been implemented, employing some techniques adapted from Starburst [12].

There are several other relational active database systems, not as closely related to Starburst as the systems described above. Two projects, *DATEX* [8] and *DIPS* [38], implement the OPS5 rule language using an underlying database system and special indexing techniques to support efficient processing of large rule and data sets. The *PARADISER* project also uses a database system for efficient processing of expert system rules, but in PARADISER the focus is on distributed and parallel rule processing [23]. *RPL* (for *Relational Production Language*) was an early project in relational active database systems; RPL includes an OPS5-like rule language based on relational queries and a prototype implementation in which rule processing is loosely coupled to a commercial relational DBMS [22]. *A-RDL* is an extension to the *RDL* deductive database system that supports active rules [39]. The *Alert* project explores how active rules can be supported on top of a passive database system with minimal extensions [37]. Finally, *Heraclitus* is a relational database programming language with *delta relations* as first-class objects; a primary goal of the Heraclitus language is to simulate and support active rule processing [29].

One early project and numerous recent efforts (including Chimera) consider active object-oriented database systems. Although some issues in active database systems are common to both relational and object-oriented environments, there are many significant differences; furthermore, to date most object-oriented active database systems do not have implementations that are as advanced as their relational counterparts. *HiPAC* was a pioneering project in the area of active database systems; *HiPAC* includes a very powerful rule language for an object-oriented data model, a flexible execution semantics, and several main-memory experimental prototypes [20]. Recently there has been an explosion of projects in object-oriented active database systems—many of these projects are still preliminary; see e.g. [3,4,6,7,10,11, 24,25,26,27,33,35].

Several previous papers have described language, implementation, or application development issues related to the Starburst Rule System. An initial proposal for the Starburst rule language appears in [49]. [48] describes how the extensibility features of the Starburst prototype are used in implementing the rule system. Details of Starburst's rule priority system are given in [1]. A series of papers describe how rules in the Starburst language can be generated automatically from specifications for particular applications: integrity constraints are considered in [15], view maintenance in [16], deductive databases in [19], and heterogeneity management in [18]. A denotational semantics for the Starburst rule language is given in [45], while [2] describes methods for static analysis of Starburst rules. Finally, [17] discusses how the Starburst Rule System can be extended for parallel and distributed database environments. Except for a short overview in [46] and an unpublished user's guide [47], this is the first paper to provide a complete description of the final, operational, Starburst Rule System.

3 Syntax of Rule Language

The syntax of the Starburst rule language is based on the extended version of SQL supported by the Starburst database system [30]. The Starburst rule language includes five commands for defining and manipulating rules: **create rule**, **alter rule**, **deactivate rule**, **activate rule**, and **drop rule**. In addition, rules may be grouped into *rule sets*, which are defined and manipulated by the commands **create ruleset**, **alter ruleset**, and **drop ruleset**. We describe each of these eight commands below. The Starburst Rule System also includes some simple user commands for querying and displaying rules, which we omit from this paper (see [47] for details), and commands for user or application initiation of rule processing, which we describe in Section 4.

3.1 Rule Creation

Rules are defined using the **create rule** command. The syntax of this command is:

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

The *name* names the rule, and each rule is defined on a *table*. Square brackets indicate clauses that are optional.

The **when** clause specifies what causes the rule to be *triggered*. Rules can be triggered by any of the three relational data modification operations: **inserted**, **deleted**, and **updated**. The **updated** triggering operation may include a list of columns; specifying **updated** without a column list indicates that the rule is triggered by updates to any column. Each rule specifies one or more triggering operations in its **when** clause; any of the specified operations on the rule's table will trigger the rule.

The **if** clause specifies a condition to be evaluated once the rule is triggered. A rule condition is expressed as an unrestricted **select** statement in Starburst's SQL. The condition is true if and only if the **select** statement produces at least one tuple. The **if** clause may be omitted, in which case the rule's condition is always true. Note that using an unrestricted SQL **select** statement as the condition part of a rule is equivalent to using an unrestricted SQL predicate: any SQL predicate can be transformed into an equivalent SQL **select** statement (on a dummy table), while any SQL **select** statement can be transformed into an equivalent SQL predicate (using **exists**).

The **then** clause specifies a list of actions to be executed when the rule is triggered and its condition is true. Each action may be any database operation, including data manipulation commands expressed using Starburst's SQL (**select**, **insert**, **delete**, **update**), data definition commands (e.g. **create table**, **drop rule**), and **rollback**. The actions are executed sequentially in the order listed.

The optional **precedes** and **follows** clauses are used to specify priority orderings between rules. When a rule R_1 specifies a rule R_2 in its **precedes** list, this indicates that if both rules are triggered at the same time, then R_1 will be considered first, i.e. R_1 precedes R_2. If R_1 specifies R_2 in its **follows** list, this indicates that if both rules are triggered at the same time, then R_2 will be considered first. Cycles in priority ordering are not permitted.

Rule conditions and actions may refer to arbitrary database tables; they also may refer to special *transition tables*. There are four transition tables: **inserted**, **deleted**, **new-updated**, and **old-updated**. If a rule on a table T specifies **inserted** as a triggering operation, then transition table **inserted** is a logical table containing the tuples that were inserted into T causing the rule to be triggered; similarly for **deleted**. Transition table **new-updated** contains the current values of updated tuples; **old-updated** contains the original values of those tuples.[1] A transition table may be referenced in a rule only if it corresponds to one of the rule's triggering operations.

Note that Starburst rules do not include a **for each row** option, or a **before** option for triggering operations. (Readers may be familiar with these options from commercial SQL-based trigger systems [32,34].) Neither option is appropriate in the context of rules that are evaluated over arbitrary transitions; this issue is addressed further in Section 8.

3.2 Other Rule Commands

The components of a rule can be changed after the rule has been defined; this is done using the **alter rule** command. The syntax of this command is:

> **alter rule** *name* **on** *table*
> [**if** *condition*]
> [**then** *action-list*]
> [**precedes** *rule-list*]
> [**follows** *rule-list*]
> [**nopriority** *rule-list*]

[1] If a rule is triggered by **updated** on any column, then transition tables **new-updated** and **old-updated** contain tuples for which any column was updated. If a rule is triggered by **updated** on particular columns, then transition tables **new-updated** and **old-updated** contain the entire tuples for which at least one of the specified columns was updated.

The **if**, **then**, **precedes**, and **follows** clauses in this command use the same syntax as the corresponding clauses in the **create rule** command. The **if** clause specifies a new rule condition that replaces the existing one. Similarly, the **then** clause specifies a new list of actions that replaces the existing list. The **precedes** and **follows** clauses specify rules to be added to the existing **precedes** and **follows** lists, while the **nopriority** clause is used to remove priority orderings. Notice that the **when** clause of a rule may not be altered; to change triggering operations, a rule must be dropped and then re-created (this restriction is due to implementation details).

