Integrating Triggers and Declarative Constraints in SQL Database Systems

Roberta Cochrane Hamid Pirahesh Nelson Mattos

IBM Almaden Research Center, San Jose, CA

{bobbiec, pirahesh}@almaden.ibm.com, mattos@vnet.ibm.com

Abstract

This paper describes a model that integrates the execution of triggers with the evaluation of declarative constraints in SQL database systems. This model achieves full compatibility with the 1992 international standard for SQL (SQL92). It preserves the set semantics for declarative constraint evaluation while allowing the execution of powerful procedural triggers. It was implemented in DB2 for common servers and was recently accepted as the model for the emerging SQL standard (SQL3).

1 Introduction

Active databases are taking a prominent role in commercial database applications [6, 30, 29, 13]. With client/server solutions, applications are being developed by small, autonomous groups of developers with narrow views of the overall enterprise; the enterprise information system is very vulnerable to integrity violations because it lacks strict enforcement of the enterprise business rules. *Active data* proactively monitors events and, without user intervention, protects its own integrity or invokes actions either within or external to the database. Active data features can be used to bind frequently used application logic to data for invocation directly within the server, decreasing client/server communication.

As the SQL standard progresses to its next release (SQL3)[21], it is under pressure to include a well-defined model for active data. Prior research [3, 11, 34] promotes event condition action (ECA) rules as the uniform mechanism to support constraints, authorization and application logic. The only prior work that considers the simultaneous existence of separate specifications and models for the execution of ECA rules and declarative constraints is [16]. However, most commercial DBMSs support a rich set of declarative constraint constructs compliant with SQL92, and commercial applications rely on the fact that this specification will not change. Hence, the active database model that will be used in practice must integrate procedural triggers (one form of ECA rules) with the existing model for declarative constraints defined by the SQL international standard (SQL92) [20, 27, 7].

There are many practical reasons why, when applicable, declarative constraints should be used in lieu of triggers. However, declarative constraints alone are not powerful enough to support all the active database requirements of applications. Most popular active database models developed thus far bear some resemblance to either OPS5 [2] style production rule systems or deductive logic. Unfortunately, the triggering mechanisms emerging in the commercial DBMSs [24, 12, 35, 17] do not follow this approach. Instead, they are procedural in nature, and lack many reasoning advantages inherent in such rule-based systems. However, this procedural model is a natural mapping of existing commercial application logic.

This paper describes the first integrated model for reconciling the execution of triggers with the evaluation of SQL92 declarative constraints. It resolves the problems with the co-existence of triggers and declarative constraints introduced in [16], and it is the only model that (a) allows triggers to coexist with deferred constraints and cascading referential constraints, (b) incorporates before-triggers, after-triggers, row-level triggers, and statement-level triggers in a uniform framework that is integrated with the set semantics for modifications required by SQL92 (not previously understood), and (c) defines scoping rules for transition variables and tables in the context of the aforementioned types of triggers.

SQL3 [21] includes advanced database features and

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

has started the final publication process. Until now, SQL3 has received little input from the active database community. Yet there is increasing interest from both commercial SQL vendors and customers for a well-defined specification of SQL triggers. Our model was recently accepted as the model for SQL3 triggers [5], and we hope that this paper will stimulate further contribution from the research community.

The structure of the paper is as follows. Section 2 discusses the relative advantages of declarative constraints and triggers. Section 3 presents declarative constraints and their fixpoint evaluation model as defined in SQL92. Section 4 proposes our model for SQL3 triggers and defines their execution in the absence of declarative constraints. Our model for integrating the evaluation of declarative constraints and the execution of triggers in SQL is presented 3 sections: Section 5 formally describes the integrated execution model, demonstrated by an example in Section 6, and discussed further in Section 7 which highlights the contributions of our work with respect to other products and prototypes.

2 Constraints or Triggers?

An active database system (DBMS) should support constraints as well as event-driven application logic. However, declarative constraints should be used in lieu of triggers whenever possible. First, several triggers are required to enforce one declarative constraint; even then, the system has no way of guaranteeing the validity of the constraint in all cases. Consider the database load utilities in which the database checks the declarative constraints against the loaded data before it can be accessed. There is no way to determine which triggers should be checked since triggers are also used for transitional constraints and for event-driven application logic. This behavior also applies when constraints and triggers are added to a database with pre-existing data.

The database engine can use knowledge of declarative specifications to optimally evaluate constraints. For example, concurrency control hot spots can be avoided during referential integrity enforcement by reducing the isolation level to cursor stability [28]. Guaranteeing the presumed semantic intent of triggers, in general, requires an isolation level of repeatable read [10]. Furthermore, semantic query optimization is not possible if the declarative semantics are hidden in triggers. Tool vendors may also take advantage of declarative constraints for pre-checking entries at the client (e.g. a mobile laptop) before sending data to the server.

The declarative constructs provided by most systems and defined in SQL92 only support a small, albeit useful, set of static constraints that define the acceptable states of the value of the database, e.g. $\texttt{salary} > \texttt{commission} + \texttt{hourlyWage} * \texttt{hoursWorked}$. They do not support *transitional constraints* that restrict the way in which the database value can transition from one state to the next e.g. `salary increases must be less than 10%`. They also do not support event-driven invocation of application and business logic. Hence, triggers are required to enhance the declarative constraint constructs and to capture application specific business rules. Triggers provide a procedural means for defining implicit activity during database modifications. They are used to support event-driven invocation of application logic, which can be tightly integrated with a modification and executed in the database engine by specifying a trigger on the base table of the modification. Triggers should not be used as a replacement for declarative constraints. However, they extend the constraint logic with transitional constraints, data conditioning capabilities, exception handling, and user-defined repairing actions.

In summary, there are advantages to using both declarative constraints and procedural triggers, and both types of constructs are available in many commercial systems. It is not feasible to expect applications providers to either migrate their existing applications to use only triggers or partition the tables in their database according to the type of constraints and triggers that are required. It is therefore imperative to define and understand the interaction of constraints and triggers.

