Proc. of 1990 ACM-SIGMOD Conference, pages 259–270

Set-Oriented Production Rules in Relational Database Systems

Jennifer Widom
Sheldon J. Finkelstein*

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

Abstract

We propose incorporating a production rules facility into a relational database system. Such a facility allows definition of database operations that are automatically executed whenever certain conditions are met. In keeping with the set-oriented approach of relational data manipulation languages, our production rules are also set-oriented—they are triggered by sets of changes to the database and may perform sets of changes. The condition and action parts of our production rules may refer to the current state of the database as well as to the sets of changes triggering the rules. We define a syntax for production rule definition as an extension to SQL. A model of system behavior is used to give an exact semantics for production rule execution, taking into account externally-generated operations, self-triggering rules, and simultaneous triggering of multiple rules.

1 Introduction

Recently, there has been considerable interest in integrating production rules systems and database management systems. Some work, such as [DE89, SLR88, Tzv88], focuses on using database technology to efficiently support OPS-like production rules languages [BFKM85]. Other work—including ours—focuses on extending database systems to include a production rules facility [Coh89, dMS88, Esw76, Han89, KDM88, MD89, RS89, SHP88, SJGP90]. Generally, production rules take the form *when* X *then* Y, where X is a triggering condition and Y is an action: whenever condition X is true, action Y is performed. Specific forms of such rules may be used in particular environments.

*Author's current address: Tandem Computers, 19333 Vallco Parkway, LOC 3-15, Cupertino, CA 95014

We consider production rules in the context of relational database systems. A number of potential uses motivate the addition of such a facility. Production rules are a natural mechanism for integrity constraint enforcement, as suggested by [Esw76]. More generally, production rules permit additional semantic structure for the database. In active database systems [MD89, Mor83], production rules are used to monitor particular conditions (sometimes associated with timing constraints) and, when appropriate, to trigger corresponding actions. [Esw76] suggests that production rules may be useful for authorization checking and for maintenance of derived data. Finally, production rules in database systems provide a flexible framework for building efficient knowledge-base and expert systems.

We propose a production rules facility compatible with the SQL data manipulation language [IBM88], although our framework could apply equally well to other relational database languages. Rule activation results from user-generated (or application-generated) database operations. In this paper, we focus on the syntax of rule definition and the semantics of rule execution. (Details of implementation issues will be presented in a future report.) Our syntax is straightforward, based directly on SQL. The semantics of rule execution is somewhat more complicated. A number of issues must be addressed:

- *Trigger cause*: What causes a rule to be triggered? Is it a database state, a transition from one state to another, either, both?
- *Transition granularity*: If rules can be triggered by state transitions, what exactly constitutes a transition? An operation on a single tuple? A set-oriented database update? A transaction?
- *Execution points*: When are rules executed? At any time? Only after certain operations? Only at transaction boundaries?
- *Execution scheduling*: What happens if several rules are triggered at the same time? Are all rules executed? If so, is there an order? Is only one rule executed? If so, how is it chosen?
- *Execution environments*: If rules are not always executed as soon as they are triggered, what environments are used when rules finally execute?
- *Trigger cascade*: What happens if execution of a rule causes another rule to trigger? How does the

new rule interact with other triggered rules? Can a rule trigger itself?

- *Trigger permanence*: If several rules are triggered simultaneously, what happens if execution of one rule's action negates another rule's condition?

Many existing proposals leave some of these issues unresolved or unclear, or suggest restrictive solutions. We provide a precise semantics that allows an easy understanding of rule behavior while remaining expressive and flexible.

Most other proposals for production rules in database systems (e.g., [Coh89, dMS88, Esw76, MD89, SJGP90]) consider *instance-oriented* rules: rules that are applied once for each data item satisfying the condition part of the rule. (For example, one might define an instance-oriented rule that is applied once for each tuple inserted into the database.) In contrast, we propose *set-oriented* rules: rules that are triggered by arbitrary sets of changes to the database and may perform sets of changes. (For example, a single set-oriented rule might operate on all tuples inserted into the database during the course of a transaction.) This approach conforms to the set-oriented approach of relational database languages. In particular, set-oriented processing in relational database systems permits efficient execution of non-procedural queries through extensive optimization. Such optimization is not inhibited by the presence of our set-oriented production rules; furthermore, it is directly applicable to the rules themselves. Set-oriented execution may also allow rules to be combined with database operations or with other rules [FPT88]. In some cases, instance-oriented rules can be compiled for set-oriented execution [SdM88, SJGP90], but our direct approach avoids complications and limitations arising from such a scheme. Finally, our set-oriented rules allow specification of some conditions and actions not expressible using instance-oriented rules.

The paper proceeds as follows. Section 2 describes our model of database system execution. We introduce the notion of "operation blocks"—sequences of data manipulation operations that appear to execute atomically and form the basis of the database transitions that trigger our production rules. A semantics for operation blocks is rigorously defined, based on the semantics of the component operations. In Section 3, we give a syntax for production rule definition; several examples are included. The semantics of rule execution in the presence of externally-generated database transitions is defined in Section 4. As a first step towards implementation, Section 4 also includes an algorithm for rule execution that reflects the desired semantics. A number of potential extensions are proposed in Section 5. Section 6 contains discussion of future work.

2 Model of System Execution

We assume a typical relational database structure [Cod70]: a set of named *tables* (or *relations*) is defined, each having a fixed set of named and typed *columns* (or *attributes*).[1] In a given *state* of the database, each table contains zero or more *tuples*, where a tuple assigns a single value (or **null**) to each column of the table. Duplicate tuples may appear in a table. For convenience, we assume that associated with each tuple is a system *tuple handle*—a distinct, non-reusable value identifying the tuple and its containing table. In subsequent discussion, we use tuple handles to identify certain (multi)sets of tuples—some are tuples in the database, while others are tuples that have existed in a previous state of the database but have since been deleted.

