Case study 8

Active databases

This study shows how advanced data structures of SQL can be used to built smarter databases, in particular *active databases*, that are able to react to external stimuli. It starts with a short reminder of the SQL data structures, including some of these advanced constructs, namely check constraints, views, generated columns and triggers. Then, it presents some usual applications of active databases, such as integrity control, redundancy management, updatable views, data modification logging, alerters, type-subtype implementation, repair rules, temporal databases and access control. Finally, it explores, through a simple but representative business application, the power of active databases as compared with traditional application development. In this application, we observe the impact of moving the control of business rules from the programs to the database. This unusual application architecture requires special static and dynamic validation techniques. In its conclusion, this study briefly sketches the history of the trigger concept and evaluates the benefits and disadvantages of its use in application development.

Keywords. ECA rules, trigger, business rules, active database, DAG, finding circuits, advanced SQL, data structure, check predicate, updatable view, derived data, inventory management, 2-tier architecture, 3-tier architecture

Table of content

8.1 Introduction

The main goal of a database is to store data that describe the objects and the activities of a part of the real world, generally called its *application domain*. In addition to the basic constructs through which this static view of the data is implemented (mainly tables, columns and keys), relational DBMS also include more sophisticated mechanisms that allow a database to better translate the facts and the rules of the application domain. Check constraints, generated columns, views (notably updatable ones), stored SQL procedures and triggers are the most important of them. The latter are particularly interesting in that they are the corner stone of so-called *active databases*, that are designed to automatically react to external stimuli.

This study is intended to show how these data structures can be used to built smarter databases. It starts with a short reminder of the SQL data structures, including some of the most advanced constructs. Then, it presents some usual applications of active databases. Finally, it explores, through a simple but complete business application, the power of active databases as compared with traditional application development.

8.2 The SQL data structures

We know that the SQL language offers a rich set of constructs allowing us to create databases that constitute reliable models of data oriented problems and of their solutions. These constructs range from simple static structures, such as tables and columns, to more sophisticated objects like predicates, views, stored procedures and triggers. They all form the basic toolkit to create and manage active databases.

Let us briefly recall and describe them.

8.2.1 Static structures

Let us call *static structures* of a database the constructs that allow us to store data, that is, the basic *data containers*. We recall their main constituents and their properties:

- A *database* comprises tables.
- A *table* is structured into one or more columns. Each *column* is assigned a *data type*, such as numeric, character string, date, blob and some others.
- Each *data row* is stored in a table. For each column of this table, each row comprises one value of the column data type (or *null*, as will be reminded below).
- Among the columns of each table, a subset can be declared its *primary key*. No two rows of the table may include the same values of the primary key. The role of this key generally is to uniquely identify the rows of the table.

– Among the columns of each table, subsets can be declared *foreign keys*. Each value of a foreign key references a row in another table (or in its table itself).

8.2.2 Basic constraints

Three basic constraints can be specified on the static structures:

– *Mandatory column*: if a column is declared *not null*, then each row must comprise a value for this column. Symbolically, we will say that it cannot take the *null* value.

– *Uniqueness constraint*: the rows of a table are distinct. This property can be ensured for a subset of the columns. The *primary key* is one way to assert a uniqueness property[1]. If other subsets enjoy this property as well, they will be declared through *unique keys*.

– *Referential constraint*: this constraint applies to a subset of column of a source table *wrt* a target table. It states that, for any row of the source table, the values of these columns also are those of a row of the target table. The preferred way to ensure this constraint is to declare this subset of columns a *foreign key.*[2]

8.2.3 Predicates (check constraints)

The declaration of a table can include predicates, in the form of *check* constraints. A *check* constraint comprises an SQL condition that must be satisfied (more precisely *not evaluated to False*, which is a bit different) by each row of its table. In most DBMS, this condition may only apply to values of the current row (or system registers, such as *current_date*), without reference to other rows or to other tables.

8.2.4 Derived data

Tables and columns constitute the basic data structures. From them, we can define new tables (called *views*) and columns (called *generated columns*) the contents of which derive through SQL or computation formulae.

– *Views*: A view is a virtual table of which only the definition, in the form of a `select` query, is stored in the database. When a view is queried, its contents is computed through this query, or at least the relevant part of this contents. Under certain conditions the data of a view can be modified. i.e., adding, deleting and modifying rows. In this case, the DBMS propagate the requested modifications on the real data. Generally, a view can be used as a real table.

1. Though primary and unique keys are the preferred ways to ensure uniqueness, they are not the only techniques. Uniqueness can also be verified by triggers.
2. In the same way, there are other techniques to ensure referential constraints, for example through triggers, as in earlier versions of RDBMS.

– *Generated columns*: A *generated* - or *computed* - *column* is a column the values of which are computed by a formula from the values of other columns of its row (or system registers, such as *current_date*). These values can be *virtual*, in which case they are computed each time they are requested, or *stored* (or *persistent*) in the database.

8.2.5 Stored procedures

An SQL stored procedure is a named sequence of SQL statements that is stored in the database. It translates a data manipulation function shared by a community of users. A stored procedure can be called by applications programs, by triggers and by other stored procedures. If the concept of stored procedure is not available in the DBMS, it can be simulated by triggers or UDF.

8.2.6 Triggers

SQL triggers are a concrete implementation of *Event-Condition-Action* rules (aka ECA). They read: when this *Event* occurs and if this *Condition* is satisfied, then execute this *Action*. An SQL trigger is a named procedure (the Action), also stored in the database as a sequence of SQL statements, that will be automatically executed as soon as some events occur (Event), provided the Condition is met. These events comprise data modification actions: `insert`, `update` and `delete`, though some DBMS also allow other events such as opening and closing the database, DDL operations, temporal events or abstract events created by the programs.

The complete definition of a trigger, in addition to its name, generally comprises these components:

– the table or the view it is attached to

– the event(s) that trigger it: *insert*, *delete*, *update* (of any or certain columns)

– when it is executed with respect to the event (*before*, *after* or *instead of* the action). An *instead of* trigger is generally defined on a view, explaining what to do when a modification action is asked for execution.

– its *scope* or *granularity*: once for the action (*for each statement*) or for each row affected (*for each row*)

– an optional *condition* that must be met in order for the body to be executed

– its *body*, which is a sequence of SQL statements. However, many DBMS provide a richer language including *variables* and control statements such as *if-then-else* and *loops*. An important statement is the one that *cancels* or *aborts* the source event and *raises* an exception that can be caught by the application program.

– the *execution order* when more than one trigger have been defined for the same event on the same table.

Many DBMS provide a subset only of these components.[3] Triggers probably are the most powerful feature of relational DBMS. However, they are particularly delicate to use and to debug. Indeed, they rely on three different programming paradigms, namely *event-based*, *logic-based* (through the `when` and the `where` clauses) and *procedural* (within the body part). In addition, they have strong links with the transaction management policy of the DBMS.

As we will learn, the trigger mechanism is at the heart of active databases.

8.2.7 Coping with the limitations of the trigger language

The SQL-3 standard defines the body of a trigger as a *sequence of SQL statements*.[4] However, in most DBMS, the body of a trigger is a procedure written in a general-purpose language comprising, besides SQL statements, standard constructs such as *variables* and *if-then-else* control statements. We show below that we can easily compensate for the lack of such constructs. We also address the way a trigger can call an external procedure.

Simulating variables

Usually, a value used in several statements of the trigger is stored in a local variable. This is particularly recommended if the acquisition cost of this value is high, for instance, if it is computed by an aggregation query.

If the concept of local variable is missing, a technical table containing a single row, where each column implements a variable can be used instead, as illustrated by table **V** in Script 8.23.

```
create table V(A integer, B real, C char(2));

create trigger ...
begin
   ...
   insert into V(A,B)
      select count(*), avg(Account) from CUSTOMER;
   update V set C = 'C1';
   ...
   update CUSTOMER
   set CAT = (select C from V);
   where Account > (select B from V);
   ...
   delete from V;
end;
```

Script 8.23 - Single-row table V acts as a series of local variables

3. This is the case of SQLite: *instead of* triggers on views only, no *for each statement* clause, the trigger *body* is a pure sequence and trigger *execution order* is undefined.
4. SQLite strictly complies with this definition

Simulating *if-then-else* statements

If the trigger language of the DBMS does not include *if-then-else* alternative control statement, then Script 8.24, in which <C>, <C1> and <C2> denote arbitrary SQL conditions, is invalid.

```
create trigger TRG_DEL_T
before delete on T
begin
   if (<C>)
      then delete from S where <C1>;
      else update S set B = null where <C2>;
   endif;
end;
```

Script 8.24 - Standard if-then-else alternative

There are two methods to transform this trigger into pure sequential code. The first one consists in adding respectively <C> and not <C> to each branch of the alternative (Script 8.25). If the evaluation of <C> is expensive, its result can be stored in a local variable, that will be used in next statements as a substitute for <C>.

According to the second method, each branch is translated into a distinct trigger the `when` clause of which decides whether it will fire (Script 8.26).

It is important to note that the validity of these transformations depends on whether the result of the execution of the first branch affects the evaluation of the second instance of condition <C>. For example, in Script 8.25, if <C> is true, the first statement is executed. However, this execution may make condition <C> false, in which case both branches of the alternative will be executed. This point will be examined in more detail in Section 8.18.1.

```
create trigger TRG_DEL_T
before delete on T
begin
   delete from S where <C1> and <C>;
   update S set B = null where <C2> and not <C>;
end;
```

Script 8.25 - Transformation of if-then-else alternative - Method 1

Calling an external procedure

Most DBMS allow the body of a trigger to ask the execution of a procedure written in any proprietary or general programming language. Such a statement will generally look like this one:

```
create trigger TRG_DEL_T1
before delete on T
when <C>
begin
   delete from S where <C1>;
end;

create trigger TRG_DEL_T2
before delete on T
when not <C>
begin
   update S set B = null where <C2>;
end;
```

Script 8.26 - Transformation of if-then-else alternative - Method 2

```
execute procedure sendMail(host,login,pw,...);
```

as in Postgresql, or, simply, as in Oracle PL/SQL:

```
sendMail(host,login,pw,...);
```

Unfortunately, the body of an SQLite trigger must be made up of a sequence of SQL data modification queries only (`insert`, `update`, `delete`) plus a simple form of `select` query initially intended to (conditionally) raise an exception. So, no explicit procedure nor function call.

We will consider the following syntax of this variant of `select` query:

```
select <function> [where <condition>];
```

Where `<function>` denotes a UDF function that executes some desired action. So, instead of

```
sendMail(host,login,pw,...);
```

we will write:

```
select sendMail(host,login,pw,...);
```

In the same way, instead of:

```
if (<condition>) select sendMail(host,login,pw,...);
```

we will write:

```
select sendMail(host,login,pw,...) where <condition>;
```

If we find this syntax a bit awkward, we can improve it as follows:

```
set sendMail = select sendMail;
```

Then, wherever in the body of a trigger we want to write `<something>` in the output window, we simply write:

```
$sendMail$(host,login,pw,...);
```

8.3 Basic applications

There are many applications of triggers suggested and described in the literature. We will illustrate some of them in Sections 8.4 to 8.11.

8.4 Integrity management

While basic integrity constraints (see 8.2.2) are automatically controlled by the database engine, the management of more advanced constraints must be explicitly expressed into predicates, triggers or stored procedures. Constraints on the values of a row generally can be translated into a `check` condition. In the example below (Script 8.27) which defines the structure of table CUSTORDER (customer order), we state that the *order date* must be less or equal to the current date and that the *quantity* must be a positive number.

```
create table CUSTORDER(
       OrdID   char(12) not null primary key,
       DateOrd date     not null,
       CustID  char(12) not null references CUSTOMER,
       ItemID  char(16) not null references ITEM,
       Qty     integer  not null,
       Status  varchar(20) not null,
       constraint CH_DateOrd check(DateOrd <= current_date),
       constraint CH_Qty check(Qty > 0));
```

Script 8.27 - Controlling the values of columns DateOrd and Qty

Both constraints could have been controlled by a single check condition. In this example, each constraint is named and is controlled by its own check condition to allow a more precise information to be sent to the application program. If we submit the following statement:

```
insert into CUSTORDER
values ('2020-S33','2040-02-15','D-568',15);
```

... the operation is aborted and this message is sent to the program:

```
CHECK constraint failed: CH_DateOrd
```

A constraint that makes use of values external to the current row (i.e., values of other rows in the same table or in other tables) will be expressed by one or several triggers.

Let us consider that a customer order can be recorded only if the requested quantity is not higher than the quantity available of the item referenced by column ItemID. We assume that this quantity is stored in column Qavail of table ITEM, the primary key of which is (ItemID).

Usually, a trigger that controls a constraint will check whether *the constraint is violated*, in which case exception is raised (Script 8.28).

```
create trigger TRG_ORDER_QTY
before insert or update of Qty,ItemID on CUSTORDER
for each row
when new.Qty > (select Qavail from ITEM
                where  ItemID = new.ItemID)
begin
   raise('Quantity error in order '||old.OrdID);
end;
```

Script 8.28 - Trigger controlling the values of column OrdQty (CUSTORDER side)

This trigger controls the value of Qty against three events affecting the data of CUSTORDER table: inserting a new row, modifying the current value of Qty of an existing row and changing the current item referenced by an existing row.

To fully control the evolution of the data, we must also ensure that any modification that decreases the value of column Qavail of an ITEM row does not invalidate the value of Qty of all the dependent CUSTORDER rows. Symbolically, the constraint Qavail <= sum(Qty) must be maintained in any condition. This is left as an exercise.

The basic integrity constraints, predicates and the kind of trigger shown above all contribute to define the **valid states** of the database. Let's say that checking whether such constraints have been satisfied, we only need to examine the snapshot of the data after the modification.

To control the **valid transitions** between states, we need to compare two snapshots, *before* and *after* the modification. Column Status of table CUSTORDER records the evolution of an order through its successive processes: recording, validation, execution, payment, archiving, etc. So, the successive values of Status of a definite order must reflect the sequence of these processes: 'recorded' ⟶ 'validated' ⟶ ... ⟶ 'executed' ⟶ 'paid' ⟶ 'archived'.

Checking that this order is respected can only be performed by triggers, in which both previous and new states are simultaneously available through row aliases **old** and **new**. The trigger shown in Script 8.29 checks that Status can be set to 'archived' only if the previous value was 'paid'. old.Status denotes the value of Status before the update and new.Status the new value it would take, should the update succeed. Completing this code to address the other valid value transitions is straightforward.

```
create trigger TRG_ORDER_STATUS
before update of Status on CUSTORDER
for each row
when new.Status = 'archived' and old.Status <> 'paid'
begin
   raise('Invalid Status update in '||old.OrdID);
end;
```

Script 8.29 - Checking valid Status value transitions

8.5 Non standard system behavior

The way the SQL engine reacts to data modification operations may not meet the requirements of the users. Let us consider the foreign key construct, which is the preferred way to enforce referential integrity. When an operation is likely to violate referential integrity, the DBMS will either refuse it (*no action*, *restrict*), or execute the operation then perform additional actions in order to make the data correct again (*cascade*, *set null*, *set default*).

We suppose that our database comprises, among others, the two tables:

– `CUSTORDER(OrdID,...)`, that records the data of customer orders
– `DETAIL(OrdID,ItemID,...)`, each row of which records the reference of one of the item ordered by order OrdID.

Quite naturally, column OrdID of DETAIL is declared a foreign key to table CUSTORDER. We have found it convenient to assign this foreign key a cascade mode for delete and update operations. DETAIL rows can be deleted freely. However, we wouldn't like to see the last DETAIL row of a customer order being deleted. In such a case, we could either prevent this last row to be deleted (a kind of reverse no action mode) or execute an additional action, namely deleting the CUSTORDER row (a kind of reverse cascade mode). This can be controlled through triggers. Let us adopt the second rule: *when the last detail of an order disappears, this order also disappears*.

This rule is implemented by the trigger of Script 8.30.

```
create trigger TRG_LAST_DETAIL
after delete on DETAIL
for each row
when (select count() from DETAIL where OrdID = old.OrdID) = 0
begin
   delete from CUSTORDER where OrdID = old.OrdID;
end;
```

Script 8.30 - Deleting a CUSTORDER row when its last dependent DETAIL row has been deleted

8.6 Updatable view

In most DBMS, updating data through a view is allowed only in very limited cases. For instance, the query that defines the view cannot include joins, subqueries, aggregate functions but must include all the not null columns (therefore also the primary key of the source table). This means that, in general, we **can** query the data through the view but we **must** update them on the source table, which is particularly awkward!

Thanks to appropriate triggers we can design SQL views that allow us to query **and** update the data.

To show how we can build such views, we consider that the database contains the following tables:

– `CUSTOMER(CustID,Name,Address,...)`, that records the data of the customer,

– `CUSTORDER(OrdID,CustID,...)`, that records the data of orders placed by the customer

We would like to view the data of customer orders augmented with the name and the address of the parent customer. This is what the definition of Script 8.31 gives us.

```
create view CUSTORDER_FULL(OrdID,CustID,Name,Address,...)
as  select CustID,O.CustID,Name,Address
    from   CUSTORDER O, CUSTOMER C
    where  O.CustID = C.CustID;
```

Script 8.31 - A view that is not updatable (so far)

Typically, being based on a join, this view is not updatable. The SQL engine does not understand what we mean when we ask it to execute `insert`, `delete` and `update` queries on this view. So, we need to explain our own interpretation of these operations. This is why we define the triggers of Script 8.32.

Now, a query such as

```
update CUSTORDER_FULL set Name='O''Neil' where OrdID='3017';
```

becomes perfectly valid, so that the programmers can ignore the difference between base tables and views!

```
create trigger TRG_CORD_FULL_INSERT
instead of delete on CUSTORDER_FULL
begin
   insert into CUSTORDER values (new.OrdID,new.CustID,...);
end;
create trigger TRG_CORD_FULL_DELETE
instead of delete on CUSTORDER_FULL
begin
   delete from CUSTORDER where OrdID = old.OrdID;
end;
create trigger TRG_CORD_FULL_UPDATE
instead of update of OrdID,Name,Address on CUSTORDER_FULL
begin
   update CUSTOMER
   set    Name = new.Name
   where  CustID = new.CustID and new.Name <> old.Name;
   update CUSTOMER
   set    Address = new.Address
   where  CustID = new.CustID and new.Address <> old.Address;
   update CUSTORDER
   set    OrdID = new.OrdID
   where  new.OrdID <> old.OrdID;
   ...
end;
```

Script 8.32 - The triggers that make view `CUSTORDER_FULL` updatable

8.7 Redundancy management and derived data

According to a popular proverb among database people, *each real world fact must be represented once and only once*. So, theoretically, a normalized database includes no redundant data. Practically, things may be different. It may be useful to record some information in more than one place, or to record data derived from other data, and this for various reasons.

Among them:

– *Expressiveness and readability* of the data structures. Some essential data (from the point of view of users or even programmers) may be missing from the database schema because they can be calculate from more elementary values. Considering table CUSTORDER described in Script 8.27, a naive user may be troubled by the absence of a column giving the *amount* of each order. This amount is easy to compute for each row: we just multiply the value of Qty by the unit price, supposedly stored in column UnitPrice of the referenced ITEM row. So, adding a new column that stores this amount value provides a more natural representation of customer orders, but at the cost of increased storage space and processing time.

– *Efficiency.* Processing a customer order requires the extraction of additional data from the CUSTOMER table (notably their name and address) and from the ITEM table (the item unit price for example). By permanently maintaining a copy of these data in each CUSTORDER row, we avoid two costly joins. This is even more critical when the database is distributed among several sites. Some data can be replicated in several sites to avoid internet transmission cost and latency.

– *Robustness against data corruption.* It is usual to copy critical data in several places. This makes it easier to recover them when a copy has been accidentally or intentionally destroyed.

To illustrate the principles of redundancy management, we will develop further the example of the amount of customer orders mentioned above.