3 Declarative Constraints in SQL92

Declarative constraints in SQL92 include two forms of static constraints: check constraints and referential constraints.[1] SQL92 also supports the definition of symmetric views (sometimes referred to as the *with check option*), which are transitional constraints applied to updates and inserts through a view.

Although declaratively specified, each constraint implies a set of *events* for which the constraint is checked and an *action* to be taken if the constraint is not satisfied (or, more accurately, evaluates to false). When a static constraint is defined, the system verifies it against the existing data, and once defined, guarantees that the constraint is always satisfied. After load utilities are used, the system verifies all declarative static constraints before the loaded data can be accessed.

A *check constraint*, typically used to validate input data, is a condition that is true for every row in a given table. In general, a check constraint can be

[1] In SQL92, primary keys, unique keys, and NOT NULL are viewed as special cases of check constraints.

any SQL condition, including multi-table assertions. Since such assertions are extremely expensive to support, this feature is practically restricted to the special case key constraints and one variable aggregate-free check constraints, i.e., value constraints, in most existing products. These constraints allow the definition of uniqueness constraints, value restrictions and intra-row value checks on a single table. In object-oriented extensions of SQL, check constraints can contain functions applied to the values as long as the functions are deterministic and do not have side-effects. Value constraints need only be evaluated when the constraint is defined, after special load utilities are run, or when any table on which the constraint is defined is modified by an UPDATE or INSERT statement. In addition, multi-table assertions may also need to be checked when any table referenced in the condition is modified. If the constraint evaluates to false for any updated or inserted rows (or, for a multi-table assertion, any row in any referenced table), then the statement that caused the modification[2] is rejected and any changes made by the statement are undone.

A *referential integrity (RI) constraint* establishes a relationship between a set of columns designated as a *foreign key* and a unique key such that the non-null values of the foreign key must also appear as values in the unique key. The foreign key's table is the constraint's *child table* and the unique key's table is the constraint's *parent table*. Note that the parent table and child table can be the same, referred to as a *self-referencing* RI constraint. Like check constraints, RI constraints must also be checked when the constraint is defined and after special load utilities are run. However, since there are two tables involved in the constraint, there are four modifications that cause constraint evaluation: (a) insertions into the child table, (b) updates a foreign key column, (c) deletions from the parent table, and (d) updates of a unique key column. Whenever modifications to the child table violate the constraint, the statement that caused the modification is rejected and any changes made by the statement are undone. However, the action to be taken when the parent is modified by a DELETE or UPDATE statement is one of a set of pre-defined actions that is specified with a delete rule or update rule, respectively, when the constraint is defined. This pre-defined set includes: NO ACTION and RESTRICT, which reject the violating statement[3]; SET NULL, which sets the nullable columns of the foreign key of any child rows that match the modified parent rows to null; SET DEFAULT, which sets the foreign key columns for any child rows that match the modified parent to their default values; and CASCADE, which deletes any matching child rows.

A *symmetric view* is defined by specifying the WITH CHECK OPTION clause during the creation of the view. It defines a transitional constraint that is evaluated whenever the view is modified by an INSERT or UPDATE statement. The most general form of this clause indicates that any row modified through an UPDATE or INSERT statement on a symmetric view must remain in the view after the modification. If the modification causes the row to disappear from the view, then the statement causing the modification is rejected and any changes made by the statement are undone.

Constraint Evaluation In the Absence of Triggers

The evaluation of declarative constraints, well-defined by SQL92, is complex and if done incorrectly can lead to anomalous non-deterministic behavior [26]. Difficulties arise when an SQL statement modifies several rows or causes the evaluation of multiple RI constraints. The result of the statement must not depend on the order in which the rows are modified or the order in which the constraints are applied. Hence, several existing products restrict the combinations of constraints that can be defined and the classes of updates and deletions allowed on unique keys [19][4]. According to SQL92, the constraints must effectively be processed only after all modifications of the original statement are applied. However, there is a common optimization employed that evaluates constraints *in-flight* as the rows are modified. This optimization must only be used when the success or failure of the statement is not dependent on the order in which the rows or the constraints are processed.

The interaction of simultaneously enforcing check constraints, RI constraints, and symmetric views when the update or delete rule of one of the RI constraints is either SET NULL or CASCADE is another potential source of non-determinism. This ambiguity was removed by a run-time marking algorithm defined in [15] which defines a fixpoint computation for enforcement of the declarative constraints. With a few subtle exceptions, this run-time algorithm results in the following evaluation order for constraints:

1. Evaluate the original statement's modifications.
2. Evaluate all constraints with RESTRICT semantics. If any are violated, return an error after undoing any changes made by the original statement.
3. Perform all cascaded modifications, including SET NULL. As modifications are cascaded, any fur-

[2] We refer to UPDATE, DELETE and INSERT statements collectively as modifications.

[3] NO ACTION and RESTRICT have very subtle differences which are not illuminated here as these distinctions do not affect the model presented in this paper.

[4] DB2 for Common Servers Version 2 has removed these restrictions.

ther activated RESTRICT constraints must also be checked.

4. Perform all cascaded deletes. As deletes are cascaded, any further activated RESTRICT and cascaded modifications must be checked.

5. Evaluate all constraints with NO ACTION semantics, including check constraints, symmetric views, unique keys and not-null constraints. If any of these constraints are violated, raise an error and undo any changes made by the original statement and any resulting cascaded actions.

Thus, SQL92 has a very explicit specification for deterministic evaluation of constraints with respect to database modifications. This specification defines a fixpoint evaluation that is independent of the order in which rows or constraints are processed. Not only has this evaluation model been published as the SQL standard for over four years, but more than a dozen SQL database vendors have a conforming implementation of SQL92 constraints.

4 A Proposal for SQL Triggers

In contrast to declarative constraints, triggers are procedural. The *event* governing the execution of a trigger is explicitly specified in its definition. This triggering event is an UPDATE, DELETE or INSERT statement applied to a base table. An optional column list can be specified in the case of updates to further restrict the set of update events that activate the trigger.

