Chapter 2

Syntax and Semantics of Active Databases

The first chapter of this part is dedicated to the understanding of the syntax and semantics of active rules. This topic is developed by examples, through the careful analysis of four representative active database systems. We start with the Starburst active rule system and conclude with Chimera. These are research prototypes for relational and object-oriented databases, respectively, that emphasize advanced semantic features, such as set-oriented database processing, sophisticated event composition and management, and the ability to access transition values representing past states. In between these research prototypes, we present Oracle and DB2, two of the most popular relational products. The emphasis on relational products is justified by the fact that all relational products support *triggers*. By this term they denote active rules; therefore we will use the terms "trigger" and "active rule" as synonyms. We will conclude this chapter with a comparative review of the various features of active rules and triggers that were progressively introduced.

It is interesting to position the two relational systems relative to the evolution of the SQL3 standard. An effort for defining a standard for SQL triggers has been ongoing since the late 1980s; however, triggers missed the publication deadline for SQL-92, probably because of the inadequacy of the standard document, which was very intricate, especially in a section listing measures for avoiding mutual triggering. Given the importance of standard compliance, all vendors have tried to produce systems as close as possible to the preliminary standard document by disregarding some of its most exotic features, but the document left a number of open issues, resolved by vendors

in different ways. Oracle is a typical example of a system that was carefully developed along the guidelines of the SQL3 standard document dominating between 1990 and 1995.

More recently, in the summer of 1996, the designers of DB2 presented a change proposal for the SQL3 standard that introduces a novel semantics of triggering, in particular with respect to integrity checking. This proposal, which adapts well to the features of most relational products, was recently evaluated and accepted by the SQL3 standard body. Thus, the definition of triggers for DB2 gives us the most recent snapshot on the current SQL3 proposal for triggers, both syntactically and semantically; we cannot easily predict now when and how such a standard proposal will become definitive and be published.

2.1 Starburst

Starburst is the name of a project developed at the IBM Almaden Research Center; an active database extension developed in this framework, called the Starburst Active Rule System, has gained popularity mainly because of its simple syntax and semantics, which adopts the classical set-oriented syntax and semantics of relational databases.

The Active Rule System component extends the data definition language (DDL) of Starburst by enabling the creation and deletion of active rules. Active rules are based on the Event-Condition-Action (ECA) paradigm:

- Events are SQL data manipulation primitives (**INSERT**, **DELETE**, **UPDATE**).
- The condition is a boolean predicate on the database state, expressed in SQL.
- Actions may perform arbitrary SQL queries (including **SELECT**, **INSERT**, **DELETE**, **UPDATE**); in addition, actions may include rule manipulation statements and the transactional instruction **ROLLBACK WORK**.

The ECA paradigm has a simple, intuitive semantics: *when* the event occurs, *if* the condition is satisfied, *then* perform the action. This basic semantics is followed by most active rule systems; we say that a rule is *triggered* when its relevant event occurs, is *considered* when its condition is evaluated, and is *executed* when its action is performed. However, as it will become clear, there are subtle differences in the way in which different

systems define the semantics of triggering, consideration, and execution of active rules.

Active rules are added to the schema and shared by all applications; however, each transaction can dynamically activate and deactivate existing rules. In Starburst, rules can be grouped, and a user can selectively activate and deactivate the rules of a group.

As an initial example, consider the following SalaryControl rule, which monitors the hiring, firing, and change of salary of employees. The rule is defined by a conservative manager who keeps the salary of his employees under control by imposing a salary reduction whenever the average salary goes beyond a given threshold.

Example 2.1 Salary control rule

```
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF      (SELECT AVG (Sal) FROM Emp) > 100
THEN  UPDATE Emp
        SET Sal = .9 * Sal
```

We now proceed with a systematic description of the syntax and semantics of active rules in Starburst.

2.1.1 Syntax of the Starburst CREATE RULE Statement

The main DDL extension for active rules enables the creation of active rules. Its syntax is defined in the following:[1]

```
<Starburst-rule> ::=CREATE RULE <rule-name> ON <table-name>
                    WHEN <triggering-operations>
                    [ IF <SQL-predicate> ]
                    THEN <SQL-statements>
                    [ PRECEDES <rule-names> ]
                    [ FOLLOWS <rule-names> ]
```

[1]Throughout this part of the book we adopt a simple BNF notation where optionality is denoted by square brackets and alternatives by curly brackets; pluralized grammar production names (such as <rule-names>) indicate one or more repetitions of that grammatic construct, separated by commas.