An existing rule can be deleted by issuing the **drop rule** command:

> **drop rule** *name* **on** *table*

Sometimes it is useful to temporarily *deactivate* rules (particularly for debugging purposes). When a rule is deactivated, it will not be triggered and its actions will not be executed, even if its triggering operations occur. A deactivated rule behaves as if the rule were dropped, except it remains in the system and can easily be reactivated. A rule is deactivated by issuing the command:

> **deactivate rule** *name* **on** *table*

To reactivate a rule that has been deactivated, the following command is issued:

> **activate rule** *name* **on** *table*

3.3 Rule Sets

We have provided a basic facility in the Starburst Rule System for grouping rules into sets. Rule sets can be used to structure rule applications in conjunction with the **process ruleset** command, described in Section 4.3.[2] A rule set is defined using the **create ruleset** command:

> **create ruleset** *name*

Rules are added to and deleted from a rule set using the **alter ruleset** command:

> **alter ruleset** *name*
> [**addrules** *rule-list*]
> [**delrules** *rule-list*]

Each rule may be in any number of rule sets (including none), and each set may contain any number of rules. A rule set is deleted by issuing the command:

> **drop ruleset** *name*

[2]Rule sets might also be used to group rules for the purposes of shared priorities, activation/deactivation of multiple rules, or inheriting common components, but such features are not provided in the current Starburst Rule System.

4 Semantics of Rule Execution

In this section we explain the semantics of rule execution in Starburst, including the relationship of rule processing to query and transaction processing.[3] For the descriptions of rule behavior in this section, we assume that some number of rules already have been created, and we assume that these rules are not altered, deactivated, activated, or dropped. (The subtle interactions between transactions in which rules are changed and other concurrently executing transactions are discussed in Section 7.3.)

Rules are processed automatically at the end of each transaction that triggers at least one rule. In addition, rules may be processed within a transaction when special user commands are issued. The semantics of rule execution is closely tied to the notion of database state transitions. Hence, we begin by describing transitions, then we describe end-of-transaction rule processing, and finally we describe command-initiated rule processing.

4.1 Transitions

When we determine whether a rule is triggered, and when we evaluate a rule's transition tables, this is based on a precise notion of database state *transition*. A transition is the transformation from one database state to another that results from the execution of a sequence of SQL data manipulation operations. Since rule processing always occurs within a transaction and is defined with respect to the operations performed in that transaction only, we need not consider issues such as concurrent transactions and failures in defining rule semantics. Furthermore, since rules are triggered by data modification only, and not by data retrieval, execution of SQL **select** statements also need not be considered.

Suppose a sequence of SQL data modification operations (**insert**, **delete**, and/or **update**) is executed, transforming the database from a state S_0 to a state S_1. We depict the resulting transition τ as:

$$S_0 \xrightarrow[\tau]{} S_1$$

Rather than considering the individual operations creating a transition, rules consider the *net effect* of transitions. The net effect of a transition consists of a set of inserted tuples, a set of deleted tuples, and a set of updated tuples. Considering transition τ above, we associate with each inserted tuple its value in state S_1, with each deleted tuple its value in state S_0, and with each updated tuple its (old) value in S_0 and its (new) value in S_1. If a tuple is modified more than once during a transition, it still appears in at most one set in the net effect of the transition. Specifically:

[3] More detailed and formal treatments of Starburst's rule execution semantics can be found in [45,49].

- If a tuple is inserted and then updated, we consider this as an insertion of the updated tuple.

- If a tuple is updated and then deleted, we consider this as a deletion of the original tuple.

- If a tuple is updated more than once, we consider this as an update from the original value to the newest value.

- If a tuple is inserted and then deleted, we do not consider it in the net effect at all.

For clarity, we use dashed arrows to denote transitions that result from user- or application-generated data manipulation operations, while we use solid arrows denote transitions that result from rule-generated operations. For example, the following depicts a user-generated transition followed by three rule-generated transitions:

$$\begin{tikzcd} S_0 \arrow[r, dashed, "\tau_1"'] & S_1 \arrow[r, "\tau_2"'] & S_2 \arrow[r, "\tau_3"'] & S_3 \arrow[r, "\tau_4"'] & S_4 \end{tikzcd}$$

Rules often consider *composite* transitions. For example, a rule might be triggered by a composite transition τ that is the net effect of transitions τ_1, τ_2, and τ_3. We depict this as:

$$\begin{tikzcd} S_0 \arrow[r, dashed, "\tau_1"'] \arrow[rrr, bend left, "\tau"] & S_1 \arrow[r, "\tau_2"'] & S_2 \arrow[r, "\tau_3"'] & S_3 \end{tikzcd}$$

4.2 End-of-Transaction Rule Processing

Suppose a transaction X is executed and suppose that the net effect of the data modification operations performed by X includes at least one operation that triggers at least one rule; then rule processing is invoked automatically at the end of transaction X, before X commits. Transaction X itself creates the initial triggering transition. As rules are executed, they create additional transitions that may trigger additional rules or may trigger the same rules again. If a rule action executes **rollback**, then the entire transaction aborts. Otherwise, the entire transaction commits when rule processing terminates.

Rule processing itself consists of an iterative loop. In each iteration:

1. A triggered rule R is selected for *consideration* such that no other triggered rule has priority over R (details of rule selection are discussed in Section 4.4 below).

2. R's condition is evaluated.

3. If R's condition is true, R's actions are executed.

For step 1, a rule is triggered if one or more of its triggering operations occurred in the composite transition since the last time the rule was considered, or since the start of the transaction if the rule has not yet been considered.

As illustration, suppose a user transaction creates transition τ_1. Suppose a rule R is triggered by transition τ_1, it is selected for consideration, its condition is true, and its actions are executed:

$$\begin{tikzcd} S_0 \arrow[r, dotted, "\tau_1"'] \arrow[rr, bend left, "\tau"] & S_1 \arrow[r, "\tau_2\,(R)"'] & S_2 \end{tikzcd}$$

At this point, any rule that was not considered in state S_1 is triggered if one or more of its triggering operations occurred in the composite transition τ; R is triggered (again) if one or more of its triggering operations occurred in transition τ_2.

We have chosen this particular semantics for rule execution in part because it has the useful property that every rule considers every change exactly once.[4] This property is illustrated by the following example, which shows the (composite) transitions considered by a rule R during several steps of rule processing:

$$\begin{tikzcd} S_0 \arrow[r, dotted, "\tau_1"'] \arrow[rr, bend left] & S_1 \arrow[r, "\tau_2\,(R')"'] & S_2 \arrow[r, "\tau_3\,(R)"'] \arrow[r, bend left] & S_3 \arrow[r, "\tau_4\,(R)"'] \arrow[rrr, bend left] & S_4 \arrow[r, "\tau_5\,(R')"'] & S_5 \arrow[r, "\tau_6\,(R')"'] & S_6 \end{tikzcd}$$

The first time rule R is considered, at state S_2, R uses the changes since initial state S_0, i.e. the changes made by the initial user transaction and subsequent execution of a rule R'. In its second consideration, at state S_3, R uses the changes since S_2. If R is considered a third time, at state S_6, it uses the changes since state S_3. The upper arrows depict the (composite) transitions used by rule R each time it is considered, illustrating clearly that R considers every change exactly once.