The trigger also has an *activation time* that specifies if the trigger is executed before or after its event and a *granularity* that defines how many times the trigger is executed for the event. *Before-triggers* execute before their event and are extremely useful for conditioning of the input data before modifications are applied to the database and the relevant constraints evaluated. *After-triggers* execute after their event and are typically used to embed application logic, which typically runs after the modification completes.

The granularity of a trigger can be specified as either FOR EACH ROW or FOR EACH STATEMENT, referred to as *row-level* and *statement-level* triggers respectively. When the event of a row-level trigger occurs, the trigger is executed once for each row affected by the event. If no rows are affected, then the trigger is never evaluated. However, a statement-level trigger is executed exactly once whenever its event occurs, even if the event does not modify any rows. Like to constraints, both row-level and statement-level after-triggers must (appear to) execute only after the triggering event finishes executing; before-triggers must appear to execute entirely before the triggering event's modifications are applied. Note that the granularity does not dictate when the trigger executes. Obviously, there are many cases when row-level triggers can be safely and optimally processed in-flight. However, if they or any constraints that must be evaluated require access to either the base table of the triggering event or any table that is modified by the trigger or the constraints, then row-level triggers can only be processed in a set-oriented fashion. E.g., if a row-level update trigger on table `T` accesses the average of a column in `T`, then the average must not be computed in the middle of the triggering update; it must either be computed entirely before or after the application of any modifications to `T`.

Each trigger has access to the before-transition values and after-transition values of the event through the declaration of *transition variables* and *transition tables*. The declaration "`referencing` NEW `as N`" defines `N` as a single row correlation variable that contains the value of a row in the database immediately after the modification is applied. Similarly, "`referencing` OLD `as O`" declares `O` as a single row correlation variable containing the value of the same row in the database before the modification is applied. If both OLD and NEW transition variables are declared as above, then the new and old values of a given row can be compared. For example, if the trigger's table is EMP, and EMP has column `salary`, then we can check if the modification was a raise by comparing `O.salary` with `N.salary`. Transition tables are similarly declared using keywords NEW_TABLE and OLD_TABLE. If `NT` is declared as a new transition table, it is a virtual table containing the values of all modified rows immediately after the modification is applied. Similarly, if `OT` is declared as an old transition table, it is a virtual table containing the values of all modified rows before the modification is applied. The contents of transition tables is operationally defined in Section 5,

Not all types of transition variables and transition table references are valid for each type of trigger. In particular, triggers defined on insert events can only see `new` values and triggers defined on delete events can only see `old` values while triggers defined on update events can see both `old` and `new` values Furthermore, transition variables are only accessible at the granularity of one row, and hence, can only be referenced by row-level triggers. As will be described in Section 5, both statement-level and row-level before-triggers participate in a fixpoint computation of the transition tables; during the computation of this fixpoint, inherently, the entire content of the transition tables is not yet computed. Hence, before-triggers cannot reference transition tables. However, both row-level and statement-level after-triggers can reference transition tables. Such references allow row-level after-triggers to compare a single transition row value with aggregations on the transition tables, e.g., how the `new` salary in a given row compares with total impact of the salary increase.

Each trigger has an *action* that is optionally guarded by a *condition*; the action is only executed when the triggering event occurs and, if specified, the condition is satisfied. The condition is a single, unrestricted search condition and the action is a procedure containing a sequence of SQL statements. Both the action and the condition can query the transition variables and tables as well as the current value of the database, and these queries can invoke user-defined functions. Since before-triggers appear to execute entirely before the event occurs, any queries of the database in their conditions or actions must appear to read the database state prior to any modifications made by the event. Similarly, the conditions and actions of all after-triggers must appear to read the database state after all of the event's modifications are applied. If the after-trigger needs to query the state of the database prior to the event, then it can reconstruct it using the transition tables. For example, if `T` is modified by the triggering event, `NT` is the `new` transition table and `OT` is the `old` transition table, then the state of `T` prior to the event can be approximated by $(\mathtt{T} \setminus \mathtt{NT}) \cup \mathtt{OT}$.

The trigger action is an atomic procedure, e.g., SQL Procedural Stored Module (PSM), that may contain SQL statements combined with other procedural constructs. The SQL statements are executed in the order in which they occur in the evaluation of the procedure. If any one of the SQL statements fail, the entire trigger action is rolled-back, including the triggering event. After-triggers, used primarily to embed application logic in the database, can contain any data manipulation SQL, including SELECT, UPDATE, INSERT, and DELETE statements. Before-triggers can contain condition data using an assignment statement that allows the trigger body to set the values of transition variables declared as NEW. Note that this statement is not allowed in after-triggers since the modification is already applied to the database, and the NEW transition variable is no longer maleable. Conversely, before-triggers cannot modify the database using UPDATE, DELETE, or INSERT statements since this leads to a nested model of unapplied modifications.

Triggers can indirectly cause a statement-level rollback by raising an error, e.g., using the SQL signal statement. When this occurs, the containing SQL statement fails and, as described above, the entire trigger action is rolled-back, including the triggering event.

Several triggers can have the same event and activation time. Hence, multiple triggers can be simultaneously eligible for execution. When this occurs, the eligible triggers must be executed according to some discernible total order global to the entire set of triggers. Our execution model assumes the existence of some global ordering of the entire set of triggers, à la Postgres [33] or Starburst [1].

Trigger Execution in the Absence of Constraints

The execution of an after-trigger may activate other triggers, recursively activate itself, or activate the evaluation of constraints. Triggers that activate other triggers are executed in a nested procedural fashion. If a modification statement `S` in trigger action `A` causes an event `E`, and there are no constraints defined in the system, `S` is processed as follows.

