Cap. 11 y Cap. 10 secc. 10.3

After populating the table with thousands of movies and giving them ID values of 0, 1, 2, . . . , we learn that we'd really rather use 1, 2, 3, . . . , instead. An obvious UPDATE statement would be

```
UPDATE movie_titles
  SET id = id + 1 ;
```

If the constraint were checked during the update of each row, then the very first row that got updated (unless it happened to be the one with the highest ID value) would cause the constraint to be violated, and the statement would fail with an error. However, by waiting until the end of the UPDATE statement, after all rows have been processed, the constraint is not violated and the statement succeeds.

There are many examples of the usefulness of this characteristic. We leave as an exercise for the reader the reason why end-of-statement checking is necessary for a FOREIGN KEY constraint with ON DELETE CASCADE when the FOREIGN KEY points to the same table where it is defined.

```
CREATE TABLE employees (
    emp_id          INTEGER PRIMARY KEY,
    mgr_id          INTEGER REFERENCES employees
                            ON DELETE CASCADE )
```

10.9 Chapter Summary

The material covered in this chapter should help you to further specify your SQL data definitions so that a great deal of the tedious processing inherent in information systems applications will be automatically handled by your underlying database management system. Each and every column may have a range of values, a list of values, a maximum or minimum value, or some other constraint designated at definition time. Further, each table may have a primary key designated through which the data from other tables may be related to the data of that table.

Through the use of foreign keys, which typically are used in conjunction with primary keys but may also be coordinated with nonprimary candidate keys if so desired, several different types of referential integrity constraints may also be specified at data definition time. These include cascading of data value modifications to related tables, automatically deleting certain logically linked rows, and setting various default or null values upon certain actions.

Tomado del libro: "SQL:1999. Understanding Relational Language Concepts" Melton y Simon. Morgan Kaufmann Publishers 2002

TRIGGERS

10

Chapter 11

Active Databases and Triggers

11.1 Introduction

In this chapter, we're going to discuss a feature that most SQL products have long implemented but that was not standardized until the publication of SQL:1999—triggers. In fact, the specification for triggers was complete before SQL-92 was published, but pressures to keep the size of SQL-92 from being even larger than it was caused the deferral of several features.

The definition of triggers was among the deferred features, based on the expectation that it would be several years before the feature would be widely implemented by commercial products—an obvious error in judgment! In fact, almost before the ink was dry on the SQL-92 standard, all major SQL vendors were delivering versions of their products that provided trigger support.

But the trigger specification didn't remain static in the lengthy interval between SQL-92's publication and SQL:1999's. In fact, several improvements and enhancements were made, some to align with features that vendors had added and some to simplify the specifications.

This feature arguably has an unusual name, reminding many people of weaponry. In fact, some of the terminology associated with triggers is related to the concept, including the use of the word *fire*, as in "the trigger fired." We don't believe the intent was to suggest violence against anyone or anything (not even data), but the metaphor of a dramatic action being the immediate result of another, less dramatic action is certainly appropriate. In many cases, a different

8

term has been applied to this feature: *active database*. This term is especially relevant, since it implies that the database itself can take action on the data it contains. Nonetheless, we'll use the more familiar terminology in this book.

11.2 Referential Actions Redux

Before we show you the syntax of defining triggers and discuss the semantics, let's briefly review a subject that we covered earlier in section 10.5.2, "Referential Constraint Actions"—namely, referential actions.

As you'll recall from that discussion, you can define certain constraints, called *referential integrity constraints*, for tables in your database. These referential integrity constraints can be written so that they prohibit programs from making certain types of changes to the data in those tables or so that they automatically keep the data in those tables synchronized in response to certain types of changes. If a constraint is written to automatically keep the data synchronized, it uses referential actions to do so.

Some people maintain that referential actions are properly included in the notion of an active database, since the database automatically updates or deletes rows of data that are not directly identified by SQL statements in response to changes to rows that are identified by those SQL statements. We don't disagree with that belief, but we do note that the changes caused by referential actions are rather closely related to the changes directly caused by an SQL statement. By contrast, the changes that can be made by a trigger in response to changes directly caused by an SQL statement may have no obvious relationship at all to those more direct changes.