First, we augment the schema of table CUSTORDER as follows (we ignore the columns that are of no use in this development):

```
create table CUSTORDER(
       OrdID     char(12) not null primary key,
       ...,
       ItemID    char(16) not null references ITEM
                                   on update cascade
                                   on delete cascade,
       Qty       integer  not null,
       Amount    integer  not null,
       ...);
create table ITEM(
       ItemID    char(16) not null primary key,
       ...,
       UnitPrice integer not null,
       ...);
```

Script 8.33 - New computed column Amount is added to table CUSTORDER

The value of column Amount of CUSTORDER row **ord** is computed by this formula:

```
      ord.Amount = ord.Qty * (select UnitPrice
                              from   ITEM
                              where  ItemID = ord.ItemID)
```

Though Amount is a computed column, it cannot be declared a **generated** column since its formula involves external columns. Therefore, we must rely on triggers to make its values comply with the formula.[5]

Let us see this formula as an *equality relation* between two quantities. We have to identify which events may alter this equality. The formula uses the data of two tables, CUSTORDER and ITEM. In table CUSTORDER, three columns are involved, ItemID, Qty and Amount. In table ITEM, two columns are involved, ItemID and UnitPrice. The events of interest are those which modify the state of these objects. For each of them, we decide the action to perform.

- **Events of table CUSTORDER**
  - *insert row*: compute the value of Amount.
  - *update of ItemID*: change of referenced item; recompute the value of Amount.
  - *update of Qty*: recompute the value of Amount.
  - *update of Amount*: forbidden.
  - *delete row*: no effect.

- **Events of table ITEM**
  - *insert row*: no effect (no customer order yet).
  - *update of ItemID*: no action; automatically managed according to the cascade update mode of the foreign key.
  - *update of UnitPrice*: recompute the Amount value of all the dependent CUSTORDER rows.
  - *delete row*: no action; automatically managed according to the cascade delete mode of the foreign key.

Now, we have enough information to build the triggers that preserve the equality relation.

Managing CUSTORDER table

The `insert` event and the `update` of Qty and ItemID events can all be coped with by a single trigger (Script 8.34).[6]

5. An alternative structure could be thought of: we add in CUSTORDER new column UnitPrice, the value of which is that of column UnitPrice of the row referenced by ord.ItemID. This redundancy is simple to manage (e.g., there is no need to control the value of Qty). Now, Amount can be declared a *generated* column.
6. In SQLite, a trigger is fired by one event only. So, this trigger must be split into two distinct triggers.

```
create trigger TRG_ORDER_MODIF
after insert, update of Qty,ItemID on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = new.Qty*(select UnitPrice from ITEM
                         where  ItemID = new.ItemID)
   where OrdID = new.OrdID;
end;
```

Script 8.34 - Recalculation of derived column Amount due to events on table CUSTORDER

Prohibiting the explicit modification of Amount can be done in several ways. The most obvious is to *revoke* the right of updating this column from all users:

```
    revoke update (Amount) on CUSTORDER from public;
```

Another technique uses a trigger that cancels any operation attempting to change the value of Amount. The problem with this technique is that it will also cancel other valid modifications requested by the same update statement.

Finally, we suggest a third technique that accepts the modification but silently recovers the correct value through a trigger similar to that of Script 8.34. This leads to the modified code of Script 8.35.

```
create trigger TRG_ORDER_MODIF
after insert, update of Qty,ItemID,Amount on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = new.Qty*(select UnitPrice from ITEM
                         where  ItemID = new.ItemID)
   where OrdID = new.OrdID;
end;
```

Script 8.35 - Extending trigger 8.34 to prevent users from updating the *Amount* column

Managing ITEM table

We only have to cope with update actions on UnitPrice (Script 8.36).

```
create trigger TRG_ITEM_MODIF
after update of UnitPrice on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = Qty*(select new.UnitPrice from ITEM
                     where  ItemID = new.ItemID);
   where OrdID = new.OrdID;
end;
```

Script 8.36 - Propagating UnitPrice modification in table ITEM to the Amount column in CUSTORDER table

8.8 Data modification journaling

We want to keep a precise record of data modification operations on a database. The description of the operations can be written in a text file or in a table of the database itself. We chose the latter approach and we create table JOURNAL, in which the history of `insert`, `update` and `delete` operations affecting each table is recorded.

The content of journal entries depends on the objective we assign to the journal. If the goal is to compute activity statistics, only minimal information will be needed, such as a timestamp, the table name and the nature of the operation. We could want to build a *forward journal* from which the operations can be **replayed** later if necessary. In this case, we must record the complete description of the SQL operations. Or, we could build a *backward journal*, that can be used to **undo** some of the operations that have been successfully executed.

In this exercise, we describe the management of a forward journal. A journal entry will comprise the following information:
– **LogID** : sequence number of the journal entry,
– **OpTime**: date (`aaaa-mm-jj`) and time (`hh:mm:ss.mmm`) of the operation,
– **Operation**: nature of the operation ('insert', 'update', 'delete'),
– **TableName**: name of the table,
– **RowID**: primary key of the row,
– **Arguments**: detail of the operation, e.g., old/new values.

The structure of table JOURNAL is shown in Script 8.37.

```
create table JOURNAL(
    LogID     integer not null primary key autoincrement,
    OpTime    datetime not null,
    Operation char(32) not null,
    TableName varchar(64) not null,
    RowID     varchar(128) not null,
    Arguments varchar(1024));
```

Script 8.37 - Structure of the journal table

Three triggers are suggested to control `insert` (Script 8.38), `update` (Script 8.39) and `delete` operations (Script 8.40). Function `current_timestamp_full()` is a UDF[7] (user-defined function) that returns, in ISO format, the current timestamp with a precision of millisecond.

The arguments of the triggers controlling `insert` and `update` operations collect the new column values in a character string in a format that makes it easy to extract their individual values. Here we have chosen the CSV format, which is more concise than XML and JSON.

```
create trigger TRG_ORDER_LOG_INS
after insert on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID,Argument)
         values(current_timestamp_full(),
               'insert',
               'CUSTORDER',
               new.OrdID,
               '"'||new.DateOrd||'"'
              ||',"'||new.CustID||'"'
              ||',"'||new.ItemID||'"'
              ||','||cast(new.Qty as char)
              ||',"'||new.Status||'"') $;$
end;
```

Script 8.38 - Generating the JOURNAL entries for the **insert** operations

Let us execute this sequence of statements, applied to an empty table:

```
insert into CUSTORDER
       values('123','2020-02-16','B512','PA45',12,'recorded');
insert into CUSTORDER
       values('184','2020-02-17','H054','PA60',5,'recorded');
delete from CUSTORDER where OrdID = '123';
insert into CUSTORDER
       values('270','2020-02-19','F400','PH222',2,'recorded');
```

7. This function is a member of UDF library **SQLiteUDFlib.py** of the SQLfast distribution.

```
update CUSTORDER set Status = 'validated' where OrdID = '184';
```

```
create trigger TRG_ORDER_LOG_UPD
after update on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID,Argument)
         values(current_timestamp_full(),
                'update',
                'CUSTORDER',
                old.OrdID,
                '"'||case when new.DateOrd = old.DateOrd then ''
                          else new.DateOrd end||'"'
             ||',"'||case when new.CustID = old.CustID then ''
                          else new.CustID end||'"'
             ||',"'||case when new.ItemID = old.ItemID then ''
                          else new.ItemID end||'"'
             ||','|| case when new.Qty = old.Qty then ''
                          else cast(new.Qty as char) end
             ||',"'||case when new.Status = old.Status then ''
                          else new.Status end||'"');
end;
```

Script 8.39 - Generating the JOURNAL entries for the **update** operations

```
create trigger TRG_ORDER_LOG_DEL
after delete on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID)
         values(current_timestamp_full(),
               'delete',
               'CUSTORDER',
               old.OrdID) $;$
end;
```

Script 8.40 - Generating the JOURNAL entries for the **delete** operations

It will create the following state of table CUSTOMER.

```
+-------+-----------+--------+--------+-----+-----------+
| OrdID | DateOrd   | CustID | ItemID | Qty | Status    |
+-------+-----------+--------+--------+-----+-----------+
| 184   | 2020-02-17 | H054  | PA60   | 5   | validated |
| 270   | 2020-02-19 | F400  | PH222  | 2   | recorded  |
+-------+-----------+--------+--------+-----+-----------+
```

As to the the content of table JOURNAL, it will look like this one (shown in two parts to make it readable):

```
+-------+-------------------------+-----------+-----------+----
| LogID | OpTime                  | Operation | TableName | ...
+-------+-------------------------+-----------+-----------+----
| 1     | 2020-02-08 17:54:10.971 | insert    | CUSTORDER | ...
| 2     | 2020-02-08 17:54:11.058 | insert    | CUSTORDER | ...
| 3     | 2020-02-08 17:54:11.532 | delete    | CUSTORDER | ...
| 4     | 2020-02-08 17:54:11.604 | insert    | CUSTORDER | ...
| 5     | 2020-02-08 17:54:12.113 | update    | CUSTORDER | ...
+-------+-------------------------+-----------+-----------+----
+-------+-------+----------------------------------------+
| LogID | RowID | Arguments                              |
+-------+-------+----------------------------------------+
| 1     | 123   | "2020-02-16","B512","PA45",12,"recorded" |
| 2     | 184   | "2020-02-17","H054","PA60",5,"recorded"  |
| 3     | 123   | --                                     |
| 4     | 270   | "2020-02-19","F400","PH222",2,"recorded" |
| 5     | 184   | "","","","","validated"                |
+-------+-------+----------------------------------------+
```

We observe that this table provides sufficient information to rebuild the source SQL DML queries.[8]

A complete application is available in script **Scripts\Case-Studies\Case_Active _DB/Data-modification-logging.sql**.

8.9 Alerters

An *alerter* is a mechanism intended to inform some agent (e.g., a person or an external process) about specific events that have occurred. It is somehow similar to *journaling*, except that the information is sent in real time outside the database.

In the context of a database application, sending an email is a popular way to transmit this information and, quite naturally, the *sending agent* is a **trigger** associated with the table in which the events occur.

Let us illustrate the concept of alerter with a table called STOCK, (Script 8.41) in which we store the *quantity on hand* of a set of products (column **QonHand**).

Whenever a quantity **q** of a product is consumed, **q** is subtracted from the quantity on hand of this product. When the latter quantity becomes too low, the employee responsible for the resupply of this product receives a warning email. What *too low* exactly means is specified by column **Qresupply**, that tells below what quantity a resupply must be carried out.

For instance, row `('P001',100,20)` indicates that product `P001` has a quantity on hand of `100` units and a resupply level of `20`. When the value of QonHand falls below that of Qresupply, it is time to replenish the stock.

8. Actually, this is only true for modification operations that apply to *individual rows*. For multirow queries, such as this one: "`delete from CUSTORDER where ItemID = 'PA45'`", that is likely to affect several rows, the journal will include a distinct entry for each of these rows.

```
createOrReplaceDB PRODUCTS.db;
create table STOCK(
       ProdID    char(4) not null primary key,
       QonHand   integer not null,
       Qresupply integer not null);
insert into STOCK values ('P001',50,20),
                         ('P002',40,10),
                         ('P003',90,40);
```

Script 8.41 - Creating the test database

Script 8.42 shows the code of the trigger that implements the alerter. It fires when the value of QonHand is modified in such a way that it *crosses* that of Qsupply, a condition that can be translated as follows[9]:

```
old.QonHand >= new.Qresupply and new.QonHand < new.Qresupply
```

The action of the trigger is quite simple: it sends an email to the supply manager, asking her to replenish the stock of the product that just gets out of stock.

The `set` statement builds the body of the mail and the `exec` statement sends the mail to an SMTP server through the sendMail procedure. For obvious privacy reasons, the critical sending parameters are acquired through an `ask` statement.

If we execute the following `update` queries,

```
update STOCK set QonHand = QonHand - 35 where ProdID = 'P001';
wait 1000;
update STOCK set QonHand = QonHand - 25 where ProdID = 'P002';
wait 1000;
update STOCK set QonHand = QonHand - 60 where ProdID = 'P003';
```

... the supply manager will receive two messages:

```
Quantity on hand of item #P001
has dropped from 50 to 15
on 2020-04-13 11:31:33


Quantity on hand of item #P003
has dropped from 90 to 30
on 2020-04-13 11:31:35
```

9. This condition also addresses the simultaneous modification of column Qsupply.

```
ask host,login,pw = [Initialize mail parameters]
                     Host:|Login:|Pw:;
set from = alert@acme.com;
set to   = supply.manager@acme.com;
set subj = Replenishment required;

openDB PRODUCTS.db;

create trigger TRG_STOCK_ALERT
after update of QonHand on STOCK
for each row
when old.QonHand >= new.Qresupply
and  new.QonHand < new.Qresupply
begin
  set body = 'Quantity on hand of <b>item #'
           ||old.ProdID
           ||'</b><br>has dropped from <b>'
           ||cast(old.QonHand as char)||'</b> to <b>'
           ||cast(new.QonHand as char)
           ||'</b><br>on '
           ||datetime(current_timestamp,'localtime');

  exec sendMail('$host$','$login$','$pw$',
                '$from$','$to$','','html',
                '§subj§',body,'');

closeDB;
```

Script 8.42 - The trigger that implements the alerter

About the *sendMail* procedure

The **sendMail** procedure is a general-purpose procedure that sends an e-mail through an SMTP server. It requires the following parameters:

```
sendMail(
     Host,         # SMTP server address
     Login,        # (server) user id
     Pw,           # (server) password
     From,         # sender address
     To,           # recipient address list
     Cc,           # carbon-copy address list
     Format,       # message format: 'plain' or 'html'
     Subject,      # subject text
     Body,         # body text
     Attach)       # list of attached file names
```

In the SQLfast version, calling this procedure in the body of the trigger is slightly different due to the limitation of the allowed syntax (see Section 8.2.7). The modified version is shown in Script 8.43.

A more informative alerter is available in the scripts of the case study. It not only reacts to the low level of a product but also to any replenishment operation.

```
create trigger TRG_STOCK_ALERT
after update of QonHand on STOCK
for each row
when old.QonHand > new.Qresupply
and  new.QonHand <= new.Qresupply
begin
  select raise(IGNORE)
  where  sendMail('$host$','$login$','$pw$',
         '$from$','$to$','',
         'html','§subj§',
         'Quantity on hand of item <b>#'
        ||old.ProdID
        ||'</b><br>has dropped from <b>'
        ||cast(old.QonHand as char)||'</b> to <b>'
        ||cast(new.QonHand as char)
        ||'</b><br>on '
        ||datetime(current_timestamp,'localtime'),'')
        = 'x'; end;
closeDB;
```

Script 8.43 - The SQLite version of the alerter [Alerter-by-mail-v1.sql]

8.10 Type-subtype implementation

This application shows how complex semantic structures, namely supertype-subtype hierarchies, can be implemented through active database techniques.

8.10.1 About database design

Designing a complex database generally starts with an important phase, *conceptual modeling*, during which the application domain[10] is analyzed. The result is called the *conceptual schema* of the database. This schema identifies in the application domain the categories - or types - of entities (*customers* and *orders*), their attributes (*name* of customer, *date* of order) and their relationships (customers *place* orders). So, the main components of a conceptual schema are the entity types, their attributes and their relationship types.

10. Reminder: the application domain is that part of the real world we are interested in and about which we intend to collect, memorize and exploit information. Also called the *universe of discourse*.

This way of describing things[11] is a trade off between two conflicting requirements: to produce a natural and intuitive description of the application domain (the *user view*) and to easily translate it into database structures (the *developer view*). Indeed, generating an SQL schema from a conceptual schema seems to be fairly easy: each entity type is represented by a table, each attribute by a column, each relationship type by a foreign key and, finally, each entity by a row.

8.10.2The Type-subtype concept

However, the real world we are interested in is not always that simple. It may include many other important fact types that we would like to represent in the database but that are much less straightforward to translate into SQL data structures. Among them, the fact that *an entity may belong to more than one type*, which would imply, in its SQL translation, that a row may belong to more than one table, which is, at first glance, impossible.

For instance, it is generally agreed that *professors* and *students* are *persons*. So, Stonebraker, who is a *professor*, will be classified as a member of type PROFESSOR, and, as such, is an entity of type PERSON as well. In other words, the set of PROFESSOR entities is a subset of the set of PERSON entities.

Obviously, the conceptual schema must report on these facts. It does so in this way:

– We declare entity type PERSON with all its properties (attributes, relationship types, constraints, and the like).

– Then, we declare entity type PROFESSOR a *subtype* of PERSON. PERSON is therefore the *supertype* of PROFESSOR. All the properties of PERSON also are those of PROFESSOR. This mechanism is called *inheritance*: attributes PersID and Name are *inherited* attributes of PROFESSOR. In addition, entity type PROFESSOR may have proper attributes, here Specialty.

– And of course the same for STUDENT.

Synthetically, one says that there is an *is-a* relation between (CUSTOMER, SUPPLIER) and PERSON, an expression that tells that each professor *is-a* person.

These entity types and their *is-a* relations are illustrated in Figure 8.1.[12] Symbol **D** in the triangle indicates that the subtypes are *disjoint*, that is, they cannot share common entities.

Such patterns are at the basis of most programming languages but are generally less frequent in relational databases for reasons we will discuss below.

11. In this example, we refer to the Entity-relationship model, the most popular database conceptual modeling language.

12. The graphical conventions are those of the GER, a general-purpose entity-relationship language intended to represent database schemas according to various data models. This schema has been drawn in the DBMain Case tool.

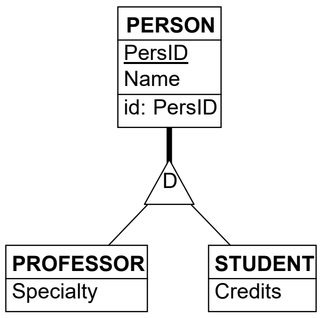


Figure 8.1 - A conceptual schema showing an **is-a** relation

8.10.3 Is-a relations in SQL3

Now, we can formulate the central question of this study: how can we translate *is-a* relations into database structures? The answer depends on the level of the *SQL standard* the DBMS implements.

Besides standard table definition (as in Script 8.27 for example), SQL3 defines the concept of *table type*, from which we can create any number of *typed tables*. In the example below, we create type TPERSON, then two tables of this type. Each of these tables comprises columns PersID and Name as they are defined in type TPERSON:

```
create type TPERSON(PersID char(5),Name varchar(32));
create table PROFESSOR of TPERSON;
create table STUDENT of TPERSON;
insert into STUDENT values('P345','Ullman');
```

An interesting property of table type organization is that a *type* can be declared a *subtype* of another one, the latter being the *supertype* of the former. In the example that follows, TTEACHER and TSTUDENT are declared subtypes of TPERSON through clause `under`. This relation also exists between the tables we create from these types: PROFESSOR and STUDENT are *subtables* of PERSON, which in turn is their *supertable*.[13] Since TTEACHER *inherits* from TPERSON, columns PersID and Name need not be declared in this subtype.

```
create type TPERSON(PersID char(5),Name varchar(32));
create type TTEACHER(Specialty varchar(24) under TPERSON;
create type TSTUDENT(Credits integer) under TPERSON
```

13. This example is simplified, the definitions of subtype structures being fairly complex. It is merely intended to give the reader the taste of SQL subtypes, no more.

```
create table PERSON of TPERSON;
create table STUDENT of TSTUDENT under PERSON;
create table PROFESSOR of TTEACHER under PERSON;
insert into PERSON values ('E0123','Smith');
insert into STUDENT values ('S0234','Stonebraker',45);
```

By definition, table PERSON contains a set of PERSON rows. In addition, all the rows of STUDENT and those of PROFESSOR also are rows of PERSON. So, the following query is valid, though it will only return column PersID and Name of the rows of the subtables:

```
select * from PERSON where Name = 'Stonebraker';
```

The SQL3 specifications discussed above are an attempt to offer database designers an easy way to translate the *is-a* relations of conceptual schemas, and, on the other hand, to bridge the database world with the object-oriented paradigm of most programming languages.[14]

8.10.4Is-a relations in SQL2

The SQL3 specifications are both complex and limited.[15] In addition their implementations in major DBMS often are proprietary and therefore not compatible.[16]

Some DBMS, including SQLite, even ignore table types. As a result, despite their undoubted usefulness, these structures are not popular and are rarely used in practice.

So, why not develop home-made subtable-supertable structures? In this way, we can profit from their modeling advantages without suffering from their drawbacks.