2.1 Operation Blocks

In relational database systems, users (or application programs) submit streams of data manipulation operations for execution. We consider a model of system behavior in which these operations are grouped into *operation blocks*. We assume that operation blocks always finish and are executed indivisibly—we consider a level of abstraction in which multiple users, concurrent processing, and failures are all transparent. (The correspondence between operations blocks and database transactions is discussed in Section 4.) During execution of an operation block, sets of tuples may be inserted, deleted, or updated; for concreteness, we let each block correspond to a non-empty sequence of SQL **insert**, **delete**, and **update** operations.[2] We consider operation blocks (rather than individual operations) for generality and to permit a formalism that adapts easily to most relational database languages. Using a variant of the SQL data manipulation language [IBM88] and assuming some familiarity with SQL, a syntax for operation blocks is given as follows:

op-block	::=	*sql-op* ; *sql-op* ; ... ; *sql-op*
sql-op	::=	*insert-op* | *delete-op* | *update-op*
insert-op	::=	**insert into** *table* **values** $\langle v_1, v_2, \ldots, v_n \rangle$
	|	**insert into** *table* (*select-op*)
delete-op	::=	**delete from** *table* [**where** *predicate*]
update-op	::=	**update** *table* **set** *columns* = *expressions* [**where** *predicate*]
select-op	::=	**select** *columns* **from** *tables* [**where** *predicate*]

The predicate in **delete**, **update**, and **select** operations may be arbitrarily complex and may include em-

[1] In some cases the database schema may change over time. For simplicity, we assume a fixed schema. Also, in this paper we do not consider views.

[2] For simplicity, we do not include **select** operations here since they leave the database state unchanged. A possible extension is to consider **select** operations here, which would allow rules to be triggered by data retrieval; see Section 5.

bedded **select** operations. If the predicate is omitted, the meaning is equivalent to including **where true**.

We describe the behavior of the SQL operations with respect to a given database state. For each operation we also define an *affected set*: a set of tuple handles identifying those tuples "affected" by the operation. For **update** operations, the affected set also indicates which columns are updated. (Affected sets are used below to define the semantics of operation blocks.)

- **insert** with values: A new tuple containing values $\langle v_1, v_2, \ldots, v_n \rangle$ is added to the specified table. The affected set for this operation contains the handle for the inserted tuple.
- **insert** with select operation: The embedded select operation is evaluated producing a group of tuples. Each tuple is inserted into the specified table. The handles for the inserted tuples form the affected set for this operation.
- **delete**: The tuples in the specified table satisfying the given predicate are identified. The handles for these tuples form the affected set for this operation. The tuples are deleted from the table.[3]
- **update**: The tuples in the specified table satisfying the given predicate are identified. For each tuple, the expressions are evaluated and the results are assigned to the specified columns. The affected set contains handle-column pairs identifying each updated tuple and specified column.[4] (Note that it is possible for the values of columns affected by an **update** operation to actually remain unchanged.)

Now consider an operation block. Execution begins in some initial database state. Each operation in the block is executed as described above, producing a new database state which is used by the subsequent operation. Since operation blocks are executed indivisibly, we need not consider the intermediate states, but only the state transition that results from executing the entire sequence of operations. Hence, we refer to execution of an operation block as a *transition*. A transition labeled $\mathcal{T}$ taking a database from a state S to a state S' is denoted graphically as:

$$S \xrightarrow[\mathcal{T}]{} S' \tag{1}$$

2.2 Transition Effects

For production rules, we are particularly interested in the "effect" of a transition: those tuples in the database that have been inserted, deleted, or updated. Since operation blocks are executed indivisibly, we want to consider the overall effect of a transition, rather than the individual effects of the component operations. For example, if a tuple is updated by several operations and then deleted, we consider only the deletion, since this is the net effect of the transition. Similarly, multiple updates of a tuple are considered as a single update, an insertion followed by an update is considered as an insertion of the updated tuple, and an insertion followed by a deletion is not considered at all. In contrast, however, we never consider deletion of a tuple followed by insertion of a new tuple as an update to the original tuple.

Formally, the *effect* of a transition is a triple, $[I, D, U]$: I is a set of handles identifying those tuples inserted by the transition, D is a set of handles identifying those tuples deleted by the transition, and U is a set of handle-column pairs identifying those tuples and columns updated by the transition. Note that since $[I, D, U]$ represents the overall effect of a transition, a given tuple handle may appear in at most one of the three sets.

Although sets I and D of a transition effect are always derivable from the database states preceding and following the transition (i.e., states S and S' in (1) above), set U is not. As mentioned above, U contains handle-column pairs for those tuples selected for an update, regardless of whether a value is actually changed. The effect of a transition is, however, always derivable from the operation block producing the transition, along with the state of the database as the operation block executes. (Computing transition effects this way is necessary for efficiency, in any case.) We give a rigorous description of the derivation, since the concepts and definitions introduced here also are used in Section 4 to define the semantics of production rule execution.

Let B be an operation block. The effect of a transition corresponding to execution of block B, denoted $\mathcal{E}(B)$, is defined inductively. Suppose first that B consists of a single operation op, and let $\mathcal{A}(op)$ denote the affected set for op (as defined in Section 2.1):

- If op is an **insert**, then $\mathcal{E}(B) = [\mathcal{A}(op), \emptyset, \emptyset]$.
- If op is a **delete**, then $\mathcal{E}(B) = [\emptyset, \mathcal{A}(op), \emptyset]$.
- If op is an **update**, then $\mathcal{E}(B) = [\emptyset, \emptyset, \mathcal{A}(op)]$.

Next, we give a general procedure for composing consecutive transition effects. Suppose an operation block B_1 is executed, producing a transition $\mathcal{T}_1$ with effect $\mathcal{E}(B_1) = [I_1, D_1, U_1]$. Then an operation block B_2 is executed, producing a transition $\mathcal{T}_2$ with effect $\mathcal{E}(B_2) = [I_2, D_2, U_2]$. In our graphical notation, the effect of a transition is indicated above the arrow for the transition. Thus, we have:

$$S \xrightarrow[\mathcal{T}_1]{\mathcal{E}(B_1)} S' \xrightarrow[\mathcal{T}_2]{\mathcal{E}(B_2)} S''$$

We are interested in the transition effect that corresponds to treating the two transitions as an indivisible unit, i.e., when $B_1 \,;\, B_2$ is a single operation block:

$$S \xrightarrow[\mathcal{T}]{\mathcal{E}(B_1 \,;\, B_2)} S'' \tag{2}$$

[3]Recall that tuple handles are not reused. After the deletion, the affected set identifies tuples in a previous state of the database.

[4]For convenience, we define the affected set to include one element for each column of each updated tuple. In practice, this set can contain one element for each updated tuple, identifying all updated columns.