As you learned in section 10.5.2, "Referential Constraint Actions," referential actions are invoked whenever the referential integrity constraint with which they are defined is violated by the actions of some SQL statement. The referential actions that you can define can take action only when row values are modified or when rows are deleted, and the actions that can be taken are limited. The referencing rows can be modified in the same way as the rows they reference, they can be deleted, or the columns containing the referencing values can be set to null or to their default values. In particular, no referential actions can be specified to address the insertion of rows into a referencing or referenced table.

11.3 Triggers

Triggers, on the other hand, are considerably more flexible in the events that cause them to take action and in the actions that they are allowed to take.

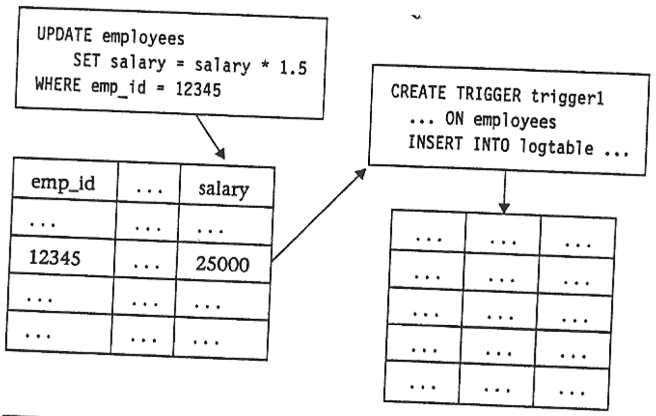
As we will show you shortly, you can define triggers that are invoked (or fired, if you prefer) whenever you insert one or more rows into a specified table, update one or more rows in a specified table, or delete one or more rows from a specified table. These triggers can, generally speaking, take any sort of action you find appropriate for your applications. They can be fired once per `INSERT`, `UPDATE`, or `DELETE` statement or once per row being inserted, updated, or deleted. The table with which the trigger is defined is called the *subject table* of the trigger, and the SQL statement that causes a trigger to be fired is called the *triggering SQL statement*. When a trigger is fired, it causes another SQL statement, called the *triggered SQL statement*, to be executed.

Triggers can be fired even *before* the actions of the triggering SQL statement. Perhaps even more interesting is the fact that the trigger can have access to the values of the row or rows being inserted, updated, or deleted; and, for rows being updated, the values before the update takes place and the values after the update can both be made available.

Triggers, even though they are "schema objects" themselves, are always associated with exactly one base table. SQL:1999 does not allow them to be associated with views, although some SQL products provide extensions to allow that capability.

What uses might triggers serve? The potential uses are numerous, but we narrow them down to just a few categories:

