Proc. of 1991 VLDB Conference, pages 275–285

Implementing Set-Oriented Production Rules as an Extension to Starburst

Jennifer Widom
Bruce G. Lindsay

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
widom@ibm.com, bruce@ibm.com

Roberta Jo Cochrane*

Department of Computer Science
University of Maryland
College Park, MD 20742
bobbie@cs.umd.edu

Abstract. This paper describes the implementation of a set-oriented database production rule language proposed in earlier papers. Our implementation uses the extensibility features of the Starburst database system, and rule execution is fully integrated into database query and transaction processing.

1 Introduction

In database systems, a *production rules* facility allows definition of database operations that are executed automatically whenever certain events occur or conditions are met. Production rules in database systems can be used for enforcing integrity constraints, maintaining derived data, and building efficient knowledge-base and expert systems. In [WF89,WF90] we propose a syntax for specifying production rules in relational database systems and a semantics for rule execution. In keeping with the set-oriented approach of relational data manipulation languages, our production rules are set-oriented—they are triggered by sets of changes to the database and may perform sets of changes. The condition and action parts of a production rule may refer to the current state of the database as well as to the sets of changes that triggered the rule. Our rule language has now been implemented at the IBM Almaden Research Center as an extension to *Starburst.* Starburst is a prototype relational database system with a focus on extensibility [H^+90]. This paper serves the dual role of describing our implementation of the rule system and illustrating how the extensibility features of Starburst facilitated its rapid development.

Several other research efforts also consider production rules in database systems. Descriptions of a variety of rule languages and prototype systems appear in [Coh89, DE89, dMS88, Esw76, Han89, MD89, SLR88, SJGP90], among others. The most significant difference between our rule system and others is our focus on set-orientation, whereby rules can be triggered by and process arbitrary sets of changes to the database. We believe this approach permits a flexible framework and adapts well to the database setting. However, it requires a quite different implementation strategy than more traditional (instance-oriented) database rule systems, as described in this paper. In addition, our rule system fully integrates rule definition and execution with database processing, including features such as concurrency control and rollback; these features are omitted in many other database rule systems. Further discussion of the rule language and comparison with related work can be found in [WF89,WF90].

Section 2 reviews the syntax and semantics of the rule language. Section 3 briefly describes Starburst, highlighting the extensibility features used by the rule system. Section 4 gives the overall design of the rule system and contrasts our approach with other systems; details of particular components are presented in the subsequent sections. Section 5 describes how rules are created, dropped, and altered (hereafter referred to as *rule definition*), and how they are stored in the database. Section 6 explains how *transition information* for rules is accumulated at run-time—this information is used to determine which rules are triggered and to evaluate references to the database changes that triggered them. Rule execution itself is described in Section 7. Section 8 explains how we handle rule ordering, including the rule priority feature in our language. Section 9 discusses concurrency control issues, in particular our approach to enforcing serializability of transactions that may include rule definition. How the rule system reacts to complete and partial rollback is described in Section 10. Section 11 covers authorization issues in both rule definition and execution. Finally, in Section 12 we discuss the current status of the rule system and propose a number of enhancements and extensions to it.

2 Rule Language

We provide a brief overview of our set-oriented relational database rule language. It is based on SQL, since an extended version of SQL is the query language used in Starburst [H^+90]. The production rules facility is fully integrated into the database system. That is, all the usual database tasks are performed; in addition, a set of rules may be defined. A rule's *transition predicate* controls triggering; when triggered, a rule's *condition* is checked before it may execute its *action*, which is a sequence of database operations. Rules are not activated until the commit point of each transaction, at which time all triggered rules are considered.[1] The database oper-

*Work performed while this author was visiting IBM Almaden

[1] We will soon extend the system with a flexible mechanism for additional rule triggering points; see Section 12.

ations executed as part of a rule's action may trigger additional rules. Once there are no triggered rules left to consider, the transaction is committed. Details are given below.

To facilitate a quick prototype implementation, the rule language supported by our system differs slightly from that proposed in [WF90]. Some changes are purely syntactic, others are restrictions that we intend to remove, and in one case we have added expressive power to the language. To alleviate any confusion in readers familiar with [WF89,WF90], all changes are documented in footnotes.

Our production rules are based on the notion of *transitions*. A transition is a database state change resulting from execution of a sequence of database operations. We consider the *net effect* of transitions, meaning that: (1) if a tuple is updated several times, we consider only the composite update; (2) if a tuple is updated then deleted, we consider only the deletion; (3) if a tuple is inserted then updated, we consider this as inserting the updated tuple; (4) if a tuple is inserted then deleted, it is not considered at all. A formal presentation of transitions and their net effects appears in [WF89].