Definition 2.1 (Transition Effect Composition) Let $[I_1, D_1, U_1]$ and $[I_2, D_2, U_2]$ be transition effects. The *composition* of these two effects, denoted $[I_1, D_1, U_1] \circ [I_2, D_2, U_2]$, is effect $[I, D, U]$ where:

- $I = (I_1 \cup I_2) - D_2$.
  I identifies all tuples that were inserted by either transition and were not subsequently deleted. Note that there is no need to subtract set D_1, since handles in D_1 cannot appear in I_1 or I_2.
- $D = (D_1 \cup D_2) - I_1$.
  D identifies all tuples that were deleted by either transition and existed prior to the first transition (i.e., were not inserted by the first transition).
- $U = (U_1 \cup U_2) - (D_2 \cup I_1)$.
  U identifies all tuples that were updated by either transition, were not subsequently deleted, and existed prior to the first transition. Note our "misuse" of the set difference operator here. The intended meaning is to delete from set $U_1 \cup U_2$ every handle-column pair where the handle appears in set $D_2 \cup I_1$. □

The effect of indivisibly executing two consecutive operation blocks is thus defined as the composition of the effects of the individual blocks. That is, in (2) above, $\mathcal{E}(B_1 ; B_2) = \mathcal{E}(B_1) \circ \mathcal{E}(B_2)$.

Definition 2.1 allows us to complete our formal derivation of transition effects. Let operation block $B = op_1 \; ; \; op_2 \; ; \; \ldots \; ; \; op_n$. Then $\mathcal{E}(B) = \mathcal{E}(op_1) \circ \mathcal{E}(op_2 \; ; \; \ldots \; ; \; op_n)$. By induction and since composition operator $\circ$ is associative, without ambiguity we can write $\mathcal{E}(B) = \mathcal{E}(op_1) \circ \mathcal{E}(op_2) \circ \ldots \circ \mathcal{E}(op_n)$.

3 Rule Definition

Production rules are composed of three main components: a *transition predicate*, an optional *condition*, and an *action*. The transition predicate controls rule triggering; the condition specifies an additional predicate that must be true if a triggered rule is to automatically execute its action. An SQL-like syntax is used for production rule definition:

prod-rule-def	::=	**create rule** *name*
		when *trans-pred*
		[**if** *condition*]
		then *action*

We elaborate on the definition and informal semantics of the transition predicate, condition, and action. A detailed semantics of rule execution is given in Section 4.

Production rules are triggered by database state transitions—i.e., by execution of operation blocks. After a given transition, those rules whose transition predicate holds with respect to the effect of the transition are triggered. A transition predicate is a list of *basic transition predicates*, which specify particular operations on particular tables. For example, basic transition predicate **inserted into t** (where **t** is a table name) holds with respect to any transition effect in which the I component identifies one or more tuples in table **t**; basic transition predicate **deleted from t** is similar. Basic transition predicate **updated t.c**, where **c** names a column of table **t**, holds with respect to any transition effect in which the U component contains a handle-column pair identifying column **c** of a tuple in table **t**. We also allow simply **updated t**, which holds whenever U identifies a tuple in table **t**, regardless of column. In a rule, the list of basic transition predicates is a disjunction: the rule is triggered by any transition with an effect satisfying one or more of the basic predicates in the list. (In [WF89a], we show that it is possible to use the condition part of a rule to obtain the effect of arbitrary boolean combinations of basic transition predicates.) The syntax for transition predicates is thus:

trans-pred	::=	*basic-trans-pred*
	|	*basic-trans-pred* **or** *trans-pred*
basic-trans-pred	::=	**inserted into** *table*
	|	**deleted from** *table*
	|	**updated** *table.column*
	|	**updated** *table*

The condition part of a rule is an SQL predicate. The predicate may be arbitrarily complex and may include embedded **select** operations; it may also be omitted, in which case the meaning is equivalent to including **if true**. Embedded **select** operations allow SQL predicates to refer to the "current" state of the database—the contents of the tables at the time the predicate is evaluated. In addition, we want a rule's condition to be able to refer to the changes that triggered the rule. For example, if a rule is triggered because transition predicate **inserted into t** holds, then in the condition part of the rule we might want to refer to the set of tuples in table **t** that were inserted by the transition that triggered the rule. Hence, for each basic transition predicate in a rule's list, the rule may refer to one or two corresponding logical tables, as follows (**t** is any table name):

- If **inserted into t** is a basic transition predicate, then logical table **inserted t** refers to the tuples of table **t** in the current state of the database that were inserted by the transition that triggered the rule.
- If **deleted from t** is a basic transition predicate, then logical table **deleted t** refers to the tuples of table **t** in the previous state of the database that were deleted by the transition that triggered the rule.
- If **updated t.c** is a basic transition predicate, then logical table **old updated t.c** refers to the tuples of table **t** in the previous state of the database in which column **c** was updated by the transition that triggered the rule; logical table **new updated t.c** refers to the current values of the same tuples.
- If **updated t** is a basic transition predicate, then logical table **old updated t** refers to the tuples of table **t** in the previous state of the database that

were updated by the transition that triggered the rule; logical table **new updated t** refers to the current values of the same tuples.

We call these logical tables *transition tables*.[5] A rule may refer to transition tables corresponding to its basic transition predicates by including them in a **select** statement in the usual way; for example:

> **select** *columns*
> **from** ..., **inserted t tvar** , ...
> **where** *predicate*

References to table variable **tvar** in the **select** and **where** clauses of this operation refer to logical table **inserted t**. The variable name (**tvar**) may be omitted as long as there is no conflict with other references to table **t** (see the examples below). The syntax for the condition part of a rule is thus:

> *condition* ::= *predicate**

where the * indicates that references to transition tables are permitted. Note that our syntax does not enforce the restriction that a rule's condition may only refer to transition tables corresponding to its basic transition predicates. This restriction is syntactic, however, therefore easily checked.

The action part of a production rule is an operation block, as defined in Section 2.1. We may eventually choose to obtain more flexibility by permitting actions to call external procedures (which may include data manipulation operations); see Section 5. However, for defining the semantics of rule execution with respect to the database system, it is sufficient to assume that the action is an arbitrary sequence of SQL operations. As in the condition part of a rule, it may be useful in the action to refer to the changes that triggered the rule. Therefore, we allow embedded **select** operations in the action to refer to transition tables corresponding to the rule's basic transition predicates. Finally, we also permit the action to simply request a rollback of the current transaction. Thus, the syntax for actions is:

> *action* ::= *op-block**
> | **rollback**

where, again, the * indicates that (legal) references to transition tables are permitted.