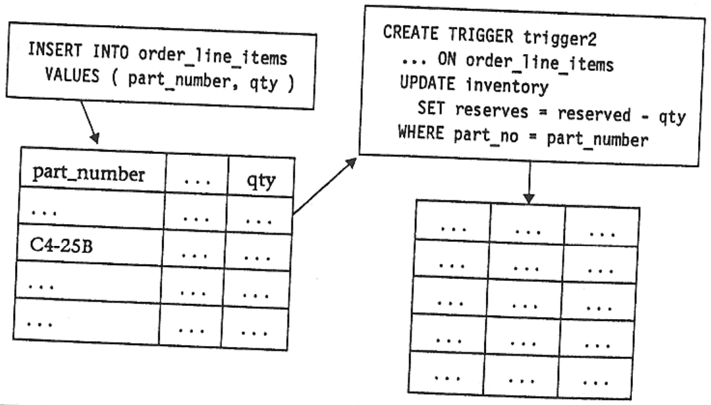
- **Logging and auditing**: You can define triggers on certain tables—especially tables that have security implications, such as salary information and competitive data—to record information about changes made to those tables. That information can be recorded in other tables and might include such information as the authorization identifier under whose control the changes were made, the current time when the changes were made, and even the values being affected in the subject table. This is illustrated in Figure 11-1.
- **Consistency and cleanup**: Your application might benefit from allowing relatively simple sequences of SQL statements on certain tables to be supported by triggers whose responsibilities include making corresponding changes to other tables. Thus, an SQL statement that adds a line item to an order might cause a trigger to be fired that updates an inventory table by reserving the quantity of material that was ordered. We illustrate this alternative in Figure 11-2.
- **Non-database operations**: Your triggers are not restricted to merely performing ordinary SQL operations. Because of the power available in SQL-invoked routines, your triggers can invoke procedures that send e-mail messages, print documents, or activate robotic equipment to retrieve inventory to be shipped. You can see an illustration of this use in Figure 11-3.



```
UPDATE employees
    SET salary = salary * 1.5
WHERE emp_id = 12345
```

```
CREATE TRIGGER trigger1
  ... ON employees
  INSERT INTO logtable ...
```

Figure 11-1 Using Triggers for Logging and Auditing



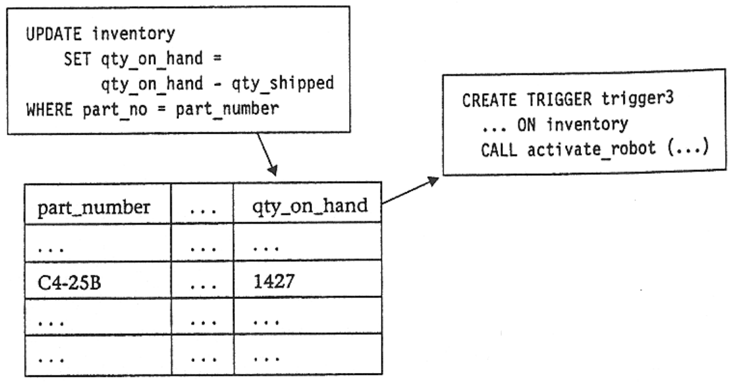
```
INSERT INTO order_line_items
  VALUES ( part_number, qty )
```

```
CREATE TRIGGER trigger2
  ... ON order_line_items
  UPDATE inventory
    SET reserves = reserved - qty
  WHERE part_no = part_number
```

Figure 11-2 Using Triggers for Consistency and Cleanup

Having thus whetted your appetite, we're ready to show you SQL:1999's syntax for defining (and eliminating) triggers and to analyze in detail the behaviors associated with their various features.

We first consider Syntax 11-1, which specifies the syntax of a trigger definition.



```
UPDATE inventory
    SET qty_on_hand =
        qty_on_hand - qty_shipped
WHERE part_no = part_number
```

```
CREATE TRIGGER trigger3
  ... ON inventory
  CALL activate_robot (...)
```

Figure 11-3 Using Triggers for Non-Database Operations

Syntax 11-1 *Syntax of <trigger definition>*

```
<trigger definition> ::=
    CREATE TRIGGER <trigger name>
      <trigger action time> <trigger event>
      ON <table name> [ REFERENCING <old or new values alias list> ]
      <triggered action>

<trigger action time> ::=
    BEFORE
  | AFTER

<trigger event> ::=
    INSERT
  | DELETE
  | UPDATE [ OF <trigger column list> ]

<trigger column list> ::= <column name list>

<triggered action> ::=
    [ FOR EACH { ROW | STATEMENT } ]
      [ WHEN <left paren> <search condition> <right paren> ]
      <triggered SQL statement>
```

```
<triggered SQL statement> ::=
    <SQL procedure statement>
  | BEGIN ATOMIC
      { <SQL procedure statement> <semicolon> }...
    END

<old or new values alias list> ::=
    <old or new values alias>...

<old or new values alias> ::=
    OLD [ROW ][AS ]<old values correlation name>
  | NEW [ROW ][AS ]<new values correlation name>
  | OLD TABLE [ AS ] <old values table alias>
  | NEW TABLE [ AS ] <new values table alias>

<old values table alias> ::= <identifier>

<new values table alias> ::= <identifier>

<old values correlation name> ::= <correlation name>

<new values correlation name> ::= <correlation name>
```

Like much of SQL's syntax, that seems complex at first glance, but we think you'll understand it pretty easily as we explain the various pieces of it.