1. Execution of `A` is temporarily suspended, saving all local state on a stack. `New` and `old` transition tables for `S` are computed.
2. All before-triggers activated by `E` are executed in sequence according to their creation order. If a before-trigger contains an assignment statement, then the appropriate fields in the `new` transition table are modified for the row associated with the `new` transition variable.
3. The current values in the `new` transition table are applied to the database.
4. All after-triggers activated by `E` are executed in sequence according to their creation order. If any such execution causes another event, then the statement causing the event is processed recursively starting in step 1.
5. `A`'s state is recovered from the stack and processing resumes with the statement that follows `S` in `A`.

Recalling that an event is the entire SQL statement, the event-to-condition binding is deferred (with the exception of the first trigger executed), and condition-to-action binding is immediate. The difference is that in these prototype systems, the event is a modification to a single row and in SQL systems the event is an entire SQL statement that may involve modifications to several rows.

In the above model, there is at most one SQL statement being applied to the database at any given point. That is, there is only one set of transitions being computed for a given event. Recall that we do not allow before-triggers to modify the database. If we did, many statements could be in progress simultaneously. Because before-triggers execute prior to the application of their triggering modification, allowing modifications causes an endless chain of unapplied modifications to build up. These modifications are not visible to the nested invocations of other triggers and result in an ambiguity as to what persists in the database when the before-triggers and any cascaded actions due to such statements complete.

5 An Integrated Model

This section defines a model that integrates the execution of before-triggers and after-triggers defined in Section 4 with the fix-point evaluation of SQL92 constraints summarized in Section 3. This model is incorporated in the IBM DB2 for Common Servers Version 2 product and was recently proposed and accepted as the model for trigger execution in SQL3.

One of the motivating principles behind the design of our execution model is to minimize the window in which the declarative constraints of the database are not guaranteed. This is particularly important for application logic that is embedded in after-triggers. Application logic is more easily written and debugged if one can assume that the declarative constraints are enforced. Furthermore, many semantic optimization techniques can only be applied if the declarative constraints are enforced. Otherwise, the resulting plans may yield erroneous results. Our model guarantees that any statement issued externally by an application or internally by an after-trigger will only read database states that are consistent with respect to the declarative constraints.

Figure 1 depicts the execution of an SQL modification statement `S`. During the processing of `S`, data is conceptually stored in two different repositories: the database and the working storage. The database represents the persistent repository of the enterprise data that is shared among applications and whose visibility is protected by the authorization and locking mechanisms of the DBMS. On the other hand, the *working storage* is local to and contains transitions computed during the execution of `S`. A transition associates the before-value and after-value of a single row resulting from an event. If θ is a transition representing a modification of a row for some event `E`, then θ.`new` (applicable for insert and update events) is the value of the row to be applied to the database and θ.`old` (applicable for delete and update events) is the value of the row before `E` is executed. Transitions are grouped into sets $\triangle_1, \ldots, \triangle_n$ according to their event type, which is update, delete or insert [5] qualified by a single table.

A trigger is activated by a transition set if the event type of the trigger is the same as the event type of the transition set. When a row-level trigger is executed for a transition, the value of the `new` and `old` transition variables are derived directly from the `new` and `old` values of the transition. When either a row-level trigger or statement-level trigger is executed, the value of the `new` and `old` transition tables are computed from the trigger's activating $\triangle$. The `new` transition table is a table containing one row for each transition in $\triangle$ whose value is the `new` value of the transition. Similarly, the `old` transition table is computed from the `old` values of the transitions.

An **U**pdate, **D**elete, or **I**nsert statement `S` is executed by the procedure **Execute_UDI** as depicted in Figure 1 and described below.

Compute $\triangle_1$ for `S`. The execution of statement `S` begins with the initialization of $\triangle_1$, which represents transitions made by `S` or by any resulting cascaded actions with the same modification type and table as `S`'s event. The values of $\triangle_1$ are determined based on the current values of the database and the semantics of `S`. If `S` is a DELETE(T), then $\triangle_1$ is initialized as the set of all transitions for all rows that satisfy the search condition of `S`. If θ is a transition in $\triangle_1$, θ.`old` is the value of the row that is to be deleted, and θ.`new` is not applicable. If `S` is INSERT(T), then $\triangle_1$ is initialized as the set of transitions for all rows identified by the VALUES clause or query expression of `S`. If θ is a transition in $\triangle_1$, θ.`new` contains the new values to be inserted into the database and θ.`old` is not applicable. If `S` is UPDATE(T), then $\triangle_1$ is initialized as the set of transitions for all rows that satisfy the search condition of `S`. If θ is a transition in $\triangle_1$, θ.`old` is the value of the row before the update and θ.`new` is the value of the row after the update.

Execute before-triggers for `S`. Let $B_1, B_2, \ldots, B_p$ be all of the before-triggers activated by $\triangle_1$'s event in increasing order according to the global trigger ordering. B_1 is considered first. If B_1 is a row-level trigger, it is executed once for each row in $\triangle_1$, which implies that it is not executed if $\triangle_1$ is empty. If B_1 is a statement-level trigger, it is executed exactly once. Note that a statement-level trigger is executed even if $\triangle_1$ is empty, because the triggering event, `S`, occurred. All statements in a row-level trigger are executed for a given row before proceeding to the next row. The remaining before-triggers $B_2, \ldots, B_p$ are considered in turn. Before-triggers can read data from the database. A row-level before-trigger can also use an assignment statement to modify columns of a given rows `new` transition value. Because restrictions on before-triggers prevent them from modifying the database, they cannot cause cascaded execution of triggers or cascaded evaluation of constraints. However, a triggered action may explicitly request an error to be returned. There is no need to rollback any changes made by `S` since $\triangle_1$ has not yet been applied to the database.

Apply $\triangle_1$ to the database. This is the first point

[5] In the case of update, the event type is further partitioned by column sets if there are triggers whose event specifies a column list. This level of detail is omitted for the purposes of this description.