Finally, note that during condition evaluation and action execution, the contents of a rule's transition tables always reflect the rule's triggering transition.

---

[4]Certainly there are many other possible choices for the semantics of rule execution. Our choice seems appropriate for many applications; however, it is our belief that for every choice of semantics it is possible to concoct a reasonable example for which that semantics is inconvenient or inappropriate. The recent *Chimera* active rule system addresses this issue by allowing its users to choose between a number of alternative semantics [14].

4.3 Rule Processing Commands

While end-of-transaction rule processing is sufficient for many applications, we have found that in some cases it is useful for rules to be processed within a transaction (for example, to verify consistency after some operations have been executed but before the transaction is complete). For this, the Starburst Rule System provides three commands:

process rules
process ruleset *set-name*
process rule *rule-name*

Execution of the **process rules** command invokes rule processing with all rules eligible to be considered and executed. The behavior of rule processing in response to a **process rules** command is identical to end-of-transaction rule processing. In particular, recall from the previous section that a rule is triggered if one or more of its triggering operations occurred in the composite transition since the last time the rule was considered, or since the start of the transaction if the rule has not yet been considered. This behavior is valid even if rules are processed multiple times within a transaction as well as at the end of the transaction, and this behavior retains the semantic property that every rule considers every change exactly once.

Execution of the **process ruleset** command invokes rule processing with only those rules in the specified set eligible to be considered and executed. Again, the behavior of rule processing is identical to end-of-transaction rule processing, except in this case any rules that are not in the specified set will not be considered for execution during rule processing, even if they are triggered. (Such rules eventually will be considered for execution, however, at end-of-transaction rule processing if not sooner.) The **process ruleset** command is useful, for example, when rules are used to maintain integrity constraints [15] or materialized views [16]. In this case, the rules associated with a particular constraint or view are grouped into one set S. Whenever the constraint should be checked or the view refreshed (before the end of a transaction), a **process ruleset** command is issued for set S.

Execution of the **process rule** command invokes rule processing with only the specified rule eligible to be considered and executed. Once again, the behavior of rule processing is identical to end-of-transaction rule processing, except in this case any rules other than the specified rule will not be considered for execution. Note that although only one rule is eligible to be considered and executed, rule processing still may involve several rule executions if the rule triggers itself.

Since **process rules**, **process ruleset**, and **process rule** are executable Starburst commands, these commands may be used in rule actions. Execution of such rule actions results in "nested" invocations of rule processing. This behavior is acceptable and well-defined, and it may be useful in certain scenarios, however we have found that it can be difficult to understand and frequently it results in infinite rule triggering.

4.4 Rule Selection

The **precedes** and **follows** clauses in rules allow them to be ordered in any way, as long as a cycle is not produced. During rule processing, these user-specified priorities influence which rule is selected for consideration when more than one rule is triggered (recall step 1 of the rule processing algorithm in Section 4.2). Since the user-specified ordering on rules may be only a partial ordering (indeed, no ordering is required), it still may be necessary for the system to choose between multiple triggered rules. This selection is performed deterministically by using an algorithm that induces a total ordering on all currently defined rules. The total ordering is consistent with the user-specified partial ordering, and consequently also is consistent with any ordering transitively implied by the user-specified ordering. (That is, if rule R_1 is specified to precede rule R_2, and rule R_2 is specified to precede rule R_3, then R_1 will precede R_3.) As a "tie-breaker", rules that have no user-specified or transitively implied ordering are ordered based on rule creation time (i.e. R_1 is ordered before R_2 if and only if R_1 was created before R_2), unless this ordering is impossible given the user-specified and transitively implied orderings. Details and a formalization of this deterministic rule ordering strategy can be found in [1].

5 Examples

We now provide examples to illustrate the syntax of rule creation and the semantics of rule execution. Our examples are relatively simple and contrived, but they serve to compactly illustrate the salient features of the Starburst rule language syntax and semantics. For more comprehensive examples making up a full rule application, the reader is referred to [15].

We use the following generic employee-department relational database schema:

emp(emp-no, name, salary, dept-no)
dept(dept-no, mgr-no)

Our first example rule, **cascade**, implements a variation on the *cascaded delete* method of enforcing referential integrity constraints. The rule is triggered whenever managers are deleted; its action deletes all employees in departments managed by deleted employees, then deletes the departments themselves. We assume a hierarchical structure of employees and departments, and we assume that employee numbers are not immediately reused—that is, a single transaction will not delete an employee and then insert a new employee with the same employee number.

```
create rule cascade on emp
when deleted
then delete from emp
     where dept-no in
        (select dept-no from dept
         where mgr-no in (select emp-no from deleted));
     delete from dept
     where mgr-no in (select emp-no from deleted)
```

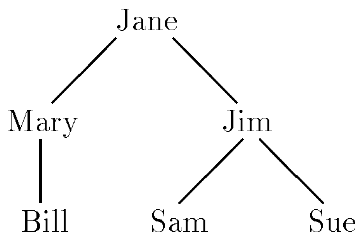
Notice in particular that this rule has no condition (i.e. its condition is always true), it has two actions to be executed in order, and it references transition table **deleted**. As will be shown below, the self-triggering property of this rule under the semantics specified in Section 4 correctly reflects the rule's recursive nature.

Our second example rule, **sal-control**, controls employee salaries: Whenever employees are inserted or salaries are updated, the rule checks the average salary. If the average salary exceeds 50, then the rule deletes all inserted or updated employees whose salary exceeds 80.

```
create rule sal-control on emp
when inserted, updated(salary)
if (select avg(salary) from emp) > 50
then delete from emp
     where emp-no in (select emp-no from inserted
                      union select emp-no from new-updated)
     and salary > 80
precedes cascade
```

Notice in particular that this rule has two triggering operations (either of which will trigger the rule), it has a condition, it references transition tables **inserted** and **new-updated**, and it is specified to have priority over rule **cascade**.