3.1 Examples

Although we have not yet defined a complete semantics of rule execution, we give examples illustrating rule definition. The examples included here and in Section 4.5 are somewhat unrealistic, but they serve to illustrate important aspects of production rule definition and execution. Additional examples pertaining to a fairly large case study appear in [CW90].

[5] Transition tables allow access to a subset of a previous database state in a rule's execution environment. Determining which database versions may be referenced in production rules involves difficult trade-offs among simplicity, expressiveness, and implementation. A precise semantics for our transition tables appears in Section 4.

Consider a database schema with two tables, **emp** and **dept**. Table **emp** maintains employee data, having columns for **name**, **emp_no** (which is unique—a key for the table), **salary**, and **dept_no**. Table **dept** maintains department data, having columns for **dept_no** and **mgr_no** (both are unique). Thus, we have:

> **emp(name, emp_no, salary, dept_no)**
> **dept(dept_no, mgr_no)**

We use this schema for examples throughout the paper.

Example 3.1 Our first example illustrates the "cascaded delete" method of enforcing referential integrity [IBM88]: Whenever departments are deleted, delete all employees in the deleted departments.

```
when deleted from dept
then delete from emp
      where dept_no in
         (select dept_no from deleted dept)
```

No **if** clause is needed in this rule—we want it to execute whenever one or more departments are deleted.

Example 3.2 Our second example is more complex: Whenever employee salaries are updated, if the total of the updated salaries exceeds their total before the updates, then give all employees of department #2 a 5% salary cut and give all employees of department #3 a 15% salary cut.

```
when updated emp.salary
if (select sum(salary)
    from new updated emp.salary) >
   (select sum(salary)
    from old updated emp.salary)
then update emp
        set salary = 0.95 * salary
        where dept_no = 2;
      update emp
        set salary = 0.85 * salary
        where dept_no = 3
```

Example 3.3 Our third example illustrates a composite transition predicate: Whenever employees are inserted or deleted, or employee salaries or department numbers are updated, check if any employee's salary exceeds twice the average salary for his department. If so, delete the manager of department #5.

```
when inserted into emp
       or deleted from emp
       or updated emp.salary
       or updated emp.dept_no
if exists (select * from emp e1
             where salary >
                   2 * (select avg(salary)
                        from emp e2
                        where e2.dept_no =
                              e1.dept_no))
then delete from emp
      where emp_no =
          (select mgr_no from dept
           where dept_no = 5)
```

4 Rule Execution

Production rules are activated automatically as a result of database state transitions caused by externally-generated operation blocks. Suppose that a stream of operation blocks is submitted for execution. If execution begins in a state S_0 and production rules are not considered, then the system behaves as follows:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1 \xrightarrow[\mathcal{T}_2]{\mathcal{E}_2} S_2 \xrightarrow[\mathcal{T}_3]{\mathcal{E}_3} \cdots \quad (3)$$

$\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3 \ldots$ are unique transition labels; $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3 \ldots$ are the effects of the transitions. Recall that each transition effect is actually a triple $[I, D, U]$.

In this paper, we assume that externally-generated operation blocks correspond to database transactions.[6] That is, each state in execution sequence (3) above corresponds to a state in which a transaction begins execution; the subsequent state corresponds to the point at which the user or application program requests that the transaction be committed. This one-to-one correspondence between transitions and transactions holds only for externally-generated operations. When rules are executed, a single transaction is composed of one externally-generated transition followed by some number of rule-generated transitions (that is, rules are considered and executed just before committing each externally-generated transaction). If a rule with a **rollback** action is executed, the system immediately rolls back to the start state for the transaction, i.e., to the state preceding the initial externally-generated transition. A detailed semantics follows.

4.1 A Single Rule

Since the semantics of rule execution when only one rule is defined is straightforward, we begin by describing this case. In Section 4.2 we generalize to multiple rules. Consider one production rule R (with a non-**rollback** action):

when *trans-pred*
if *condition*
then *op-block*

and consider an externally-generated transition $\mathcal{T}_1$ with effect $\mathcal{E}_1$:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1$$

Rule R is triggered in state S_1 if R's transition predicate holds with respect to transition effect $\mathcal{E}_1$, as defined in Section 3. Triggering alone is not enough to initiate execution of the action part of a rule—the specified condition must also hold. R's condition may refer to the current state of the database (state S_1); additionally, R's condition may refer to the logical transition tables introduced in Section 3. Recall that transition tables are based on transition effects and may refer to the current state or the pre-transition state of the database. (For example, transition table **inserted t** contains those tuples of table **t** in state S_1 identified by tuple handles in the I component of $\mathcal{E}_1$, while transition table **old updated t.c** contains those tuples of table **t** in state S_0 identified by a handle appearing with column **c** in the U component of $\mathcal{E}_1$.) Thus, evaluation of R's condition may depend on $\mathcal{E}_1$, S_1, and S_0.

If R's condition holds, then, before execution of another externally-generated operation block, R's action is executed. As with evaluation of R's condition, execution of R's action may depend on the tables in state S_1 as well as on transition tables. Execution of R's action causes a new transition:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1 \xrightarrow[\mathcal{T}_2\,[R]]{\mathcal{E}_2} S_2 \qquad (4)$$

Notation $[R]$ is used to indicate that transition $\mathcal{T}_2$ was generated by rule R.

Despite the fact that transition $\mathcal{T}_2$ is caused by a rule rather than by an externally-generated operation block, $\mathcal{T}_2$ is indistinguishable from other transitions—the action part of a rule is an operation block producing an effect and a new state. Thus, we allow the transition generated by execution of a rule to trigger other rules, or even trigger the same rule again. Since we are restricting attention to a single rule, consider an example of the latter case. Suppose the triggering condition of rule R is an **update** to column **c** of table **t** and, as its action, rule R further updates **c**:

when updated t.c
if *condition*
then update t
  set c = *expression*
  where *predicate*

Suppose rule R is triggered by transition $\mathcal{T}_1$, its condition holds, and its action is executed, as in (4) above. Transition $\mathcal{T}_2$ then corresponds to an operation block consisting of a single **update t ...** operation. Assume that the U component of $\mathcal{E}_2$ is non-empty, i.e., at least one tuple is selected in R's update. Then rule R is triggered again in state S_2. If R's condition again holds, then R's action is executed a second time and we have:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1 \xrightarrow[\mathcal{T}_2\,[R]]{\mathcal{E}_2} S_2 \xrightarrow[\mathcal{T}_3\,[R]]{\mathcal{E}_3} S_3$$

If the U component of $\mathcal{E}_3$ is non-empty, R is triggered again in state S_3, and its condition may still hold. In fact, if R is always triggered by the transition caused by executing its action, and its condition always holds, then execution of R repeats indefinitely.[7]

---

[6] Our approach can be extended to permit more flexibility with respect to transitions and transactions; see Section 5.