As you will notice, one piece of syntax is the `<table name>` that identifies the subject table of the trigger. This `<table name>`, like all `<table name>`s, can be fully qualified, including the name of the schema in which it resides. If you specify a `<schema name>` as part of the `<table name>`, then that `<schema name>` must be the same as the `<schema name>` that (explicitly or implicitly) qualifies the `<trigger name>`—that is, a trigger must always reside in the same schema as its subject table.

One requirement that doesn't show in the syntax used to define a trigger is the fact that you must have a particular privilege on tables in order to define triggers on them. If you're the owner of a table, then you inherently have the `TRIGGER` privilege. However, if you have been granted the `TRIGGER` privilege on a table that you do not own, you have the ability to define triggers on that table.

We'll refer back to Syntax 11-1 a number of times in the remaining sections of this chapter, typically by using a piece of BNF as we explain a feature of triggers.

11.3.1 Types of Triggers

SQL:1999's triggers can be described, or categorized, from multiple perspectives. For example, as you can infer from `<trigger action time>` in Syntax 11-1, you can define a trigger to fire before the application of the effects of the triggering SQL statement or after those effects are applied. Similarly, the `<trigger event>` lets you decide which of three types of SQL statement will cause the trigger to be fired, and the `<triggered action>` lets you decide whether the trigger fires once per triggering statement or once per affected row. The `<triggered action>` also lets you determine whether additional conditions must be met to cause the trigger to be invoked.

As a result of applying this taxonomy to triggers, we can say that a given trigger is, for example, a *before insert statement-level trigger* or an *after update conditional row-level trigger*, depending on the choices made when the trigger was defined. (We hasten to add that those terms are our own and may or may not appear in the documentation of the SQL product you're using.)

BEFORE and AFTER Triggers

As you just read, the syntax that Syntax 11-1 calls `<trigger action time>` allows you to specify whether the trigger that you're defining is fired before or after the effects of the triggering SQL statement are applied to the table. The syntax, as you have observed, is trivial—merely a choice between two keywords.

The semantics are not really more complex than the syntax. In fact, the only effect of this bit of syntax is to determine whether the triggered actions are applied immediately before the effects of the triggering statement, or immediately after them.

INSERT, UPDATE, and DELETE Triggers

The `<trigger event>` specifies the nature of the SQL statement (or other event, such as a change caused by a referential action) that causes a trigger to fire. If you specify `INSERT`, then only an `INSERT` statement that specifies the subject table can cause the trigger to fire.

If you specify `UPDATE`, then there are several statements that can cause the trigger to fire. These include an `UPDATE` statement that specifies a search condition (that is, a searched update statement), an `UPDATE` statement that specifies a cursor name (a positioned update statement), and the corresponding dynamic SQL statements (see Chapter 18, "Dynamic SQL"). Similarly, if you specify `DELETE`, there are several statements that cause the trigger to fire, including the searched delete statement, the positioned delete statement, and the corresponding dynamic SQL statements.

Materiali-
zación de
vistas

We think it's worth noting that SQL:1999 does not provide SELECT as a <trigger event>. The capability to fire a trigger when information is retrieved from a table has been discussed a number of times by the designers of SQL, but it is difficult to justify for any purpose other than logging and auditing (for which there are other solutions in many products). We do not expect this capability to be added to SQL in a future revision of the standard, in spite of the fact that a few SQL products do have something corresponding to it. However, political requirements, such as the European Union's strong privacy laws, might change our expectations in the future.