The syntax for creating production rules is:[2]

> **create rule** *rule-name* **on** *table-name*
> **when** *transition-predicate* ,
> [**if** *condition* ,]
> **then** *action-list* ,
> [**precedes** *rule-list* ,]
> [**follows** *rule-list*] ;

The transition predicate specifies the rule's *triggering operations*; it is a nonempty subset of

> (**inserted, deleted, updated** [(*column-list*)])

where *column-list* is a list of columns in *table-name*. A rule is triggered by a given transition if at least one of the specified operations occurred on *table-name* in the net effect of the transition. In the case of **updated**, one of the columns in *column-list* must be updated; if no columns are specified, the rule is triggered by updates to any column. Once a rule is triggered, its condition is checked. A rule condition is an arbitrary SQL predicate over the database. If the condition evaluates to true, then the rule's list of actions is executed.[3] (The condition may be omitted, in which case it is always true.) Rule actions are arbitrary Starburst database operations, including **select**, **insert**, **delete**, and **update** expressions, as well as data definition commands and **rollback** requests.[4] The optional **precedes** and **follows** clauses list existing rules and are used to specify rule priorities. If a rule R_1 includes a rule R_2 in its **precedes** list, then if R_1 and R_2 are both triggered, R_1 will be considered first; conversely for **follows**.[5]

The condition and action parts of a rule may refer to the current state of the database through SQL operations. In addition, these components may refer to *transition tables*—logical tables reflecting the changes that have occurred during a rule's triggering transition. There are four transition tables: **inserted**, **deleted**, **new_updated**, and **old_updated**. A rule may refer to any transition table corresponding to one of its triggering operations. Consider a rule R on a table T. At the end of a transition triggering rule R:

- **inserted** refers to those tuples of table T in the current state that were inserted by the transition.
- **deleted** refers to those tuples of table T in the pre-transition state that were deleted by the transition.
- **new_updated** refers to those tuples of table T in the current state for which at least one of the triggering columns was updated.[6]
- **old_updated** refers to those tuples of table T in the pre-transition state for which at least one of the triggering columns was updated.

Transition tables may be referenced in the **from** clauses of **select** operations using Starburst's *table expression* syntax (see Section 3.2) as shown in this example:

> **select** ... **from** ... **v as (inserted())** ... **where** ...

In the **select** and **where** clauses, references to table variable **v** indicate transition table **inserted**.

Rules are dropped by issuing the command:

> **drop rule** *rule-name* **on** *table-name* ;

Existing rules may be altered; we permit changes to all aspects of a rule except its name, table, and triggering operations. Rules are altered using the command:

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

where each specified clause replaces the existing attribute for that rule. (Actually, for **precedes** and **follows** we allow rule names to be added and removed from the existing lists.) Finally, for convenience, rules may be temporarily *deactivated*. Deactivated rules remain in the

[2]Note minor changes from the syntax in [WF90]. The syntax used here allows us to take advantage of general commands provided by Starburst (see Section 5). Also note that rules are triggered by operations on only one table, while in [WF90] they could be triggered by operations on any number of tables. We intend to permit multiple tables in the future.

[3]For simplicity, in our current implementation rule conditions actually are SQL **select** expressions: if the **select** expression is nonempty then the condition is true. The two forms are interchangeable, and we intend to convert to predicates in the near future.

[4]In [WF90] we proposed only data modification and **rollback**. The more general actions were readily implemented in Starburst and mesh with our intention to eventually permit rule actions to be arbitrary procedure calls.

[5]In [WF90] rule priorities are specified in separate commands; the expressive power is equivalent.

[6]This form combines the two forms given in [WF90].

system but cannot be executed. The syntax for deactivation and reactivation is:

alter rule *rule-name* **on** *table-name* **deactivate ;**
alter rule *rule-name* **on** *table-name* **activate ;**

We now describe the semantics of rule execution. First, a user or application executes a transaction—a sequence of SQL operations; rules are processed at the commit point of the transaction. The state change resulting from this initial transaction creates the first relevant transition, and some set of rules are triggered by this transition. One rule is chosen from this set for *consideration.* The rule is chosen such that no other triggered rule should precede it according to the rules' **precedes** and **follows** clauses. The chosen rule's condition is checked; if the condition is false then another triggered rule is chosen for consideration. Otherwise, the rule's list of actions is executed. Let R be the first rule whose actions are executed and assume for the moment that it does not specify **rollback**. At this point, one rule has been executed, although several rules may have been considered. All rules *not* previously considered are now triggered only if their transition predicate holds with respect to the composite transition created by the initial transaction and subsequent execution of R's action. That is, these rules see R's action as if it were executed as part of the user-generated transaction. Rules already considered (including R) have already "processed" the initial transaction. Thus, these rules are triggered again if their transition predicate holds with respect to the transition created by R's action.

Consider now an arbitrary point in rule processing, where zero or more triggered rules have been considered, and those whose conditions were true have been executed. A given rule is triggered at this point if its transition predicate holds with respect to the (composite) transition since the point at which it was most recently considered.[7] If a rule has not yet been considered, then it is considered with respect to the transition since the start of the transaction. If a **rollback** action is encountered during rule execution, the system rolls back to the start of the transaction (including undoing all effects of previously executed rule actions) and rule processing terminates. Otherwise, rule processing terminates when the set of triggered rules is empty or when no triggered rule has a true condition; the entire transaction is then committed.