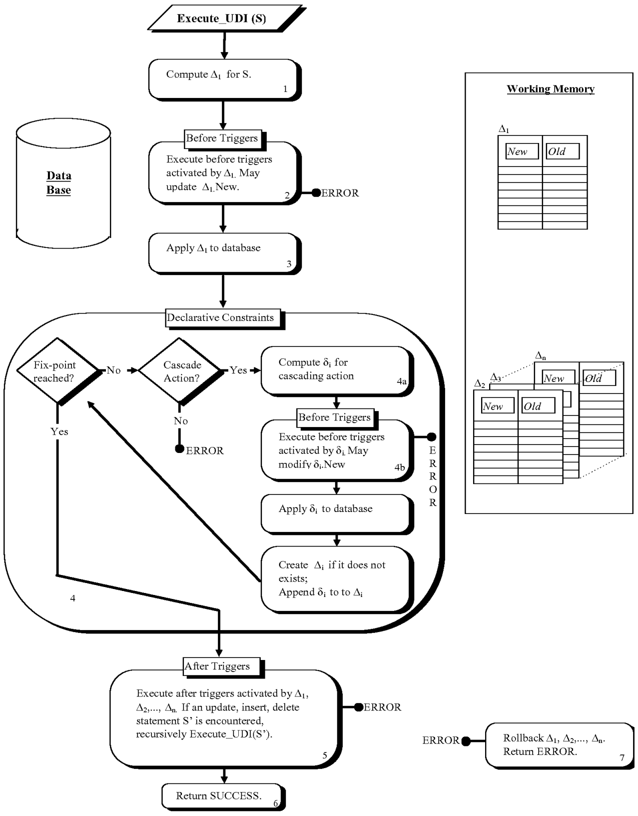

Figure 1: Execution of a Database Modification Statement

in the processing of S when S's modifications are made to the database. Note that any changes made to $\triangle_1$ by row-level before-triggers are reflected in the modifications applied to the database.

Evaluate the declarative constraints for S. If $\triangle_1$ is empty, this step does not have any effect. Otherwise, constraints (referential constraints, check constraints and symmetric views) are evaluated for all modifications made by S according to the SQL92 semantics summarized in Section 3. The figure refers to a fixpoint, which is reached only when (a) no constraints are violated and (b) statement-level before-triggers for all cascaded actions of the evaluated constraints are executed. This is symmetric with the execution of statement-level before-triggers for statements that do not actually perform any modifications. If the fixpoint is not reached because of the evaluation of some constraint C, then either C is not satisfied or C is satisfied and has a cascading action for which statement-level before triggers must be processed. So, if C does not have a cascading action, then an error must be returned. If the evaluation of C requires a cascading action S_i (CASCADE, SET NULL, SET DEFAULT), then all before-triggers activated by S_i must be processed before the S_i's modifications are applied to the database. First, transitions for S_i's modifications are computed and stored in a temporary transition set δ_i. If C is satisfied, then δ_i will be empty. But we must still execute statement-level before-triggers for S_i. We must also create $\triangle_i$ for S_i if it does not already exists to schedule the execution of the corresponding statement-level after-triggers.

Let B_{i1}, B_{i2}, …, B_{ip} be the before-triggers activated by δ_i's event in increasing order according to the global trigger ordering. B_{i1} is considered first. If it is a newly activated statement-level trigger, i.e., it was not activated previously by S or by any other cascading action, then it is executed once. If it is a row-level trigger, then it is executed once for each row in δ_i. The remaining before-triggers B_{i2}, …, B_{ip} are similarly considered in turn. Note that a statement-level before-trigger B_{stmt} is evaluated at most once for a statement and its cascading actions. If B_{stmt} is activated by S's event, then it is never executed by a cascading action of a declarative constraint for S. Otherwise, it is executed the first time it is activated by a cascaded action.

The modifications of δ_i are then applied to the database. If there does not already exist a transition table $\triangle_i$ in the working storage with the same event as δ_i, then $\triangle_i$ is created and initialized with the values of δ_i. Otherwise, the transitions in δ_i are appended to $\triangle_i$. The declarative constraint processing continues in this manner until a fixpoint is reached, i.e., all of the declarative constraints are satisfied and all statement-level before-triggers for cascaded actions required for the evaluation of a constraints are executed.

If the evaluation of a constraint causes an error to be returned, then any modifications made to the database by Execute_UDI are first undone. These modifications are represented by the transitions in $\triangle_1, \ldots, \triangle_n$ in the working storage. Processing for Execute_UDI terminates and returns an error.

Execute after-triggers for `S` and its cascading actions. Let $A_1, A_2, \ldots, A_q$ be all statement and row-level after-triggers activated by events of $\triangle_1, \ldots, \triangle_n$ in increasing order according to the global trigger ordering. Processing of the after-triggers is the same as previously described for before-triggers, with the following exceptions: (a) the set of simultaneously activated triggers contain triggers with different events (rather than a single event) since the entire contents of the working storage is considered, (b) statement-level triggers execute once for the transition set $\triangle_i$ of their respective activating event, (c) after-triggers cannot change the `new` values of $\triangle_i$, and (d) any explicit error returned by the action of a trigger must first undo changes made to the database by Execute_UDI. Because the action of an after-trigger can contain SQL statements that modify the database, such statements must be processed in a recursive manner. When a DELETE, UPDATE, or INSERT statement S' is encountered during the execution of after-trigger A_i, the current contents of the working storage is pushed on a stack clearing its current contents, and Execute_UDI is recursively invoked to process S'. After Execute_UDI finishes executing S', the saved contents of the working storage is popped from the stack. If the recursive invocation causes an error, all modifications made to the database by Execute_UDI are undone and processing terminates in error.

Return. If the above processing of the before-triggers, declarative constraints, and after-triggers completes successfully with no explicit errors during trigger execution, constraint violation errors, or errors caused from recursive invocations of Execute_UDI, then processing returns successfully. Otherwise, processing returns an error after restoring the database to its state prior to this execution of Execute_UDI. Note that, as described, when any error occurs, the database will eventually be restored to the state of the database before the execution of the outermost modification event (i.e. to the beginning of the original statement).