Let us try to develop the PERSON/PROFESSOR/STUDENT example described above with standard database structures. First, we must choose an appropriate implementation of the data that underlie this structure. There are several simple solutions. Among them, the following are the most popular:

1. **Supertable representation**: a single PERSON table gathers all the rows of the three tables. Its schema is the following:

```
create table PERSON(PersID not null,Name not null,
                    Subtype,Specialty,Credit)
```

14. Purely object-oriented database systems have been developed in the 1980s. However, they never met with the hoped-for success in the industrial world, and therefore gradually disappeared in the 1990s.
15. Two examples: (1) two subtables of the same supertable are disjoint (a PROFESSOR row may not be a STUDENT row) and (2) a type may not have more than one supertype (PostgreSQL is an exception).
16. SQL standards are just *recommendations*.

Nullable column Subtype indicates whether the row represents a professor, a student or none of them.[17] Subtables are reconstructed by selecting rows based on column Subtype.

2. **Subtable representation**: three tables represents the *partition* of PERSON, i.e., POPULATION_ONLY (people who are neither professor nor student), PROFESSOR (people who are professor only) and STUDENT (people who are student only). Supertable PERSON is reconstructed through a `union` operator. This representation is not recommended to implement non disjoint subtables.
3. **Table representation**: the supertable is implemented by table PERSON as defined in the structure. PersID is its primary key. Each subtable is represented by a table comprising primary key PersID plus its proper columns. PersID is also a foreign key that references PERSON. Subtables are reconstructed through `join` operators.

Let us choose the last implementation, which will prove the most flexible. Script 8.44 creates the three base technical tables. Since they (hopefully) will never be directly referenced by the users (neither by the programmers), we prefix their name with an underscore symbol.

```
create table _PERSON (
        PersID  char(5) not null primary key,
        Name    varchar(32) not null);
create table _PROFESSOR (
        PersID  char(5) not null primary key
                references _PERSON
                on update cascade
                on delete cascade,
        Specialty varchar(24) not null);
create table _STUDENT (
        PersID  char(5) not null primary key
                references _PERSON
                on update cascade
                on delete cascade,
        Credits integer not null);
```

Script 8.44 - Representing each supertable and subtable by a standard table

Then, we create three views, named PERSON, PROFESSOR and STUDENT, that will be the only external interface for anyone interested in these data. We decide that view PERSON provides all the available data of all its members, be they professor, student or none of them. It is created by left outer joins. As to views PROFESSOR

17. Or both of them (non disjoint subtables), though this case may be more complex to understand, to manage and to process.

and STUDENT, they show all the data respectively of professors and students. They are created by inner joins.

```
create view PERSON (PersID,Name,Specialty,Credits)
    as  select P.PersID,Name,Specialty,Credits
        from _PERSON P
             left join _PROFESSOR F using (PersID)
             left join _STUDENT S   using (PersID);
create view PROFESSOR (PersID,Name,Specialty)
    as  select P.PersID,Name,Specialty
        from  _PERSON P, _PROFESSOR F
        where  P.PersID = F.PersID;
create view STUDENT (PersID,Name,Credits)
    as  select P.PersID,Name,Credits
        from  _PERSON P, _STUDENT S
        where  P.PersID = S.PersID;
```

Script 8.45 - Representing each supertable and subtable by a standard table

We populate these tables with a sample of initial rows (Script 8.46).

```
insert into _PERSON    values ('P0123','Smith');
insert into _PERSON    values ('P0234','Stonebraker');
insert into _PROFESSOR values ('P0234','DB Theory');
insert into _PERSON    values ('P0345','Ullman');
insert into _STUDENT   values ('P0345',45);
insert into _PERSON    values ('P0456','Van Bolle');
insert into _PROFESSOR values ('P0456','Advanced SQL');
insert into _STUDENT   values ('P0456',30);
```

Script 8.46 - Introducing the data of four persons (in technical tables)

The content of the views (see below) shows an interesting variety of persons: `'P0123'` is neither a professor nor a student, `'P0234'` is a professor only, `'P0345'` is a student only and `'P0456'` is both a professor and a student.

View PERSON

```
+--------+-------------+--------------+---------+
| PersID | Name        | Specialty    | Credits |
+--------+-------------+--------------+---------+
| P0123  | Smith       | --           | --      |
| P0234  | Stonebraker | DB Theory    | --      |
| P0345  | Ullman      | --           | 45      |
| P0456  | Van Bolle   | Advanced SQL | 30      |
+--------+-------------+--------------+---------+
```

View PROFESSOR

```
+--------+------------+-------------+
| PersID | Name       | Specialty   |
+--------+------------+-------------+
| P0234  | Stonebraker| DB Theory   |
| P0456  | Van Bolle  | Advanced SQL|
+--------+------------+-------------+
```

View STUDENT

```
+--------+-----------+---------+
| PersID | Name      | Credits |
+--------+-----------+---------+
| P0345  | Ullman    | 45      |
| P0456  | Van Bolle | 30      |
+--------+-----------+---------+
```

Querying data through these views is quite natural. However, we cannot say the same for data creation. Indeed, we must insert data in the three technical table that should remain hidden, a state of affair we cannot accept.

This is where *active databases* come in.

Our new objective is to allow data modification operations to be performed on these views instead of on the technical tables. In other words, we must make these views *updatable*. In most DBMS, modifying data through a view is very restricted (or even forbidden, as in SQLite). In particular, the view cannot be defined by a join.

Getting around this difficulty is simple: since the DBMS does not understand what `insert`, `delete` and `update` queries mean, we will attach to each view a set of triggers that explain how to execute these operations.

8.10.5The insert operations

We first state how we would like to proceed to insert data. Script 8.47 shows a first correct approach. For a person who is neither a professor nor a student, the data are inserted in view PERSON (e.g., 'P0123'). The data of a person who is either a professor or a student (but not both) are inserted into views PROFESSOR or STUDENT respectively (e.g., 'P0234' and 'P0334').

The last case is a bit more complex: the data of a person who is both a professor and a student are inserted in both subtables (e.g., `'P0456'`). So, two `insert` queries for one person, a ratio which is far from natural!

```
insert into PERSON(PersID,Name) values ('P0123','Smith');
insert into PROFESSOR values ('P0234','Stonebraker','DB Theory');
insert into STUDENT   values ('P0345','Ullman',45);
insert into PROFESSOR values ('P0456','Van Bolle','Advanced SQL');
insert into STUDENT   values ('P0456','Van Bolle',30);
```

Script 8.47 - Introducing of the data of four persons (in views) - Method 1

In the second approach (Script 8.48), all the data are inserted into supertable PERSON. If the person is a professor, then column Specialty must have a value. If the person is a student, column Credits must have a value. For a *professor-student*, both columns must be valued.

```
insert into PERSON(PersID,Name) values ('P0123','Smith');
insert into PERSON(PersID,Name,Specialty)
       values ('P0234','Stonebraker','DB Theory');
insert into PERSON(PersID,Name,Credits)
       values ('P0345','Ullman',45);
insert into PERSON(PersID,Name,Specialty,Credits)
       values ('P0456','Van Bolle','Advanced SQL',30);
```

Script 8.48 - Introducing of the data of four persons (in views) - Method 2

The insert triggers

The `insert` triggers coded in Script 8.49 are straightforward and allow for both approaches.

When the data are inserted in view PERSON (as in Script 8.48), trigger TRG_PER_INSERT fires. It first inserts a fragment in table _PERSON. Then, if Specialty is not null, it inserts a fragment in table _PROFESSOR. Similarly, if Credits is not null, it inserts a fragment in table _STUDENT. The subtype(s) indicator derives from the presence of a value of a *not null* column of the subtable(s), namely, here, Specialty for PROFESSOR and Credits for STUDENT).

When the data are inserted in view PROFESSOR (as in Script 8.47), trigger TRG_PRO_INSERT fires. It inserts a fragment in table _PERSON if it didn't already exist (if value new.PersID is not already present in this table). Then, it inserts the remaining fragment in table _PROFESSOR. Trigger TRG_STU_INSERT (not shown) behaves in the same way when data are inserted in view STUDENT.

8.10.6The update operations

Update operations are more powerful than they use to be in standard databases. Indeed, not only are they used to change the value of columns but they also allow us to assign, remove or change the status of persons.

- Changing the value of proper attributes of an existing person

  The name of professor 'P0123' changes

```
update PERSON
set   Name = 'Branson'
where PersID = 'P0123';
```

```
create trigger TRG_PER_INSERT
instead of insert on PERSON
for each row
begin
   insert into _PERSON
      values (new.PersID,new.Name);
   insert into _PROFESSOR
      select new.PersID,new.Specialty
      where  new.Specialty is not null;
   insert into _STUDENT
      select new.PersID,new.Credits
      where new.Credits is not null;
end;

create trigger TRG_PRO_INSERT
instead of insert on PROFESSOR
for each row
when new.Specialty is not null
begin
   insert into _PERSON
      select new.PersID,new.Name
      where  new.PersID not in (select PersID from _PERSON);
   insert into _PROFESSOR
      values (new.PersID,new.Specialty);
end;
```

Script 8.49 - The triggers that control **data insertions** in the user views(trigger TRG_STU_INSERT not shown)

- Changing the value of attributes of an existing professor
  The speciality of professor 'P0234' changes

  ```
  update PROFESSOR
  set   Specialty = 'DB Theory and Practice'
  where PersID = 'P0234';
  ```

  or

  ```
  update PERSON
  set   Specialty = 'DB Theory and Practice'
  where PersID = 'P0234';
  ```

- Assigning a status to an existing person
  Person 'P0123' becomes a professor

  ```
  update PERSON
  set   Specialty = 'Business models'
  where PersID = 'P0123';
  ```

- Removing a status of an existing person
  Person 'P0234' is no longer a professor

```
update PERSON
set   Specialty = null
where PersID = 'P0234';
```

or

```
delete from PROFESSOR
where PersID = 'P0234';
```

The update triggers

The update triggers are shown in Script 8.50.

Trigger TRG_PER_UPDATE1 manages the modifications of the proper columns of the supertable. For performance reasons, we have assigned a distinct update query to each of these columns instead of one big query that updates all the columns, whether their value has changed or not. If the primary key is updated, the modification is automatically propagated to the foreign keys in _PROFESSOR and _STUDENT thanks to their `on update cascade` clauses.

Trigger TRG_PER_UPDATE2 manages the modification of the proper columns of view PROFESSOR imported into the supertable view (here, column Specialty). If the new value of Specialty is not *null*, the effect of the operation depends on whether the person concerned already is a professor or not. In the first case, column Specialty is updated in the dependent _PROFESSOR row. In the second case, a new _PROFESSOR row is created. Assigning a *null* value to Specialty means that this person is no longer a professor. His _PROFESSOR row must therefore be deleted. A similar trigger must be devoted to STUDENT updates.

Trigger TRG_PRO_UPDATE manages the modification of the proper columns of view PROFESSOR. There are two cases. If the new value is not *null*, the column is updated. If this value is null, the person loose his status and the corresponding _PROFESSOR row is deleted.

Similar triggers are built to manage STUDENT updates.

```
create trigger TRG_PER_UPDATE1
instead of update of PersID,Name on PERSON
for each row
begin
   update _PERSON
   set   PersID = new.PersID
   where new.PersID <> old.PersID;

   update _PERSON
   set   Name = new.Name
   where PersID = new.PersID
   and   new.Name <> old.Name;
end;

create trigger TRG_PER_UPDATE2
instead of update of Specialty on PERSON
for each row
begin
   update _PROFESSOR
   set   Specialty = new.Specialty
   where PersID = new.PersID
   and   new.Specialty is not null;

   insert into _PROFESSOR
      select new.PersID,new.Specialty
      where  new.PersID not in (select PersID from _PROFESSOR);

   delete from _PROFESSOR
   where  PersID = new.PersID
   and    new.Specialty is null;
end;

create trigger TRG_PRO_UPDATE
instead of update of Specialty on PROFESSOR
for each row
begin
   update _PROFESSOR
   set   Specialty = new.Specialty
   where PersID = new.PersID
   and   new.Specialty is not null;

   delete from _PROFESSOR
   where  PersID = new.PersID
   and    new.Specialty is null;
end;
```

Script 8.50 - The triggers that control data **update** operations on the user views

8.10.7The delete operations

We can delete a person (with his status) or merely one of his statutes.

- Deleting a person
  Person 'P0234' is deleted

```
delete from PERSON where PersID = 'P0345';
```

- Removing a status of an existing person
  Person 'P0234' is no longer a professor

```
delete from PROFESSOR where PersID = 'P0234';
```

Delete triggers

The delete triggers are particularly straightforward (Script 8.51).

When a _PERSON row is deleted, the dependent rows in the subtables are automatically deleted, due to the `on delete cascade` clauses of the foreign keys.

```
create trigger TRG_PER_DELETE
instead of delete on PERSON
for each row
begin
   delete from _PERSON
   where  PersID = old.PersID;
end;

create trigger TRG_PRO_DELETE
instead of delete on PROFESSOR
for each row
begin
   delete from _PROFESSOR
   where  PersID = old.PersID;
end;
```

Script 8.51 - The triggers that control **delete** operations on the user views

8.10.8Automating type-subtype manager production

Maintaining the consistency of a hierarchy of *is-a* relations requires a fairly large set of triggers. However, if we examine carefully these triggers, we find that their structure is quite simple. Once we know:

– the schema of the technical tables (Script 8.44)

– the *is-a* relations, for instance stored in a temporary table ISA(Supertable,Subtable) writing them is straightforward. In other words, a simple script based on the SQLfast dictionary (`createDictionary`) and the content of table ISA, can automatically generate the complete set of triggers managing the *is-a* hierarchy.

Developing such a script is left to the reader's initiative.

8.11 Other applications

Before discussing and developing the prototype business application that closes this study, we mention three more of the many applications of active databases.

Repair rules

Instead of rejecting `insert` and `update` queries that attempt to introduce erroneous data in the database, these operations are accepted but the data they convey are sanitized or quarantined according to specific repair rules. Some examples:

– *value clipping*: if a value must be greater than or equal to **v**, but actually is lower than this value, it is replaced by **v**

– *dummy rows*: attempting to insert a CUSTORDER row that references a customer who doesn't exist (yet) should be rejected. Instead, a dummy CUSTOMER row is inserted with the CustID value comprised in the invalid row. Later, this row will be completed with additional information on this customer.

– *alternative reference*: a CUSTORDER row to insert references a product that is no longer available. This reference is automatically replaced by that of a similar product.

In all these cases, it is recommended to report in a log table these anomalies and the fixes that have been applied.

Access control

Most SQL implementations provide information about the execution context of queries. This makes it possible to control whether a query can be executed or should be rejected. Let us consider that only employee HR_008 is authorized to update table SALARY, and only on business days from 9 a.m. to 12 a.m. This rule can be enforced by the following trigger:

```
create trigger TRG_SALARY_SECURITY
before update on SALARY
for each row
when not (current_user = 'HR_008'
          and day_of_week(current_date)  between 2 and 6
          and hour_of_time(current_time) between 9 and 12)
begin
   raise('Illegal update of SALARY table');
end;
```

Temporal databases

A database is said *temporal* - or *historical* - when it stores the current and past states of its content.[18] When rows are inserted, updated or deleted, the current state is saved and a new one is added. Each state has a time interval associated with it, indicating the period during which the report is/was valid. These data manipulations may be complex but can be managed automatically by appropriate triggers. Temporal databases are studied in detail in two case studies of this series, *Temporal databases - Part 1* and *Part 2*.

8.12 Building a business application

This application is of a nature different from those we have developed so far. While the latter provide some sort of ancillary services for the benefit of higher level application, this one is an active database that implements a full scale business application (except the graphical user interface of course).

8.12.1About the case

We choose a business application that is sufficiently simple to be described in less than half a page, but sufficiently rich to require the implementation of some non-trivial rules.

This application is to support some basic business processes of a small retail shop that sells building materials to private customers. We identify three actors in this scenario: the *customer*, the *retail shop employee* and the *supplier*(s).

The responsibilities of the customer are simple: placing orders and executing payments.

An employee of the shop receives orders from customers and encodes them *in the computer*. A customer order specifies a quantity of a definite item. If this item is physically available in this quantity, the order is executed, an invoice is printed and sent, and the customer is invited to take away the materials ordered. When the payment of the invoice is received, the order is said to be closed. If there is not enough of this item in the inventory, the order is suspended, the employee writes a purchase order and sends it to the supplier which proposes the best offer for this item: i.e., the lowest price, and in case of tie, the shortest delivery time.

From time to time, each supplier examines the purchase orders it has received for a definite item and decides to execute them. On the retail shop side, this replenishment allows pending orders to be executed and invoices to be sent.

From time to time too, customers pay their invoices, which closes their orders.

18. Actually, true temporal database also record future states.

Since we concentrate our discussion on the retail shop management, we assume that the inventories of suppliers are infinite so that their items never become out of stock. We also consider that the items as viewed by the retail shop also are those viewed by the suppliers.

8.12.2 About the case study

We intend to develop this application according to two different paradigms that can be abstractly described as *3-tier* and *2-tier architectures*.[19]

In a standard **3-tier architecture**, the application is distributed in three components, that usually reside on three distinct hardware/software platforms: the *database*, the *business logic* and the *presentation* (or *client*). See Figure 8.2, left.

The *database* comprises all the persistent data required to allow the business to work. The *business logic* is made up of the application program(s) that collect, check, transform, process, compute, store, retrieve the data exchanged with the clients and with the database. The *presentation* component is in charge of managing the user dialogues through a GUI, either proprietary (such as through a specific app in a smartphone) or generic (such as a web browser).

Theoretically, the database platform, generally called the *database server*, requires a specific hardware architecture comprising large, fast external data stores. The application programs will generally reside on a CPU intensive *application server*, with many-core processors and large RAM, while the *presentation* component will run on a mobile device or on a standard PC. The communication channels between the components are expected to be as fast as possible. Actually, one must distinguish the logical architecture (the three components) from the physical one (the platforms). A 3-tier architecture can be distributed on three platforms, as described above, but also on a single machine, in which case the communications are particularly fast. At the opposite, the database can be distributed and replicated among several machines, to improve the availability and security of the data.

In a **2-tier architecture**, two of these components are merged. For instance, the database component and the application programs are integrated. This integration can be understood in two ways. First, the application programs *absorb* the database component; it accesses the data directly, without the intermediary of a database server. Programs accessing SQLite databases are of this kind.[20] The second interpretation is much more interesting: the application programs *are absorbed* by the database, which becomes an **active database**. This variant of application only comprises the *presentation* and *database* components (Figure 8.2, right).

This is the architecture we will explore in this case study. We first study, in an informal and intuitive way, the sequence of actions that must be performed whenever customer orders are encoded in the database, customers execute their payments

19. https://en.wikipedia.org/wiki/Multitier_architecture
20. The SQLite DBMS, that is a function library, is said *serverless*.

and suppliers replenish out of stock items, irrespective of the technique that could be applied to implement these actions.

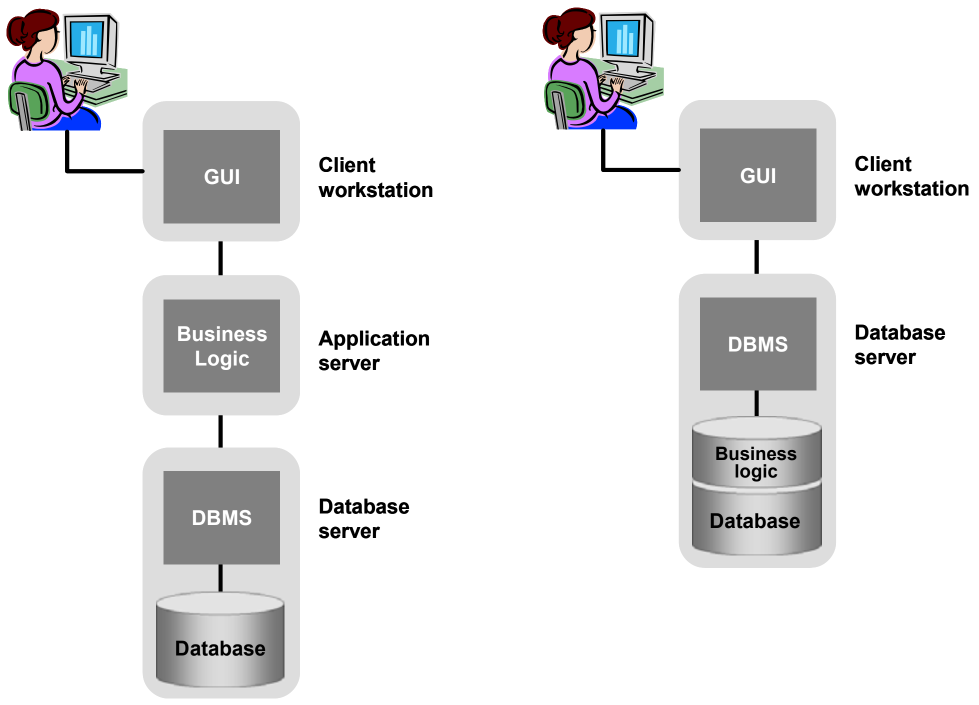


Figure 8.2 - Symbolic representation of 3-tier (left) and 2-tier (right) architecture

8.13 The static structure of the database

The short description developed above suggests that the application domain (the part of the world we are interested in) comprises seven kinds of core entities: the *customers*, the *items*, the *suppliers*, the *offers* (the items the suppliers propose to sell, with their price and delivery time), the *customer orders*, the *customer invoices* and the *supplier orders*.