<triggering-operation> ::= **INSERTED** | **DELETED** |
UPDATED [(<column-names>)]

Each active rule in Starburst has a unique name and is associated to a specific table, called the rule's *target*. Each rule monitors multiple events, specified as the rule's triggering operations; the same SQL statement can be monitored by multiple rules. The ordering of rules is regulated by means of a *partial order* between rules; at rule creation time, it is possible to specify that a given rule precedes given rules and follows given rules. The precedence relationships must be acyclic.

2.1.2 Semantics of Active Rules in Starburst

Rules in Starburst are processed in the context of a given transaction; rule processing is implicitly initiated when the transaction issues a **COMMIT WORK** command, thus yielding to a *deferred* execution; in addition, rule processing can be initiated with an explicit **PROCESS RULES** command.

A rule is either *triggered* or *untriggered*; it is untriggered at the transaction's start, and it becomes triggered because of the occurrence of its triggering events (we will define triggering more precisely later). The triggered rules at a given time form the *conflict set*. Starburst's rule processing algorithm consists of the iteration of the following steps:

Algorithm 2.1 *Starburst's rule processing algorithm*
While the conflict set is not empty

1. *Select one rule R from the conflict set among those rules at highest priority; R's state becomes untriggered.*
2. *Evaluate the condition of R.*
3. *If the condition of R is true, then execute the action of R.*

Rule selection at step 1 of the above algorithm produces one of the rules at highest priority (i.e., those rules in the conflict set that are not preceded by any other rule in the set); given that rules are partially ordered by their rule creation statements, there may be many rules that satisfy the above condition. In order to guarantee repeatability of execution (i.e., the same system's behavior on the same database state and input transaction), the system maintains a total order of rules, consistent with the user-specified partial order; rules at the same user-specified priority level are further ordered by the system on the basis of their creation time.

Action execution may cause the triggering of rules, including the re-triggering of the rule being executed. Thus, the above algorithm could lead to an infinite execution, caused by rules that cyclicly trigger each other; in this case, we say that the rule execution is *nonterminating*. As we will see, ensuring termination is one of the main problems of active rule design. When instead rule execution terminates with an empty conflict set, we denote the corresponding database state as a *quiescent state*.

The precise notion of rule triggering requires the definition of *state transitions*; these are transformations from a given state S_1 of the database to another state S_2 because of the execution of SQL statements by transactions. State transitions are defined by means of the sets *inserted*, *deleted*, or *updated*; these sets include the tuples that are, respectively, inserted, deleted, or updated by the SQL data manipulation operations that transform S_1 into S_2. From the set *updated* we can extract subsets containing tuples affected by updates on specific attributes, for example, *updated*(Sal).

Rules consider the *net effect* of operations between two well-defined database states; the net effect is obtained by composing multiple operations occurring on the same tuple. Because of the net effect, each tuple affected by a database operation appears at most in one of the above *inserted*, *deleted*, and *updated* sets. For instance, the net effect of the insertion of a tuple t followed by its deletion is null, while the insertion of a tuple t_1 followed by the update of t_1 into t_2 appears as the insertion of the tuple t_2.

We are now in the position of specifying whether a given rule is triggered. Intuitively, a rule R is triggered if any of the sets corresponding to its triggering operations is not empty, relative to a given state transition. At the first consideration of a rule we consider the transition from the initial state S_0 of the transaction to the state S_1 that exists when R is selected by the rule processing algorithm; this is called the *initial transition* for R. After consideration, R is detriggered (regardless of whether the rule is executed or not). In order to determine whether R is triggered in a later state S_2, we consider the transition from S_1 to S_2, and so on, until rule processing terminates on a quiescent state S_f, in which rule R, as well as all other rules, is not triggered.

The use of net effects and the notion of transitions as defined above has the consequence that each rule processes each database change to individual tuples exactly once.