2.1 Examples

We illustrate our rule language with three simple examples; for numerous additional examples see [CW90, CW91, WF89, WF90, Wid91]. The first rule controls salaries in a database of employees:

[7]In [WF90] we specified that a rule is considered with respect to the (composite) transition since the point at which it was most recently *executed.* Although either semantics (or both) could have been implemented, our current choice seems intuitive and useful.

```
create rule sal_control on emp
when (inserted, updated(salary)),
if 'exists (select * from i as (inserted())
          where i.salary > 100) or
    exists (select * from nu as (new_updated())
          where nu.salary > 100)',
then ('update emp
       set salary = 50
       where emp.id in
        (select v.id from v as (inserted()))',
      'update emp
       set salary = .9 * salary
       where salary > 100');
```

This rule is triggered whenever employees are inserted or salaries are updated. The condition holds if any inserted or updated employee has a salary greater than 100. If true, the action sets the salaries of all inserted employees to 50 and reduces each existing employee's salary by 10% if it is greater than 100. Notice that this rule triggers itself until all salaries are reduced to less than or equal to 100.

Suppose that in the extreme case, when an inserted or updated salary exceeds 150, the entire transaction should be rolled back. This is implemented by the following rule:

```
create rule sal_extreme on emp
when (inserted, updated(salary)),
if 'exists (select * from i as (inserted())
          where i.salary > 150) or
    exists (select * from nu as (new_updated())
          where nu.salary > 150)',
then 'rollback',
precedes sal_control;
```

Since both rules will be triggered at the same time, the **precedes** clause in this rule specifies that it is considered first, so that salaries greater than 150 are not simply reduced by rule `sal_control`.

As a final example, we show a rule that implements the cascaded delete method of enforcing referential integrity. Whenever employees are deleted, the rule deletes all employees managed by the deleted employees:

```
create rule del-cascade on emp
when deleted,
then 'delete from emp
      where emp.mgr-id in
       (select v.id from v as (deleted()))',
precedes sal_control,
follows sal_extreme;
```

This rule has no condition, so its action is executed whenever it is triggered. The rule triggers itself, with termination occurring when no employees satisfy the predicate in the action, i.e., no deletions occur. For efficiency, this rule is specified to follow `sal_extreme` but precede `sal_control`.

3 Starburst

Starburst is a prototype relational database system at the IBM Almaden Research Center. One of the goals of Starburst is to build an extensible system—a system that can support non-traditional applications and can serve as a testbed for innovations and improvements in database technology. For a detailed description of Starburst, its extensibility architecture, and some of its current extensions, see [H+90]. Our production rules facility is a substantial extension that takes advantage of several features included in Starburst for extensibility. We briefly introduce those features.

3.1 Attachments

The *attachment* mechanism in Starburst permits extensions to the system that require procedures to be called after each tuple-level database operation. (Our description of attachments here is somewhat simplified since the rule system does not use all attachment features.) A new *attachment type* is created by registering a set of procedures: a procedure to be invoked when an *attachment instance* is created on a given 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.[8] Once an attachment type is established by registering these procedures, instances of that type are created, dropped, and altered using general data definition commands provided by Starburst. When an attachment instance is created, the procedure registered for creation may build an *attachment descriptor*. This is an arbitrary data structure stored by the system in the database and provided to the extension (for examination or modification) whenever subsequent attachment procedures are invoked. It is interesting to note that the Starburst attachment mechanism already provides the internal power of a tuple-oriented database rule system.

3.2 Table Functions

Table functions extend the flexibility and expressive power of the Starburst query language (an enhanced version of SQL). A table function is created by registering a function name, parameter specifications, a table schema, and a procedure for producing the tuples of the table function. Any table listed in the **from** clause of a Starburst query can instead be a table function; the syntax is "**from ... v as** (*fn-name*($p_1 \ldots p_n$)) **...**". Appearances of table variable **v** elsewhere in the query are references to table function *fn-name*. The table produced by *fn-name* at run-time has the schema that was registered for the table function. To produce the table, parameters $p_1 \ldots p_n$ are passed to the table function's procedure. The procedure may perform any computations as long as it generates a set of tuples with the specified schema.

3.3 Event Queues

The *event queue* mechanism in Starburst is used for deferred execution of procedures. Once an event queue is declared, arbitrary parameterized procedures can be placed on the queue to be executed when the queue is invoked. Currently there are 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 upon actual *commit*. These queues also are invoked if a transaction rolls back; their procedures are passed a flag indicating that a rollback is occurring. Starburst permits *partial rollback*, whereby the process is rolled back to a user-specified save point within the current transaction. During partial rollback, all procedures placed on event queues during the portion of the transaction being rolled back are removed from the queue and executed with the "rollback" flag. Procedures may be placed on event queues with parameters that will then be available when the procedure is executed. Event queue procedures are executed in reverse order of arrival (i.e., the queue behaves as a stack).