[7] The potential for such behavior is similar to the potential for infinite loops in conventional programs. We might want to provide a static rule analysis facility that issues a warning when the possibility of divergence is detected; see Section 6. Run-time detection using a timeout mechanism could also be incorporated.

4.2 Multiple Rules

Now consider multiple rules. Like the semantics given for a single rule, the interaction of externally-generated operation blocks and rules is:

1. Execute an externally-generated operation block, creating a transition.
2. Repeatedly execute rules (creating new transitions) until there are none left to execute or a **rollback** occurs.
3. Go to 1.

We elaborate on step 2 in the case when several rules may be triggered by a single transition. Assuming we are not interested in a semantics based on parallel execution, the triggered rules must be activated in some order, taking into consideration the transitions created as rules are executed (which may also trigger new rules). We begin by considering in detail the initial rule-generated transitions of step 2. We then generalize by describing system behavior at an arbitrary point during rule processing.

Let $\mathcal{T}_1$ be the externally-generated transition in step 1, and suppose that its effect $\mathcal{E}_1$ contains at least one non-empty set. (If all three sets in $\mathcal{E}_1$ are empty, then no rules can be triggered and step 2 is trivial.)

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1$$

As a result of transition $\mathcal{T}_1$, some set of rules $R_1, R_2, \ldots, R_n$ are triggered. A rule is chosen from this set for consideration; we defer discussion of how a rule is chosen—several possibilities are discussed in Section 4.4 below—and assume that we have some rule selection method. Suppose a rule R_i is chosen. R_i's condition is evaluated with respect to current state S_1 and transition tables based on $\mathcal{E}_1$, S_1, and S_0. If R_i's condition does not hold, then R_i's action is not executed and a new triggered rule is chosen for consideration.

Suppose R_i's condition does hold, and assume that R_i's action is not **rollback**. Then the action is executed and we have:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1 \xrightarrow[\mathcal{T}_2\ [R_i]]{\mathcal{E}_2} S_2$$

Now we must carefully define which rules are (still) triggered in state S_2, and, for the triggered rules, how transition tables are to be evaluated.

As in the single-rule case described in Section 4.1, rule R_i is triggered a second time in state S_2 if its transition predicate is satisfied by effect $\mathcal{E}_2$ of rule-generated transition $\mathcal{T}_2$. If R_i is chosen for consideration, it uses S_2 as the new current state, and its transition tables are based on $\mathcal{E}_2$, S_2, and S_1.

For all rules other than R_i, we treat rule-generated transition $\mathcal{T}_2$ as if it were an additional sequence of operations executed as part of the preceding externally-generated transition. That is, transitions $\mathcal{T}_1$ and $\mathcal{T}_2$ are considered as a single transition with effect $\mathcal{E}_1 \circ \mathcal{E}_2$ (recall Definition 2.1 of transition effect composition):

$$\begin{tikzcd} S_0 \arrow[r, "\mathcal{E}_1", "\mathcal{T}_1"'] \arrow[rr, bend left, "\mathcal{E}_1 \circ \mathcal{E}_2"] & S_1 \arrow[r, "\mathcal{E}_2", "\mathcal{T}_2\ [R_i]"'] & S_2 \end{tikzcd}$$

Any rule other than R_i is triggered in state S_2 if its transition predicate is satisfied by composite effect $\mathcal{E}_1 \circ \mathcal{E}_2$. Such a rule's transition tables are based on effect $\mathcal{E}_1 \circ \mathcal{E}_2$, current state S_2, and pre-transition state S_0. As an example, consider a rule R_j that was triggered in state S_1 but was not considered prior to rule R_i. Rule R_j is still triggered in state S_2 as long as transition $\mathcal{T}_2$ does not "undo" the changes that initially caused R_j to be triggered. Similarly, a rule R_k that was not triggered in state S_1 is now triggered in state S_2 as long as the net effect of transitions $\mathcal{T}_1$ and $\mathcal{T}_2$ satisfy the rule's transition predicate. Finally, a rule that was triggered in S_1 but whose condition was found to be false may be reconsidered in S_2 as long as it is still triggered by the composite effect.

In summary, the set of triggered rules in state S_2 contains R_i if it is re-triggered by transition $\mathcal{T}_2$, along with all rules whose transition predicate is satisfied by effect $\mathcal{E}_1 \circ \mathcal{E}_2$. From this set, rules are chosen for consideration until one is found with a condition that holds or until there are none left. If a rule with a true condition is found, its action is executed, creating a new transition.

Consider now an arbitrary point in rule processing. A state S_n has been reached, resulting from execution of a user-generated operation block followed by some number of rules:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1 \xrightarrow[\mathcal{T}_2\ [R_i]]{\mathcal{E}_2} \cdots \xrightarrow[\mathcal{T}_n\ [R_j]]{\mathcal{E}_n} S_n$$

In state S_n, rules are triggered based on composite transition effects; the transition tables in their conditions and actions are defined accordingly. If a rule R_k has not yet generated any transitions in the sequence (i.e., R_k's action has not been executed since the most recent externally-generated transition), then R_k is triggered in state S_n if its transition predicate is satisfied by the composite effect since state S_0:

$$\begin{tikzcd} S_0 \arrow[r, "\mathcal{E}_1", "\mathcal{T}_1"'] \arrow[rrr, bend left, "\mathcal{E}_1 \circ \mathcal{E}_2 \circ \cdots \circ \mathcal{E}_n"] & S_1 \arrow[r, "\mathcal{E}_2", "\mathcal{T}_2\ [R_i]"'] & \cdots \arrow[r, "\mathcal{E}_n", "\mathcal{T}_n\ [R_j]"'] & S_n \end{tikzcd}$$

If rule R_k is indeed triggered and it is chosen for consideration, then its transition tables are based on transition effect $\mathcal{E}_1 \circ \mathcal{E}_2 \circ \cdots \circ \mathcal{E}_n$, current state S_n, and pre-transition state S_0.