In the condition or action parts of the rule, it is possible to make references to temporary relations storing the instances of the tuples that characterize state transitions. More precisely, four transition tables are defined, respectively called **INSERTED**, **DELETED**, **OLD-UPDATED**, and **NEW-**

UPDATED. The **INSERTED** and **DELETED** transition tables store the tuples belonging to the sets of *inserted* and *deleted* tuples as defined above; **OLD-UPDATED** stores the content of these tuples *before* the update, while **NEW-UPDATED** stores the content of updated tuples *after* the update. Referring to transition tables within SQL statements of active rules may be more efficient than referring to the current content of regular tables because transition tables are usually small and are resident in main memory. Transition tables are not defined outside of the scope of active rules.

2.1.3 Other Active Rule Commands

The DDL extensions provided by the Active Rule Language of Starburst also include commands for dropping, altering, activating, and deactivating rules:

<deactivate-rule> ::= **DEACTIVATE RULE** <rule-name> **ON** <table>
<activate-rule> ::= **ACTIVATE RULE** <rule-name> **ON** <table>
<drop-rule> ::= **DROP RULE** <rule-name> **ON** <table>

Rules in Starburst can be organized within a *ruleset*. Rulesets are created and manipulated by DDL instructions:

<create-ruleset> ::= **CREATE RULESET** <ruleset-name>
<alter-ruleset> ::= **ALTER RULESET** <ruleset-name>
[**ADDRULES** <rule-names>]
[**DELRULES** <rule-names>]
<drop-ruleset> ::= **DROP RULESET** <ruleset-name>

Rule processing can be invoked within a transaction on a specific ruleset, or even on a specific rule, by means of commands:

<rule-processing-commands> ::= **PROCESS RULES**
| **PROCESS RULESET** <ruleset-name>
| **PROCESS RULE** <rule-name>

2.1.4 Examples of Active Rule Executions

Let us consider an active rule system with only one rule, the SalaryControl rule presented before and repeated here for convenience:

```
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF      (SELECT AVG (Sal) FROM Emp) > 100
THEN  UPDATE Emp
        SET Sal = .9 * Sal
```

Consider the following initial database state:

Employee	*Sal*
Stefano	90
Patrick	90
Michael	110

Consider a transaction that performs the inserts of tuples (Rick,150) and (John,120). The insertion triggers the rule, which is selected at step 1 of the rule processing algorithm. The condition holds (the average is now 112), and the action is executed; the new database state is shown below:

Employee	*Sal*
Stefano	81
Patrick	81
Michael	99
Rick	135
John	108

The action again triggers the rule, this time because of an **UPDATE** operation to the Sal attribute; the rule is again considered, the condition holds again (the average is now 101, which is still too high), and the action is again executed, yielding the database state below:

Employee	*Sal*
Stefano	73
Patrick	73
Michael	89
Rick	121
John	97

At this point, the action again triggers the rule, the rule is again considered, but the condition evaluation yields FALSE (the average is 91); thus, rule processing terminates on a quiescent state. Note that termination occurs because the rule's action decreases the average salary and the rule's condition checks that the average salary is below a given threshold; thus,

rule processing *converges* to a quiescent state where the rule's condition is met. If, instead, the multiplicative factor were equal to or greater than 1 in the action, then the execution would have been infinite. Generally, it is the responsibility of the rule designer to ensure that all executions terminate.

Let us consider now the effect of adding to the active database the rule HighPaid, which inserts within a view the tuples of those newly inserted employees whose salary exceeds 100.

Example 2.2 High paid rule

```
CREATE RULE HighPaid ON Emp
WHEN INSERTED
IF      EXISTS (SELECT * FROM INSERTED
                WHERE Sal > 100)
THEN  INSERT INTO HighPaidEmp
        (SELECT * FROM INSERTED
        WHERE Sal > 100)
FOLLOWS AvgSal
```

Consider again the transaction adding tuples for Rick and John to a database storing tuples for Stefano, Patrick, and Michael as illustrated above. Both rules SalaryControl and HighPaid are triggered by the insert, but the latter follows the former in rule ordering. Thus, rule SalaryControl is considered twice by the rule processing algorithm, yielding to the same final state relative to the Emp table. At this point, rule SalaryControl is not triggered, while rule HighPaid is still triggered because of the initial **INSERT** operations. Thus, the rule is considered, and its condition is satisfied. Because of net effects, the rule considers as **INSERTED** the following temporary table:

Employee	*Sal*
Rick	122
John	97

Note that this table is quite different from the value of the tuples that were actually inserted, as it reflects the application of two updates to each of the inserted tuples. The rule's action is executed next, causing the tuple (Rick,122) to be inserted into the view HighPaid. The tuple (John,97) is not inserted, although John was originally inserted by the transaction with a salary that was higher than 100, because this salary was rectified by the execution of the SalaryControl rule. Rule prioritization, in this case, serves the purpose of enabling the consideration of rule HighPaid only after the (recursive) consideration and execution of rule SalaryControl.