4 Rule System Design

In this section we describe the overall structure of the rule system. Details of particular components are given in the remainder of the paper. Figure 1 illustrates most of the rule system's execution modules and data structures, showing how they fit together and how they interact with Starburst. 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 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; the arrows are labeled by the event causing a call to occur. When these arrows pass through or originate from a star, this indicates that the call is made through an extensibility feature of Starburst.

The data maintained by the rule system can be divided into:

- *Rule Catalog*: This resides in the database and stores the set of currently defined rules.
- *Global Rule Information*: For efficiency, some information regarding the set of rules also is stored in main memory. This information is shared by all user processes. (We assume that the number of rules does not exceed the capacity of (virtual) memory. An argument for this assumption appears in [HCKW90].)
- *Transition Log*: This is a highly structured log of those operations occurring within a transaction that are relevant to currently defined rules. It is stored in main memory and is called a *transition* log since, during rule processing, information about the net effect of 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.

[8] A table may have many instances of a given attachment type, and instances of a given type may be created on any table. Each instance, however, is associated with exactly one table.

Figure 1: *Overall Structure of the Rule System*

- *Rule Processing Information*: This also is local. 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, an attachment type *Rule* has been registered 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 Sections 5 and 6 for details).

The execution modules depicted in Figure 1 are:

- *Rule Definition Module*: This component processes all rule definition commands. It is responsible for maintaining the Rule Catalog and updating the Global Rule Information. It also creates, deletes, and modifies rule attachment instances 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 occurs on a table with at least one rule.
- *Transition Table Procedures*: This set of procedures produces the transition tables that may be referenced in rule conditions and actions. Transition tables are implemented as (parameterless) table functions, so these procedures are registered with Starburst as described in Section 3.2. At run-time, they produce transition tables one tuple at a time, extracting the tuples from the Transition Log. Further details are given in Section 6.
- *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. 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. Rule ordering must be considered here; it is represented in the Global Rule Information (see Section 8). Rule conditions are checked and actions are executed by fetching them from the Rule Catalog (they are not stored in the Global Rule Information) and calling the Starburst query processor. When there are no triggered rules

left to execute and the Rule Execution Module terminates, Starburst commits the transaction.

The rule system also contains several components not illustrated in the diagram:

- *System Start-Up*: When 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. They are used during the course of each transaction, then reset after rule processing.
- *Rollback Handler*: The rule system must be prepared for a partial or complete rollback at any time. The Rule Catalog and attachment information is 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. Details of this component are in Section 10.

Our design differs from other database rule systems in several ways. The generality of the Rule Attachment Procedures, Transition Log, and Transition Table Procedures is unique to our system; it is necessary because rules are based on arbitrary state transitions (rather than tuple-level or, for some rule systems, statement-level changes) and can refer to the net effect of these transitions. While other systems have achieved varying degrees of interaction between rules and query processing, we have fully integrated rules into the database system. Rule definition is no different than data definition, and rule actions may perform all database operations. (In fact, the rule system will be used to implement future Starburst features.) Concurrency control, authorization, and transaction processing all are handled. Finally, we note that Starburst transactions for which no rules are applicable incur no overhead due to the existence of the rule system (see Section 7).

5 Rule Definition and Storage

For rule definition, we were able to take advantage of Starburst's extensible commands for creating, dropping, and altering attachment instances. Hence, the parser did not need to be extended, and procedures to handle rule definition commands could be registered with Starburst for automatic invocation. The general syntax for creating an attachment instance in Starburst is:

create *attach-type instance-name* **on** *table-name*
attribute-name attribute-value ,
. . .
attribute-name attribute-value ;

It is clear how we have adapted this syntax to rules. The *attachment-type* is **rule**, the *instance-name* is the rule name, and the allowable attributes are **when**, **if**, **then**, **precedes**, and **follows**, with values as described in Section 2. We do not, however, actually create one rule attachment instance (i.e., a separate attachment descriptor) for each rule created on a table. Rather, when our procedure for attachment instance creation is invoked, a new descriptor is created only if one does not already exist for that table, as described below.

When a new rule is defined, it is entered into the Rule Catalog. The Rule Catalog has a structured format (rather than containing the text of the rule) so that each separate component can be accessed quickly.[9] Some information on the new rule also must be entered into the Global Rule Information data structure. Concurrency control issues must be considered here, since other transactions may be affected by the new rule (see Section 9). An authorization scheme for rule definition also has been implemented (see Section 11).

If there currently are no rules on the new rule's table, then a rule attachment descriptor is created for the table; from that point on Starburst will invoke the procedures registered for rule attachments after every insert, delete, and update operation on the table. When invoked, these procedures are provided with the attachment descriptor. Since the information that is written to the Transition Log by these procedures depends on the triggering operations and transition table references in the rule, an *information code* to reflect this is stored in the attachment descriptor. If there already are rules on the new rule's table, then the only change necessary is recomputation of the information code to incorporate the new rule's triggering operations and transition table references (see Section 6).