The first four categories are *basic entities*, that define the intrinsic static structure of the shop. These entities evolve slowly. For example, the life span of a customer is measured in years. Although new items may appear at any time, we do not expect more than 10% of them to be introduced in the inventory each year.

The last three entity categories are *transactional entities* that describe the operations that take place in the application domain. They evolve rapidly, most of them living only two or three days.

To record data about these entities, the database comprises four *base tables* and three *transaction tables*.

8.13.1 The base tables

The *base tables* describe stable entities: the customers (CUSTOMER), the items sold by the retail shop (ITEM), the suppliers (SUPPLIER) and the offers (table OFFER).

Table CUSTOMER

Table CUSTOMER stores identification and contact data of customers (columns CustID, Name, Address and City) plus the current balance of each customer account (column Account). Once an invoice has been sent, this value is decreased by the amount of this invoice. The balance is negative if some invoices still are to be paid.[21]

```
create table CUSTOMER (
        CustID  char(10) not null,
        Name    char(32) not null,
        Address char(60) not null,
        City    char(30) not null,
        Account decimal(9,2) not null,
        primary key (CustID));
```

Table ITEM

The rows of table ITEM describe the items of the retail shop that the customers may buy.

```
create table ITEM (
       ItemID      char(15) not null,
       Description char(60) not null,
       Price       decimal(4,2) not null,
       SuppID      char(10),
       SuppPrice   decimal(4,2),
       QonHand     integer not null,
       Qord        integer default 0,
       Qavail      integer not null,
       Qeco        integer not null,
       constraint PKITEM primary key (ItemID));
```

In addition to its Id (column ItemID) and description (column Description), each item is characterized by the following properties.

21. In this small case study, we do not consider purchase *reimbursement*. Therefore, the balance will never be positive.

- Price: the current customer unit price of the item.
- SuppID: Id of the reference supplier of the item. The reference supplier is selected as the one that provides the best offer for the item (lower price and lower shipment time). This choice is re-evaluated each time the item is replenished.
- SuppPrice: the unit price requested by the reference supplier to replenish the inventory.
- QonHand: the *quantity on hand* is the quantity that **physically exists** in the inventory of the retail shop. If a customer order requests a quantity **qty** not greater than QonHand, it is immediately executed. Otherwise, the order is *pending*, waiting for the replenishment the reference supplier will execute. In both cases, QonHand is decreased by **qty**. This means that QonHand may be negative.
- Qord: the *quantity on order* is the total quantity that has been ordered to the reference supplier *but not shipped yet*. When a customer order, the quantity qty of which cannot be satisfied by the quantity on hand (qty > QonHand) nor by the future replenishment (qty > Qavail), then an order is issued for a certain quantity **mQeco** (explained below) and sent to the reference supplier. Qord is then augmented by mQeco.
- Qavail: the *available quantity* is the sum of QonHand and Qord. This is the quantity that can be consumed to execute future customer orders without the need to place a new replenishment order with the reference vendor. If a customer order requests more than the quantity on hand but no more than the available quantity (QonHand < Qty ≤ Qavail), it will be executed as soon as the pending supplier orders are executed.
- Qeco: this is a simplified variant of the *economic order quantity*, that is, the optimal quantity[22] to be ordered from the reference supplier to replenish the item inventory. Actually, the quantity **mQeco** requested from the supplier is a *multiple* of Qeco computed as follows:

  mQeco = m x Qeco, m being the smallest integer such that

  Qavail + mQeco > 0.

Table SUPPLIER

Table SUPPLIER describes the suppliers of the retail shop. It plays no role and is mainly symbolic.

22. In real inventory management, this quantity is computed so that the total cost of replenishment is minimal. See https://en.wikipedia.org/wiki/Economic_order_quantity for more detail.

```
create table SUPPLIER (
       SuppID  char(10) not null,
       Name    char(32) not null,
       City    char(30) not null,
       primary key (SuppID));
```

Table OFFER

Table OFFER describes, for each supplier (SuppID) and for each item this supplier can supply (ItemID) its unit price (Price) and its delivery time in days (Delay).

```
create table OFFER (
       SuppID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2) not null,
       Delay   integer not null,
       primary key (SuppID,ItemID),
       foreign key (SuppID) references SUPPLIER,
       foreign key (ItemID) references ITEM);
```

8.13.2The transaction tables

The *transaction tables* describe the documents processed by the employees of the retail shop: the customer orders (CUSTORDER), the invoices sent to the customers (CUSTINVOICE) and the orders sent to the suppliers (SUPPORDER).

The entity states

Before translating these entities into database structures, we examine the timeline of the life of transactional entities, from their initial recording to their closure. Their lives comprise a sequence of states.

The first state of a **customer order** is *'recorded'*. If the item requested is available in sufficient quantity (Qty ≤ QonHand), an invoice is created and the state of the order is *'invoiced'*. When the customer has made the corresponding payment, the order is in the *'closed'* state. However, if there is no sufficient quantity (Qty > QonHand), the order is in the *'pending'* state, followed, later by *'invoiced'* and *'closed'* states.

The first state of a **customer invoice** is *'recorded'*. Then, if the order is in *'invoiced'* state, the invoice is set in *'sent'* state. If the order is in *'pending'* state, the state of the invoice also is *'pending'*. When the customer has made the payment, the invoice is in the *'paid'* state.

The sequence of states of a **supplier order** is simpler: *'recorded'*, *'assigned'* when the supplier with the best offer has been identified, and *'closed'* when the order has been executed.

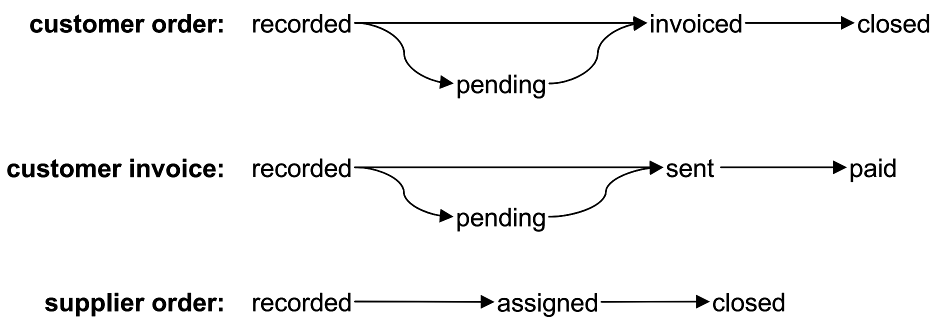


Figure 8.3 - Diagram of state transitions of transactional entities

Table CUSTORDER

Table CUSTORDER contains the data of the customer orders received so far. In addition to the Id (column OrdID) and date received (column DateOrd) of each order, its row specifies:

- CustID: the reference of the customer who placed the order.
- ItemID: the reference of the item ordered.
- Price: the unit price of the item *at the time the order has been registered*. This column is nullable; it will be updated after the row has been inserted.
- Qty: the quantity requested of the item.
- State: the current state of the order, among the following: *'recorded'*, *'invoiced'*, *'closed'*, *'pending'*.

```
create table CUSTORDER (
       OrdID   integer  not null,
       DateOrd date     not null,
       CustID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2),
       Qty     integer  not null,
       State   char(20) not null,
       primary key (OrdID),
       foreign key (CustID) references CUSTOMER,
       foreign key (ItemID) references ITEM);
```

Table CUSTINVOICE

Table CUSTINVOICE contains the data of the invoices generated for each customer order as soon as the latter has been processed. Since each invoice is attached to an order, it shares most of its data with the latter.

These data include the Id of the invoice (column InvID), the date it has been generated (column DateInv). In addition, they comprise, for each invoice:

– OrdID: the reference of the customer order with which it is associated.

– CustID: the reference of the customer who placed the order (and to whom the invoice will be [or has been] sent)

– ItemID: the reference of the item ordered.

– Price: the unit price of the item (same as in the CUSTORDER row).

– Qty: the quantity requested of the item.

– State: the current state of the invoice, among the following: *'recorded'*, *'sent'*, *'paid'*, *'pending'*.

```
create table CUSTINVOICE (
       InvID   integer  not null,
       DateInv date not null,
       OrdID   integer  not null,
       CustID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2) not null,
       Qty     integer  not null,
       Amount  integer  not null,
       State   char(20) not null,
       primary key (InvID),
       foreign key (OrdID) references CUSTORDER);
```

Table SUPPORDER

Each row of table SUPPORDER describes an order issued by the retail shop to replenish an item that is out of stock. This order is sent to the reference supplier of this item. A supplier order is sent for each customer order that cannot be executed due to insufficient quantity available (Qavail).
In addition to the Id (column OrdID) and date sent (column DateOrd) of each order, its row specify:

– CustID: the reference of the customer who placed the order.

– ItemID: the reference of the item ordered.

– SuppID, Price: the supplier Id and the unit price of the item as specified by the best offer.

– Qty: the quantity requested of the item.

– State: the current state of the order, among the following: *'assigned'*, *'closed'*.

```
create table SUPPORDER (
       OrdID   integer not null,
       DateOrd date not null,
       ItemID  char(15) not null,
       SuppID  char(10) not null,
       Price   decimal(4,2) not null,
       Qty     integer  not null,
       State   varchar(20) not null,
       primary key (OrdID),
       foreign key (SuppID,ItemID) references OFFER);
```

Script 8.52 - Schema of the last of the seven tables of the database

8.13.3The initial state of the database

New, we will observe how the data evolve when we execute a series of representative transactions according to the scenario described in the next sections. To make things concrete, we consider that the base tables contain the initial data shown in Appendix A. The subset of these data that will actually be used by the transactions are depicted in Figure 8.4.

```
CUSTOMER CustID Name    Address              City      Account
======== ====== ======= ==================== ========= =======
         ...
         C400   NEUMANN 454, Kirchenstraße   Berlin    0
         ...
         K111   JANSEN  5B, Grote Halstraat  Amsterdam 0
         ...

ITEM ItemID ...  Price SuppID SuppPrice QonHand Qord Qavail Qeco
==== ====== ==== ===== ====== ========= ======= ==== ====== ====
     ...
     PA45   ...  48    --     --        80      0    80     80
     ...

SUPPLIER SuppID Name         City
======== ====== ============ =========
         ...
         F-725  BricoMat     Paris
         ...

OFFER SuppID ItemID Price Delay
===== ====== ====== ===== =====
      ...
      D-109  PA45   35    3
      E-034  PA45   37    1
      F-725  PA45   35    1
      N-601  PA45   39    4
      ...    ...    ...   ...
```

Figure 8.4 - Initial state of OFFER table (limited to item PA45)

8.13.4The transactions

We first introduce and process some customer orders. Then, we will let the supplier ship the items requested by the retail shop. Finally, the customers will pay the invoices they have been sent.

A first customer order

This first customer order is placed by German customer C400 (*NEUMANN*). It requests 60 units of item PA45 (*Steel nails of 45 mm*). This order is encoded as row **1** in table CUSTORDER (Figure 8.5).

```
CUSTORDER OrdID DateOrd    CustID ItemID Price Qty State
========= ===== ========== ====== ====== ===== === ========
          1     2020-03-27 C400   PA45   48    60  invoiced
```

Figure 8.5 - Customer C400 has ordered 60 units of item PA45

The customer price (48) of the item has been inserted into this row. Since the quantity on hand of this item is 80 (Figure 8.4), this order **can be executed immediately**. Therefore:

– an invoice has been created in the database (Figure 8.6),

– this invoice has been printed and sent (Figure 8.7); so, its state is set to 'sent' (Figure 8.6),

– as a natural consequence, the state of the order is set to 'invoiced' (Figure 8.5),

```
CUSTINVOICE InvID DateInv    OrdID CustID ItemID Price Qty Amount State
=========== ===== ========== ===== ====== ====== ===== === ====== =====
            1     2020-03-27 1     C400   PA45   48    60  2880   sent
```

Figure 8.6 - The invoice as it has been created in the database

```
INVOICE no 1
--------------------
 CUSTOMER: C400
     Date: 2020-03-27
     Item: PA45
      Qty: 60
    Price: 48
--------------------
    Total: 2,880
```

Figure 8.7 - The invoice has been printed (simplified layout) and sent to the customer

In table ITEM, the row describing item PA45 has been updated: the quantity of the order (Qty = 60) is subtracted from the *quantity on hand* of the item (80 - 60 = 20), and from its *quantity available* (Figure 8.8).

ITEM	ItemID	...	Price	SuppID	SuppPrice	QonHand	Qord	Qavail	Qeco
	...	...	...	...	...	...	...	...	...
	PA45	...	48	--	--	20	0	20	80
	...	...	...	...	...	...	...	...	...

Figure 8.8 - The new state of table ITEM

Finally, we have to notify that the customer's account is decreased by the amount of the invoice, i.e., 2,880 (Figure 8.9).

CUSTOMER	CustID	Name	Address	City	Account
	...	...	...	...	...
	C400	NEUMANN	454, Kirchenstraße	Berlin	**-2880**
	...	...	...	...	...

Figure 8.9 - Since the order has been executed and the invoice has been sent, the account level of the customer is updated

Another customer order

We decide to process a second order, from customer K111 (*JANSEN*), that also specifies item PA45 of which it requests 35 units. Since there are only 20 units left of PA45, this order **cannot be executed**. It is created with a *pending* state. An invoice is also created with the same *pending* state (Figure 8.10).

CUSTORDER	OrdID	DateOrd	CustID	ItemID	Price	Qty	State
	1	2020-03-27	C400	PA45	48	60	invoiced
	2	2020-03-27	**K111**	**PA45**	**48**	**35**	**pending**

CUSTINVOICE	InvID	DateInv	OrdID	CustID	ItemID	Price	Qty	Amount	State
	1	2020-03-21	1	C400	PA45	48	60	2880	sent
	2	**2020-03-27**	**2**	**K111**	**PA45**	**48**	**35**	**1680**	**pending**

Figure 8.10 - Customer order n° 2 cannot be executed and is left *pending* as well as its associated invoice

The quantity on hand of item PA45 becomes negative (20 - 35 = -15), as well as its quantity available, so, we decide to replenish the stock of item PA45 by sending an order to a supplier. The quantity requested (Qord) is the *economic order quantity* of this item (Qeco = 80), which will be sufficient to replenish the stock. The question is, which supplier will be chosen? First, of course, the suppliers offering the lowest price are selected. There are two contenders, namely D-109 and F-725, both with a price of 35 (Figure 8.4). From them, we choose the offer that guarantees the shortest delivery time, F-725, to which the order is assigned. This *best offer* is translated into the data of Figure 8.11. A supplier order is created. Its components are shown in Figure 8.12.

```
ITEM ItemID ...  Price SuppID SuppPrice QonHand Qord Qavail Qeco
==== ====== ==== ===== ====== ========= ======= ==== ====== ====
     ...    ...  ...   ...    ...       ...     ...  ...    ...
     PA45   ...  48    F-725  35        -15     80   65     80
     ...    ...  ...   ...    ...       ...     ...  ...    ...
```

Figure 8.11 - 80 units of P45 have been ordered according to the best offer.

```
SUPPORDER OrdID DateOrd    ItemID SuppID Price Qty State
========= ===== ========== ====== ====== ===== === ========
          1     2020-03-27 PA45   F-725  35    80  assigned
```

Figure 8.12 - The supplier order has been assigned to supplier F-725

About next customer orders

If the next customer order specifies a quantity **qty** of PA45 not greater than 65 units, it could be executed once the supplier order is executed (theoretically the next working day according to the data of Figure 8.4). In this case, there is no need to issue another supplier order. On the contrary, if this quantity is greater than 65 a new supplier order of 80 units, so that Qord is increased to 160.

On the supplier side

Now, let us go and see what happens on the supplier side once the first supplier order has been received (that depicted in Figure 8.12). This order asks supplier F-725 to replenish item PA45. When this request is satisfied, 80 units of this item are sent to the retail shop to replenish the PA45 stock.

The ITEM row is changed as follows (Figure 8.13):

– the quantity on hand is now QonHand = -15 + 80 = 65 units

– the quantity ordered is reset to 0

– the available quantity does not change (65).

```
ITEM ItemID ...  Price SuppID SuppPrice QonHand Qord Qavail Qeco
==== ====== ==== ===== ====== ========= ======= ==== ====== ====
     ...    ...  ...   ...    ...       ...     ...  ...    ...
     PA45   ...  48    F-725  35        65      0    65     80
     ...    ...  ...   ...    ...       ...     ...  ...    ...
```

Figure 8.13 - Item PA45 has been replenished

The supplier order has been successfully processed and can be closed (Figure 8.14).

```
SUPPORDER OrdID DateOrd    ItemID SuppID Price Qty State
========= ===== ========== ====== ====== ===== === ======
          1     2020-03-27 PA45   F-725  35    80  closed
```

Figure 8.14 - The supplier order is closed

Item PA45 has been replenished, what next?

Now, we go back to the retail shop, where the second customer order can be executed, with the following consequences (Figure 8.15):

– its *pending* invoice is sent to the customer,

– the customer order is declared invoiced ,

– the Account value of customer K111 is changed to -1680.

```
CUSTINVOICE InvID DateInv    OrdID CustID ItemID Price Qty Amount State
=========== ===== ========== ===== ====== ====== ===== === ====== =====
            1     2020-03-27 1     C400   PA45   48    60  2880   sent
            2     2020-03-27 2     K111   PA45   48    35  1680   sent
```

```
CUSTORDER OrdID DateOrd    CustID ItemID Price Qty State
========= ===== ========== ====== ====== ===== === ========
          1     2020-03-27 C400   PA45   48    60  invoiced
          2     2020-03-27 K111   PA45   48    35  invoiced
```

```
CUSTOMER CustID Name    Address              City      Account
======== ====== ======= ==================== ========= =======
         ...    ...     ...                  ...       ...
         C400   NEUMANN 454, Kirchenstraße   Berlin    -2880
         ...    ...     ...                  ...       ...
         K111   JANSEN  5B, Grote Halstraat  Amsterdam -1680
         ...    ...     ...                  ...       ...
```

Figure 8.15 - Processing pending customer orders and invoices after item replenishment

Finally, customer C400 pays his invoice

The last operation to consider is the responsibility of the customer: paying for the materials she has been delivered. Let us examine the case of customer C400, who ordered, and took away, 60 units of item PA45, and for which he has to pay an amount of 2,880. When this payment has been executed, the employee of the retail shop encodes it *in the computer*. This operation triggers three data modifications (Figure 8.16):

– the invoice state is set to 'paid'

– the customer order is closed

– and the account level of the customer is increased (becomes less negative) by the amount of the payment.

```
CUSTINVOICE InvID DateInv    OrdID CustID ItemID Price Qty Amount State
=========== ===== ========== ===== ====== ====== ===== === ====== =====
            1     2020-03-27 1     C400   PA45   48    60  2880   paid
            2     2020-03-27 2     K111   PA45   48    35  1680   sent

 CUSTORDER OrdID DateOrd    CustID ItemID Price Qty State
 ========= ===== ========== ====== ====== ===== === ========
           1     2020-03-27 C400   PA45   48    60  closed
           2     2020-03-27 K111   PA45   48    35  invoiced

 CUSTOMER CustID Name    Address              City      Account
 ======== ====== ======= ==================== ========= =======
          ...    ...     ...                  ...       ...
          C400   NEUMANN 454, Kirchenstraße   Berlin    0
          ...    ...     ...                  ...       ...
```

Figure 8.16 - Registering the payment of the first invoice

This scenario is great for intuitively understanding the principles of the mechanisms we will have to implement. However, to translate them into code, we would need a more precise definition of the operations. In order not to make this presentation too cluttered, we have moved these definitions to Appendix B.