As we'll see in the subsection below called "Triggering Conditions," the semantics of triggers are affected by <trigger event> more than merely by identifying which type of triggering SQL statement causes the trigger to fire.

Statement and Row Triggers

Syntax 11-1 includes a piece of syntax called the <triggered action>; that syntax has several components, one of which determines whether the trigger being defined is a *row-level trigger* or a *statement-level trigger.*

As the names imply, a row-level trigger is one whose triggered SQL statement is executed for every row that is modified by the triggering statement. Similarly, a statement-level trigger is one whose triggered SQL statement is executed only once every time a triggering SQL statement is executed.

The subsection below called "Triggered SQL Statement" has more detail about the implications of making a trigger a row-level trigger or a statement-level trigger. By the way, since the syntax with which you specify FOR EACH ROW or FOR EACH STATEMENT is optional, SQL:1999's default (if you don't specify either) is FOR EACH STATEMENT.

Triggering Conditions

Another component of a <triggered action>—this one optional—is a search condition that allows you to limit the circumstances under which a trigger will fire. You do this by specifying one or more predicates that determine whether or not your criteria have been met. If the search condition evaluates to True, then the trigger will be fired; otherwise (that is, if it evaluates either to False or Unknown), then the trigger will not be fired.

The use of triggering conditions can add considerable expressive power to your triggers. For example, you can use them to cause the trigger to fire only when the triggering SQL statement indicates the sale of more than 100 DVDs in a single operation, or when the price of a CD is lowered by more than 70%. You might also use it to cause your webstore application to issue an e-coupon that the customer can use in a future purchase.

But you must always keep in mind that the triggering conditions are never even considered except at the relevant <trigger action time>, and then only when the triggering SQL statement generates the proper <trigger event>.

Triggered SQL Statement

In Syntax 11-1, you can see that there are two alternatives for <triggered SQL statement>. It is immediately obvious why the first of these (a single SQL statement) is required: the SQL statement allows the trigger to take some action when it is fired.

The need for the other might not be as readily apparent, but a short explanation should clear it up. In SQL:1999, unless conformance to SQL/PSM (or some proprietary analog) is provided, there is no way to make a trigger's single SQL statement perform a sequence of operations, even though that is very often required. Implementation of SQL/PSM would provide your triggers with a compound statement (BEGIN...END), so SQL:1999 makes that facility available in the definition of a trigger even without the availability of SQL/PSM. Using this alternative allows you to specify triggers whose actions include multiple SQL statements executed sequentially. (If conformance to SQL/PSM is provided by your SQL product, then you can use looping, conditional statements, and the other features of computational completeness provided in SQL/PSM.)

The mandatory keyword ATOMIC is explained in section 11.3.2.

Some Examples

Before we delve into the details of how triggers are processed in an SQL implementation, let's have a look at a couple of examples of trigger definitions. You can see these, along with a few comments in SQL syntax, in Example 11-1.

Example 11-1 *Example Trigger Definitions*

```
/* We try to name our triggers meaningfully. In this example, we choose
   "short" names to respect most SQL implementations identifier limits */
CREATE TRIGGER bef_upd_inv_qty
```

```
    BEFORE UPDATE ON inventory (qty_in_stock)
    FOR EACH STATEMENT    -- We like to explicitly specify the defaults
      BEGIN ATOMIC
        CALL send_email ('ShippingManager', 'Inventory update beginning');
        INSERT INTO log_table
          VALUES ('INVENTORY', CURRENT_USER, CURRENT_TIMESTAMP);
      END;

CREATE TRIGGER limit_price_hikes
  BEFORE UPDATE ON movie_titles
  REFERENCING OLD ROW AS oldrow
              NEW ROW AS newrow
  FOR EACH ROW
    WHEN newrow.regular_dvd_sale_price >
         1.4 * oldrow. regular_dvd_sale_price
    -- Uses a PSM feature to prevent large price hikes
    SIGNAL PRICE HIKE_TOO_LARGE;
```