Starburst also provides a general syntax for dropping and altering attachment instances, as illustrated for rules in Section 2. When a rule is dropped, it is deleted from the Rule Catalog, the Global Rule Information is modified, and the information code in the relevant attachment descriptor is recomputed. If the dropped rule is the last rule on its table, the attachment descriptor is deleted; subsequently, rule attachment procedures for that table no longer are invoked and operations on that table no longer are written to the Transition Log. When a rule is altered, the changes must again be reflected in the Rule Catalog, the Global Rule Information, and the information code in the attachment descriptor. In the case of deactivation, a flag is set for that rule in the Global Rule Information (and in the Rule Catalog) to indicate that the rule should not be executed. Activation causes the flag to be reset.

6 Transition Information

The attachment procedures that write to the Transition Log save information during a transaction so that the Rule Execution Module can determine which rules are triggered, and so the transition table references appearing in rule conditions and actions can be evaluated. Since the effect of rule action execution also is considered

[9] Currently, rule conditions and actions are stored as text, however we ultimately intend to store and perhaps cache them in compiled form.

by rules, the Transition Log must be maintained during rule processing as well; in our design this happens automatically.

The semantics of rule execution dictates that, at any given time, different rules may be considered with respect to different start points. (Recall Section 2: Each rule is triggered with respect to the transition since it was last considered, or since the start of the transaction if it has not yet been considered.) Hence, during rule processing, we may need to construct the net effect of many different transitions. To do this, entries in the Transition Log include a (logical) time-stamp which is obtained from Starburst. In the Rule Processing Information we track the most recent time at which each rule has been considered. The transition for a given rule is then constructed 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; we omit their enumeration here. From the set of rules on each table we compute the composite set of triggering operations and transition table references for that table. Based on this set, we produce the *information code* 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; our approach guarantees that we log all and only the necessary information.

The data structure used 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 given 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 (i.e., most recent first). It is sometimes 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 table and 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 table and tuple identifier. (In practice, we expect the tuple list to be short, since each tuple generally is not modified many times within a single transaction.)

We have developed a number of efficient algorithms for maintaining and traversing the Transition Log structure. The algorithms for checking if rules are triggered and for producing transition tables are almost identical—by definition a triggering operation has occurred if and only if the corresponding transition table is non-empty. However, for triggering, no values are needed and the algorithm stops as soon as it detects one instance of the operation.

We note one remaining issue regarding transition tables. Starburst table functions expect a fixed schema, declared when the function is registered (as described in Section 3.2). The schema of a transition table, however, depends on the schema of the table for the rule in which it is referenced. Hence, transition table schemata may change with each invocation. Fortunately, Starburst provides a mechanism allowing us to install the correct schema at compile-time. When a table function is registered with Starburst, an optional *semantic function* may also be registered. Before a table function reference is compiled, the semantic function is invoked. Our semantic functions for transition tables look up the schema of the appropriate table and use it to replace the compile-time (dummy) schema; compilation then proceeds normally.

7 Rule Execution

The Rule Execution Module must be invoked at the commit point of every transaction for which rules may have been triggered. 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. Hence, when the transaction is ready to commit, rule execution is invoked automatically. Note that we queue only the fact that rule execution should occur (and only once per transaction); which rules are triggered is determined at rule execution time.

A pseudo-code algorithm for rule execution is shown in Figure 2. *Potential-Rules* is part of the local (per-process) Rule Processing Information; it contains references to those rules potentially triggered at a given point in time. We say "potentially" triggered because the set is conservative—every triggered rule is in the set, but there may be rules in the set that actually are not triggered: At the start of rule processing (i.e., at the end of the initial transition) and at the end of each subsequent transition, we add to Potential-Rules all rules triggered by operations that occurred during the transition; we do not consider the net effect of the transition. Hence, for example, if tuples were inserted into table T during the transition,

```
/* find potentially triggered rules */
set-potential-rules();
/* process rules until none left */
while Potential-Rules != empty do
  /* look for triggered rule */
  found := false;
  while not found and Potential-Rules != empty do
    Rule := delete-next(Potential-Rules);
    Start-Time :=
      lookup-start-time(Rule, Rule-Processing-Info);
    found := check-triggered(Rule, Start-Time);
  if found then
    /* check condition, execute action */
    reset-start-time(Rule, Rule-Processing-Info);
    cond-holds := check-condition(Rule, Start-Time);
    if cond-holds then
      execute-action(Rule, Start-Time);
    /* new transition has occurred */
    /* find new potentially triggered rules */
    set-potential-rules()

set-potential-rules():
  scan Transition-Log for all operations in most
   recent transition;
  for each operation Op on table T do
    scan Global-Rule-Info for rules triggered
     by Op on T;
    for each rule R do insert R into Potential-Rules
```

Figure 2: *Rule Execution Algorithm*

then all rules triggered by **inserted** on T are added to Potential-Rules, regardless of whether the inserted tuples subsequently were deleted. We expect that it will be rare for operations in a transition to be "undone" in the net effect, so our set should not be overly conservative. However, before processing a rule from Potential-Rules, we must check that it is indeed triggered by considering the net effect (as described in Section 6). Recall that when a rule is fetched from Potential-Rules for consideration, it must be chosen such that no rules with higher precedence also are triggered. This is achieved by maintaining Potential-Rules as a sort structure based on rule ordering; see Section 8.