8.14 Implementation of the classical application architecture

We can ignore the standard ancillary functions for registering, modifying and deleting data of basic tables (CUSTOMER, ITEM, SUPPLIER and OFFER). On the one hand, they do not depend on the application architecture and, on the other hand, their code is quite simple. So, we can concentrate the discussion on the components and algorithms of the standard 3-tier architecture.

The GUI comprises three modules that collect the data required to register a *customer order*, a *customer payment* and the execution of *supplier orders* to replenish the item stock of the retail shop. These modules are implemented by three SQLfast scripts (Figure 8.17):

– Register-CUSTORDER-GUI.sql (Script 8.53 and Figure 8.18),

– Register-PAYMENT-GUI.sql,

– Replenish-ITEM-GUI.sql.

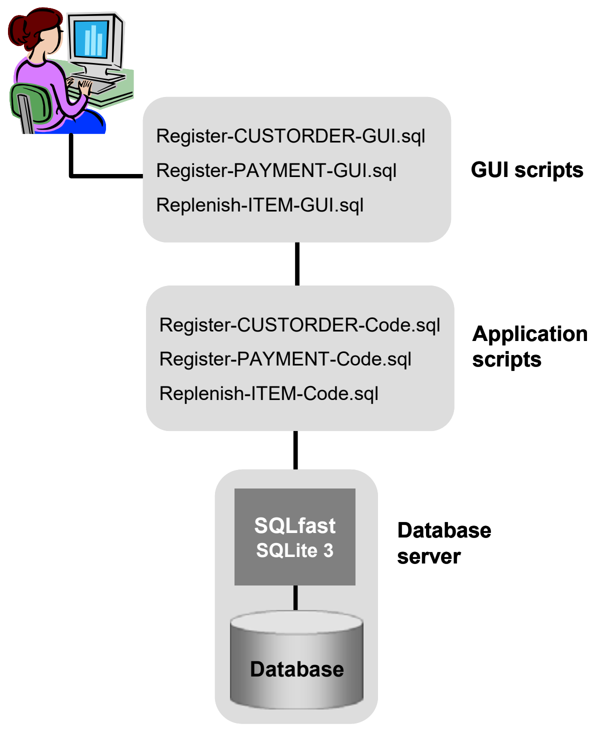


Figure 8.17 - The standard implementation of the 3-tier architecture

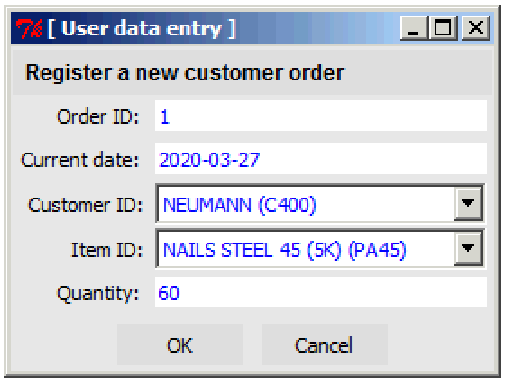
```
extract oid = select coalesce(max(OrdID),0)+1 from CUSTORDER;
set dat = $date$;
ask-u oid,dat,cus,itm,qty = [/bRegister a new customer order]
          Order ID:
         |Current date:
         |Customer ID:[select Name||' ('||CustID||')',CustID
                       from   CUSTOMER order by CustID]
         |Item ID:[select Description||' ('||ItemID||')',ItemID
                   from   ITEM order by ItemID]
         |Quantity:;
execSQL Register-CUSTORDER-Code.sql;
```

Script 8.53 - Dialogue box to register a new customer order (simplified version)

The first script computes the next order Id (or 0 for the very first order), extracts the current date, execute the dialogue box, then call application script RETAIL-Register-CUSTORDER-Code.sql that implements in a straightforward way the business logic developed in Section 8.24.2 of Appendix B. Considering the emphasis of this document on active database implementation, the code of Register-CUSTORDER-

Code.sql is not shown here but can be examined in directory SQLfast\Scripts\Case-Studies\Case_Active_DB\Standard-Version.

The other GUI modules and application scripts are coded in a similar way.



Register a new customer order

Order ID:	1
Current date:	2020-03-27
Customer ID:	NEUMANN (C400)
Item ID:	NAILS STEEL 45 (5K) (PA45)
Quantity:	60

Figure 8.18 - Collecting the data to register a new customer order

8.15 Implementation as an active database

The basic philosophy of the standard architecture is to decompose a complex task into simpler tasks, that typically translates into a hierarchy of procedures in which parent procedures call their children procedures for execution.

The principle of active databases is quite different. It consists in creating events that send some kind of messages to target objects materialized as rows in the tables.[23] For example, when the payment of an invoice has been received, a message is sent to this invoice by changing its state from 'sent' to 'paid'. As soon as the invoice receives this message (by catching the State update event), it closes its order and updates the account of its customer. In other words, each object is responsible for the consequences of its evolution.

The architecture specific to our case study is shown in Figure 8.19. The GUI comprises three scripts quite similar to those of the standard architecture. The only difference is that, instead of calling complex application scripts, they just send messages to database objects by executing elementary data modification queries.

8.16 Extending the data structures

The main components of our active database are the **triggers** that are responsible for applying the rules of the business logic elaborated in Appendix B.[24] Our discus-

23. Somewhat similar to object-oriented and agent-based approaches.

sion will of course focus on their development. However, there is some opportunities to alleviate their work by automating some checking and computing operations through `default`, `check` and `generated` clauses. Some of them are shown in Script 8.54.

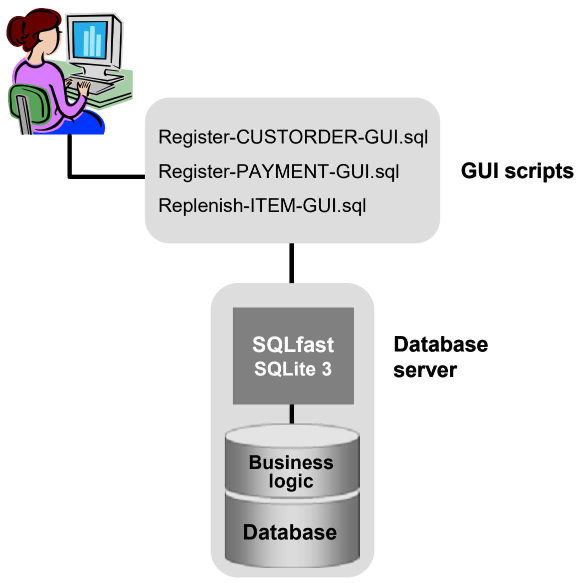


Figure 8.19 - The 2-tier architecture implementation as an active database

8.17 The user interface

The three transaction operations, namely *registering a customer order*, *registering the payment of a customer invoice* and *replenishing the stock of an item* by a supplier are carried out through the same dialogue boxes that collect data from the user. Then, they use them to update the database:

– *Registering a customer order*: a new row is inserted in CUSTORDER table from the data collected from the user (Script 8.55).

– *Registering a payment*: the user provides the Id of an invoice, the state of which is set to 'paid' (Script 8.56).

– *Replenishing the stock of an item*: the user identify all the *assigned* (that is, not *closed* yet) supplier orders of an item to a supplier. The state of this order is set to 'closed' to indicate that the quantity ordered has been shipped (Script 8.57).

24. *Stored procedures* can also be used to encapsulate some critical rules. However, since the DBMS on which our case study will be built does not offer these constructs, they will be ignored.

```
create table CUSTOMER (
        ...,
        Account decimal(9,2) not null default 0.0,
        ...);
create table ITEM (
       ...,
       Qavail  integer not null
               generated always as (QonHand+Qord),
       ...);
create table CUSTORDER (
       ...,
       State   char(20) not null default 'recorded',
       ...,
       check(Qty > 0));
create table CUSTINVOICE (
       ...,
       Amount  integer  not null
               generated always as (Qty*Price),
       State   char(20) not null default 'recorded',
       ...);
create table SUPPORDER (
       ...,
       State   varchar(20) not null default 'recorded',
       ...);
```

Script 8.54 - Augmenting table structures with *default*, *check* and *generated* clauses

```
ask-u oid,dat,cus,itm,qty = <see Script 8.53>;
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values ($oid$,'$dat$','$cus$','$itm$',$qty$);
commitDB;
```

Script 8.55 - Dialogue box to register a new customer order (simplified version)

```
ask inv = [/bRegister a payment]
    Invoice:[select InvID||': '||CustID||', '||DateInv,InvID
             from   CUSTINVOICE
             where  State = 'sent' order by InvID];
update CUSTINVOICE
set    State = 'paid'
where  InvID = $inv$;
commitDB;
```

Script 8.56 - Dialogue box to register the payment of an invoice (simplified version)

```
ask itm = [/bSelect an item to replenish]
           Item:[select distinct ItemID from SUPPORDER
                 where  State = 'assigned' order by ItemID];
ask sup = [/bSelect a supplier of "$itm$"]
           Supplier:[select distinct SU.SuppID||' ('||SU.Name
                                   ||')' ,SU.SuppID
                     from   SUPPORDER SO,SUPPLIER SU
                     where  SO.SuppID = SU.SuppID
                     and    ItemID = '$itm$'
                     and    State = 'assigned'
                     order by SU.SuppID];
update SUPPORDER
set    State = 'closed'
where  SuppID = '$sup$' and ItemID = '$itm$'
and    State = 'assigned';
commitDB;
```

Script 8.57 - Dialogue box to execute supplier orders (simplified version)

The actions of the user interface merely consist in sending messages to objects (i.e., table rows). They are symbolized in Figure 8.20.

$$\begin{tikzcd}
\textbf{Register-CUSTORDER-GUI} \arrow[r, "\text{insert}"] & \textbf{CUSTORDER} \\
\textbf{Register-PAYMENT-GUI} \arrow[r, "\text{State = 'paid'}"] & \textbf{CUSTINVOICE} \\
\textbf{Replenish-ITEM-GUI} \arrow[r, "\text{State = 'closed'}"] & \textbf{SUPPORDER}
\end{tikzcd}$$

Figure 8.20 - The GUI scripts send messages through data modification events

Now, the real work is performed by triggers that react to these messages by carrying out the data processing operations implementing the business tasks.

8.18 The transactions

Now we examine how the logic of the three main transactions are translated into a trigger system.

8.18.1 Registering a customer order

When a CUSTORDER row has been inserted in the database, the first question is, does the stock contain enough of the item to execute this order (**Case 1** in Appendix B, Section 8.24.2), or not (**Case 2** in the same section)?

We find it clearer to create a trigger for each branch of this alternative (Script 8.58).

```
create trigger TRG_CORD_INS1
after insert on CUSTORDER
when new.Qty <= (select QonHand from ITEM
                 where  ItemID = new.ItemID)
begin ... end;

create trigger TRG_CORD_INS2
after insert on CUSTORDER
when new.Qty > (select QonHand from ITEM
                where  ItemID = new.ItemID)
begin ... end;
```

Script 8.58 - Controlling CUSTORDER insert operations - First attempt

Beware of side effects!

As a general rule, the conditions of the `when` clauses of these triggers (let us call them **c1** and **c2**) must be *exclusive* and *covering*:

– exclusive: **c1** and **c2** = False

– covering: **c1** or **c2** = True

In other words, when a new CUSTORDER row is inserted, one and only one of these triggers will fire. Unfortunately, as they are expressed in Script 8.58, these triggers may not always execute as expected. The analysis of Section 8.24.2 shows that, in **Case 1** and **Case 2**, the value of QonHand of the referenced ITEM row is decreased by that of Qty of the order. Indeed, we read, in both cases:

- update referenced ITEM row **it**:
  subtract co.Qty from it.QonHand

Let us consider that an `insert` occurs when condition Qty ≤ QonHand is met. The database engine selects and fires one of the `after insert` triggers that meet this condition, that is, TRG_CORD_INS1. During the execution of its body, this trigger decreased the value of QonHand of the referenced ITEM row. When this execution is finished, the database engine examines one of the other `after insert` triggers, i.e., TRG_CORD_INS2, and evaluates its condition.

This is where the aforementioned problem may arise: if, initially, Qty ≤ QonHand < 2 * Qty, then the first trigger assigns to QonHand a new value such that QonHand <

Qty, which means that the **`when` condition of the second trigger is also met**. Therefore, both triggers will fire!

The pattern of the problem is easy to identify: the body of TRG_CORD_INS1 modifies the values used in the `when` clause of TRG_CORD_INS2, in such a way that conditions **c1** and **c2** no longer satisfy the *exclusive* condition.

This problem can be solved in two ways. First we can encapsulate all the different cases in a single (large and complex) trigger. Secondly, we refine the `when` conditions in such a way that they remain exclusive, whichever the order in which they are executed. Let us choose the last solution.

We examine the operations of **Case 1** and **Case 2** and we observe that both create an invoice. So we modify the `when` condition as follows: each trigger will fire if the `when` condition of Script 8.58 is met and *no invoice has been created yet*. This means that, whatever the order in which the triggers are considered for execution by the SQL engine, only the first one will fire.

Now, the triggers of Script 8.59 work as expected.

Important remark

The behavior of the trigger system of a database depends on the execution model implemented by the DBMS. Theoretically, when an event occurs, the SQL engine identifies all the candidate triggers that this event may fire. From them, only those that satisfy the `when` condition will actually fire. This leads to two questions: in which order will these triggers fire and when will their `when` condition be evaluated?

Most DBMS allow the database designer to specify this order, in a deterministic way, through various means. Some others, SQLite being one of them, leave the order undefined so that the designer must resort to some adhoc trick to force a definite execution order. This is what we have done by checking the absence of a dependent CUSTINVOICE row.

The question of the `when` condition is more delicate. According to a first model, all the conditions of the candidate triggers are evaluated before any of them fires. In this model, there is no side effect and the *exclusive* property of conditions Case 1 and Case 2 is guaranteed by construction. In another model, the condition of the first candidate trigger is evaluated, and if *True*, its body is executed. After that, the SQL engine consider another trigger, evaluates its condition, and, if *True*, executes its body. This process goes on until all the candidate triggers have been examined. So, the value of the condition of a trigger depends on the execution of the triggers executed before it, which leads to the undesirable side effect described in this section. The execution model of SQLite is of this kind.

```
create trigger TRG_CORD_INS1
after insert on CUSTORDER
when new.Qty <= (select QonHand from ITEM
                 where  ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where  OrdID = new.OrdID)
begin ... end;

create trigger TRG_CORD_INS2
after insert on CUSTORDER
when new.Qty > (select QonHand from ITEM
                where  ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where  OrdID = new.OrdID)
begin ... end;
```

Script 8.59 - Controlling CUSTORDER insert operations - Final version

Local variables

Triggers of CUSTORDER tables will use four local variables. They are simulated by technical table V_CORDER:

```
create table V_CORDER(Price,Qavail,mQeco,InvID);
```

Controlling CUSTORDER row in Case 1

Script 8.60 shows the code of **TRG_CUSTORDER_INS1**, the trigger responsible for the propagation of data modification due to the insertion of a new CUSTORDER according to **Case 1.**, that is, when the quantity on hand is sufficient to immediately execute the order. It uses two local variables: Price, the customer unit price of the item, and InvID, the Id of the next invoice. It comprises five queries:

1. The first query (`insert`) stores in V_CORDER the current values of the local variables.
2. The next query (`update`) subtract the quantity ordered from the quantity on hand.
3. Then, the CUSTORDER row just inserted is completed (`update`) with the unit price of the item.
4. The fourth query creates the invoice (`insert`) and initializes its state to 'send'. This latter setting is a message that instructs the target row to execute a series of actions deriving from its new state. These actions are specified in Script 8.64.
5. Finally, the local variables are reset.

```
create trigger TRG_CUSTORDER_INS1
after insert on CUSTORDER
for each row
when new.Qty <= (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   insert into V_CORDER(Price,InvID)
       select Price,
              (select coalesce(max(InvID),0)+1 from CUSTINVOICE)
       from   ITEM where ItemID = new.ItemID;
   update ITEM
   set   QonHand = QonHand - new.Qty
   where ItemID = new.ItemID;
   update CUSTORDER
   set    Price = (select Price from V_CORDER)
   where  OrdID = new.OrdID;
   insert into CUSTINVOICE(InvID,DateInv,OrdID,CustID,ItemID,
                           Price,Qty,State)
       values ((select InvID from V_CORDER),new.DateOrd,
                new.OrdID,new.CustID,new.ItemID,
               (select Price from V_CORDER),new.Qty,'sent');
   delete from V_CORDER;
end;
```

Script 8.60 - Trigger that controls the creation of a customer order when the quantity on hand is sufficient

Controlling CUSTORDER row in Case 2

The task of trigger **TRG_CUSTORDER_INS2** is to cope with the cases of insufficient quantity on hand, according to the rules of **Case 2.** It uses four local variables: Price, the customer unit price of the item, Qavail the quantity available without issuing a new supplier order, mQeco, the quantity that will be needed to replenish the stock of the item (the smallest multiple of Qeco such that Qavail will become positive), and InvID, the Id of the next invoice. It comprises the following queries (Script 8.61):

1. The first query (`insert`) stores in V_CORDER the current values of the local variables.
2. The next query (`update`) subtract the quantity ordered by the customer from the quantity on hand (QonHand) of the ITEM row.
3. A block of three queries is executed *if the quantity available is not sufficient.*[25]
   - The quantity to order from the supplier (mQeco) is added to the quantity on order (Qord) of the ITEM row.

25. For the current SQLfast version, that uses the SQLite DBMS, this conditional block must be transformed according to the rules of Section 8.2.7.

- A best offer for the item is identified if none has been selected yet (`SuppID is null`) and notified (`update`) in the ITEM row.
- A supplier order is created by inserting a row in SUPPORDER table.

```
create trigger TRG_CUSTORDER_INS2
after insert on CUSTORDER
for each row
when new.Qty >  (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   insert into V_CORDER(Price,Qavail,mQeco,InvID)
       select Price,Qavail,
              Qeco*(truncate(1.*(new.Qty-Qavail)/Qeco)+1),
              (select coalesce(max(InvID),0)+1 from CUSTINVOICE)
       from   ITEM where ItemID = new.ItemID;
   update ITEM
   set   QonHand = QonHand - new.Qty
   where ItemID = new.ItemID;
   if (new.Qty > (select Qavail from V_CORDER));
      update ITEM
      set   Qord   = Qord + (select mQeco from V_CORDER)
      where ItemID = new.ItemID;
      update ITEM
      set   (SuppID,PriceSupp) = (select SuppID,Price from OFFER
                                  where   ItemID = new.ItemID
                                  order by Price,Delay limit 1)
      where ItemID = new.ItemID and SuppID is null;
      insert into SUPPORDER(OrdID,DateOrd,ItemID,SuppID,
                            Price,Qty,State)
          select  (select coalesce(max(OrdID),0)+1 from SUPPORDER),
                   new.DateOrd,new.ItemID,SuppID,PriceSupp,
                  (select mQeco from V_CORDER),'assigned'
          from    ITEM
          where   ItemID = new.ItemID;
   endif;
   update CUSTORDER
   set    Price = (select Price from V_CORDER),
          State = 'pending'
   where  OrdID = new.OrdID;
   insert into CUSTINVOICE(InvID,DateInv,OrdID,CustID,ItemID,
                           Price,Qty,State)
       values ((select InvID from V_CORDER),new.DateOrd,
                new.OrdID,new.CustID,new.ItemID,
               (select Price from V_CORDER),new.Qty,'pending');
   delete from V_CORDER;
end;
```

Script 8.61 - Trigger that controls the creation of a customer order when the quantity on hand is **not** sufficient

4. Then, the CUSTORDER row just inserted is completed (`update`) with the unit price of the item. In addition, its state is set to 'pending'.
5. An invoice is created by inserting a row in table CUSTINVOICE. Its state is also that of the customer order: 'pending'.
6. Finally, the local variables are reset.