11.3.2 Execution of Triggers

All triggers execute in a *trigger execution context*, and a particular SQL session might have one or more such contexts (or none) at any instant in time. The reason is probably obvious once you think about it. Execution of an SQL statement may cause one or more triggers to fire, and execution of the `<triggered SQL statement>`s of those triggers will probably cause other changes to various tables. The execution of those `<triggered SQL statement>`s is performed in a trigger execution context. But what if the tables changed by those `<triggered SQL statement>`s have their own triggers defined, triggers that are fired by the actions of the first set of triggers? These "secondary" triggers get their own trigger execution context to make it possible to define their actions precisely and independently of the actions taken by the first set of triggers. These secondary triggers can cause still more triggers to be fired—nested arbitrarily deep—and each level gets its own trigger execution context. (In fact, each trigger and its `<triggered SQL statement>` gets its own trigger execution context, but that's a detail that doesn't affect our discussion in any substantive way.)

SQL:1999 has several types of execution context, of which triggers involve just one. Some of these execution contexts are not required to be "atomic," meaning that they may complete some of their actions without all of their actions succeeding. However, trigger execution contexts are required to be atomic: either the entire triggered SQL statement completes successfully, or its effects are not allowed to become persistent in the database. In SQL/PSM, not all

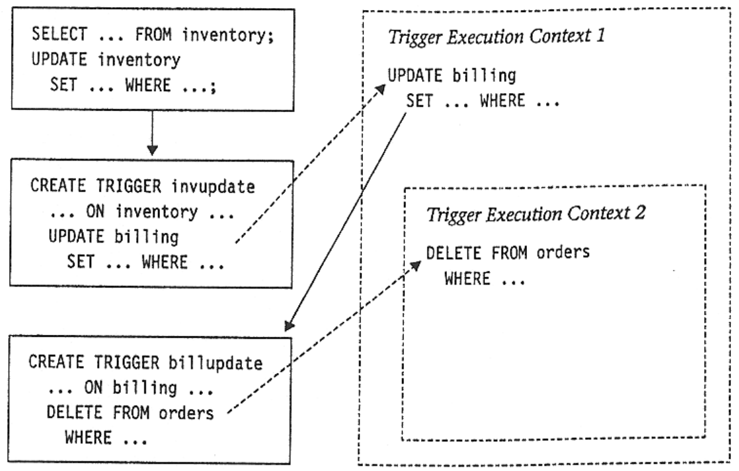


Figure 11-4 Trigger Execution Contexts

`BEGIN...END` blocks are required to be atomic, so the use of `BEGIN...END` in a `<triggered SQL statement>` requires the use of the keyword `ATOMIC` to act as a reminder that this particular `BEGIN...END` block will always be handled as a unit.

Let's look at trigger execution contexts in a little more detail. If you understand how they work, you can design your triggers a little more effectively.

Trigger Execution Contexts

In Figure 11-4, we have illustrated how execution of certain SQL statements cause the creation of trigger execution contexts and how only one of those trigger execution contexts is *active* at any one time.

The `UPDATE` statement that updates the INVENTORY table causes trigger INVUPDATE to be fired; that trigger's triggered SQL statement is another `UPDATE` statement (this time, BILLING is being updated). INVUPDATE's triggered SQL statement is executed in a trigger execution context that we've labeled "Trigger Execution Context 1." (In the SQL standard, trigger execution contexts don't have names, so we've named this one just for the purposes of our discussion.) When Trigger Execution Context 1 was created, it was also made active.

When INVUPDATE's triggered SQL statement is executed, it makes changes to another table (BILLING) that is the subject table of another trigger (BILLUPDATE). As a result of the changes made by INVUPDATE's UPDATE statement, the second trigger (BILLUPDATE) is fired. At that moment, Trigger Execution

Context 1 is "pushed onto the stack" and a second trigger execution context, Trigger Execution Context 2, is created and made active. Trigger Execution Context 1 is not destroyed or released; it is merely suspended, or made inactive, while Trigger Execution Context 2 is active.