6 Our Model in Action

Consider an application that with the following tables, constraints, and a symmetric view:

```
DISTRIBUTORS(id,location)
AUDITPARTS(modtype, updated_by, moddate, numrecs)
PARTS(codenum, supplier, cost, super_part,
      updated_by,record_date)

MINVAL:      check (cost >= 10.0)
VALIDDIST:   PARTS(supplier) references DISTRIBUTOR(id)
             on delete set null
PARTSUPPART: PARTS(super_part) references PARTS(codenum)
             on delete cascade

NONCAPITAL view:  (select * from PARTS
                   where COST < 500) with check option
```

Check constraint MINVAL verifies that only parts with cost greater than 10 are tracked. The RI constraint VALIDDIST ensures that any non-null value of PARTS.supplier is a valid distributor. Because the delete rule is SET NULL, the deletion of a distributor will set the supplier field for any part supplied by the deleted distributor to NULL. The self-referencing RI constraint PARTSUPPART ensures that all values of super_parts are also listed as parts. The delete rule of cascade will delete a part whenever its super part is deleted. Symmetric view NONCAPITAL selects only parts that are not classified as capital equipment. Any insertion of capital equipment or update that reclassifies non-capital equipment as capital is not allowed through this view.

The following row-level before-trigger ensures that the supplier field is never updated to any value other than NULL. If this occurs the originating statement is rolled-back and an error is reported.

```
create trigger ONESUPPLIER
before update of supplier on PARTS
referencing new as N
for each row
when (N.supplier IS NOT NULL)
 signal sqlstate '70005' ('Cannot change supplier');
```

The following row-level before-trigger computes the time and user of each update.

```
create trigger USERDATE
before update on PARTS
referencing new as N
for each row
      set N.updated_by = USER,
          N.record_date = CURRENT DATE;
```

The following statement-level after-trigger uses a transition table to audit the number of parts updated. Assume there is a similar statement-level trigger RECORDDEL after deletions on PARTS.

```
create trigger RECORDUPD
after update on PARTS
referencing old_table as OT
for each statement
  insert into AUDITPARTS
     values('U', USER, CURRENT DATE,
            (select count(*) from OT));
```

When all suppliers located in California are deleted from table DISTRIBUTOR, the referential constraint VALIDDIST will be evaluated, causing the supplier column for all rows in PART whose supplier column matches the id of any of the deleted distributors to be set to NULL. This update, in turn, activates the two before-triggers ONESUPPLIER and USERDATE, which are then executed. Assume that the global order is the same as the creation time order of the triggers, so that ONESUPPLIER is executed before USERDATE First, the new value for supplier is checked. Because it is NULL, the condition of trigger ONESUPPLIER is not satisfied, so the error is not returned. Second, update values for columns updated_by and record_date are computed from the environment values for the current user and date. Once these triggers are evaluated for all of the identified rows, the resulting updates are applied to the database, which activates the statement-level trigger RECORDUPD. This trigger will insert one row into table AUDITPARTS to record number of rows in PARTS updated as a result of a statement issued by the current user.

Now, suppose that all parts with NULL suppliers are deleted. The referential constraint PARTSUPPART is evaluated and subsequently deletes any row in PART whose super_part was deleted. This process continues until there are no rows left in PART with either a NULL supplier or a super_part value that does not match some codenum value of a remaining row in PART. Once the evaluation of PARTSUPPART reaches this fixpoint, the statement-level trigger RECORDDEL is activated and inserts one record in table AUDITPARTS to record the total number of rows deleted from PARTS by the issued delete statement and the resulting cascading actions.

To demonstrate symmetric views in this scenario, consider an update to the cost column of the view NONCAPITAL. This update will first activate before-triggers. In this case, only USERDATE is activated because ONESUPPLIER is only activated on updates to the supplier column. Then, the update is applied with the value of cost specified in the original statement and the values for updated_by and record_date computed by trigger USERDATE. Subsequently, the check constraint MINVAL and the symmetric view are evaluated to verify the new cost value is between 10 and 500; a value less than 10 indicates the part is not valuable enough to record, and a value of 500 or more would cause the updated rows to be reclassified as capital equipment and disappear from the view.

7 Discussion of our Model

The main contributions of our model are as follows.

- It is integrated with the set semantics of SQL and the fixpoint semantics for the evaluation of declarative constraints, achieving full compatibility with SQL92, the standard to which commercial SQL DBMSs are committed. Assumptions made in previous models are either incompatible with SQL or compromise the generality and expressive power of SQL, which is unacceptable.
- It allows both before-triggers and after-triggers and distinguishes between them. After-triggers are used primarily as an extension to the application logic while before-triggers are primarily used to extend the constraint logic.
- It incorporates the before-triggers as data conditioners and extensions to the constraint logic into the fixpoint computation.
- It ensures that application logic embedded in after-triggers see a constraint-consistent database.
- It clearly defines the contents and scoping rules for both transition tables and transition variables.
- It recognizes that triggers with different events may be triggered simultaneously and hence, supports a global ordering of all triggers.

Prior to the acceptance of our model, SQL3 contained a specification for triggers that was incomplete, did not address any of the above issues [5], and imposed many arbitrary restrictions. For instance, it explicitly disallowed the definition of any trigger on a table that was the target of a cascading referential action.

We integrate trigger execution in the context of the set-semantics of SQL by defining the triggering event as a single SQL statement, even for row-level triggers. We also make this model upward compatible with the evaluation of constraints defined by SQL92. We recognize that the integration of triggers in the presence of this fixpoint constraint model requires certain restrictions for before-triggers that are not required for after-triggers.

An important aspect of the model described in this paper is that it distinguishes between before-triggers and after-triggers. Before-triggers participate in the fixpoint computation of declarative constraints, and consequently, the fixpoint computation of the transition tables. After-triggers execute only after declarative constraints are satisfied. By definition, a before-trigger must be executed before the activating modifications are applied to the database, but any such before-trigger executed during the fixpoint evaluation of constraints may be exposed to an inconsistent database. One way to prevent this is to simulate the evaluation of constraints, e.g., using the run-time marking algorithms defined in [15], to pre-compute the transition tables before executing any of the before-triggers or applying any of the modifications. The disadvantage of this approach is that it prevents the use of before-triggers to condition the data as the transition tables would be pre-computed before the triggers are executed. Because we view data conditioning as one of the primary advantages of before-triggers, we clearly distinguish between the uses and properties of before-triggers and after-triggers as previously described.