8.18.2Executing a supplier order

Script 8.57 shows that the execution of all the orders of a definite item sent by the retail shop to a supplier is asked for by sending closing messages to the corresponding SUPPORDER rows, i.e., by changing their state from 'assigned' to 'closed'. The reaction consists in adjusting the quantities of the referenced ITEM row: the value of Qord of the order is added to the quantity of hand (QonHand) of the item and subtracted from its quantity on order (Qord) (Script 8.62).

```
create trigger TRG_SUPPORDER_UPD
after update of State on SUPPORDER
for each row
when new.State = 'closed'
begin
   update ITEM
   set   QonHand = QonHand + Qord,
         Qord    = Qord - new.Qord
   where ItemID = new.ItemID;
end;
```

Script 8.62 - Execution of a supplier order that (partially) replenish the stock of an item

When the last supplier order has been executed, the value of Qord of the ITEM row falls to 0. Setting the quantity on order to zero is a message to the ITEM row telling it that its pending invoices can now be processed. This is implemented in the trigger of Script 8.63. It comprises two queries:

1. The first one changes the state of pending invoices to 'sent'.
2. The second one re-evaluates the best offer for this item.

So, by changing the state of pending invoices, this trigger sends them a message to tell them that they can now be sent to their customers. The reaction is coded in trigger CUSTINVOICE_UPD1 of Script 8.64.

This trigger comprises three operations:

1. Calling procedure writeFile that prints (in a text file) the invoice to be sent to the customer. This procedure uses two arguments, the name of the text file and the character string to write in it. In the latter argument, symbol `'@n'` denotes a *new line* and function thousandSep formats a numeric character string by inserting thousands separators ('1234567' becomes '1,234,567').

```
create trigger TRG_ITEM_UPD
after update of Qord on ITEM
for each row
when new.Qord = 0
begin
   update CUSTINVOICE
   set   State = 'sent'
   where ItemID = new.ItemID
   and   State = 'pending';
   update ITEM
   set   (SuppID,PriceSupp) = (select SuppID,Price from OFFER
                               where  ItemID = new.ItemID
                               order by Price,Delay limit 1)
   where ItemID = new.ItemID;
end;
```

Script 8.63 - When the stock of an item is fully replenished, its pending customer orders and invoices can be processed

```
create trigger TRG_CUSTINVOICE_UPD1
after insert, update of State on CUSTINVOICE
for each row
when new.State = 'sent'
begin
   writeFile('D:/SQLfast/Files/INVOICE-'||new.InvID||'.txt',
             'INVOICE no '||new.InvID
           ||'@n--------------------'
           ||'@n CUSTOMER: '||new.CustID
           ||'@n     Date: '||new.DateInv
           ||'@n     Item: '||new.ItemID
           ||'@n      Qty: '||new.Qty
           ||'@n    Price: '||new.Price
           ||'@n--------------------'
           ||'@n    Total: '
           ||thousandSep(cast(new.Qty*new.Price as char),',');
   update CUSTORDER set State = 'invoiced'
   where OrdID = new.OrdID;
   update CUSTOMER set Account = Account - new.Amount
   where CustID = new.CustID;
end;
```

Script 8.64 - Sending the invoice to the customer of an order

2. Setting the state of the (pending) order of the invoice to 'invoiced'.
3. Subtracting the amount of the invoice from the account balance of the customer.

8.18.3Registering the payment of an invoice

According to Script 8.56, the registration of the payment of an invoice is triggered by the sending of a message to this invoice. This message is created by changing its status to 'paid'. When fired, trigger CUSTINVOICE_UPD2 closes the order of this invoice then updates the account balance of the customer (Script 8.65).

```
create trigger TRG_CUSTINVOICE_UPD2
after update of State on CUSTINVOICE
for each row
when new.State = 'paid'
begin
   update CUSTORDER
   set   State = 'closed'
   where OrdID = (select OrdID from CUSTINVOICE
                  where InvID = new.InvID);
   update CUSTOMER
   set   Account = Account + new.Amount
   where CustID = new.CustID;
end;
```

Script 8.65 - This invoice has been paid

8.18.4Event architecture of the active database

The graph of Figure 8.21 extends that of Figure 8.20. It shows the flow of messages that control the cascade of operations triggered by the three user interactions: *placing a customer order*, *paying an invoice* and *replenishing an item stock*. We observe that some tables appear more than once in this graph. Indeed, the messages sent by a table depend on the messages it receives. So, merging table nodes would reduce the information content of the graph.

8.19 Verification of an active database

The set of triggers in a database can make up a complex system in that the activation of one trigger can fire other triggers. So, it is not unusual that the first executions of an active database either crash or produce unexpected results.

Hence the need for specific techniques to analyze a trigger system to identify as early as possible flaws in its architecture. There are two main families of analysis techniques: static and dynamic. Static analysis techniques rely on the visual or automated examination of the source code of the triggers. Dynamic techniques extract and analyze information on the behavior of the triggers at run time.

We will examine informally some of these techniques.

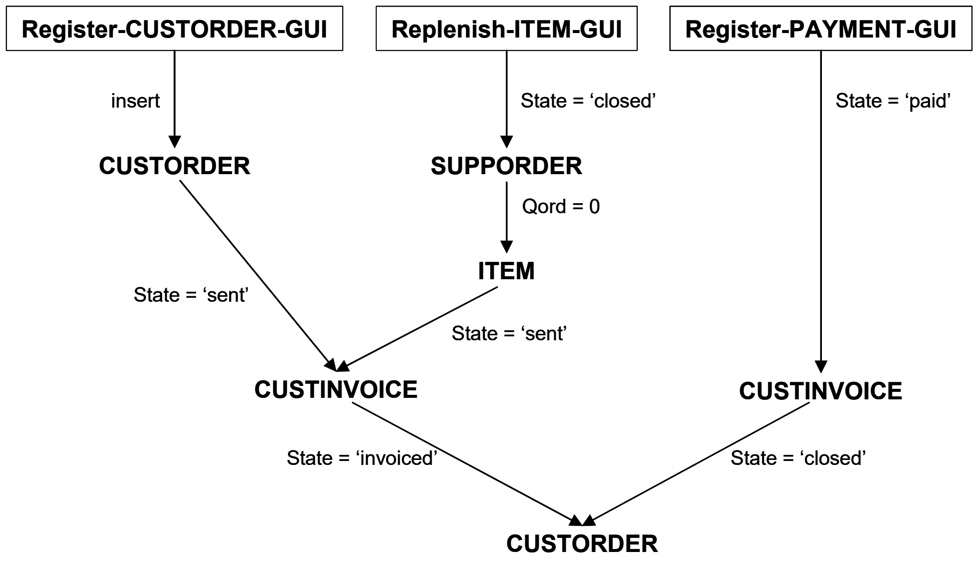


Figure 8.21 - The graph of the messages controlling the business application

8.19.1 Static analysis: checking trigger circuits

One of the major concerns of active database development is whether the execution of the trigger system being built will *terminate* whatever the initial event that triggers it. *Infinite execution* may occur when the code includes some form of *recursivity*, either in the explicit form of recursive triggers or when a trigger T executes data modifications that start a chain of modifications that ultimately fires T itself.

Let us consider directed graph **F** defined as follows. Each node represents a trigger and each arc (**tp**,**tq**) indicates tells that the body of **tp** includes modification queries that *may* fire trigger **tq**. If graph **F** is *acyclic*, that is, it *includes no circuit*, then the all the possible executions of the trigger system are ensured to terminate in a finite number of steps (lesser or equal to the number of triggers). If, on the contrary, **F** includes one or more circuits, then there is a risk of infinite execution.

Hence the importance of checking the acyclicity of **F**. If the test fails, we must identify the circuits of **F** for in-depth examination. These processes are carried out by two algorithms that are developed and illustrated in Appendix C.

The trigger system of the RETAIL.db database is depicted in Figure 8.22. A simple visual inspection shows that it is acyclic, so that there is no risk of infinite execution, whatever the initial data.

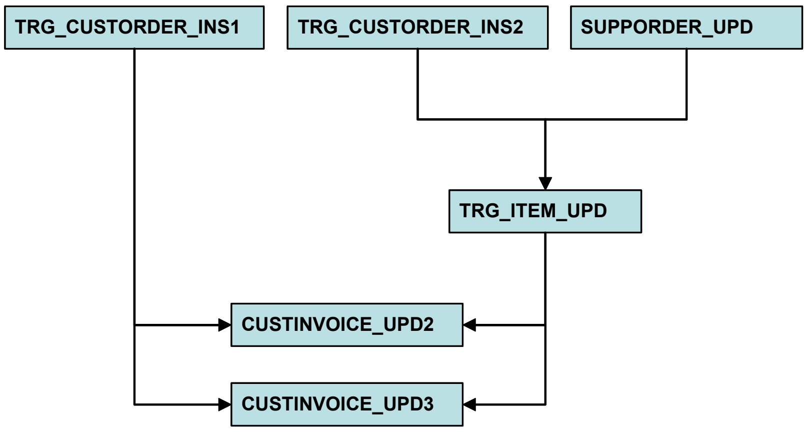


Figure 8.22 - The trigger graph of the RETAIL database

8.19.2Dynamic analysis: tracing the execution of the active database

Trigger tracing techniques consists in extracting, storing and analyzing information on the execution of a trigger system. This information constitutes the trace of this execution. By examining the traces produced for a representative set of initial data, we can derive some important properties of these triggers. In particular, such traces can show us when and where the triggers exhibit unexpected behavior.

An in-depth description of tracing techniques in the SQLfast environment has been presented in Chapter 23 (*Aid to SQLfast script development*) of the SQLfast manual. In this chapter, Section 23.7 is devoted to a short introduction to *tracing trigger execution*. The reader is referred to this chapter before proceeding with this reading.

We will describe two complementary categories of traces: *event-based* and *operation-based*. Both rely on UDF functions provided by the SQLfast language that can be invoked from within trigger body. They are specific to SQLite but their generalization to other DBMS is straightforward.[26]

About the concept of *process level* in trigger tracing

Before describing these types of trace, we must refine the concept of *process level* introduced in Chapter 23 of the SQLfast manual and used in the tracing of SQLfast code. The process level specifies the depth of a process[27] in the call hierarchy at run time. If process **P**, with level **k** calls script **Q**, thus creating a process of **Q**, then the level of the latter process is **k**+1. The main script is at level 0.

26. All the more since big DBMS already include tracing features that are missing in SQLite.

The process level associated with each trace of a statement is stored in column Tlevel of table TRACES. To trace trigger executions, we also need to define some form of process level. We consider each trigger activation as a process. When an event created by a statement of a process (of a script or of a trigger) with level **k** is caught by a trigger, the activation of the latter is said to have level **k**+1.

In script execution, the process level if provided by system variable processLevel. However, this variable is not available from within a trigger execution, so that we will simulate it as a global variable that must be controlled *manually*. Since it must be available by all the triggers, it is translated into elementary table TLEVEL comprising a single column, Level, and a single row (see Section 8.2.7). Its definition and management are specified as shown in Script 8.66.

```
create table TLEVEL as select 0 as Level;
...
create trigger TRG_CUSTORDER_INS1
...
begin
   update TLEVEL set Level = Level + 1;
      ...
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.66 - Implementation of the process level in triggers

Event-based tracing

Event-based tracing reports on the events that fire triggers. For each event, it generates a short description of the SQL data modification query and the start and end of the execution of the body of the triggers that fired. This is done by the instrumentation of the code of the body with procedure writeMessage(<message>) that writes character string <message> in the output window.[28] A summary of the event is crafted from states old and new of the current row. The expected result, produced by the execution of trigger TRG_CUSTORDER_INS1, is shown in Figure 8.23.

It must be noticed that, in addition to the six functional triggers `TRG_CUSTORDER_INS1`, `TRG_CUSTORDER_INS2`, `TRG_ITEM_UPD` (renamed `TRG_ITEM_UPD2`), `TRG_CUSTINVOICE_UPD1`, `TRG_CUSTINVOICE_UPD2` and `TRG_SUPPORDER_UPD` (see Figure 8.22), we have created six auxiliary triggers (`TRG_CUSTOMER_UPD`, `TRG_CUSTORDER_UPD`, etc.) the sole objective of which is

27. A process is not the same thing as a script. It is the execution of a script. In a recursive execution, in which, for example, script A calls script B, which itself calls script A, this execution includes two processes of A and one process of B.
28. Remember that this procedure is implemented as a UDF function. Due to SQLite's limitations regarding trigger body syntax, the call to this function is made in the form `$message$`.

to write *we were here* in the trace. These additional triggers catch the events created by the functional triggers but that do not fire any other functional trigger.

For each trigger activation, the trace comprises four sections:

- the text of the event (`insert into CUSTORDER ...`),
- a label marking the start of the body, together with the identification of the trigger (> `Enter TRG_CUSTOMER_INS1`),
- for each query of the body that fires a trigger, and for each row affected by this query, the trace of this trigger; this trace is indented *wrt* its parent trace to make the parent-child hierarchy explicit,
- a label marking the end of the body (< `Exit TRG_CUSTOMER_INS1`).

The trace of Figure 8.23 shows that the recording of a customer order, initiated by event "`insert into CUSTORDER` ...)", fires trigger `TRG_CUSTORDER_INS1`. The execution of its body creates four events that fire successively triggers `TRG_ITEM_UPD1`, `TRG_CUSTORDER_UPD`, `TRG_CUSTINVOICE_INS` and `TRG_CUST INVOICE_UPD2`. The execution of the latter then fires triggers `TRG_CUST INVOICE_UPD2` and `TRG_CUSTOMER_UPD`.

```
EVENT: insert into CUSTORDER (1,'2020-08-30','C400',
                              'PA45',--,60,'recorded')
  > Enter TRG_CUSTORDER_INS1
    EVENT: update ITEM set QonHand = 20
      > Enter TRG_ITEM_UPD1
      < Exit TRG_ITEM_UPD1
    EVENT: update CUSTORDER set Price = 48
      > Enter TRG_CUSTORDER_UPD
      < Exit TRG_CUSTORDER_UPD
    EVENT: insert into CUSTINVOICE (1,'2020-08-30',1,'C400',
                                    'PA45',48,60,2880,'recorded')
      > Enter TRG_CUSTINVOICE_INS
      < Exit TRG_CUSTINVOICE_INS
    EVENT: update CUSTINVOICE set State = 'sent'
      > Enter TRG_CUSTINVOICE_UPD2
        EVENT: update CUSTORDER set State = 'invoiced'
          > Enter TRG_CUSTORDER_UPD
          < Exit TRG_CUSTORDER_UPD
        EVENT: update CUSTOMER set Account = -2880
          > Enter TRG_CUSTOMER_UPD
          < Exit TRG_CUSTOMER_UPD
      < Exit TRG_CUSTINVOICE_UPD2
  < Exit TRG_CUSTORDER_INS1
```

Figure 8.23 - Event trace of the creation of a customer order when the quantity ordered is available

The trace has been generated by the specific instrumentation of trigger TRG_CUSTORDER_INS1 (in blue in Script 8.60). The indentation of child traces within the trace of their parent is produced by the expression "`(select repeat(' ',Level*4) from TLEVEL)`"[29] that creates a space string from the process level stored in TLEVEL.

```
create trigger TRG_CUSTORDER_INS1
...
begin
   update TLEVEL set Level = Level + 1;
   -- Tracing --------------------------------------------
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
               ||'EVENT: insert into CUSTORDER ('
               ||new.OrdID||','
               ||''''||new.DateOrd||''''||','
               ||''''||new.CustID||''''||','
               ||''''||new.ItemID||''''||','
               ||'--'||','
               ||new.Qty||','
               ||''''||new.State||''''||')');
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
               ||'  > Enter TRG_CUSTORDER_INS1');
   -- ----------------------------------------------------
   ...
   -- Tracing --------------------------------------------
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
               ||'  < Exit TRG_CUSTORDER_INS1');
   -- ----------------------------------------------------
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.67 - Instrumentation of trigger TRG_CUSTORDER_INS1 for event tracing

Operation-based tracing

The operation-based tracing of triggers is an extension of the tracing of scripts described in Chapter 23 of the SQLfast manual. It consists in inserting in table TRACES of database TRACING.db the information on the execution of each query of the trigger bodies.

In the same way as we did to implement event-based tracing, we will instrument the body of the triggers we want to control through procedure `writeTrace(idx, tim,pro,lev,nat,raw,prep)`.[30] If we insert this procedure just before each SQL query, we will store a description of this query similar to those of pure SQLfast statements. In Script 8.68, procedure **writeTrace** will store in table TRACES the state of query "`update ITEM ... where ItemID = new.ItemID`" of trigger `TRG_CUST_ORDER_INS1` before its execution. We observe that:

– the value of column Tindex is set *manually*,

29. Function `repeat(s,n)` produces a character string formed by **n** occurrences of string **s**.
30. That will appear as `$trace$(...)` in SQLfast scripts.

- the name of the procedure (column Tproc) is built as the concatenation of the database name and the trigger name; this combination is unique,
- the process level (Tlevel) is drawn from column Level of table TLEVEL,
- the nature (Tnature) of the query is 0 (native elementary),
- the prepared version of the query (PrepStat) is built from the column values of the new state of the current row.

When processed by procedure writeTrace, the value of Tlevel will be augmented by the value of system variable procLevel. In this way, the activation of a trigger will appear in the trace as the calling of a procedure. In a procedure at level **k**, a trigger fired by a data modification query will appear at level **k**+1. So, the trace of the triggers will be seamlessly integrated in that of the SQLfast statements.

```
create trigger TRG_CUSTORDER_INS1
after insert on CUSTORDER
for each row
when new.Qty <= (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   update TLEVEL set Level = Level + 1;
   insert into V_CORDER(Price,InvID) ...;
   writeTrace(5,current_timestamp_full(),
            'RETAIL.db/TRG_CUSTORDER_INS1',
            (select Level from TLEVEL),0,
            'update ITEM set QonHand = QonHand - new.Qty
             where ItemID = new.ItemID',
            'update ITEM set QonHand = QonHand - ' || new.Qty||'
             where ItemID = ''' ||new.ItemID|| '''',1);
   update ITEM
   set    QonHand = QonHand - new.Qty
   where  ItemID = new.ItemID;
   update CUSTORDER ...;
   insert into CUSTINVOICE ...;
   delete from V_CORDER;
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.68 - Writing the trace of query "`update ITEM`" in trigger `TRG_CUSTORDER_INS1`

Instrumenting triggers for event-based or operation-based tracing may be felt particularly convoluted. This is true. However, the good news is that the structure of the tracing statements appears to be quite systematic, in such a way that they can be automatically generated in the trigger code from the information provided by dictionary tables SYS_TRIGGER_FULL and SYS_TRIGGER_COMP_FULL. The code of Script 8.69 is a skeleton of such a generator. Completing it is fairly easy and is left as an exercise.

```
set db = RETAIL.db;
openDB Dictionary-of-$db$;
for trid,taid,tanam,trnam,trwh,trev,trcol,trcon
   = [select TrigID,TableID,TableName,TrigName,TrigWhen,
             TrigEvent,TrigCol,TrigCond
      from   SYS_TRIGGER_FULL order by TrigID];
   if ('$trcol$' = '')
      write-b trigger $trnam$ $trwh$ $trev$ on $tanam$;
   if ('$trcol$' <> '')
      write-b trigger $trnam$ $trwh$ $trev$ of $trcol$
              on $tanam$;
   write begin;
   for seq,exp = [select StatSeq,StatExpr
                  from SYS_TRIGGER_COMP_FULL
                  where  TrigID = $trid$ order by StatSeq];
      if (startswith(lower('§exp§'),'select raise')) next;
      write @S2 writeTrace($seq$,...,$db$/$trnam$,1,0,...);;
      write @S2 $exp$;;
   endfor;
   write end;;
endfor;
```

Script 8.69 - Automating the instrumentation of the triggers of the RETAIL.db database

8.19.3 Replaying transactions

Let us consider a sequence of transactions such as those of the scenario of Section 8.13.4: registering two customer orders, then replenishing the stock of an item and finally registering the payment of a customer. Let us also consider that we want to execute them again. There can be several reasons for this:

- a bug has been found and fixed, then we want the four transactions being carried out again,
- we want to show newly hired employees how the application works,
- if the RETAIL application we have developed actually is just a playable prototype for a future, full scale, application, this scenario (and its effect on the database) is a part of the specifications[31] of the latter,

31. The specifications of an application is a document that defines precisely the goals of the application, without anticipating the way it will be implemented [short and incomplete definition!]

– we find this scenario to be an excellent test case, to be used to verify that the future versions of the application work as expected.

What we want is called *replaying* these transactions. In this context, the architecture of the RETAIL application in the form of an active database presents an interesting property. Since starting each interaction between a user and the application translates into a single data modification query, collecting then running these queries allow us to *replay* a series of transactions.

For example, our scenario results from the execution of the GUI scripts detailed in Section 8.17. These scripts have translated the user requests into the four queries recalled in Script 8.70. So, if we execute the latter script, we replay this scenario exactly, provided the database is reset in the same initial state of course.