Eventually, BILLUPDATE's DELETE statement, which is operating within Trigger Execution Context 2, completes. At this time, Trigger Execution Context 2 is no longer needed, so it is destroyed, and Trigger Execution Context 1 is "popped off the stack" and again becomes active. Only one trigger execution context at a time can be active, but there can be many such contexts "on the stack."

A trigger execution context provides the information required by the SQL system to permit a triggered SQL statement to be executed correctly. This information comprises a set of *state changes*, where each state change describes a change to the data in the target table of the trigger. A state change contains all of the following pieces of information:

- The trigger event: INSERT, UPDATE, or DELETE.
- The name of the subject table of the trigger.
- The names of the subject table's columns that are specified in the trigger definition (only for UPDATE; INSERT and DELETE triggers cannot specify a column list).
- A set of *transitions* (that is, a representation of the rows being inserted, updated, or deleted from the subject table), along with a list of all statement-level triggers that have already been executed in some existing (not necessarily active) trigger execution context and a list of all row-level triggers that have already been executed in some existing (not necessarily active) trigger execution context along with the rows for which they were executed.

The reason for tracking triggers that have already been executed is the prevention of triggers being executed multiple times for the same event, which can cause all sorts of undesirable effects, including triggers that never complete.

When a trigger execution context is first created, its set of state changes is initially empty. State changes are added to a trigger execution context's set of state changes whenever a "new" state change is encountered—one that does not duplicate any existing state change's trigger event, name of subject table, or name of subject table's columns. Each state change is initially empty of transitions, but transitions are added to a state change whenever a row is added to, updated in, or deleted from the subject table associated with the state change. That includes changes that result from the checking of referential integrity constraints and their associated referential actions.

Referencing New and Old Values

One aspect of SQL:1999's triggers that we haven't discussed yet is the optional syntax in Syntax 11-1: REFERENCING <old or new values alias list>. This syntax gives you the ability to reference values in the subject table of your trigger, either in the trigger condition or in the triggered SQL statement.

As you see in the BNF, an <old or new values alias list> is a list of phrases. Each of those phrases can be specified at most one time. They each create a new alias, or correlation name, that you can use to reference values in the trigger's subject table. If you create a correlation name for new values or an alias for new table contents, you can then reference the values that will exist in the subject table *after* an INSERT or UPDATE operation. Similarly, if you create a correlation name for old values or an alias for old table contents, you can then reference the values that existed in the subject table *before* an UPDATE or DELETE operation. Naturally, you cannot use NEW ROW or NEW TABLE for a DELETE trigger, since there are no new values being created, nor can you use OLD ROW or OLD TABLE for an INSERT trigger, since there are no old values to reference. Furthermore, you can specify OLD ROW or NEW ROW only with row-level triggers—that is, if you also specify FOR EACH ROW.

The tables that are referenced by those correlation names or aliases are called *transition tables*. Of course, they are not persistent in the database, but they are created and destroyed dynamically, as they are needed in trigger execution contexts. In a row-level trigger, you can use an old value correlation name to reference the values in a row being modified or deleted by the triggering SQL statement as it existed before the statement modified or deleted it. Analogously, you can use an old value table alias to access the values in the entire table before the triggering SQL statement's effects are applied to the row. In a statement-level trigger, the old value table alias has the same use, of course. In the same manner, you can use a new value correlation name in a row-level trigger to reference the values in a row being modified or inserted by the triggering SQL statement as it will exist after the statement's effects. You can also use a new value table alias to access the values in the entire table after the triggering SQL statement's effects are applied to the row. Finally, a new value table alias has the same use in a statement-level trigger.

You may find this restriction surprising: if you define a BEFORE trigger, you are not allowed to specify either OLD TABLE or NEW TABLE, nor may the trigger's triggered SQL statement make any changes to the database. Those are allowed only for AFTER triggers. The reason is perhaps a bit subtle. The transition tables implied by OLD TABLE and NEW TABLE are too likely to be affected by referential constraints and referential actions that are activated by the changes being caused by the triggered SQL statement; therefore, the values of the rows in that table are not stable or adequately predictable until after the triggering SQL statement has been executed.