Now consider rule processing when both of these rules are defined. Let the initial state of the database include six employees—*Jane*, *Mary*, *Jim*, *Bill*, *Sam*, and *Sue*—with the following management structure:



Refer to Figure 5. Suppose the initial user transaction τ_1 deletes employee Jane, and the same transaction updates Mary's salary to exceed 80 so that the average salary exceeds 50. Both rules **cascade** and **sal-control** are triggered in state S_1; note that **cascade** is triggered

$$S_0 \xrightarrow[\tau_1]{} S_1 \xrightarrow[\tau_2]{} S_2 \xrightarrow[\tau_3]{} S_3 \xrightarrow[\tau_4]{} S_4 \xrightarrow[\tau_5]{} S_5$$

S_0: {Jane, Mary, Jim, Bill, Sam, Sue}
τ_1: deletes Jane, sets Mary salary > 80, average salary > 50
S_1: {Mary, Jim, Bill, Sam, Sue}
τ_2: **sal-control** deletes Mary
S_2: {Jim, Bill, Sam, Sue}
τ_3: **cascade** deletes Bill and Jim
S_3: {Sam, Sue}
τ_4: **cascade** deletes Sam and Sue
S_4: {}
τ_5: **cascade** deletes nothing
S_5: {}

Figure 1: Transitions for example rules

with respect to set {Jane} of deleted employees. Since rule **sal-control** has priority over rule **cascade**, **sal-control** is chosen for consideration. Its condition is true so it executes its action, deleting employee Mary and creating transition τ_2; **sal-control** is not triggered again. Now, in state S_2, rule **cascade** is triggered by the composite transition since the initial state (transitions τ_1 and τ_2), so its set of deleted employees is {Jane, Mary}. Rule **cascade** executes its actions, deleting all employees and departments whose manager is either Jane or Mary. Employees Bill and Jim are deleted, creating transition τ_3, and rule **cascade** is triggered a second time. Now, in state S_3, the rule considers only the most recent transition (τ_3), so the set of deleted employees is {Bill, Jim}. The rule's actions delete all employees and departments managed by either Bill or Jim—employees Sam and Sue are deleted. Finally, **cascade** executes a third time for transition τ_4 with deleted employees {Sam, Sue}, but no additional employees are deleted.

6 System Architecture

The Starburst rule language as described in Sections 3 and 4 is fully implemented, with all aspects of rule definition and execution integrated into normal database processing. The implementation took about one woman-year to complete; it consists of about 28,000 lines of C and C++ code including comments and blank lines (about 10,000 semicolons). Along with the core capabilities of rule management and rule processing, we also have included considerable infrastructure for program tracing, debugging, and user interaction.

The implementation relies heavily on three extensibility features of the Starburst database system: *attachments*, *table functions*, and *event queues*. We describe these extensibility features here only in enough detail to understand how they are used by the rule system implementation; further details on these and other extensibility features of Starburst can be found in [30].

- The *attachment* feature is designed for extensions that require procedures to be called after each tuple-level database operation on certain tables. An extension creates a new *attachment type* by registering a set of procedures: a procedure to be invoked when an *attachment instance* is created on a table, a procedure to be invoked when an instance is dropped, a procedure to be invoked when an instance is altered, and procedures to be invoked after each tuple-level insert, delete, or update operation on a table with one or more attachment instances. Once an attachment type is established by registering these procedures, instances of that type may be created, dropped, and altered on any table. When an attachment instance is created on a table T, the procedure registered for creation may build an *attachment descriptor*. This data structure is stored by the system and provided to the extension whenever subsequent attachment procedures are invoked for T.

- A *table function* is a virtual table whose contents are generated at run time by a host language procedure, rather than stored in the database. A new table function is created by registering a name along with a procedure for producing the tuples of the table. The procedure may perform any computations as long as it generates tuples of the appropriate schema. Any table listed in the **from** clause of a Starburst **select** operation may be a table function. When a query referencing a table function is processed, the table function's registered procedure is called to produce the contents of the table.

- The *event queue* feature is designed for deferred execution of procedures. Once an event queue has been declared, arbitrary procedures can be placed on the queue at any time, to be executed the next time that queue is invoked. The rule system uses two built-in event queues: one for procedures to be executed during the *prepare-to-commit* phase of each transaction, and one for procedures to be executed in the case of *rollback*.

Figure 2 illustrates the general architecture of the rule system, showing most of the execution modules and data structures, how they fit together, and how they interact with Starburst itself. In the diagram, Starburst, its query processor, and its data repository appear on the left. The ovals in the center column indicate execution modules of the rule system. The rectangles on the right represent memory-resident data structures maintained

Figure 2: Architecture of the Starburst Rule System

by the rule system. An arrow from an execution module to data indicates that the execution module creates the data, while the reverse arrow indicates that the execution module uses the data. A (double-headed) arrow from one execution module to another indicates that the first module calls the second. When these arrows pass through or originate from a star, this indicates that the call is made through an extensibility feature of Starburst. The invocation arrows are labeled by the event causing a call to occur:

(a) Tuple-level insert, delete, or update on a table with one or more rules

(b) Reference to a transition table (transition tables are implemented as table functions)

(c) Evaluation of a rule condition or execution of a rule action

(d) Prepare-to-commit (event queue) or execution of a **process rules**, **process ruleset**, or **process rule** command

(e) Execution of a rule definition command (**create rule**, **alter rule**, **drop rule**, etc.)

The data maintained by the rule system is divided into:

- *Rule Catalog*: The Rule Catalog resides in the database; it stores all information about the currently defined rules and rule sets.

- *Global Rule Information*: For efficiency, some information regarding rules and rule sets also is stored in main memory. This information is shared by all user processes, and includes facts such as each rule's triggering operations, the sets rules belong to, priorities between rules, and whether rules have been deactivated.

- *Transition Log*: This is a highly structured log of those operations occurring within a transaction that are relevant to the currently defined rules. It is stored in main memory and is called a *transition* log since, during rule processing, information about triggering transitions is extracted from the log. The log also is used to produce transition tables. This data structure is *local*, i.e. one Transition Log is maintained for each user process.[5] Further details on the Transition Log are given in Section 7.1.

- *Rule Processing Information*: This also is local to each process. It includes all information pertinent to executing rules within a given transaction, including which rules have been considered and when, and which rules are potentially triggered at a given point in time.

In addition, we have registered an attachment type *Rule* in Starburst. A table has one instance of this attachment type if and only if at least one rule is defined on the table. The attachment descriptor for an instance contains an indicator of what information needs to be written to the Transition Log when operations occur on the table (see Section 7.1 for details).

The execution modules depicted in Figure 2 are:

- *Rule Definition Module*: This component processes all eight rule definition commands described in Section 3. (Here we use "rule definition" generically to mean any command that manipulates rules or rule sets.) The Rule Definition Module is responsible for maintaining the Rule Catalog and updating the Global Rule Information. It also creates, deletes, and modifies rule attachment instances and descriptors as appropriate.

- *Rule Attachment Procedures*: This set of procedures writes to the Transition Log whenever relevant table modifications occur. A rule attachment procedure is called automatically whenever an insert, delete, or update operation occurs on a table with at least one rule.