```
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values (1,'2020-03-27','C400','PA45',60);
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values (2,'2020-03-27','K111','PA45',35);
update SUPPORDER set State = 'closed'
where  SuppID = 'F-725' and ItemID = 'PA45'
and    State = 'assigned';
update CUSTINVOICE set State = 'paid' where InvID = 1;
```

Script 8.70 - Replaying the scenario of Section 8.13.4

The nice aspect of this exercise is that such replay scripts can be automatically generated. By enabling the tracing of the SQLfast scripts, limited to SQL queries (for this we check button Trace SQL statements only), these queries are stored into the TRACES table. We just extract them by the simple query of Script 8.71 to build the replay script.

```
select PrepStat from TRACES where Tproc like '%-GUI';
```

Script 8.71 - Extracting the SQL queries for replaying a scenario

8.20 Limits of this case study

It is worth mentioning some of the simplifying assumptions which our active database is based on.

– The suppliers are idealized and abstract entities: their stock of items are infinite, they are absolutely reliable and they never require to be paid. In addition, their item catalogue is the same as that of the retail shop (notably same Id and same description).

– Then, the customers are quite liable and trusted people: they always come and take their items away, they never cancel their orders and they always pay their invoices in due time.
– The shop and supplier employees never make reading, encoding, typing or operation errors.
– The IT infrastructure and the application programs never crash.
– The best offer selection policy guarantees that only one supplier at a time is asked to replenish the stock of an item.

Coping with these limitations to build a really usable application would require hundreds of code pages. This is left as an exercise!

8.21 Conclusions, history and extensions

In this case study, we have tried to illustrate two outstanding properties of active databases in general and triggers in particular: their power and the wide range of their applications.

It is interesting to note that triggers have been available for several decades in the early versions of most DBMS. One of their first use was to implement referential integrity maintenance well before foreign keys were available. This use is now deprecated unless one wants to implement non standard foreign key behavior.

From deductive databases to triggers

The concept of ECA (Event-Condition-Action) rule, or active rule, comes from the domain of *artificial intelligence* as it was perceived in the eighties.[32] The objective was to describe in a declarative way the dynamic evolution of knowledge. Another source of the concept is *logic programming*, where knowledge is represented by a collection of raw facts plus a set of derivation rules, possibly recursive. These concepts have been introduced in database models and technology to form the family of *deductive databases*. Some prototype implementations have been developed in the eighties but due to complex logic problems and low performance, none have really met industrial success. Most have disappeared in the nineties.[33]

32. In these years, statistical analysis of large data sets still was considered a domain distinct from that of A.I.

33. Datalog is another attempt to couple databases and logic programming. It is an *extended subset* of the Prolog language (https://en.wikipedia.org/wiki/Datalog). Many Datalog implementations still are available, either as standalone interpreters or as language extensions (e.g. PyDatalog, a Python module).

In some sense, active databases (mainly based on SQL views and triggers) are the remains of the historical deductive database dream. Instead of relying on a built-in generic deductive engine (similar to that of Prolog interpreters) the database designer now must build, from scratch, for each database, an adhoc deductive engine through which new data and operations are derived from base data and queries. Of course, implementing this engine with triggers often proves a non trivial task for large applications.

Recursive programming

Managing and exploiting *recursive data structures* can be done with recursive queries but also through recursive triggers. This is a point that we have not addressed in this study though it would deserve a full chapter in its own. Several application examples are developed in the SQLfast manual, Chapter 19 (*Recursive programming*).

Modeling and design

Developing large business applications has long been supported by a hierarchy of models and design processes.[34] To develop a database, the standard approach starts by building an abstract, technology independent conceptual schema, which is then translated into a specific technology, such as a relational or NoSQL DBMS. Solid models and design processes also exist to help develop the programs and the GUI of business applications.

Despite many attempts in the nineties,[35] there are few such comprehensive guidelines to create active databases.

We have also shown that a trigger system (the set of triggers attached to a database) is more difficult to model and to design than standard 3-tiers systems. Obviously, there is no single approach to design them.

– *Integrity maintenance* and *redundancy control* can be modeled by a set of equations for which we identify the events that may disturb them. For each of them, we specify the reaction that restores the equation.

– Building a *temporal database* will require quite different reasoning based on the consistency of a temporal sequence of database states (see the case studies on *Temporal Databases*).

– For *business applications* implementation where business rules are moved from the programs to the database, we could think of an approach inspired by

34. Look for example at the UML ecosystem that offers more than a dozen specific models to describe the different aspects of complex, interactive, data intensive, applications.

35. In particularly the IDEA approach to information system design, the modeling language of which, named Chimera included ECA rules. See Ceri S. and Fraternali, P. *The Story of the IDEA Methodology*, 1996, Springer [https://link.springer.com/content/pdf/10.1007/3-540-63107-0_1.pdf].

object-oriented modeling methods. We identify in the database representative objects classes (implemented as tables or aggregates of tables), with which we associate attributes and methods. These methods are activated by sending messages to objects through events.

Finally, what may active databases be good for?

Active databases generally are known for implementing non trivial integrity properties, that is, those that cannot be controlled by the standard uniqueness, referential and not null constraints nor by check predicates.

We have shown that the triggers of a databases can also be used to *silently* perform useful ancillary functions that simplify the task of application programmers. Redundancy management, updatable views, logging data changes and managing temporal data are some of the most useful applications.

All these examples share a common goal, showing that, far beyond being a mere data store, a database can be a very powerful knowledge management system.[36]

However, implementing full size business applications as active databases, that is, moving business rules from application programs to the database, is a quite different matter!

The experiment described in Section 8.15 suggests that such an enterprise is realistic, at least for small size applications. However, whether this architecture will prove better (and according to which criteria) than the standard one is, at least, questionable.

There are a number of arguments against trigger based architecture.

– Developing and understanding the code of a trigger system may prove more complex than that of a standard architecture. Indeed, there exist since long a rich variety of models and methods intended to help developers analyze, specify, verify and implement complex application programs. (Almost) none of them is able to usefully support the development of trigger-based technology. More in this in the next section.

– Active databases technology relies on strongly intricated programming paradigms, namely procedural, logic and event-based, in addition to the complexity of the RDBMS data structures. Many programmers, otherwise proficient in classical development languages (e.g., Java, Swing and ODBC) are not familiar with these concepts.

– Trigger body languages generally lack the resources that usually are available in standard programming languages. For instance, there is no easy way to execute a GUI dialogue, to catch an exception, to control the current transaction or use some API.

36. In addition, gathering and storing in a unique, secure, place (the database schema) a set of rules that are common to many applications contribute to the integrity of the data.

- There is no unique syntax nor execution model for triggers. Though the trigger concept is understood by most DBMS, its implementation may considerably vary, in particular concerning the way conflict situation are resolved. We have already discussed this issue in Section8.18.1. This means that a non trivial trigger system cannot be ported to another DBMS without an in-depth revision.

Reference

Paton, N., W. and Díaz, O. Active Database Systems, *ACM Computing Surveys*, Vol. 31, No. 1, March 1999, pp. 63-103 [*https://www.academia.edu/7023990/Active_database_systems*]

One of the most comprehensive study on active databases. Includes a very large bibliography.

8.22 The implementation

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Active_DB**.

They can be run from main script **ActiveDB-Main.sql**, that displays the selection box of Figure 8.24.

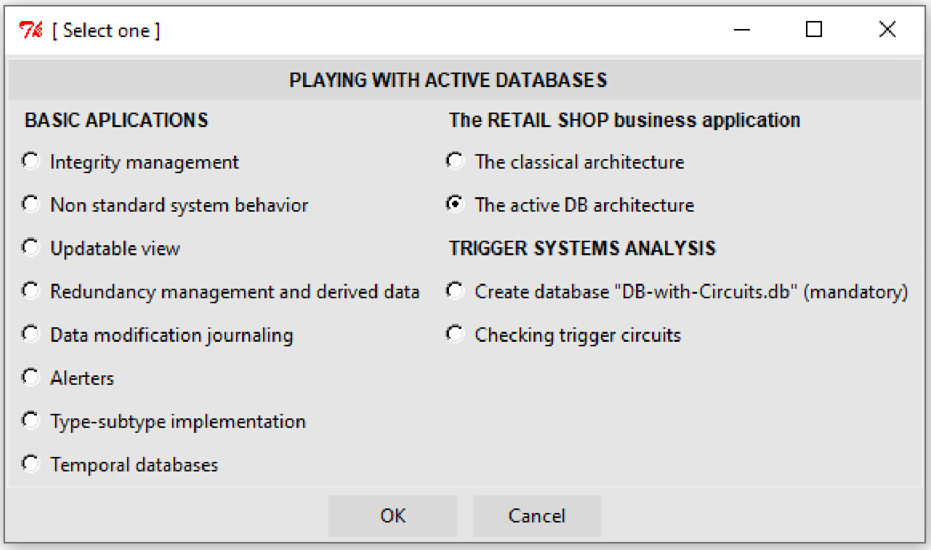

[Select one]
PLAYING WITH ACTIVE DATABASES
BASIC APLICATIONS
Integrity management
Non standard system behavior
Updatable view
Redundancy management and derived data
Data modification journaling
Alerters
Type-subtype implementation
Temporal databases
The RETAIL SHOP business application
The classical architecture
The active DB architecture
TRIGGER SYSTEMS ANALYSIS
Create database "DB-with-Circuits.db" (mandatory)
Checking trigger circuits
OK
Cancel

Figure 8.24 - Selection of an Active DB appplication

The last action, **Checking trigger circuits**, evaluates the triggers of two databases, namely RETAIL.db, created by the RETAIL Shop application (active DB version) and DB-with-Circuits.db, created by action **Create database "DB-with-Circuits.db"**.

Action **The active DB architecture** opens the selection box of Figure 8.25. It shows four categories of operations:

– **INITITIALIZATION**: creating the RETAIL.db database, ditto with event tracing, ditto with query tracing, load initial data in the base tables.

– **RETAIL OPERATIONS**. The operations to execute by the retail shop employee: creating a customer, creating an item, updating an item, recording customer order data and recording the payment of a customer.

– **SUPPLIER OPERATIONS**. The operations to execute by the employee(s) of the supplier(s): creating a supplier, adding an offer, updating an offer, and replenishing an item at the retail shop.

– **SHOW DATA**. Show the content of a table. Print a selected invoice.

These are the operations of interest to observe the behavior of the application. Other operations (e.g., updating and deleting elements) can be easily added.

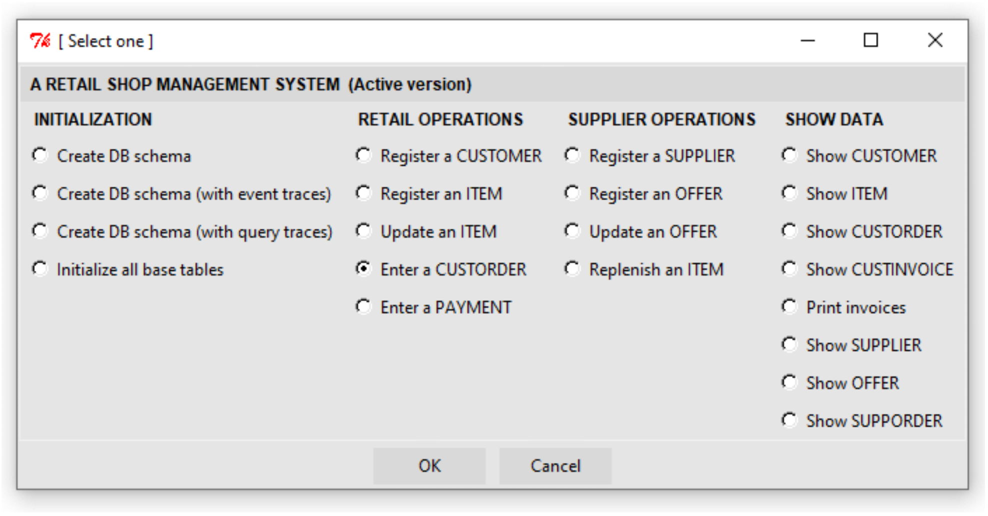


Figure 8.25 - The main menu of the retail shop application (active DB version)

Grouping the retail shop and the supplier(s) operations in a single selection box is quite artificial, as is a single work station were shared by the employees of all the actors.

We suggest a more natural organization, in which the retail shop and each supplier are assigned their own application. Its implementation is easy:

- Split the main script functions into two versions: ActiveDB-Shop-Main.sql and ActiveDB-Supplier-Main.sql.
- In the SQLfast directory, create as many copies of file SQLfast.exe as there are suppliers (named, for instance, SQLfast_D-109.exe, SQLfast_E-034.exe, etc.)
- The original file will be run by the shop employee, who then executes the first main script. Each supplier employee runs its own copy of SQLfast.exe and execute the supplier main script. These applications run in parallel and access the same database.

This is a good way to experiment with parallel execution of scripts!

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

8.23 Appendix A - The initial state of the database

The initial content of base tables is depicted in Figures 8.26 to 8.29. Observations:

- In table CUSTOMER, no unpaid invoice, so, Account = 0.
- In table ITEM, no quantity on order (Qord = 0, therefore QonHand = Qavail) and the inventory has just been replenished (QonHand = Qeco). No best offer selected yet.
- Table OFFER is fairly large (29 rows!). Since we will concentrate on item PA45, we have shown the offers of this item only.

CUSTOMER	**CustID**	**Name**	**Address**	**City**	**Account**
	B062	TAYLOR	139, Elm Park	Atlanta	0
	B112	WEBER	127, Hauptstraße	Berlin	0
	B332	MORETTI	1604, via Cavour	Roma	0
	B512	ROBERTS	108, Baker Street	London	0
	C003	WRIGHT	27, Portobello Road	London	0
	C123	SMITH	95B/2, Park Street	Atlanta	0
	C400	NEUMANN	454, Kirchenstraße	Berlin	0
	D063	SMITH	1083, King's Road	London	0
	F010	SCHMIDT	56, Bahnhofplatz	Berlin	0
	F011	JONES	10, Downing Street	London	0
	F400	LOPEZ	54B, Calle Carretas	Madrid	0
	K111	JANSEN	5B, Grote Halstraat	Amsterdam	0
	K729	EDWARDS	5, Abbey Road	London	0
	L422	GARCIA	40, W. Lake Avenue	Atlanta	0
	S127	JOHNSON	2164, Fifth Avenue	Atlanta	0
	S712	DUBOIS	58, avenue Montaigne	Paris	0

Figure 8.26 - Initial state of the CUSTOMER table

ITEM	**ItemID**	**...**	**Price**	**SuppID**	**SuppPrice**	**QonHand**	**Qord**	**Qavail**	**Qeco**
	CS262	...	70	--	--	40	0	40	40
	CS264	...	85	--	--	125	0	125	125
	CS464	...	125	--	--	100	0	100	100
	PA45	...	48	--	--	80	0	80	80
	PA60	...	55	--	--	60	0	60	60
	PH222	...	105	--	--	160	0	160	160
	PS222	...	94	--	--	110	0	110	110

Figure 8.27 - Initial state of ITEM table

SUPPLIER	SuppID	Name	City
	D-109	Wood&Steel	Berlin
	E-034	Golden Key	London
	F-725	BricoMat	Paris
	N-601	Materia-2000	Amsterdam
	U-542	Smith&Son	Atlanta

Figure 8.28 - Initial state of SUPPLIER table