Suppose instead that rule R_k's action has been executed since initial transition $\mathcal{T}_1$. Then R_k is considered with respect to the composite transition effect starting with the most recent transition generated by that rule. That is, if $\mathcal{T}_x$ is the most recent transition generated by rule R_k, then R_k is triggered in state S_n if its transition predicate is satisfied by $\mathcal{E}_x \circ \mathcal{E}_{x+1} \circ \cdots \circ \mathcal{E}_n$:

$$\begin{tikzcd} S_0 \;\cdots\; S_{x-1} \arrow[r, "\mathcal{E}_x", "\mathcal{T}_x\ [R_k]"'] \arrow[rr, bend left, "\mathcal{E}_x \circ \mathcal{E}_{x+1} \circ \cdots \circ \mathcal{E}_n"] & \cdots \arrow[r, "\mathcal{E}_n", "\mathcal{T}_n\ [R_j]"'] & S_n \end{tikzcd}$$

In this case, rule R_k's transition tables are based on transition effect $\mathcal{E}_x \circ \mathcal{E}_{x+1} \circ \cdots \circ \mathcal{E}_n$, current state S_n, and pre-transition state S_{x-1}. Note that this is a natural extension of the semantics given above for a single self-triggering rule.[8]

Following this definition, there is some set of rules triggered in state S_n. Rules are repeatedly chosen from this set and their conditions are evaluated (with transition tables as defined above). If the set is empty or if no rule with a true condition is found, then rule processing terminates and the current transaction (which began in state S_0) is committed. Otherwise, if some rule with a true condition is found, the rule's action is executed. If the action is **rollback**, then the system rolls back to state S_0; no rules are triggered and a new externally-generated operation block may begin execution. Otherwise, if the rule's action is an operation block, a new transition is created and there is a new set of triggered rules to consider.

4.3 Rule Execution Algorithm

We now give an algorithm that reflects our semantics of rule execution. The algorithm is somewhat more "procedural" than the semantics as described above. In particular, we do not assume access to all previous states and transition effects. Rather, with each defined rule we associate composite transition information starting with the state in which the rule's action was last executed (or the state preceding the initial externally-generated transition). This information can be used at any time to determine if the rule is triggered and, if so, to construct the relevant transition tables. Each rule's transition information is modified incrementally as operation blocks are executed and new transitions are created.

Many aspects of the algorithm are still simplified considerably from an implementation. For example, the entire database state need not be saved before each transition. Rather, the necessary transition information can be accumulated within transitions similarly to the way, in this algorithm, we maintain composite information between transitions (see function *modify-trans-info* in Fig. 1 below, which follows Definition 2.1 of transition effect composition). Also, for simplicity, in the algorithm we associate with each rule the full set of transition information; in actuality we need only save the subset of that information relevant to the particular rule. Of course, in many cases associating transition information on a rule-by-rule basis will introduce considerable redundancy—there is substantial need and room for optimization here.

The main algorithm is given in Fig. 1, along with three functions it calls. The algorithm loops indefinitely as long as there are externally-generated operation blocks to be executed. Each iteration of the loop corresponds to a single transaction containing an externally-generated transition followed by consideration and possible execution of some number of rules. The following variables are used:

- *old-state* of type *database state*;
- $\mathcal{E}$ of type *transition effect* (we let $I(\mathcal{E})$, $D(\mathcal{E})$, and $U(\mathcal{E})$ denote the I, D, and U components of $\mathcal{E}$, respectively);
- *ins* of type *set of tuple handle*;
- *del* of type *set of tuple value*;
- *upd* of type *set of* (*tuple handle*, *column name*, *tuple value*);
- R, R' of type *rule*;
- *cond-holds* of type *boolean*;
- h, h' of type *tuple handle*;
- c of type *column name*;
- v of type *tuple value*.

With each rule R we associate an attribute *R.trans-info*. This attribute is a triple [*ins,del,upd*] where *ins*, *del*, and *upd* have types as above and are used with the following meaning:

- *ins* contains handles for inserted tuples (values may be obtained from the database);
- *del* contains values for deleted tuples;
- *upd* contains handles and columns for updated tuples along with relevant old values (new values may be obtained from the database).

We assume the existence of the following functions:

- *current-state*() takes no arguments and returns the current state of the database;
- *execute-external-block*() takes no arguments, executes the next externally-generated operation block, and returns the effect of the created transition;
- *rules*() takes no arguments and returns the set of currently defined production rules;
- *rollback-action*(R) takes a rule R, returns *true* if R's action is **rollback**, and returns *false* otherwise;

[8] Other semantics are possible here. For example, a rule could be evaluated with respect to the transition since the most recent point at which it was chosen for consideration, regardless of whether its action was executed. Or, a rule could be evaluated with respect to the transition since the state preceding the most recent triggering of the rule, as specified in our initial proposal [WF89b]. For each interpretation, it is certainly possible to contrive examples for which the semantics is inappropriate. As an extension, we might permit a choice of interpretations to be specified as part of rule definition.

- *execute-rule-block*(R) takes a rule R with a non-**rollback** action, executes R's action with respect to the current database state and transition tables derived from R.*trans-info*, and returns the effect of the created transition;
- *select-triggered-rule*(ρ) takes a set ρ of rules and returns some member R of the set such that R's transition predicate is satisfied by R.*trans-info* (if no such rule exists, **null** is returned);
- *check-condition*(R) takes a rule R, returns *true* if R's condition holds with respect to the current database state and transition tables derived from R.*trans-info*, and returns *false* otherwise;
- *get-value*$(h,$ *old-state*$)$ takes a tuple handle h and a database state *old-state* and returns the value of the tuple in *old-state* identified by h;
- *get-values* is the same as *get-value* except its first argument is a set of tuple handles and a set of values is returned;
- *get-old-value*$(h,$ *old-state*, *upd*$)$ takes a tuple handle h, a database state *old-state*, and a set *upd* of handle-column-value triples. If h does not appear in *upd*, then the function returns the value of the tuple in *old-state* identified by h. If h does appear in *upd*, then the function returns the v component of any (h,c,v) in *upd* (all (h,c,v)'s in *upd* will have the same v).