2.2 Oracle

Oracle supports general-purpose triggers, developed according to preliminary documents on the SQL3 standard for triggers. Triggers in Oracle may execute actions that contain arbitrary PL/SQL code; this feature makes triggers in Oracle particularly powerful and expressive. Triggers in Oracle respond to data manipulation operations (insert, delete, update) on a given target table. They support two distinct granularities, *row-level* and *statement-level*. With row-level triggers, triggering occurs for each row that is affected by the operation; thus, row-level triggers have an *instance-oriented semantics*, different from the one supported by Starburst. On the other hand, statement-level triggers have a set-oriented semantics, as in Starburst.

Another distinguishing feature of triggers in Oracle is that they are *immediate*: trigger consideration is tightly coupled with the execution of the triggering SQL statement. Recall that, in contrast, Starburst has a *deferred* execution mode, where triggers are executed either because of an explicit transactional statement or at commit time. In particular, triggers in Oracle can be considered and executed either *before* or *after* the triggering operation. Thus, by combining two granularities and two raising times, we get four possible combinations:

1. row-level before-triggers
2. statement-level before-triggers
3. row-level after-triggers
4. statement-level after-triggers

2.2.1 Syntax of the Oracle CREATE TRIGGER Statement

The syntax for creating triggers in Oracle is

<Oracle-trigger> ::= **CREATE TRIGGER** <trigger-name>
{ **BEFORE** | **AFTER** } <trigger-events>
ON <table-name>
[[**REFERENCING** <references>]
FOR EACH ROW
[**WHEN** (<condition>)]] <PL/SQL block>

<trigger event> ::= **INSERT** | **DELETE** | **UPDATE**
[**OF** <column-names>]

<reference> ::= **OLD AS** <old-value-tuple-name> |
NEW AS <new-value-tuple-name>

Let us discuss each syntactic feature in detail. Each trigger may monitor any combination of the three data manipulation operations (insert / delete / update) on the target table; when the trigger's reaction to events being monitored depends on the event and the trigger is row-level, special predicates **INSERTING**, **DELETING**, and **UPDATING** may be used in the action to detect which of the triggering events has occurred.

The granularity of the trigger is dictated by the clause **FOR EACH ROW**, which needs to be included for tuple-level triggers; when this clause is omitted, the granularity is statement-level. A condition is optionally supported only when the trigger is row-level, and consists of a simple row-level predicate that has to be true in order for the action to be executed on the specific row. For statement-level triggers, the condition part is not present; a condition can nonetheless be expressed within the action as a **WHERE** clause of an SQL statement, but in this case its failure does not prevent the action from being taken, which may result in the endless activation of the trigger.

For both row- and statement-level triggers, the action is an arbitrary block written in PL/SQL, with the limitation that the block should not include DDL or transactional commands.

References to past states are possible only when the trigger is row-based, and they are restricted to the specific tuple being changed; it can be referenced by means of correlation variables introduced by the **REFERENCING** clause, or implicitly by built-in variables called **OLD** and **NEW**.

2.2.2 Semantics of Oracle Triggers

Oracle versions prior to 7.1 impose a severe limitation, namely, that each operation can be monitored by at most one trigger per mode; thus, each operation has at most four associated triggers, corresponding to the options defined above. The limitation on the number of triggers is dropped in Oracle 7.2 and subsequent versions, however without providing an explicit mechanism for prioritizing the triggers that respond to the same event and mode, whose order is system-controlled and cannot be influenced by the database administrator.