```
OFFER SuppID ItemID Price Delay
===== ====== ====== ===== =====
      D-109  CS262  45    3
      D-109  CS264  68    2
      D-109  CS464  102   4
      D-109  PA45   35    3
      D-109  PA60   36    2
      D-109  PH222  165   5
      D-109  PS222  60    3
      E-034  CS262  40    3
      E-034  CS264  73    2
      E-034  CS464  90    1
      E-034  PA45   37    1
      E-034  PA60   38    2
      E-034  PH222  160   1
      E-034  PS222  72    3
      F-725  CS262  51    1
      F-725  CS264  62    4
      F-725  PA45   35    1
      F-725  PA60   38    3
      F-725  PH222  168   4
      N-601  CS262  42    1
      N-601  CS264  75    4
      N-601  CS464  85    5
      N-601  PA45   39    4
      N-601  PA60   30    3
      N-601  PH222  155   4
      U-542  CS262  40    6
      U-542  CS264  62    2
      U-542  CS464  88    3
      U-542  PS222  85    2
```

Figure 8.29 - Initial state of OFFER table

The transaction tables are empty.

8.24 Appendix B - RETAIL application: task analysis

Before coding our application, whatever architecture is chosen, we must identify the tasks to implement and precisely specify their effect on the database.

We will consider a limited set of representative tasks to be performed by the employees of the retail shop and of the suppliers: registering the data of new basic entities, registering a customer order, executing a supplier order and registering a customer payment. Other data management tasks, such as updating the contents of base tables, would be quite simple to analyze and implement.

8.24.1 Register basic entities

Registering a customer, an item, a supplier and an offer pose no particular problems. We just have to take care of the initial value of some critical attributes:

– The Amount column of the new **CUSTOMER** row is set to 0.

– For each new **ITEM** row, columns SuppID and SuppPrice, that define the best offer, are left undefined (i.e., *null*). Columns QonHand and Qavail are set to the same non negative value and Qord is set to 0. The values of Price (customer unit price) and of Qeco (the economic order quantity) are specified by the retail shop manager.

8.24.2 Register a customer order

The first operation consists in inserting a new row in table CUSTORDER. Columns OrdID, DateOrd, CustID and ItemID are extracted from the registering dialogue box. The value of Price is null and that of State is 'recorded'.

Let **co** be the CUSTORDER row just inserted, **it** the ITEM row referenced by co.ItemID and **cu** the CUSTOMER row referenced by co.CustID.

We consider two main cases, depending on the value of co.Qty compared to that of QonHand in the referenced ITEM row.

– **Case 1**. co.Qty ≤ it.QonHand: there is enough *quantity on hand*, **the order can be executed.**

- update ITEM row **it**:

  subtract co.Qty from it.QonHand

  (Qavail, being a generated column, is automatically updated)

- update CUSTORDER row **co**:

  set co.Price = it.Price

  set co.State = 'invoiced'

- insert row **ci** in table CUSTINVOICE:

set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
(co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price, 'sent')

- generate an invoice document from the data of row **ci**
- update CUSTOMER row **cu**:
  subtract ci.Amount from cu.Account

– **Case 2**. co.Qty > it.QonHand: there is not enough *quantity on hand*, **the order cannot be executed.**
Considering whether the *quantity on order* can suffice to satisfy the customer order leads us to two subcases.

– **Case 2.1**. co.Qty ≤ it.Qavail : there is enough *quantity available*, **the order can be executed** as soon as the quantity on order (it.Qord) is delivered by the reference supplier.

- update referenced ITEM row **it**:
  subtract co.Qty from it.QonHand
  subtract co.Qty from it.Qavail
- update CUSTORDER row **co**:
  set co.Price = it.Price
  set co.State = 'pending'..
- insert row **ci** in table CUSTINVOICE:
  set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
  (co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price, 'pending')

– **Case 2.2**. co.Qty > it.Qavail : there is not enough *quantity available*, **a new order must be sent to the reference supplier**.

- compute mQeco, the total quantity to order:
  mQeco = smallest multiple of it.Qeco such that new value of it.Qavail is non negative
- update referenced ITEM row **it**:
  subtract co.Qty from it.QonHand
  add mQeco to it.Qord
  subtract co.Qty from it.QonHand then add mQeco
- if order **co** is the first one for item **it**, select best offer and update it.SuppID and it.PriceSupp
- insert row **so** in table into SUPPORDER:
  set so.(OrdID, DateOrd, ItemID, SuppID, Price, Qty, State) =
  (<next value of SUPPORDER.OrdID>, co.DateOrd, it.ItemID, it.SuppID,

it.SuppPrice, mQeco, 'assigned')

- update CUSTORDER row **co**:

  set co.Price = it.Price

  set co.State = 'pending'.

- insert row **ci** in table CUSTINVOICE:

  set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
  (co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price, 'pending')

8.24.3 Register a customer payment

Through the dialogue box, the employee of the retail shop specifies an invoice Id. Let **ci** be the corresponding CUSTINVOICE row, **co** the CUSTORDER row referenced by ci.OrdID and **cu** the CUSTOMER row referenced by ci.CustID.

- update CUSTINVOICE row **ci**:

  set ci.State = 'paid'

- update CUSTORDER row **co** referenced by ci:

  set co.State = 'closed'

- update CUSTOMER row **cu** referenced by ci:

  add ci.Amount to cu.Account

8.24.4 Execute a supplier order

An employee of a supplier decides to execute supplier orders from the retail shop. Through the dialogue box, the employee selects a couple of Id's (sup, itm) that identifies the set of SuppOrder rows where SupplierID = **sup**, ItemID = **itm** and State = 'assigned' (which means *still awaiting processing*). It is important to note that the reference supplier does not change between two replenishments. Consequently, all the (supplier) orders for an item are sent to the same supplier.[37]

let **it** be the ITEM row where ItemID = **itm**

let **totQord** the sum of Qord of the SUPPORDER rows
where SupplierID = **sup**, ItemID = **itm** and State = 'assigned'

- update ITEM row **it**:

  add totQord to it.QonHand'

  set it.Qord = 0

37. Allowing orders to be sent to different suppliers before the next replenishment would make the rules more complex.

- recalculate the best offer (result: OFFER row denoted by **bo**) and update ITEM row **it**:

  set it.SuppID = bo.SuppID

  set it.SuppPrice = bo.Price
- update SUPPORDER rows **so**
  where SupplierID = **sup**, ItemID = **itm** and State = 'assigned':

  set so.State = 'closed'
- update CUSTORDER rows **co** that reference **it** and where State = 'pending':

  set co.State = 'invoiced'
- for **ci** in CUSTINVOICE rows that reference **it** and where State = 'pending':

  let **cu** be the CUSTOMER row referenced by ci:

  - update CUSTOMER row **cu**:

    subtract ci.Amount from cu.Account
- generate an invoice document from the data of each CUSTINVOICE row **ci** that references **it** and where State = 'pending'
- update CUSTINVOICE rows **ci** that reference **it** and where State = 'pending'

  set co.State = 'sent'

8.25 Appendix C - Circuit detection in directed graphs

Avoiding infinite executions of a trigger system is not an easy problem. In this section, we will suggest some graph analysis algorithms that can help us identify directed graphs to detect and identify circuits.

Let us consider event **e0** that fires trigger **t0**, the body of which includes data modification queries. If the latter create events that are caught by other triggers, initial event **e0** starts a chain of data modification operations that can be of any length. This is what Figure 8.30 illustrates. When fired through an `insert into R` query, trigger TRG_R_INS executes two data modification queries that fire triggers TRG_S_DEL and TRG_T_UPD. These triggers, if their body also include data modification queries, will create events that are likely to fire other triggers, and so on.

This event-trigger pattern creates a *graph* in which nodes represent triggers and directed arcs represent the direct firing relation between two triggers. The initial question on trigger termination reduces to the nature of this graph. If the graph is *acyclic*, that is, if it contains no circuits, we are sure that, once started, the trigger system will terminate. If the graph contains circuit(s), then its execution could cause termination problems. Checking whether a directed graph is acyclic is a standard graph theory problem.[38] Solving it is particularly simple and intuitive. However, finding these circuits may be a bit more difficult.

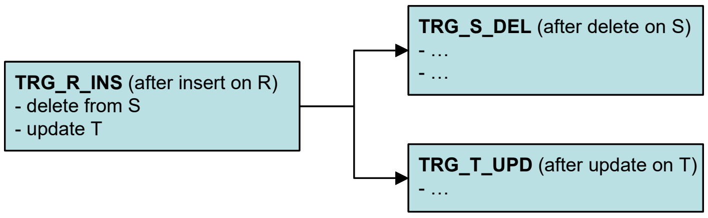

TRG_R_INS (after insert on R)
- delete from S
- update T
TRG_S_DEL (after delete on S)
- …
- …
TRG_T_UPD (after update on T)
- …

Figure 8.30 - When fired through an *insert into R* query, trigger TRG_R_INS executes data modification queries that fire triggers TRG_S_DEL and TRG_T_UPD

To study the acyclicity of trigger graphs in a practical way, we will consider the small schema of Script 8.72, that creates database DB-with-Circuits.db. This schema comprises the definitions of five tables and five triggers.

To derive the trigger graph from these definition, we will rely on the system tables of the dictionary. Among them, tables SYS_TRIGGER and SYS_TRIGGER_COMP include all the information needed to create this graph. These tables have been described in detail in Chapter 20 of the SQLfast Manual (*Metadata - Walking on the wild side*). As a reminder, table SYS_TRIGGER comprises the information of the header of triggers:

– TrigID: unique trigger id in the schema

38. It is addressed in two other case studies: *The book of which you are the hero* and *From data bulk loading to database book writing.*

- TableID: id of the parent table
- TrigName: name of the trigger
- TrigWhen: when the trigger will fire (`before`, `after`, `instead of`)
- TrigEvent: the event that fires the trigger (insert, delete, update)
- TrigCol: if the event is 'update', the list of columns (none for a general update)
- TrigScope: the scope or granularity of the trigger (`statement`, `row`)
- TrigCond: the condition of the `when` clause.

System table SYS_TRIGGER_COMP describes the statements of the body part of the trigger:

- TrigStatID: unique id of the statement in the schema
- TrigID: id of the parent trigger
- StatSeq: sequence number of the statement in the trigger body
- StatTable: the name of the table processed by the statement
- StatOper: the operation performed by the statement (`insert`, `delete`, `update`, others)
- StatCol: in case of `update` operation, the list of columns updated
- StatExpr: the integral text of the statement.

Chapter 20 also suggests that examining and processing the system tables may be more comfortable when their data are permanently stored into an independent database. This dictionary database can be created by script Create-persistent-dictionary.sql, available in the directory of this case study. Its name is formed as follows: 'Dictionary-of-' + <current database name>. Let us create it for our illustration database, with name Dictionary-of-DB-with-Circuits.db.

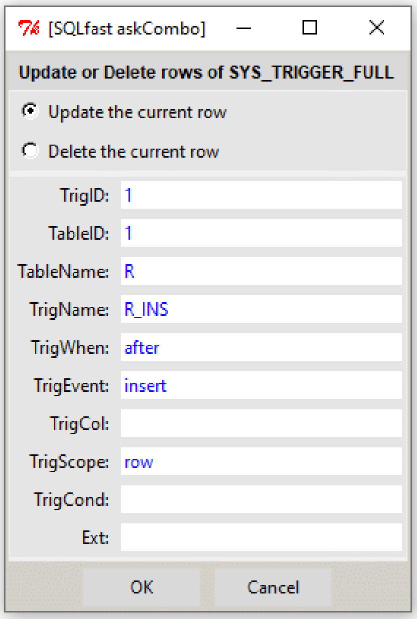
We observe that this dictionary database comprises additional tables the name of which is postfixed by '_FULL'. They are extensions of other system tables in order to ease their examination and their processing. In particular:

- table SYS_TRIGGER_FULL comprises new column TableName (name of the parent table)
- table SYS_TRIGGER_COMP_FULL comprises two new columns, TrigName (name of the trigger) and TableID (id of the trigger parent table).

Figure 8.31 shows the description of trigger R_INS (left) and of its second statement (right) in these extended system tables. The trigger has been assigned TrigID = 1. The id of the other triggers are shown in the right side of Script 8.72.

```
createOrReplaceDB DB-with-Circuits.db;
create table R (R1 int,R2 int,R3 int);
create table S (S1 int,S2 int,S3 int);
create table T (T1 int,T2 int,T3 int);
create table U (U1 int,U2 int,U3 int);
create table W (W1 int,W2 int,W3 int);
create trigger R_INS after insert on R                    [1]
    begin insert into S values(1,2,3);
          update T set T2=2,T3=3; end;
create trigger S_INS after insert on S                    [2]
    begin delete from U; end;
create trigger T_UPD after update on T                    [3]
    begin update T set T1=1;
          update W set W3=3; end;
create trigger U_DEL after delete on U                    [4]
    begin insert into S values(1,2,3);
          update W set W1=1; end;
create trigger W_UPD after update of W3 on W              [5]
    begin update W set W2=2; end;
closeDB;
```

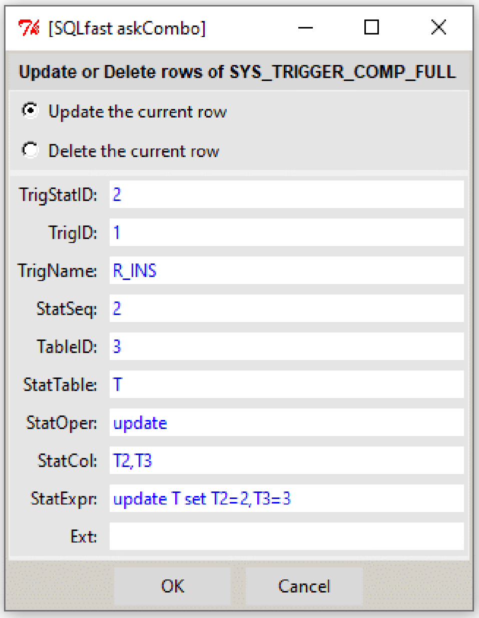
Script 8.72 - A small database that ***may*** include trigger circuits



Update or Delete rows of SYS_TRIGGER_FULL

[✓ Update the current row]
Delete the current row

TrigID:	1
TableID:	1
TableName:	R
TrigName:	R_INS
TrigWhen:	after
TrigEvent:	insert
TrigCol:	
TrigScope:	row
TrigCond:	
Ext:	



Update or Delete rows of SYS_TRIGGER_COMP_FULL

[✓ Update the current row]
Delete the current row

TrigStatID:	2
TrigID:	1
TrigName:	R_INS
StatSeq:	2
TableID:	3
StatTable:	T
StatOper:	update
StatCol:	T2,T3
StatExpr:	update T set T2=2,T3=3
Ext:	

Figure 8.31 - Description of trigger R_INS and of its second body component "update T set T2=2,T3=3"

Before deriving and processing the trigger graph, we will apply a little modification in table SYS_TRIGGER_FULL in order to make the information more regular. When the event of the trigger is a general `update`, that specifies no columns, it implicitly specifies all the columns of the parent table. The code of Script 8.73 complete column TrigCol of all triggers fired by a general `update` event.

```
update SYS_TRIGGER_FULL
set    TrigCol = (select group_concat(ColName,',')
                  from   SYS_COLUMN
                  where  TableID = SYS_TRIGGER_FULL.TableID)
where  TrigEvent = 'update' and TrigCol = '';
```

Script 8.73 - SYS_TRIGGER_FULL is completed with "TrigCol" of general "update"

Now, let us build the trigger graph. It will be stored in table BINARY as binary relation (TrigID1, TrigID2) where TrigID1 denotes a trigger and TrigID2 a trigger that the former is likely to fire. The content of this table is based on a simple syntactic analysis of the code of the triggers. Practically, row (t1,t2) is inserted into table BINARY if one of these conditions is met:

1. (1) the body of trigger t1 comprises a statement `insert into X` and (2) trigger `t2` is fired by any event of the form `insert on X`,
2. (1) the body of trigger t1 comprises a statement `delete from X` and (2) trigger `t2` is fired by any event of the form `delete on X`,
3. (1) the body of trigger t1 comprises a statement `update X` affecting columns StatCol, (2) trigger t2 is fired by any event `update on X`, affecting (explicitly or implicitly) columns TrigCol and (3) column lists TrigCol and StatCol share at least one common column.

According to the last rule, the statement "`update T set T2=2,T3=3`" of trigger R_INS (TrigID = 1) will fire trigger T_UPD (TrigID = 3), the event of which is "`update on T`", equivalent to "`update T1,T2,T3 on T`". Indeed, sets StatCol = {`T2,T3`} and TrigCol = {`T1,T2,T3`} have elements {`T2,T3`} in common. Therefore, (1,3) is a row of table BINARY.

On the contrary, the execution of trigger U_DEL will never fire trigger W_UPD because sets StatCol = {`W1`} and TrigCol = {`W3`} have no common elements.

These rules are translated into the code of Script 8.74. The join associates the rows of table SYS_TRIGGER_COMP_FULL (from which the value of TrigID1 will be extracted) with those of table SYS_TRIGGER_FULL (from which the value of TrigID2 will be extracted) when they share the same table name (`C1.StatTable = T2.TableName`) and event (`C1.StatOper = T2.TrigEvent`). The third condition selects the couples of rows that satisfy the three rules described above. The state of BINARY is shown in Figure 8.32 and its graphical representation in Figure 8.33.

```
create temp table BINARY as
select TrigID1,TrigID2
from (select
             C1.TrigID as TrigID1,C1.StatOper,C1.StatCol,
             T2.TrigID as TrigID2,T2.TrigCol
      from   SYS_TRIGGER_COMP_FULL C1,
             SYS_TRIGGER_FULL T2
      where  C1.StatTable = T2.TableName
      and    C1.StatOper = T2.TrigEvent
      and    case
             when C1.StatOper = 'update'
            then iif(itemInter(C1.StatCol,T2.TrigCol,',') = '',
                       False,True)
             else True
             end);
```

Script 8.74 - Building the BINARY table

```
+---------+---------+
| TrigID1 | TrigID2 |
+---------+---------+
| 1       | 2       |
| 1       | 3       |
| 2       | 4       |
| 3       | 3       |
| 3       | 5       |
| 4       | 2       |
+---------+---------+
```

Figure 8.32 - Table BINARY derived from the schema of Script 8.72.

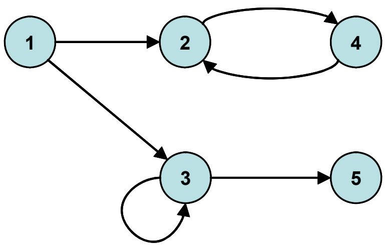


Figure 8.33 - Graphical representation of the rows of table BINARY

A first interesting question is whether the graph includes one or more circuit(s). To belong to a circuit, a node must be the source of some arc(s) **and** the destination of some (not necessary distinct) arc(s). A node that does not meet these conditions cannot belong to a circuit.

Considering the graphical interpretation of Figure 8.33, we observe that arcs (1,2) and (1,3) cannot participate in circuits since node 1 is the destination of no arc (it is not accessible from another node). Similarly, node 5 is the source node of no arc. Obviously, nodes 1 and 5 can be discarded from the graph without modifying its cyclicity property. Finally, we observe that nodes 2, 3 and 4, that are both source and destination of some arcs, cannot be eliminated based on the above rules.

This suggest a simple and intuitive algorithm:[39] we eliminate the rows of BINARY the TrigID2 value of which is not in the values of TrigID1 and the rows the TrigID2 value of which is not in the values of TrigID1. We process is to iterate until there is no elimination possible. The graph is acyclic if table BINARY is empty. This algorithms is translated in Script 8.75.

```
extract N1 = select count(*) from BINARY;
set N2 = 999;
while ($N1$ < $N2$);
   delete from BINARY
   where TrigID2 not in (select TrigID1 from BINARY);
   delete from BINARY
   where TrigID1 not in (select TrigID2 from BINARY);
   set N2 = $N1$;
   extract N1 = select count(*) from BINARY;
endwhile;
if ($N1$ = 0) write The graph is acyclic;
```

Script 8.75 - Checking whether a graph is acyclic

Now, we would like to extract the circuits *hidden* in table BINARY. A circuit is a path in the graph (a suite of consecutive arcs) the end of which is the starting node. The use of a recursive query clearly is a matter of course (Script 8.76). The CTE PATHS(TFrom,TTo,Path) computes all the paths in BINARY the last node of which is not already present (ignoring the first position).

We start by considering that each row of BINARY is an elementary path. Then, to each of these paths, we add a matching row from BINARY, if any. We go on until we try to add a node that has already been inserted into the path, in which case we stop augmenting it. In this comparison, we only consider the nodes *beyond* the starting one. When all the paths have been built in PATHS, the final query selects those for which TFrom = TTo (Figure 8.34)

39. Called the *Marimont* algorithm.

```
create temp view CIRCUIT(Circuit)
as with PATHS(TFrom,TTo,Path) as
     (select TrigID1,TrigID2,'.'||TrigID1||'.'||TrigID2||'.'
      from   BINARY
         union
      select TFrom,TrigID2,Path||TrigID2||'.'
      from   PATHS,BINARY
      where  TTo = TrigID1
      and    Path not like '.%.'||TrigID2||'.%'
     )
select Path from PATHS where TFrom = TTo;
```

Script 8.76 - Computing the circuits of the graph BINARY

```
+---------+
| Circuit |
+---------+
| .3.3.   |
| .2.4.2. |
| .4.2.4. |
+---------+
```

Figure 8.34 - The circuits as shown by view CIRCUITS

Actually, we are not completely done. First, it would nice to discard the heading and trailing dots. Then, we observe that each circuit is represented in view CIRCUITS by as many rows as the circuit comprises nodes. For instance, the circuit formed by nodes 2 and 4 is represented twice, by 2.4.2 and 4.2.4. Since any node of a circuit can be, by convention, selected as its starting points, we suggest to represent each circuit by a single expression, the first node of which has the smallest denotation, lexicographically speaking, that is, here, "2.4.2". This is what Script 8.77 computes. Function trim removes leading and trailing dots, while function itemSort generates a sorted permutation of the node denotations. This function requires four parameters: the list to sort, the sorting direction (0 = ascending, 1 = descending), the uniqueness (0 = duplicates preserved; 1 = duplicates discarded) and the separator. The result is shown in Figure 8.35

```
select trim(min(Circuit),'.') as Circuit
from   CIRCUIT
group by itemSort(trim(Circuit,'.'),0,'1','.');
```

Script 8.77 - Cleaning and reducing the expressions of circuits

```
+---------+
| Circuit |
+---------+
| 2.4.2   |
| 3.3     |
+---------+
```

Figure 8.35 - The final view of the circuits of triggers

Conclusion on trigger (a)cyclic graph analysis

It is necessary to observe that the analysis of the trigger system developed above can prove that its graph is acyclic. However, the fact that circuits have been detected in the schema does not mean that they will lead to infinite execution. The real danger is the existence of circuits *at the data level*. When the triggers in a circuit fire several times, each of their executions must tend to an exit point. Practically, this condition will be evaluated manually. Hence the importance to identify all the triggers participating in a circuit and to prove that each of their executions brings them closer to the termination.

A row such as (t1,t2) can be interpreted as: *trigger t1* ***will*** *fire trigger t2*, only if t2 includes no `when` clause and if data modification queries in the body of t1 include no `where` clause, i.e., if the firing relation does not depend on the state of the database. Otherwise the interpretation of (t1,t2) must be weaken as: *trigger t1* ***may*** *fire trigger t2*.

Let us modify our example schema by adding `when` and `where` clauses (Script 8.78). In trigger R_INS, the second body statement will execute under condition `new.R2 > 0`, the value of which can only be known at execution time. If this statement is executed, then trigger T_UPD will fire under the condition that column T3 is assigned increasing values, a condition that too can only be evaluated at execution time.

From the analysis of an active database, we can tell that the graph of the trigger system is acyclic and therefore that there is no risk of infinite execution. However, if this graph includes circuits, proving (or at least getting reasonable convinced) that there is any (or no) risk of infinite execution would require either a more in-depth syntactic analysis of the code or the tracing of the execution of the concerned triggers for a representative collection of initial data.

When it is impossible to prove that a trigger circuit will never cause infinite execution, it is recommended to include protection mechanisms in the code of the incriminated triggers. See for example the section "***Recursive queries against cyclic data***" in chapter 19 of the SQLfast manual (***Recursive programming***), which proposes several techniques to stop infinite executions.

```
...
create trigger R_INS after insert on R                            [1]
begin insert into S values(1,2,3);
      update T set T2=2,T3=3 where new.R2 > 0; end;
...
create trigger T_UPD after update on T                            [3]
when new.T3 > old.T3
begin update T set T1=1;
      update W set W3=3; end;
```

Script 8.78 - Trigger R_INS **may** fire trigger T_UPD