Like declarative constraints, the execution of before-triggers must be monotonic because they participate in the fixpoint computation of transition tables. Consequently, neither statement-level nor row-level before-triggers can use aggregate information, e.g., total number of employees deleted, computed from transition tables; this information is meaningless until the fixpoint computation is complete. Because aggregation is the main reason for including transition tables, we restrict their use entirely in before-triggers. Our model also does not allow before-triggers to modify the database because such modifications can lead to non-monotonic behavior. Furthermore, such modifications may cause the activation of further triggers which may lead to confusing behavior due to the nesting of modifications yet to be applied to the database. Clearly, these restrictions on before-triggers are conservative, and identifying the situations in before-triggers in which transition table references or modifications do not lead to non-monotonic behavior are topics for further investigation.

Our model evaluates declarative constraints before executing any after-triggers. This concurs with existing implementations of triggers in the marketplace [24, 17]. However, these implementations have fundamental limitations. Some only support statement-level after-triggers, leading to very verbose triggers for repeatedly performing operations on each row modified by the triggering event. Others intermix the execution of row-level after-triggers with the evaluation of constraints in a way that does not maintain the SQL92 constraint evaluation model and does not guarantee that application logic in an after-trigger operates on a consistent database. A previous proposal [16] provides a detailed analysis of the difficulties in specifying a model for simultaneously evaluating declarative constraints and procedural triggers in SQL. In contrast to our model, this proposal attempts to define a model in which constraints are evaluated only after all of the triggers are executed. Hence logic embedded in after-triggers is possibly exposed to inconsistent data. Unfortunately, application libraries require a consistent database and, as such, cannot be used by trigger bodies. Interestingly, such semantics may also lead to a severe performance degradation of trigger execution, and, as highlighted in [32], this is an important concern in the employment of active features. Many optimizers use uniqueness and referential constraints to reduce duplicate elimination and access to parent tables; such semantic optimization techniques can only be applied when declarative constraints are guaranteed.

Non-determinism is inherent in row-level after-triggers due to the order in which rows are processed. Consider an update of two rows r_1 and r_2 in T, and a row-level after-trigger for updates to T that inserts the value of the `new` transition variable into table S, and subsequently appends table S to table R. If r_1 is processed first, then two copies of r_1 will be appended to R and only one copy of r_2. If r_2 is processed first, then two copies of r_2 will be appended to R and only one copy of r_1. It is enlightening to notice that most row-level after-triggers can be simulated with a statement-level after-trigger in which the trigger program iterates through each row of the transition table if trigger actions support iteration. In addition, these statement-level after-triggers could use the "`order by`" clause to sort the rows in the transition tables and guarantee a deterministic behavior. Another topic for future research is to support an ordering clause to help users control the processing of row-level triggers. Row-level triggers are an important feature because they greatly simplify trigger actions. In addition to providing a convenient short-hand for processing actions once per each row, they inherently provide a mechanism for pairing the `new` and `old` transition values of an updated row. Simulating this mechanism in statement-level triggers requires a join of the `old` and `new` transition tables using the columns of non-updatable primary keys. Furthermore, row-level before-triggers cannot be simulated with statement-level before-triggers because the statement-level before-triggers cannot reference transition tables.

Our model is unique in considering the semantics and scoping rules for transition table and transition variable references within the context of all the various flavors of triggers. Transition tables can be referenced in the conditions and actions of both row-level and statement-level after-triggers, and facilitate applications that perform actions based on the aggregate information about the transition. Furthermore, a transition table collects all transitions of an event type for the execution of a statement and the evaluation of its constraints. This is particularly advantageous in the case of cyclic RI, as it delivers the full transition table to the application logic, allowing it to reason about the net-effect of a given statement. Consider a deletion for a department table that causes cascaded deletes of all employees in the deleted department, which in turn causes cascaded deletes of any departments managed by the deleted employees. At the end of the constraint evaluation, the OLD transition table of a delete-trigger for the department table contains all of the deleted departments, and the OLD transition table of a delete-trigger for the employee table contains all of the deleted employees. Assuming no other after-trigger modifies the database, these transition tables can be unioned with the existing database to reconstruct its contents prior to the original delete statement.

Our model utilizes a global trigger ordering when the fixpoint execution of constraints executes many events causing the simultaneous activation of several after-triggers with different event types. A global trigger ordering is also required for the execution of deferred constraints and triggers. SQL3 currently does not provide a mechanism for a user defined global ordering, and, in the implementation of DB2, we define the global ordering according to the creation timestamp. Extending the standards to allow user specified global ordering among all triggers, similar to the Starburst Rule System [1], and extending our model to support deferred triggers and constraints are topics for future work.

Although there has been considerable research activity for active DBMSs, prior to this paper the important problem of completely integrating declarative constraints and active rules without restrictions has not been addressed. The execution models of many prototype active database systems, e.g., Ariel [14], HiPac [8, 18], RDL [22], RPL [9], and Starburst [36], consider only ECA rules and are more in the style of OPS5; the activation of triggers is considered only at preset activation points, the execution of trigger actions are non-interruptible, and trigger activation is determined based on the net-effect of multiple events. Because declarative constraints are supported using rules, these systems cannot benefit from the important semantic query optimization techniques [23]. In contrast to the OPS5 models, triggers appearing in the marketplace are more procedural, à la Postgres [33], Alert [31] and Update Dependencies [4, 25]. Once activated, a trigger will be eventually executed, and if a trigger action causes an event that activates another trigger, the newly activated triggers are executed immediately. The burden for controlling triggering based on the net-effect of database modifications is placed on the user who can program this explicitly in the trigger condition and action.