4.4 Rule Selection

We have left unspecified the method for choosing a rule for consideration when more than one rule is triggered. (This is similar to *conflict resolution* in OPS, which has received considerable attention [BFKM85].) A number of strategies are possible. Rules could be chosen arbitrarily, but such purely non-deterministic behavior is probably undesirable—in many cases it is useful or even necessary to have some degree of control over rule selection. The rules could be totally ordered, in which case the triggered rule highest in the ordering is chosen. We want the flexibility of allowing rules to be defined independently, however, in which case a total ordering may not be possible. A compromise is to partially order the rules, in which case a rule is chosen such that no other triggered rule is strictly higher in the ordering. For this it is necessary only to add a syntactic construct such as:

create rule priority *rule-name*$_1$ **before** *rule-name*$_2$

which indicates that rule *rule-name*$_1$ has higher priority than rule *rule-name*$_2$. Any acyclic group of such pairings induces a partial order on the set of defined rules. In some cases we may also want to take into account the time at which rules were last considered (or executed), i.e., preferring those rules considered least recently or those considered most recently. For a thorough comparison and evaluation of rule selection strategies we must consider a number of large-scale examples.

```
repeat forever:
  /* execute ext. block, initialize trans. info. */
  old-state ← current-state();
  E ← execute-external-block();
  [ins, del, upd] ← init-trans-info(E, old-state);
  for each R ∈ rules() do
    R.trans-info ← [ins, del, upd];
  /* begin rule execution */
  R ← select-eligible-rule();
  while R ≠ null do
    if rollback-action(R) then rollback;
    /* execute rule block */
    old-state ← current-state();
    E ← execute-rule-block(R);
    /* rule R gets new transition info. */
    R.trans-info ← init-trans-info(E, old-state);
    /* modify transition info. for all other rules */
    for each R' ∈ rules() do if R' ≠ R then
      R'.trans-info ←
        modify-trans-info(R'.trans-info, E, old-state);
    R ← select-eligible-rule()
  end while

function select-eligible-rule():
  repeat
    R ← select-triggered-rule(rules());
    if R ≠ null then
      cond-holds ← check-condition(R)
    else cond-holds ← false
  until cond-holds or R = null;
  return (R)

function init-trans-info(E, old-state):
  ins ← I(E);
  del ← get-values(D(E), old-state);
  upd ← ∅;
  for each (h, c) ∈ U(E) do
    upd ← upd ∪ { (h, c, get-value(h, old-state)) };
  return ( [ins, del, upd] )

function modify-trans-info([ins, del, upd], E, old-state):
  ins ← ins ∪ I(E);
  for each h ∈ D(E) do
    if h ∈ ins then ins ← ins − {h}
    else begin
      del ← del ∪ { get-old-value(h, old-state, upd) };
      for each (h', c, v) ∈ upd do
        if h' = h then upd ← upd − {(h', c, v)}
    end;
  for each (h, c) ∈ U(E) do
    if h ∉ ins and (h, c, v) ∉ upd for any v then
      upd ← upd ∪
        { (h, c, get-old-value(h, old-state, upd)) };
  return ( [ins,del,upd] )
```

Figure 1: Rule Execution Algorithm

4.5 Examples

Additional examples are now given to illustrate the semantics of rule execution. We continue with the two-table database schema defined in Section 3.1:

emp(name, emp_no, salary, dept_no)
dept(dept_no, mgr_no)

Example 4.1 Our first example is a variation on Example 3.1: Whenever managers are deleted, all employees in the departments managed by the deleted employees are also deleted, along with the departments themselves. We assume a hierarchical structure of employees and departments. We also assume that employee numbers are not immediately reused—that is, a single externally-generated operation block will not delete an employee and then insert a new employee with the same employee number.

```
when deleted from emp
then delete from emp
       where dept_no in
         (select dept_no from dept
          where mgr_no in
            (select emp_no from deleted emp));
       delete from dept
       where mgr_no in
          (select emp_no from deleted emp)
```

The self-triggering property of this rule under our semantics correctly reflects its recursive nature. Suppose that an externally-generated operation block deletes one or more employees. Then the corresponding transition has an effect in which set D identifies tuples in **emp**, so the rule is triggered. Since the rule has no condition part, the rule's action is executed, deleting those employees and departments whose managers were deleted by the externally-generated deletion. If this new transition deletes one or more employees, then the rule is again triggered and its condition holds, so those employees and departments whose manager's manager was part of the original deletion are deleted. This behavior continues until a transition occurs with an effect in which set D identifies no tuples in **emp**—until execution of the rule's action deletes no further employees.

Example 4.2 As a second example, consider a rule that controls salary updates: Whenever salaries are updated, check the average of the updated salaries. If it exceeds 50K, then delete all employees whose salary was updated and now exceeds 80K.

```
when updated emp.salary
if (select avg(salary)
   from new updated emp.salary) > 50K
then delete from emp
       where emp_no in
             (select emp_no from
              new updated emp.salary)
       and salary > 80K
```

Suppose that an externally-generated operation block updates Bill's salary from 25K to 30K and updates Mary's salary from 70K to 85K. The U component of the corresponding transition effect identifies column **salary** of two tuples in table **emp**, so the rule is triggered. The condition holds since the average of the updated salaries exceeds 50K. Hence, the action is executed and employee Mary is deleted.

Example 4.3 As a final example, consider system execution when both rules from Examples 4.1 and 4.2 above are defined at the same time. Let $R1$ denote the rule from Example 4.1 and let $R2$ denote the rule from Example 4.2. Suppose that an initial state of the database includes six employees, Jane, Mary, Jim, Bill, Sam, and Sue, with the following management structure:

- Jane manages Mary and Jim;
- Mary manages Bill;
- Jim manages Sam and Sue.

Now suppose that an externally-generated operation block deletes employee Jane; the same operation block also updates salaries such that the average updated salary exceeds 50K and Mary's salary is updated to exceed 80K. Both rules $R1$ and $R2$ are triggered by this transition. Note that rule $R1$ is triggered with respect to set {Jane} of deleted employees. Let the rules be ordered so that rule $R2$ has priority over rule $R1$. Rule $R2$ executes its action, deleting employee Mary; $R2$ is not triggered again. At this point, rule $R1$ is considered with respect to the composite change since the initial state, thus the set of deleted employees is now {Jane, Mary}. Rule $R1$ executes its action, deleting all employees and departments still in the database whose manager is either Jane or Mary. Employees Bill and Jim are deleted by this transition, and rule $R2$ is triggered a second time. Now the rule is considered only relative to the effect of the most recent transition, so the set of deleted employees is {Bill, Jim}. $R2$'s action deletes all employees and departments managed by either Bill or Jim—employees Sam and Sue are deleted. Finally, $R3$ executes a third time relative to set {Sam, Sue} of deleted employees, but no additional employees are deleted.