[5]In Starburst, each user or application corresponds to one process, and each such process is comprised of a sequence of transactions.

- *Transition Table Procedures*: This set of procedures produces transition tables at run time when they are referenced in rule conditions and actions. Transition tables are implemented as table functions, so we have registered procedures for **inserted**, **deleted**, **new-updated**, and **old-updated** with Starburst; these four procedures produce transition tables by extracting appropriate tuples from the Transition Log.

- *Rule Execution Module*: This component is responsible for selecting and executing triggered rules. It is invoked automatically at the commit point of every transaction for which a rule may have been triggered; it also is invoked whenever the query processor encounters a **process rules**, **process ruleset**, or **process rule** command. To determine which rules are triggered, the Transition Log, the Global Rule Information, and the local Rule Processing Information are examined to see which operations have occurred and which rules are triggered by these operations. In the case of **process ruleset** and **process rule** commands, the Rule Execution Module considers only the specified subset of rules. Rule conditions are checked and actions are executed by calling the Starburst query processor. Further details on rule execution are given in Section 7.2.

The rule system also contains several components not illustrated in Figure 2:

- *System Start-Up*: Whenever Starburst is started or restarted, the rule system initializes the Global Rule Information from the Rule Catalog. Rule attachments are initialized automatically by Starburst.

- *Process Start-Up* and *Transaction Clean-Up*: At process start-up, the rule system allocates its local data structures—the Transition Log and the Rule Processing Information. Initially, these structures are empty. They are used during the course of each transaction, then reset after end-of-transaction rule processing.

- *Rollback Handler*: The rule system must correctly handle a partial or complete rollback at any time. The Rule Catalog and attachment information are rolled back automatically by Starburst. However, the rule system must ensure that all memory-resident data structures are modified to undo any changes made during the portion of the transaction being rolled back. This is achieved by having each modification place an appropriate undo operation on the *rollback* event queue.

7 Implementation Features

In the previous section we described the general architecture of the Starburst Rule System; in this section we cover five specific and important features of the implementation in more detail: transition information management, rule execution, concurrency control, authorization, and error handling. Efficient transition information management and rule execution are crucial for system performance, while concurrency control, authorization, and error handling are necessary for full integration with database processing.[6]

7.1 Transition Information

The attachment procedures that write to the Transition Log save information during query processing so that the Rule Execution Module can determine which rules are triggered and so the transition table references in rule conditions and actions can be evaluated. Since the effect of rule action execution also is considered by rules, the Transition Log must be maintained during rule processing as well; this happens automatically since rule actions are executed by the Starburst query processor (recall Figure 2).

The semantics of rule execution dictates that, at any certain time, different rules may need to be considered with respect to different transitions. To do this, we include a (logical) time-stamp with each entry in the Transition Log. We also include with the Rule Processing Information the most recent time at which each rule has been considered; the transition for a given rule is then computed based on entries in the Transition Log occurring after that time.

The triggering operations and transition table references in rules determine which operations and what information must be written to the Transition Log. As an example, suppose a rule R is triggered by **inserted** on a table T, but does not reference the **inserted** transition table. It is necessary to log the times at which insertions occur on T; it also is necessary to log the times at which deletions occur for tuples in T that were previously inserted, since the net effect of an insert followed by a delete is empty. Now suppose R does reference the **inserted** transition table. In this case, the values of the inserted tuples must be logged. In addition, the new values of updated tuples must be logged for those tuples that were previously inserted, since the **inserted** transition table must contain current values for its tuples. Finally, suppose R also is triggered by **updated**, and suppose it references transition table **new-updated** but not **old-updated**. Now, the new values of all updated tuples must be logged; the old values need not be logged since transition table **old-updated** is not referenced. Clearly there are many cases to consider, and we do not enumerate them

[6] Readers satisfied with the implementation overview provided in Section 6 may skip this section without sacrificing the flow of the paper.

here. From the set of rules on each table, the composite set of triggering operations and transition table references is computed. Based on this set, an *information code* is stored in the table's rule attachment descriptor. When attachment procedures are invoked, they use this code to determine what information should be written to the Transition Log. This approach guarantees that all and only the necessary information is saved in the Transition Log.

The data structure we use for the Transition Log is a "double hash table" storing lists of records. Each record represents one tuple-level operation and contains the tuple identifier, operation, time-stamp and, when necessary, new and/or old values for the tuple. Often it is necessary to access all records representing a certain operation on a certain table occurring after a certain time (e.g. all tuples inserted into T since a rule was last considered). For this, a hash is performed on the operation and table to obtain a linked list of the relevant records in descending order of time-stamp. It sometimes is necessary to consider the history of a given tuple to form the net effect of a transition (e.g. to merge updates, or to detect if a deleted tuple was previously inserted). For this, records with the same tuple identifier also are linked in descending order; these lists can be traversed from a given record or can be obtained for a particular tuple by hashing on the tuple identifier. We have developed a number of efficient algorithms for maintaining and traversing the Transition Log structure.

7.2 Rule Execution

The Rule Execution Module is invoked by the query processor whenever a **process rules**, **process ruleset**, or **process rule** command is encountered. The Rule Execution Module also must be invoked at the commit point of every transaction for which rules may have been triggered. For end-of-transaction rule processing, the first time a rule attachment procedure is called during a transaction—indicating that a relevant operation has occurred—the attachment procedure places the Rule Execution Module on the *prepare-to-commit* event queue. Then, when the transaction is ready to commit, rule execution is invoked automatically.[7] An important advantage of this approach (over the straightforward approach of invoking the Rule Execution Module at the end of every transaction) is that no overhead is incurred by transactions for which no rules are triggered.

During rule processing, we maintain a data structure called *Potential-Rules* as part of the local Rule Processing Information; this data structure contains references to those rules potentially triggered at each point in time. The rules in this structure are only "potentially"

[7]If other procedures placed on the *prepare-to-commit* event queue may modify the database, then it is important for the Rule Execution Module to be invoked after such procedures during queue processing. Currently in Starburst no other prepare-to-commit procedures modify data, so the execution order of the Rule Execution Module relative to other queued procedures is unimportant.

triggered because they are a conservative estimate—every triggered rule is in the set, but there may be rules in the set that actually are not triggered: At the end of each transition, all rules triggered by operations that occurred during the transition are added to Potential-Rules without considering the net effect of the transition. Hence, for example, if tuples were inserted into table T during the transition, then all rules triggered by **inserted** on T are added to Potential-Rules, regardless of whether the inserted tuples subsequently were deleted.

In practice, it is rare for operations in a transition to be "undone" in the net effect, so Potential-Rules usually is not overly conservative. However, before processing a rule from Potential-Rules, the net effect must be computed to verify that the rule is indeed triggered. Note that by maintaining the potentially triggered rules, rather than the actually triggered rules, we compute the net effect for only one rule in each "cycle" of rule execution, rather than for all triggered rules.