Lastly, we note some efficiency issues. In procedure *set-potential-rules*(), the Transition Log and the Global Rule Information are scanned. We have implemented the Transition Log hash structure such that scanning the most recent entries is fast. The Global Rule Information is organized as a hash structure based on operation and table, so finding all rules triggered by a given operation on a given table requires hashing once to obtain a linked list of the desired rules. Note, however, that for each transition including a particular operation, the Global Rule Information yields all rules triggered by that operation, regardless of whether the rules already are in Potential-Rules. To avoid attempts at adding duplicate rules to Potential-Rules, we keep in the Rule Processing Information a local hash structure similar to the Global Rule Information, but containing only rules not currently in Potential-Rules.

8 Rule Ordering

Recall from Section 2 that when a rule is defined, it may specify that other rules should precede or follow it. (Rule priorities may also be modified through **alter rule** statements.) We refer to these specifications as *user-defined rule priorities*, and the algorithm for rule execution must enforce all such priorities. We impose two additional constraints regarding rule ordering:

- *Transitivity*: If, in the user-specified priorities, R_1 precedes R_2 and R_2 precedes R_3, then R_1 should precede R_3, whether or not R_2 is triggered.
- *Determinism* (or *Repeatability*): If the same transaction is executed twice with the same database state and same set of rules, then all rules should be considered in the same order, even if some rules have no relative priority. (This is essential for debugging.)

To satisfy the first requirement, the system considers the transitive closure of rule priorities, rather than considering only user-defined priorities. (Note that circularities are not permitted: if creation of a rule would cause a cycle in user-defined or transitive priorities, then that **create rule** statement is rejected.) To satisfy the second requirement, the system uses a deterministic scheme to order rules with no user-defined or transitive priority; details of this scheme are given in [ACL91].

To enforce rule ordering, the set of potentially triggered rules described in Section 7 is maintained as a sort structure—a balanced binary tree. Inserting a rule or fetching the highest priority rule requires time proportional to *log*(*number of rules in structure*). However, to insert rules we must be able to determine efficiently, for any two rules, which precedes the other. A method for doing this is described in [ACL91].

9 Concurrency Control

Since Starburst is a multi-user database system, we must consider the effect on the rule system of concurrently executing transactions. For most transactions, including those with triggered rules, concurrency control is handled automatically by the database system. (Recall that rule conditions and actions are executed through the Starburst query processor.) However, the rule system itself must enforce concurrency control for transactions that affect the set of rules in the system, i.e., for transactions that include any form of rule definition. Many other database rule systems do not address this issue—rule definition takes place in single-user mode or off-line.[10] In our rule system, we have implemented concurrency control mechanisms that allow rule definition to be processed as a standard operation. The algorithms we use guarantee consistency, yet they permit substantial concurrency.

[10]In the *Ariel* [Han89] and *HiPAC* [MD89] systems, object-orientation allows rules to be specified as database objects, so concurrency control is handled automatically. This is not possible in Starburst.

We first define the consistency requirements, then explain the solutions we have implemented. There are three forms of consistency to consider: *intra-transaction consistency*, which specifies that relevant rules remain consistent throughout a transaction, *inter-transaction consistency*, which specifies that transactions are serializable with respect to rules (as well as data), and *ordering consistency*, which specifies intra-transaction consistency for user-defined and transitive rule priorities. (Inter-transaction consistency for priorities follows from inter-transaction consistency for rules.) The requirements are stated as follows.

- *Intra-transaction consistency*: Let X be a transaction. The set of rules on any table modified by X cannot change after the first time X modifies the table. If X modifies a table, X cannot subsequently change the rules on that table.
- *Inter-transaction consistency*: Let X_1 and X_2 be two transactions such that X_1 precedes X_2 in the serial schedule induced by the database system's concurrency control mechanism. If X_1 performs rule definition on a table modified by X_2, then X_2 must see the effect of X_1's rule definition. If X_2 performs rule definition on a table modified by X_1, then X_1 must not see the effect of X_2's rule definition.
- *Ordering consistency*: Let X be any transaction and let R_1 and R_2 be any two rules on tables modified by X. The precedence relationship between R_1 and R_2 must not change during X from the first time the relationship is used in rule selection.