The execution of a specific SQL statement is interleaved with the execution of triggers according to the following approach:

Algorithm 2.2 *Oracle's rule processing algorithm*

1. *Execute the statement-level before-triggers.*
2. *For each row in the target table:*
   a. *Execute the row-level before-triggers.*
   b. *Perform the modification of the row and row-level referential integrity and assertion checking.*
   c. *Execute the row-level after-triggers.*
3. *Perform statement-level referential integrity and assertion checking.*
4. *Execute the statement-level after-triggers.*

Each database manipulation performed from within the action part may cause the activation of other triggers; in this case, the execution of the current trigger is suspended and the other triggers are considered by applying to them the execution algorithm illustrated above. The maximum number of *cascading* (i.e., activated) triggers is 32; when the system reaches this threshold, which indicates the possibility of nontermination, execution is suspended and a specific exception is raised. If an exception is raised or an error occurs during trigger execution, then all database changes performed as a result of the original SQL operation and all the subsequent triggered actions are rolled back. Thus, Oracle supports partial (per statement) rollbacks instead of transactional rollbacks.

2.2.3 Example of Trigger Executions

The following trigger Reorder is used to automatically generate a new order (by inserting a tuple into the table PendingOrders) whenever the available quantity (PartOnHand) for a given part in the Inventory table goes below a given threshold (ReorderPoint). The attribute Part is the key of both tables Inventory and PendingOrders.

Example 2.3 Reorder rule

```
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
WHEN (New.PartOnHand < New.ReorderPoint)
FOR EACH ROW
    DECLARE NUMBER X
    BEGIN
        SELECT COUNT(*) INTO X
```

```
        FROM PendingOrders
        WHERE Part = New.Part;
    IF X=0
    THEN
        INSERT INTO PendingOrders
        VALUES (New.Part, New.OrderQuantity, SYSDATE)
    END IF;
END;
```

The above trigger has a row-level granularity and is considered immediately after each update to the PartOnHand attribute. The condition is checked for each row by taking into account the values that are produced by the update, denoted by means of the New correlation variable. The action contains code in PL/SQL, which however is straightforward; a variable X is declared and then used to store the number of pending orders for the particular part. If that number is equal to zero (i.e., if there are no pending orders for that part), then a new order is issued by inserting into PendingOrders a tuple with the name of the part, the (fixed) part's reorder quantity, and the current date. It is assumed that tuples in the PendingOrders table are deleted by a suitable application when parts are received at the inventory. Note that values of the row that is associated to the trigger's execution can be accessed in the action part of the trigger by means of the New correlation variable.

Consider the following initial state for the Inventory table; assume that PendingOrders is initially empty:

Part	*PartOnHand*	*ReorderPoint*	*ReorderQuantity*
1	200	150	100
2	780	500	200
3	450	400	120

Consider the following transaction (executed on October 10, 1996):

```
T1: UPDATE Inventory
    SET PartOnHand = PartOnHand - 70
    WHERE Part = 1
```

This transaction causes the trigger Reorder to be executed, resulting in the insertion into PendingOrders of the tuple (1,100,1996-10-10). Next, assume that the following transaction is executed on the same day:

T_2: **UPDATE** Inventory
SET PartOnHand = PartOnHand - 60
WHERE Part >= 1

At this point, the trigger is executed upon all the tuples, and the condition holds for parts 1 and 3. However, no new order is issued for part 1 (because of the order pending to this part), while a new order is issued for part 3, resulting in the new tuple (3,120,1996-10-10), which is also inserted into the table PendingOrders. Note that, although T_1 and T_2 are separately committed, the effects of T_1 are seen by T_2, since the two transactions share the tables of the database, and active rules run within the scope of their triggering transactions.

2.3 DB2

Triggers for DB2 Common Servers were recently defined at the IBM Almaden Research Center; efforts were focused on giving to triggers a precise, unambiguous semantics, especially with respect to integrity constraint maintenance, by taking into account the experience gained in the earlier development of the Starburst system.