When a rule is fetched from Potential-Rules for consideration, it must be chosen such that no other rule with higher priority also may be triggered. This is achieved by maintaining Potential-Rules as a sort structure based on the total ordering of rules described in Section 4.4.

7.3 Concurrency Control

Since Starburst is a multi-user database system, we must ensure that all aspects of the rule system behave correctly in the presence of concurrently executing transactions.[8] For most transactions, including those with triggered rules, concurrency control is handled automatically by the database system since rule conditions and actions are executed through the Starburst query processor. However, since rules themselves may be manipulated on-line, the rule system must enforce concurrency control for transactions that perform rule definition (i.e. transactions that create, delete, or modify rules or rule sets).

As examples of consistency issues involving rule definition, consider the following scenarios:

- Suppose a transaction X modifies a table T while a concurrent transaction deactivates rule R on T. Should R be triggered by X?
- Suppose a transaction X triggers rules R_1 and R_2 while a concurrent transaction alters the relative priority of R_1 and R_2. Which ordering should be used by X?

[8]Note, however, that Starburst is not a distributed database system, so issues of distributed access to shared data and main memory structures are not relevant.

- Suppose a transaction X executes "**process ruleset** S" while a concurrent transaction adds rule R to set S. Should R be triggered by X?

We address these issues in the Starburst Rule System by ensuring that transactions are serializable not only with respect to data but also with respect to rules (including rule triggering and rule sets). Furthermore, we ensure that the equivalent serial transaction schedule with respect to rules is the same as the equivalent serial schedule with respect to data.

Let X_1 and X_2 be transactions such that X_1 precedes X_2 in the serial schedule induced by Starburst's concurrency control mechanism for data. Serializability of X_1 and X_2 with respect to rules is guaranteed by enforcing the following three consistency requirements:

(1) *Triggering consistency*: If X_1 performs rule definition on a table T (i.e. X_1 in some way modifies rules pertaining to T), and X_2 modifies data in T, then X_2's rule processing sees the effect of X_1's rule definition. If X_2 performs rule definition on a table modified by X_1, then X_1's rule processing does not see the effect of X_2's rule definition.

(2) *Rule set consistency*: If X_1 modifies a rule set S and X_2 includes "**process ruleset** S", then X_2's rule processing sees the effect of X_1's rule set modification. If X_2 modifies S and X_1 includes "**process ruleset** S", then X_1's rule processing does not see the effect of X_2's rule set modification.

(3) *Update consistency*: If X_1 and X_2 both modify the same rule or rule set, then X_2 sees the effect of X_1's modification and X_1 does not see the effect of X_2's modification.

In addition, the Starburst Rule System ensures consistency within a transaction by enforcing the following two requirements:

(4) *Intra-transaction triggering consistency*: If a transaction X modifies a table T then X cannot subsequently perform rule definition on T.

(5) *Intra-transaction rule set consistency*: If a transaction X executes a "**process ruleset** S" operation then X cannot subsequently modify rule set S.

Lastly, the Starburst Rule System ensures consistency of rule ordering:

(6) *Ordering consistency*: If X is a transaction that triggers rules R_1 and R_2, then the ordering between R_1 and R_2 does not change during X from the first time this ordering is used in rule selection.

All six consistency requirements are ensured by protocols that check and/or set locks on data, rules, or rule sets. In Starburst, locks are acquired throughout a transaction as needed and are held until the transaction commits or rolls back. Hence, the equivalent serial schedule of transactions with respect to data is based on commit time.

We enforce consistency requirements (1) and (4) as follows. When a transaction X executes a rule definition command on table T, X first checks to see if it has modified T (by checking if it holds any exclusive locks on data in T). If so, then the rule definition command is rejected. Otherwise, X obtains a table-level shared lock on T. This forces X to wait until all transactions currently modifying T have committed, and it disallows future modifications to T by other transactions until X commits.

Consistency requirement (2) is enforced by locking rule sets. Before modifying (creating, altering, or dropping) rule set S, a transaction must obtain an exclusive lock on S. Before processing rule set S, a transaction must obtain a shared lock on S. To enforce consistency requirement (5), shared rule set locks cannot be upgraded to exclusive rule set locks.

Consistency requirement (6) is enforced by locking rules. When a rule R is added to data structure *Potential-Rules* (recall Section 7.2), a shared lock is obtained on R. When a rule definition command that affects rule ordering is executed (**create rule**, **alter rule**, or **drop rule**), an exclusive lock is obtained on every rule whose ordering relative to other rules is affected by the command. Note that even the ordering between unchanged rules may be reversed, since transitive relationships may be introduced or dropped. To prevent ordering relationships from changing within a transaction, shared rule locks cannot be upgraded to exclusive rule locks.

Consistency requirement (3) is enforced automatically since rule and rule set modifications are reflected in the Rule Catalog, and the Rule Catalog is subject to Starburst's concurrency control mechanisms for data.

Further details of these locking protocols and proofs of their correctness appear in [21].

7.4 Authorization

In the authorization component of the Starburst Rule System we address a number of distinct issues, including authorization to create rules on a given table, authorization to create rules with given conditions and actions, authorization to alter or drop given rules, authorization for rule sets, and authorization at rule execution time. In Starburst, lattices of *privilege types* can be defined for arbitrary database objects, with higher types subsuming the privileges of lower types. For example, for database tables the highest privilege is *control*; below this are privileges *write*, *alter*, and *attach*; below *write* are privileges *update*, *delete*, and *insert*; below *update* and *delete* is privilege *read*. When a table is created, its creator automatically obtains *control* privilege on the table, which includes the ability to grant and revoke privileges on it.

For rules we have defined a simple linear lattice of privilege types: the highest privilege is *control*, below this is *alter*, and privilege *deactivate/activate* is lowest. As with tables, a rule's creator automatically obtains *control* privilege on the rule and may grant and revoke privileges on it. To create a rule R on table T, R's creator must have both *attach* and *read* privileges on T.[9] During rule creation, R's condition and actions are checked using the creator's privileges. If the condition or actions contain commands the creator is not authorized to execute, then the **create rule** command is rejected. To drop a rule R on table T, we require either *control* privilege on T or *attach* privilege on T with *control* privilege on R. To alter a rule, privilege *alter* is required; to deactivate or activate a rule, privilege *deactivate/activate* is required. During rule processing, each rule's condition and actions are executed using the privileges of the rule's creator (not the privileges of the transaction triggering the rule).

We have defined two privilege types for rule sets, *control* and *alter*, with *control* subsuming *alter*. A rule set's creator obtains *control* privilege on the rule set and may grant and revoke privileges on it. Privilege *control* is needed to drop a rule set; privilege *alter* is needed to add or delete rules from a rule set. No privileges on rules are needed to add or delete them from rule sets, and no privileges are needed to execute **process rules**, **process ruleset**, or **process rule** statements.