In the Starburst locking scheme, locks are acquired throughout a transaction as needed and are held until the transaction commits or rolls back. Hence, the induced serial order of transactions is based on commit time. Intra- and inter-transaction consistency for rules is enforced quite easily using the existing lock mechanism, as follows. Let X be a transaction that performs rule definition. Consider one rule definition command in X, and let T be the table of the created, dropped, or altered rule. First, X checks to see if it has modified T. If so, the rule definition statement 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. (Modifying a table requires obtaining exclusive locks on the modified tuples, which are incompatible with a shared lock on the entire table.) X itself may subsequently modify T.

This scheme ensures both intra- and inter-transaction consistency. Consider first inter-transaction consistency. Let X_1 and X_2 be two transactions such that X_1 performs rule definition on a table T modified by X_2. Suppose X_1 precedes X_2 in the serial order, i.e., X_1 commits before X_2. Then X_2 must obtain its exclusive tuple-locks for modification after X_1 commits and releases its shared table-lock for rule definition. (Otherwise—if X_2 obtains its tuple-locks before X_1 commits—X_2 would have committed before X_1.) Thus, X_2 sees the effect of X_1's rule definition, as desired. Suppose instead that X_2 precedes X_1 in the serial order. Then X_1 must obtain its shared table-lock for rule definition after X_2 commits and releases its exclusive tuple-locks for modification. Thus, X_2 does not see the effect of X_1's rule definition, as desired. Intra-transaction consistency follows directly from inter-transaction consistency and the locking scheme.

To enforce ordering consistency, we introduce locks on rules. Let an *rs-lock* be a shared rule lock and let an *rx-lock* be an exclusive rule lock. (In Starburst, introducing locks on new objects is trivial.) Recall that during its rule execution phase, each transaction maintains a sorted data structure, *Potential-Rules*, of potentially triggered rules. Each time a transaction adds a rule to Potential-Rules, it obtains an rs-lock on that rule. Consider a transaction that performs rule definition. All rule definition commands—**create rule**, **drop rule**, and **alter rule**—may force some recomputation of rule priorities; even the ordering between unchanged rules may be reversed [ACL91]. When a rule definition command is executed, there is an identifiable minimal set S of rule pairs whose precedence relationship may be reversed by the resulting recomputation of priorities. Before changing priority data in the Global Rule Information, a transaction must obtain an rx-lock on each rule that appears in set S. To see how this guarantees ordering consistency, consider the rule execution phase of a transaction X_1. The first time X_1 adds a rule R to Potential-Rules, X_1 obtains an rs-lock on R. Consequently, no other transaction X_2 can perform any priority recomputation that may affect R, since X_2 would need to obtain an rx-lock on R. To prevent ordering relationships from changing within a transaction that performs rule definition, rs-locks cannot be upgraded to rx-locks. Although this scheme is more restrictive than required by the specification of ordering consistency, we believe it will have minimal impact on concurrency.

The rule system also must maintain consistency for its shared data—read and write operations must appear atomic. Starburst performs this task for the Rule Catalog and attachment information. For the Global Rule Information, we use Starburst's *latching* mechanism for mutual exclusion—data items are latched in shared mode for the duration of a read operation and are latched in exclusive mode for the duration of a write operation.

10 Rollback

The rule system maintains two memory-resident data structures throughout execution: the Transition Log, which contains information about operations relevant to rule triggering and execution, and the Global Rule Information, which contains information about the current set of rules. A third data structure, the Rule Processing Information, is maintained during the rule execution phase of each transaction; it contains information about potentially triggered rules and their triggering transitions. When a transaction is partially or completely rolled back, the rule system must ensure that its data structures are rolled back accordingly. It turns out that the Rule Processing Information never needs to be rolled back: a partial rollback cannot occur during rule execution except within the actions of a single rule, and a complete rollback causes termination of the rule execution phase. Hence, only the Transition Log and the Global Rule Information must be considered.

In the case of complete rollback, the Transition Log is set to empty. This occurs as part of a general rule system "cleanup" procedure for local data structures. (The cleanup procedure is placed on the *commit* event queue by each transaction the first time any information is written to the Transition Log, and is executed whether the transaction commits or rolls back.) In the case of partial rollback, the rule system must remove all entries in the Transition Log corresponding to operations that occurred during the rolled back portion of the transaction. Starburst allows arbitrary procedures to be invoked whenever a partial rollback occurs; these procedures are called with a time-stamp parameter indicating the point to which the transaction is rolling back. This time-stamp corresponds to the time-stamps in Transition Log entries. Hence, a procedure for the rule system is invoked on partial rollback and removes from the Transition Log all entries with a time-stamp greater than the time-stamp received as a parameter.

Handling rollback for the Global Rule Information is somewhat different. Recall that the Global Rule Information is initialized from the Rule Catalog on system startup, then modified only when a **create rule**, **drop rule**, or **alter rule** statement is executed. The rule system must ensure that whenever a rule definition statement is rolled back, the changes that were made to the Global Rule Information are undone. This is achieved by placing a parameterized procedure on the *commit* event queue each time a rule definition statement is executed. When one of these procedures is invoked with the "rollback" flag, it modifies the Global Rule Information appropriately. For example, when a **create rule** statement is processed, a procedure is queued with a parameter that identifies the created rule. If a rollback occurs, this procedure removes the information for that rule from the Global Rule Information. (Recall that the Rule Catalog and attachment information are rolled back automatically by the database system.) **alter rule** and **drop rule** are handled similarly.