2.3.1 Syntax of the DB2 CREATE TRIGGER Statement

Every trigger in DB2 monitors a single event, which can be any SQL data manipulation statement; updates can be optionally qualified by a list of column names. As in Oracle, triggers are activated either **BEFORE** or **AFTER** their event, and have either a row- or a statement-level granularity. The syntax of triggers in DB2 is shown below:

<DB2-trigger> ::= **CREATE TRIGGER** <trigger-name>
{ **BEFORE** | **AFTER** } <trigger-event>
ON <table-name>
[**REFERENCING** <references>]
FOR EACH { **ROW** | **STATEMENT** }
WHEN (<SQL-condition>)
<SQL-procedure-statements>

<trigger-event> ::= **INSERT** | **DELETE** | **UPDATE**
[**OF** <column-names>]

<reference> ::= **OLD AS** <old-value-tuple-name> |
NEW AS <new-value-tuple-name> |
OLD_TABLE AS <old-value-table-name> |
NEW_TABLE AS <new-value-table-name>

We notice the presence of transition values both at row- and statement-level granularities. **OLD** and **NEW** introduce correlation variables describing tuple values, as in Oracle, while **OLD_TABLE** and **NEW_TABLE** describe the tuples before and after a set-oriented update statement. An insert statement is described only by the **NEW_TABLE**, and a delete statement is described only by the **OLD_TABLE**. The content of these tables is similar to the content of Starburst's **INSERTED**, **DELETED**, **OLD_UPDATED**, and **NEW_UPDATED** tables, but these three tables can all be present within a Starburst trigger, which can monitor multiple events, while a DB2 trigger is relative to only one event. Triggers cannot perform data definition or transactional commands; however, they can raise errors, which in turn can cause statement-level rollbacks.

2.3.2 Semantics of DB2 Triggers

Before-triggers are typically used to detect error conditions and to condition input values; an assignment statement allows the body of these triggers to set the values of **NEW** transition variables. Before-triggers appear to execute entirely before the event that they monitor; thus, their conditions and actions must read the database state prior to any modification made by the event. They cannot modify the database by using **UPDATE**, **DELETE**, and **INSERT** statements, so that they do not recursively activate other triggers.

After-triggers embed part of the application logic in the database; their condition is evaluated and their action is possibly executed after the event's modification. The state of the database prior to the event can be reconstructed from transition values; for instance, for a given target table T, the before state is (T **MINUS NEW_TABLE**) **UNION OLD_TABLE**.

Several triggers (with either row- or statement-level granularity) can monitor the same event. They are considered according to a system-determined total order, which takes into account the triggers' definition time.

Row- and statement-level triggers are intertwined in the total order.[2] Row-level triggers are considered and possibly executed once for each tuple, while statement-level triggers are considered and possibly executed once per statement. If the action of a row-level trigger has multiple statements, all statements are executed for one row before considering the next row.

When triggers activate each other, if a modification statement S in the action A of a trigger causes an event E, then the following procedure takes place

Procedure 2.3 *DB2's statement processing procedure*

1. *Suspend the execution of A, and save its working storage on a stack.*
2. *Compute transition values (***OLD** *and* **NEW***) relative to event E.*
3. *Consider and execute all before-triggers relative to event E, possibly changing the* **NEW** *transition values.*
4. *Apply* **NEW** *transition values to the database, thus making the state change associated to event E effective.*
5. *Consider and execute all after-triggers relative to event E. If any of them contains an action A_i that activates other triggers, then invoke this processing procedure recursively for A_i.*
6. *Pop from the stack the working storage for A and continue its evaluation.*

Steps 1 and 6 of the above procedure are not required when the statement S is part of a user's transaction; if any error occurs during the chain of processing caused by S, then processing of S returns an error condition and the database state prior to the execution of S is restored.

This execution procedure is modified when the statement S, applied at step 4, violates constraints defined in the SQL-92 standard, such as referential constraints, check constraints, and views with check option. Thus, after step 4 in the above procedure, the *compensating actions* invoked by these constraints are performed until a fixpoint is reached where all integrity constraints are satisfied. The computation of these actions, however, may activate both before- and after-triggers; thus, the following processing takes place:

[2] Recall that in Oracle, instead, the interleaving of row- and statement-level triggers is fixed, defined by the rule processing algorithm.

Procedure 2.4 *Revised step 4 of DB2's statement processing*

4. *Apply the* **NEW** *transition values to the database, thus making the state change associated to event E effective. For each integrity constraint IC violated by the current state, consider the action A_j that compensates the integrity constraint IC.*
   a. *Compute the transition values (***OLD** *and* **NEW***) relative to A_j.*
   b. *Execute the before-triggers relative to A_j, possibly changing the* **NEW** *transition values.*
   c. *Apply* **NEW** *transition values to the database, thus making the state change associated to A_j effective.*
   d. *Push all after-triggers relative to action A_j into a queue of suspended triggers.*

At this point, several after-triggers may be pending, due not only to the original statement S, but also to compensating actions A_j; they are processed according to their priority until a quiescent point is reached where all the integrity constraints violated in the course of the computation are compensated.