The Starburst Rule System currently does not enforce any authorization requirements when users examine the rules or rule sets in the system—all rules may be queried and inspected by any Starburst user. It is clear, however, that authorization requirements for reading rules should ultimately be included in any complete active database system.

7.5 Error Handling

If an error occurs during the execution of a Starburst rule definition command (due to, e.g., the creation of cyclic priorities, the inclusion of an action the creator is not authorized to execute, or a syntactic flaw), then the rule definition command is rejected. During rule processing, two types of errors can occur: an error may be generated during the evaluation of a rule's condition or execution of a rule's action, or rules may trigger each other or themselves indefinitely. In the first case, if an error is generated by the query processor when it executes a rule condition or action, then the rule system terminates rule processing and aborts the current transaction. For the second case, the rule system includes a "timeout" mechanism: Once more than some number n of triggered rules have been considered, rule processing terminates and the transaction is aborted; limit n is established by a system administrator.

[9] In general, *attach* privilege on a table indicates that the user is permitted to alter the performance of that table. We require *read* privilege on table T since rule R can implicitly read the contents of T through transition tables without accessing T directly.

8 Conclusions and Retrospective

The Starburst Rule System is a fully implemented extension to the Starburst prototype relational database system at the IBM Almaden Research Center. We have designed a rule language that is flexible and general, with a well-defined semantics based on arbitrary database state transitions. In addition to the usual commands for manipulating rules, our language includes a basic rule set facility for application structuring, and it includes commands for processing rules within transactions in addition to the automatic rule processing that occurs at the end of each transaction. Rule processing in Starburst is completely integrated with database query and transaction processing, including concurrency control, authorization, rollback recovery, and error handling.

We have learned a number of interesting lessons from our careful development of the Starburst rule language, from its thorough implementation, and from our experiments with the running system on a variety of rule applications. With respect to our design of the Starburst rule language, we make the following observations:

- Basing the semantics on arbitrary transitions offers considerable flexibility, and it generally provides a clean execution behavior. Although users may feel initially that they better understand tuple- or statement-level rule triggering, there can be surprising anomalies in such behavior that do not arise with the Starburst semantics. On the other hand, for very simple rule processing tasks, tuple-level or statement-level rule processing usually does behave as the user expects, and it can be both more natural and more efficient than the Starburst approach.[10] Note also that Starburst's transition-oriented semantics prohibits a natural **before** option for rule triggering [34]. However, again, specifying **before** may result in surprising rule interactions, where such behavior is avoided with Starburst's rule semantics.

- Rule processing based on an iterative loop, as in Starburst, is intuitive, it seems to be sufficient for most applications, and it is relatively easy to implement. Hence, we believe that the more complex recursive rule processing algorithms used in, e.g., POSTGRES [42] or HiPAC [20], probably are not worthwhile.

- Complex *conflict resolution* policies, such as those used in OPS5 [9] and Ariel [31], do not seem appropriate for most active rule applications. Simple relative priorities appear to be sufficient, and they can be implemented easily and efficiently.

- A significant drawback in the Starburst rule language, as opposed to a number of other active rule languages, is the lack of a language facility for "passing data" from a rule's

[10] Consider, e.g., a rule that performs a simple modification to each inserted tuple and doesn't trigger any other rules.

condition to its action. Note that the data associated with triggering operations is available implicitly through transition tables. However, the data satisfying a rule's condition is not directly available in the rule's action. In practice, users often write Starburst rules that explicitly repeat the condition as a subquery in the action, or that omit the condition altogether and place it in the action. A language feature for referencing, in the action, the data satisfying the condition (as suggested in [15]) would have been very useful.

- A convenient extension to the rule language would have been to allow rules that are triggered by operations on multiple tables. In fact, this feature has no effect on the semantics of the rule language [49], but was omitted due to the additional implementation effort. Another useful extension would have been to allow rule actions that invoke arbitrary host language procedures. Currently, this behavior can be simulated through Starburst's foreign function feature in SQL [30], but host language procedures cannot be called directly from rules.

- Rules in Starburst cannot be triggered by **select** operations. Although the reason for this is partly implementation-dependent (the *attachment* extensibility feature is not available for **select** operations), there are a number of semantic issues that would also need to be addressed to add **select** as a triggering operation, such as whether rules are triggered by nested **select** expressions.

With respect to our implementation of the Starburst Rule System, we make the following observations:

- The extensibility features of Starburst offered a dramatic "head start" in implementing the rule system. All three extensibility features that we used—attachments, event queues, and table functions—were used heavily. Significant additional coding would have been required had these features not been available.

- A number of main-memory data structures are maintained by the rule system (recall Section 6). Because much of the work associated with rule processing involves manipulating these structures, rule processing itself is very fast. However, each structure needed recovery procedures coded for each of its operations (in case of a complete or partial rollback), and certain important aspects of rule processing—such as the number of rules, or the number of tuple-level operations relevant to rules within a given transaction—are limited by the fact that these structures reside in memory. The system would have been easier to implement and it would be more scalable if these main-memory structures were implemented as persistent, recoverable database objects.

Unfortunately, one of the few things Starburst did not offer was a flexible facility for such objects with the performance we desired for rule operations.

- Integrating rule processing directly into the database system, as opposed to a loosely coupled approach, offers important advantages for both performance and functionality. With a loosely coupled approach it would have been impossible to fully address issues such as concurrency control, authorization, and recovery. In addition, significant overhead would have been incurred by the need to intercept user commands and/or database results at the client level rather than within the database system. Although it may be unappealing (and, sometimes, impossible) to modify or extend the core code of a database system, this appears to be a necessity if one wishes to build a fully integrated active rule system with acceptable performance.

- Initial performance measurements have revealed that the vast majority of time spent in rule processing is in fetching, compiling, and executing rule conditions and actions. In Starburst we were unable to store precompiled queries, so rule conditions and actions needed to be compiled each time they were executed. Once conditions and actions are stored in compiled form, the main cost of rule processing will be in condition evaluation and action execution (rather than other aspects pf rule processing, such as finding triggered rules, selecting the highest priority rule, etc.). This lead us to believe that performance improvements will be made not by streamlining rule management or the rule processing algorithm itself, but rather by finding ways to optimize condition evaluation and action execution.