5 Extensions

We have proposed a syntax for production rule definition and a semantics for production rule execution that takes into account externally-generated operations, self-triggering rules, and simultaneous triggering of multiple rules. For clarity and simplicity, in many cases we have omitted certain features that we may choose to include in our first or in a future implementation. Here we outline several of these potential extensions.

5.1 Data Retrieval

For simplicity, we have considered **select** as an embedded operation only. Intuitively, it seems that most production rules would be triggered by changes to the database and would perform changes, such as rules used to enforce integrity constraints. However, we may want the action part of a rule to include data retrieval; for example, we might want to define a rule that automatically delivers a summary of employee data whenever salaries are updated. In some cases, we may also want to define rules that are triggered by data retrieval; this is a necessary feature, for example, if rules are to be used for authorization checking. Note that authorization checking also requires a facility for considering rules before (rather than after) database operations.[9] Both features—triggering by **select** operations and data retrieval in rules' actions—can be added by modifying the definition of an operation block:

op-block	::=	*sql-op* ; *sql-op* ; ... ; *sql-op*
sql-op	::=	*update-op* | *delete-op*
	|	*insert-op* | *select-op*

Transition predicates of the form "**selected** *table.column*" and "**selected** *table*" must also be added. Transition effects are then extended to include an S component containing handle-column pairs for those tuples and columns that have been selected during the transition. There are a few issues left to be addressed, such as whether tuples from embedded **select** operations are included in transition effects, and how to incorporate selected tuples in transition effect composition. In general, however, it should not be difficult to integrate data retrieval into our framework.

5.2 More General Actions

We have limited the action parts of our production rules to sequences of SQL data manipulation operations. Just as database applications typically embed data manipulation operations within programs in a host language, it might be useful to allow the same flexibility in production rules. This can be done by permitting the action part of a rule to call an arbitrary external procedure. Adding such a feature need not change the semantics of rule execution, since the effect on the database of executing an external procedure still corresponds to a sequence of data manipulation operations. However, rule storage, action execution, and other aspects of the implementation could become significantly more complex. Also, note that if we allow more flexibility in the action part of production rules, we would also want to extend our semantics to accommodate errors or exception conditions that may arise during rule execution.

5.3 Transitions and Transactions

We have specified that rules are initially triggered by an externally-generated operation block which corresponds to a single externally-defined transaction. Subsequently, rules are also triggered by rule-generated operation blocks. As a single transaction, the system executes the externally-generated operation block followed by all resulting rule-generated operation blocks. Although this is a reasonable choice in many situations, we might want additional flexibility in the time at which rules are triggered and in the correspondence between rules and transactions. (See [CJL91,MD89] for further discussion of these issues.) For example, we might want the ability to specify that a rule's action should be executed in a separate transaction. In some cases, it might be advantageous to execute several externally-generated transactions before considering triggered rules, or, conversely, we might prefer to consider rules earlier than the commit point of an externally-generated transaction.[10] One way to achieve this is to allow user-defined "rule triggering points": When a rule triggering point is reached, the externally-generated transition is considered complete, rules are processed, and a new transition begins. In this case, the correspondence between transitions and transactions must be carefully defined.

6 Future Work

The production rules facility proposed here is very expressive, but it is also quite low-level—viewed as a programming language, it is highly unstructured. Rather than limiting expressiveness, we want to provide tools that support the database production rules programmer. In particular, the programmer might benefit from knowing that a set of rules may create an infinite loop, or from knowing that ordering between certain rules may affect the final database state. We plan to explore static rule analysis techniques; these techniques can then form the basis of a facility that issues warnings of potential loops and conflicts as rules are defined.

In addition, we want to provide higher-level facilities for restricted applications of our rules system. For example, database integrity constraints can automatically be maintained by production rules. We have designed a facility whereby the user defines integrity constraints in a high-level non-procedural language. The system then performs semi-automatic translation of these constraints into sets of lower-level production rules that maintain the constraints. This work is described in [CW90].

Finally, we are currently implementing a set-oriented production rules facility following the syntax and semantics proposed in this paper. The facility is being built as an attachment to the Starburst extensible database system [HFLP89,LMP87] at the IBM Almaden Research Center. Design and implementation issues will be discussed in a future report.

Acknowledgements

Stefano Ceri inspired a number of improvements and extensions to our original proposal [WF89b]. Bruce Lindsay suggested several useful modifications to the semantics and is collaborating with us on system design. Hamid Pirahesh and Manolis Tsangaris pursued initial work in this area [FPT88].

[9] This is also necessary for implementing procedures and views using rules, as in [SJGP90].

[10] The latter case is necessary to simulate integrity constraint enforcement in existing systems [IBM88].

References

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

[CJL91] M.J. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3), September 1991.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[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.

[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.

[FPT88] S.J. Finkelstein, H. Pirahesh, and M. Tsangaris. Rule support for Starburst. Unpublished notes, 1988.

[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.

[HFLP89] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–388, Portland, Oregon, May 1989.

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

[KDM88] 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.

[LMP87] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 220–226, San Francisco, California, May 1987.

[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.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the Ninth International Conference on Very Large Data Bases*, pages 34–42, Florence, Italy, October 1983.

[RS89] L. Raschid and S.Y.W. Su. A transaction oriented mechanism to control processing in a knowledge base management system. In L. Kerschberg, editor, *Expert Database Systems—Proceedings from the Second International Conference*, pages 353–373. Benjamin/Cummings, Redwood City, California, 1989.

[SdM88] E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In *Proceedings of the Second International Conference on Database Theory*, Bruges, Belgium, September 1988.

[SHP88] M. Stonebraker, E.N. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[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.

[Tzv88] A. Tzvieli. On the coupling of a production system shell and a DBMS. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 291–309, Jerusalem, Israel, June 1988.

[WF89a] 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.

[WF89b] J. Widom and S.J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems (extended abstract).

SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems, 18(3):36–45, September 1989.