One Subject Table: Multiple Triggers

By now, you may have recognized that Syntax 11-1 says nothing about multiple triggers on a single subject table. SQL:1999 certainly doesn't prohibit that capability; it's far too useful to omit. The way in which SQL:1999 supports multiple triggers on a single subject table is simple, if a bit subtle. If there are multiple triggers defined against a single subject table for a single `<trigger action time>` and a single `<trigger event>`, then the trigger that was defined first is the first whose `<triggered SQL statement>` is executed, followed by the trigger that was defined second, and so on. In other words, such triggers are fired in the same order in which they were defined. This is called the *order of execution* of a set of triggers. If, by chance, some subject table has two or more triggers that were defined "at the same time" (within the ability of the database system to measure, at least), then the order in which those triggers are executed is determined by the implementation (random, alphabetical by the definer's authorization ID, etc.).

One unfortunate implication of SQL:1999's use of the order of creation of triggers to determine their order of execution is this: if you discover late in the application development process that you need a trigger to be executed before some already existing trigger (with the same `<trigger action time>`, `<trigger event>`, and subject table), there's no standard way to insert that new trigger at the desired place in the sequence. The only standard way to deal with this is to delete all triggers that should follow your new trigger, create the new trigger, then recreate the triggers you just deleted. We certainly hope that a future revision of the SQL standard will have a more satisfactory solution to this problem.

(Your applications can find out the order of evaluation of such triggers by examining the contents of the ACTION_ORDER column of the TRIGGERS table in the Information Schema. See Chapter 22, "Information Schema," for information on the Information Schema.)

If the execution of some `<triggered SQL statement>` is unsuccessful, then it has no effect on the database, just like any other failed SQL statement execution. More importantly, its failure causes the triggering SQL statement to have no effect, either.

11.3.3 Triggers and Products

In the introduction to this chapter, we mentioned that SQL-92 failed to publish the specification for triggers, even though it was largely complete; we also said that most vendors have implemented triggers in spite of SQL-92's failure to specify a standard for them.

The sad result of this state of affairs is that the various products' implementations of triggers do not conform to one another. Different products (sometimes even different products from the same vendor) implement triggers slightly differently. There are syntax variations, which are relatively easy to overcome. Worse, there are behavioral variations, which cause more problems to application developers.

We find ourselves somewhat doubtful that the SQL products will be brought into conformance with SQL:1999's triggers in a mad rush, but we find reason to hope that they will gradually be migrated into compliance as user demand for standardization of this feature evolves.

11.4 Interaction between Referential Actions and Triggers

Some SQL products implement referential actions through the use of triggers that are created "under the covers." We think that is an unfortunate approach for the simple reason that referential integrity has been carefully designed to be nonprocedural in nature. That is, the SQL standard describes the effects of correct referential integrity in a way that precludes the simplistic sort of procedural code that you might embed in a trigger. (Such products may well implement much more complex code in these "referential action triggers" that give the correct semantics; we haven't evaluated them ourselves to ascertain whether they have or have not done so.)

However, even in SQL products that don't intertwine referential actions and triggers in that way, there is inevitably an interaction between referential actions that might make changes to a given table and the triggers that are defined on that table or that make changes to that table.

SQL:1999 simplifies this interaction a bit by specifying that all integrity constraints, including referential integrity constraints, be properly evaluated, and all referential actions taken, before the `AFTER` triggers are fired. The reason for this is undoubtedly the requirement that all constraints be satisfied before the SQL statement can be considered "successful"; that determination has to be made after all triggers have been processed, because the execution of the triggers themselves could change the contents of tables in a way that violates constraints of various sorts. While other sequences may have been just as meaningful, SQL:1999 has avoided possible different results from different products by mandating a specific sequence of application of referential actions and `AFTER` triggers.