9 Applications and Future Work

The Starburst Rule System has been used as a platform for developing a number of applications and for investigating various issues in active database systems. We have used Starburst rules for enforcing integrity constraints [15], for maintaining materialized views [16], and for implementing deductive databases [19], as well as for several other (more ad-hoc) applications. We have studied how the Starburst Rule System can be supported in a tightly-coupled distributed database environment with full distribution transparency [17]; we also have studied how the Starburst Rule System can be used to manage semantic heterogeneity across loosely-coupled databases [18]. Because predicting and understanding the behavior of active database rules is an important facet of application development, we have developed methods for statically analyzing sets of Starburst rules; these analysis methods determine (conservatively) whether a set of rules is guaranteed to terminate, and whether the rules are guaranteed to produce a unique final state [2]. Other researchers have used the

Starburst Rule System as a basis for studying and implementing secure active databases [40], dynamic integrity constraints [28,43], and automatically-generated compensating actions for static constraints [13].

Although we do consider the Starburst Rule System to be complete at this time, there are several directions in which it may be exercised, improved, and extended:

- Currently we have obtained only initial cursory performance results. We would like to elaborate these results; this requires developing a mechanism for accurate measurements and deriving a sufficient suite of test applications.

- As explained in Section 6, a rule's condition is evaluated by executing a query over the database. We do incorporate one important optimization, namely that a rule condition is understood to be true as soon as the first tuple in the query is found. However, we do not support incremental condition monitoring methods such as those used in Ariel [44] and in OPS5 [9,36]. We have explored incremental condition evaluation in the context of Starburst [5], and we plan to explore other run-time optimization methods as well. We are interested also in compile-time optimization methods, such as static combination of multiple rules that have related conditions and/or actions.

- Statement-level rule processing can be achieved in the Starburst Rule System by issuing a "**process rules**" command after each statement; it would be useful to provide a more convenient mechanism for this. For example, we could predefine a system rule set called *Statement*. Users would then add rules to this set, and the system would automatically execute "**process ruleset** *Statement*" after each statement. A similar mechanism could be provided for tuple-level rule processing.

- Currently, the Starburst Rule System includes only basic facilities for rule tracing and for interaction between rule processing and application programs. The areas of debugging and application interfaces offer considerable opportunities for useful extensions.

In addition to these Starburst-specific areas of future work, we hope and expect that the Starburst Rule System will continue to be used as a basis for further research in active database systems.

Acknowledgments

Sincere thanks go to Stefano Ceri, Bobbie Cochrane, Shel Finkelstein, and Bruce Lindsay, all of whom made important contributions to one aspect or another of the Starburst Rule System.

References

[1] 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.

[2] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[3] A.M. Alasqur, S.Y. Su, and H. Lam. A rule based language for deductive OODBS. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[4] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[5] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. Technical Report Stan-CS-93-1495, Computer Science Department, Stanford University, November 1993.

[6] C. Beeri and T. Milo. A model for active object oriented database. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 337–349, Barcelona, Spain, September 1991.

[7] M. Berndtsson and B. Lings. On developing reactive object-oriented databases. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):31–34, December 1992.

[8] D.A. Brant and D.P. Miranker. Index support for rule activation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 42–48, Washington, D.C., May 1993.

[9] 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.

[10] A.P. Buchmann, H. Branding, T. Kudrass, and J. Zimmerman. REACH: a REal-time, ACtive, and Heterogeneous mediator system. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):44–47, December 1992.

[11] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–236, Atlantic City, New Jersey, May 1990.

[12] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in Chimera. In *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, CA, 1994. (To appear).

[13] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. To appear in *ACM Transactions on Database Systems*, 1994.

[14] S. Ceri and R. Manthey. Consolidated specification of Chimera, the conceptual interface of Idea. Technical Report IDEA.DD.2P.004, Politecnico di Milano, Milan, Italy, June 1993.

[15] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[16] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[17] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 339–351, Vancouver, British Columbia, August 1992.

[18] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, August 1993.

[19] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. To appear in *Information Systems*, 1994.

[20] S. Chakravarthy, B. Blaustein, A.P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachusetts, July 1989.

[21] R.J. Cochrane. *Issues in Integrating Active Rules into Database Systems.* PhD thesis, University of Maryland, College Park, January 1992.

[22] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: A production language for relational databases. In L. Kerschberg, editor, *Expert Database Systems—Proceedings from the Second International Conference*, pages 333–351. Benjamin/Cummings, Redwood City, California, 1989.

[23] H.M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, and S. Da Silva. Incremental database rule processing in PARADISER. *Journal of Intelligent Information Systems*, 1992.

[24] O. Diaz, N. Paton, and P. Gray. Rule management in object-oriented databases: A uniform approach. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 317–326, Barcelona, Spain, September 1991.

[25] S. Dietrich, S.D. Urban, J.V. Harrison, and A.P. Karamdice. A DOOD RANCH at ASU: Integrating active, deductive, and object-oriented databases. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):40–43, December 1992.

[26] S. Gatziu, A. Geppert, and K.R. Dittrich. Integrating active concepts into an object-oriented database system. In *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, August 1991.

[27] 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.

[28] M. Gertz and U.W. Lipeck. Deriving integrity maintaining triggers from transition graphs. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 22–29, Vienna, Austria, April 1993.

[29] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[30] 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.

[31] E.N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, San Diego, California, June 1992.

[32] ISO-ANSI working draft: Database language SQL3 (X3H2/94/080 and SOU/003), 1994.

[33] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. Supporting semantic rules by a generalized event/trigger mechanism. In *Advances in Database Technology—EDBT '88, Lecture Notes in Computer Science 303*, pages 76–91. Springer-Verlag, Berlin, March 1988.

[34] J. Melton and A.R. Simon. *Understanding the New SQL: a Complete Guide*. Morgan Kaufmann, San Mateo, California, 1993.

[35] T. Risch and M. Sköld. Active rules based on object-oriented queries. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):27–30, December 1992.

[36] 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.

[37] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 469–478, Barcelona, Spain, September 1991.

[38] T. Sellis, C.-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 404–412, Chicago, Illinois, June 1988.

[39] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational DBMS. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 315–326, Vancouver, British Columbia, August 1992.

[40] K. Smith and M. Winslett. Multilevel secure rules: Integrating the multilevel and active data models. Technical Report UIUCDCS-R-92-1732, University of Illinois, Urbana-Champaign, March 1992.

[41] M. Stonebraker, E.N. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[42] 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.

[43] D. Toman. Implementing temporal integrity constraints using an active DBMS. In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)*, pages 87–95, Houston, Texas, February 1994.

[44] Y.-W. Wang and E.N. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proceedings of the Eighth International Conference on Data Engineering*, Phoenix, Arizona, February 1992.

[45] J. Widom. A denotational semantics for the Starburst production rule language. *SIGMOD Record*, 21(3):4–9, September 1992.

[46] J. Widom. The Starburst Rule System: Language design, implementation, and applications. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):15–18, December 1992.

[47] J. Widom. Starburst Rule System user's guide. Internal Technical Report, IBM Almaden Research Center, San Jose, California, July 1992.

[48] 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.

[49] 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.