8 Summary

Support for active data is crucial to the management of the world's information. Declarative constraints and triggers are two essential features that have been introduced to support user requirements in relational DBMSs. Given the differing expressive powers of declarative constraints and triggers, support for both is required for today's applications.

The semantics of the interaction of triggers and declarative constraints must be carefully defined to avoid inconsistent execution and to provide users a comprehensive model for understanding such interaction. This is the first paper that defines such a model. Our model is unique in that it maintains the set semantics for evaluating declarative constraints as defined by SQL92 while allowing the execution of powerful procedural triggers. This model integrates trigger execution with all forms of constraints defined in the SQL92 standard: primary keys, unique keys, NOT NULL specifications, check constraints, referential constraints, and symmetric views. The main advantage of this model, as described in Section 7 "Discussion of the Model", is that it maintains the SQL92 fixpoint semantics for the evaluation of constraints and incorporates before-triggers into this fixpoint computation while ensuring that application logic embedded in after-triggers operates with a consistent database. This work is also the first to define the semantics and scoping rules of both transition tables and transition variables in the context of row-level and statement-level triggers.

Acknowledgments

Richard Sidle incorporated portions of this execution model into DB2 C/S, Dan Lee, John Hornibrook, and the entire DB2 Common Server team implemented and tested DB2 C/S triggers. Don Chamberlin, Bruce Lindsay, Pat Selinger, Mike Carey and many others at Almaden provided thorough review of this work.

References

[1] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, pages 479–487, Barcelona (Catalonia, Spain), September 1991.

[2] 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.

[3] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the 16th Int. Conf. on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[4] R. Cochrane. *Issues in Integrating Active Rules Into Database Systems.* Ph.D. dissertation, University of Maryland, Department of Computer Science, College Park, MD, 1992.

[5] R. Cochrane and N. Mattos. ISO-ANSI SQL3 Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL LHR-89, X3H2-95-458, An Execution Model for After Triggers, November 1995.

[6] Database Programming and Design. *The 1996 Business Rules Summit*, February 1996.

[7] C.J. Date and Hugh Darwen. *A Guide to The SQL Standard.* Addison-Wesley Publishing Company, Reading, Massachusetts, 3rd edition, 1993.

[8] U. Dayal, B. Blaustein, A. Buchmann, S. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1), March 1988.

[9] 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 – Proc. from the Second Int. Conf.*, pages 333–351, Redwood City, California, 1989. Benjamin/Cummings.

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

[11] K.P. Eswaran and D.D. Chamberlin. Functional specifications of a subsystem for data base integrity. In *Proc. of the 1st Int. Conf. on Very Large Data Bases*, pages 48–67, Framingham, Massachusetts, September 1975.

[12] H. Fosdick. Will DB2 make the short list. *Information Week*, pages 56–64, August 28, 1995.

[13] B. Von Halle. Uncovering business rules. *Database Programming and Design*, 8(7):13–18, December 1995.

[14] E. N. Hanson. The design and implementation of the ariel active database rule system, 1995.

[15] B. Horowitz. A run-time execution model for referential integrity maintenance. In *Proc. 8th International Conf. on Data Engineering*, pages 548–556, Tempe, Arizona, February 1992.

[16] B. Horowitz. Intermediate states as a source of nondeterministic behavior in triggers. In *Fourth International Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 148–155, Houston, TX, February 1994.

[17] L. Howe. Sybase data integrity for on-line applications. Technical report, Sybase, Inc., Emeryville, CA, 1986.

[18] M. Hsu, R. Ladin, and D. McCarthy. An execution model for active database management systems. In *Proc. 3rd International Conf. on Data and Knowledge Bases - Improving Usability and Responsiveness*, Jerusalem, June 1988.

[19] *IBM Database 2 Referential Integrity Usage Guide.* IBM Corporation, International Technical Support Center - Santa Teresa, San Jose, California, 1988.

[20] ISO-ANSI Database Language SQL2 Standard; X3H2-92-154, 1992.

[21] ISO-ANSI Working Draft: Database Language SQL (SQL3), ISO/IEC JTC1/SC21/WG3 DBL LHR-004, X3H2-95-368, j. melton, editor, October 1995.

[22] J. Kiernan, C. de Maindreville, and E. Simon. Making deductive databases a practical technology: A step forward. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, Atlantic City, New Jersey, June 1990.

[23] J. King. QUIST: A system for semantic query optimization in relational database. In *Proc. 7th International Conf. on Very Large Data Bases*, Cannes, France, September 1981.

[24] G. Koch and K. Loney. *Oracle: The Complete Reference.* Osborne McGraw-Hill, Berkeley, California, 3rd edition, 1995.

[25] L. Mark. *Self-Describing Database Systems - Formalization and Realization.* Ph.D. dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1985. TR-1484.

[26] V. M. Markowitz. Safe referential integrity structures in relational databases. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, pages 123–132, Barcelona (Catalonia, Spain), September 1991.

[27] J. Melton and A. R. Simon. *The New SQL: A Complete Guide.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[28] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990.

[29] C. S. Mullins. The procedural DBA. *Database Programming and Design*, 8(7):40–45, December 1995.

[30] Ron Ross. *The Business Rule Book.* Database Research Group, 1994.

[31] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, pages 469–477, Barcelona (Catalonia, Spain), September 1991.

[32] E. Simon and A. Kotz-Dittrich. Promises and realities of active database systems. In *Proc. of the 21st Int. Conf. on Very Large Data Bases*, pages 642–653, Zurich, Switzerland, September 1995.

[33] M. Stonebraker, M. Hearst, and S. Potamianos. A commentary on the POSTGRES rules system. *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems*, 18(3):5–11, September 1989.

[34] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 281–290, Atlantic City, New Jersey, June 1990.

[35] M. Ubell. The Montage extensible datablade. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 482, Minneapolis, Minnesota, May 1994.

[36] J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, pages 275–285, Barcelona (Catalonia, Spain), September 1991.