This integration of trigger management and integrity checking was recently submitted to the SQL3 standard body and accepted for incorporation into the SQL3 standard.

2.3.3 Examples of Trigger Executions

Consider a database schema including tables Part, Distributor, and Audit. The Part table includes PartNum as primary key, and the attributes Supplier and Cost; assume that HDD is the default supplier. Integrity contraints include a referential integrity constraint from Part to Distributor, with the following clause included into Part's schema:

FOREIGN KEY (Supplier)
REFERENCES Distributor
ON DELETE SET DEFAULT

Consider the following initial state for the Part and Distributor tables; assume that Audit is initially empty:

PartNum	*Supplier*	*Cost*
1	Jones	150
2	Taylor	500
3	HDD	400
4	Jones	800

Distributor	*City*	*State*
Jones	Palo Alto	California
Taylor	Minneapolis	Minnesota
HDD	Atlanta	Georgia

Next, consider two triggers. The before-trigger OneSupplier avoids a change of the Supplier value to **NULL** by raising an exception that forces a statement-level rollback; note that this trigger does not prevent the insertion of tuples with a null value for Supplier.

Example 2.4 Supplier rule

```
CREATE TRIGGER OneSupplier
BEFORE UPDATE OF Supplier ON Part
REFERENCING NEW AS N
FOR EACH ROW
WHEN (N.Supplier IS NULL)
    SIGNAL SQLSTATE '70005' ('Cannot change supplier to NULL')
```

The after-trigger AuditSupplier enters into the Audit table, for each update to Supplier, the user's name, current date, and total number of updated tuples.

Example 2.5 Audit rule

```
CREATE TRIGGER AuditSupplier
AFTER UPDATE ON Part
REFERENCING OLD_TABLE AS OT
FOR EACH STATEMENT
    INSERT INTO Audit
      VALUES(USER, CURRENT_DATE,
             (SELECT COUNT(*) FROM OT))
```

Consider the following transaction (executed by user Bill on October 10, 1996):

```
T1: DELETE FROM Distributor
    WHERE State = 'California'
```

This transaction causes the deletion of supplier Jones, which violates referential integrity; because of the foreign key specification, the Supplier of rows with PartNum 1 and 4 is set to the default value HDD; this compensating action triggers both OneSupplier and AuditSupplier. The former, a before-trigger, is executed twice, for each row of the updated parts; since the value of column Supplier in table Part is set to HDD, for each row the trigger's condition is invalid, and consequently the action is not performed. Next, the

statement-level after-trigger is executed once, causing the entering in the Audit table of one tuple, as shown below:

User	*CurrentDate*	*UpdatedTuples*
Bill	1996-10-10	2

2.4 Chimera

Unlike the other three languages, which are all extensions of the relational query language SQL, Chimera is an object-oriented database language, integrating an object-oriented data model, a declarative query language, and an active rule language. Active rules in Chimera, called *triggers*, are set-oriented; they have several new features compared to active rules of Starburst and triggers of Oracle and DB2:

- They take advantage of object identifiers for describing the objects that are affected by events; object identifiers provide a *binding passing mechanism* for linking events to conditions and conditions to actions.
- They support different modes for processing events, called *event consumption modes.*
- They support both immediate and deferred trigger execution.
- They support mechanisms for accessing *intermediate database states.*
- They may optionally support the net effect of event computation; net effect computation is invoked by means of special predicates.

2.4.1 Summary of Chimera

Chimera is an expressive object-oriented language. In the following, we give a short summary of the essential notions that are required in order to be able to write simple schemas, triggers, and transactions in Chimera. Although this summary is fully self-contained, some of the notions introduced in this section (such as type constructors and classes) will be defined more extensively in Part VI, dedicated to object-oriented databases; declarative expressions supported by Chimera are inspired by deductive databases, which are the subject of Part III.

Schemas in Chimera include the definition of several *object classes.* An object class has a state consisting of attributes of arbitrary type, built through type constructors **record**, **set**, and **list**; atomic types include the