11 Authorization

We must consider three distinct authorization issues for rule definition: authorization to create rules on a given table, authorization to create rules with given conditions and actions, and authorization to alter or drop given rules. In addition, we must consider how authorization is handled at rule execution time. To explain how these issues have been addressed, we first briefly describe (some of) Starburst's extensible authorization component [GLL89].

Lattices of *privilege types* can be defined for arbitrary database objects, and privileges on objects can be *granted to* and *revoked from* users and groups of users. In a lattice of privilege types, higher types subsume the privileges of lower types. As an 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.

The lattice of privilege types for rules is linear: the highest privilege is *control*, below this is *alter*, and privilege *deactivate/activate* is lowest. As with tables, a rule's creator obtains *control* privilege on the rule and may grant and revoke privileges on it. To create a rule R on a table T, R's creator must have both *attach* and *read* privileges on T. During rule creation, R's condition and actions are checked using the creator's privileges. If the condition or actions contain statements the creator is not authorized to execute, then the **create rule** statement is rejected. To drop a rule R on a table T, the requirement is 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 execution, each rule is processed using the privileges of its creator. To do this, privilege information for a rule's creator is stored together with the rule's condition and actions.

12 Status, Conclusions, and Future Work

The rule system as described in this paper is fully implemented and operational. It consists of approximately 24,000 lines of C code with comments and blank lines (approximately 8,000 semicolons), which includes a number of user facilities. The actual coding took only about 9 woman-months, but we carefully designed the system before any implementation began. Currently, we are exercising the system and intend to conduct performance measurements when a sufficient suite of examples has been developed.

The fact that the rule system could be implemented so quickly reflects well on the extensibility of Starburst. One intention of Starburst is that it can easily be extended by *external customizers*—database researchers not involved in the Starburst system itself. We point out that although the rule system design was performed in collaboration with a Starburst team member (and many others were consulted), the code itself was written entirely by two people who did not participate in the design or implementation of Starburst, i.e., by external customizers.

In addition to performance measurements, a number of enhancements and extensions to the rule system are planned for the future:

- A flexible mechanism for user-specified triggering points. With this mechanism, rules may be added to rule *assertion sets*, and an **assert rules** statement may be issued at any time to invoke rule processing for the rules in a given set. (Our semantics based on arbitrary transitions accommodates this approach directly.) All rules in the system are processed at the end of each transaction, as usual. This mechanism is fully designed and will be implemented shortly.

- Allow rule actions to call arbitrary procedures (which may include database operations).
- Allow rules to be triggered by operations on multiple tables. This is simply a matter of additional coding.
- Develop an algorithm for incremental evaluation of rule conditions, similar to that in [Han89,HCKW90] but adapted for our language and semantics.
- Implement a *dependency tracking* system for rules: If a rule's condition or action can no longer be executed because, for example, a table has been deleted or privileges have been revoked, then the rule should be invalidated. This will be included as part of a general dependency tracking facility planned for Starburst.

Clearly, the behavior of large sets of production rules can be difficult to understand and control, so we are developing a rule analysis facility as an aid to users of the system. The facility uses conservative algorithms to provide information about possible non-termination of rule sets, whether rule ordering may affect the final database state, etc. In addition, we plan to build special-purpose applications on top of the rule system, such as the constraint maintenance facility described in [CW90], the incremental view maintenance facility described in [CW91], and the deductive database facility described in [Wid91].

Acknowledgements

We are grateful to Rakesh Agrawal, Manish Arya, Stefano Ceri, George Lapis, Guy Lohman, John McPherson, Joel Richardson, and Ulf Schreier, who all have helped us out one way or another along the way.

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.

[Coh89] D. Cohen. Compiling complex database transition triggers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–234, Portland, Oregon, May 1989.

[CW90] 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.

[CW91] 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.

[DE89] 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.

[dMS88] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 234–241, Los Angeles, California, February 1988.

[Esw76] K.P. Eswaran. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. IBM Research Report RJ 1820, IBM San Jose Research Laboratory, San Jose, California, August 1976.

[GLL89] R. Gagliardi, G. Lapis, and B. Lindsay. A flexible and efficient database authorization facility. IBM Research Report RJ 6826, IBM Almaden Research Center, San Jose, California, May 1989.

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

[HCKW90] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, Atlantic City, New Jersey, May 1990.

[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.

[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.

[SLR88] 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.

[WF89] J. Widom and S.J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. IBM Research Report RJ 6880, IBM Almaden Research Center, San Jose, California, June 1989. Revised March 1990.

[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.

[Wid91] J. Widom. Deduction in the Starburst production rule system. IBM Research Report RJ 8135, IBM Almaden Research Center, San Jose, California, May 1991.