

## Case study 1

# Four hours to save the library

**Objective:** This case study describes the emergency writing of a small application that was to implement the core functions of the management of a small library. The challenge was to replace the current software, lost in a recent crash of the server. All that was left after the accident was the last backup of the database, unfortunately in an unknown format.

**Keywords:** rapid application development, application prototyping, application architecture, GUI

## 1.1 An emergency situation

This one is a true story.

A friend of mine manages a public library that lends comic books. That library is very popular, counting about 3,000 readers and 20,000 books. Its activity is estimated to about 4,000 transactions (lending and return) per week.

Some time ago, he called me, desperate, to tell that his server was destroyed during the storm of the last weekend. This server hosted the library management software he has been using twice a week. Everything had disappeared: the software sources and binaries, the documentation, the database and even the name of the language in which the application was written. The software was developed decades ago by *the nephew of a friend of the brother in law of the fitness instructor of his wife* (just synthesizing) and is therefore what we can call both *proprietary* and

*orphan*. It seemed that the author of the application is now breeding sheeps in Corrèze and that he has long abandoned his software development activity.

All that was left was the last backup of the database on a USB key.

## 1.2 First mission: recovering the data

The first objective was to extract as much usable data as possible from the backup database.

That database is a collection of 18 files with extensions \*.lis, \*.fic, \*.ndx, \*.mmo, \*.id and \*.tab. It turns out that files \*.fic were *WinDev* data files and that all the other files could be ignored. Unfortunately, no (free!) converter utility for Windev data files, nor a description of the structure of these files could be found on the web.

A visual inspection of \*.fic files through notepad++ showed that some data values are Latin-1 character strings but that others are coded as pure bit strings. In particular, foreign keys are not expressed as character strings as is usual, but as 24-bit pointers. We wrote a small Python program to extract these data and to convert them into SQL `insert` statements. Three files, comprising the core of the data, were processed in this way, namely the *reader* file, the *book* file and the *borrowing* file, and were transferred into three similar relational tables (see Script 1.1).<sup>1</sup> The other files were ignored. The resulting database was about 100 MB large.

By analyzing the data stored in these tables, we determined the data types and the uniqueness constraints, from which we derived the primary keys. In table BORROW, we detected two foreign keys, respectively to table BOOK and table READER. However, we decided not to declare them to allow for the recording of *dangling* (i.e., corrupt) borrowings for which either the book or the reader (or both) were missing.

As is usual in most DBMS, an index is automatically associated with each primary key. In addition, we create an index for each foreign key, be it declared or implicit. Since foreign key {BookID} is a prefix of the primary index of BORROW,<sup>2</sup> the latter also supports this foreign key. We just had to create an index on {ReaderID}, the second foreign key.

## 1.3 The basic functions

Since the accident, the trace of the day-to-day operations has been painfully written down in a paper notebook. The most urgent tasks were, as expected, book borrowing

---

1. In a further chapter, we will develop a more powerful library management application. This one can be considered a prototype version of the latter. Hence the name **LIBRARY-*proto*.db** of the database.

2. and since this index is implemented by a B-tree ...

and book return. However, registering new readers and new books and consulting the data was also considered critical.

We decided to write a small, no-frills, interactive program that supports these activities and that made it possible to convert the handwritten backlog into electronic data. Three core tasks were identified: *registering new readers and new books, borrowing and returning books, querying reader and book data.*

```
create table BOOK(
  BookID      varchar(32) not null primary key,
  Title       varchar(96) not null,
  Collection  varchar(32),
  Authors     varchar(64),
  Publisher   varchar(32) );

create table READER(
  ReaderID    varchar(32) not null primary key,
  RegDate     char(24)    not null,
  Name        varchar(48) not null,
  Address     varchar(64) );

create table BORROW(
  BookID      varchar(32) not null,
  ReaderID    varchar(32) not null,
  BorrowDate  date not null,
  ReturnDate  date,
  primary key (BookID,BorrowDate) );

create index ndx_borrow_reader on BORROW(ReaderID);
```

**Script 1.1** - The LIBRARY-proto.db database [Script LIBRARY-Create-DB-proto.sql]

## 1.4 The main window

The main window is the control panel of the application. It lets the user select a function (Figure 1.1). Button **OK** starts the selected function while button **Cancel** closes the application. Setting the `commit` mode to *autocommit* (each data modification query is wrapped into its own transaction) allows us to ignore explicit `commitDB` statements.

Each function is implemented as an independent procedure. These procedures are called from the body of a **while-endwhile** loop by a series of **if** statements (Script 1.2). This writing style is not particularly efficient from the point of view of programming science (all the conditions are evaluated in each execution of the body of the loop), but it provides a clear and concise code, which is essential in this experiment.

There is no need to close the database explicitly since it is automatically closed when the main script finishes.

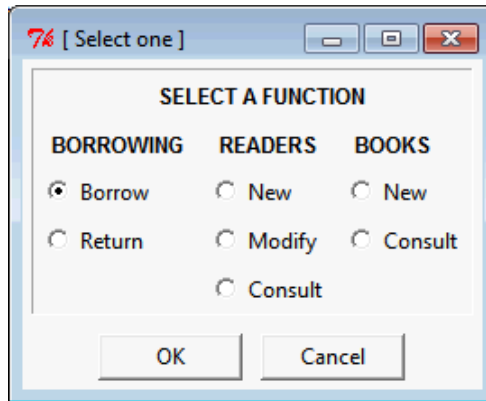


Figure 1.1 - The main window lets the user select a function

```
parameter commitmode = autocommit;
openDB LIBRARY-proto.db;
set oper = 1;
while (True);
    selectOne-u oper = [/b@S20SELECT A FUNCTION]
                        {BORROWING}|Borrow|Return
                        ||{READERS}|New|Modify|Consult
                        ||{BOOKS}|New|Consult;
    if ('$DIALOGbutton$' = 'Cancel') exit;
    if ($oper$ = 1) execSQL Library/_LIB-BORROWING-BORROW.sql;
    if ($oper$ = 2) execSQL Library/_LIB-BORROWING-RETURN.sql;
    if ($oper$ = 3) execSQL Library/_LIB-READER-NEW.sql;
    if ($oper$ = 4) execSQL Library/_LIB-READER-MODIFY.sql;
    if ($oper$ = 5) execSQL Library/_LIB-READER-CONSULT.sql;
    if ($oper$ = 6) execSQL Library/_LIB-BOOK-NEW.sql;
    if ($oper$ = 7) execSQL Library/_LIB-BOOK-CONSULT.sql;
endwhile;
```

Script 1.2 - Code of the control panel [Script LIBRARY-Main.sql]

## 1.5 Registering a new reader

The data entry box for new readers shown in Figure 1.2 was found quite appropriate.

The code is straightforward (Script 1.3):

- initializing the field variables (notably to the current date)
- acquiring field values from the user (ask-u)

- checking the validity of these values; looping until the values are valid
- inserting the values in the database (`insert`); field Address is converted into a *null* value if empty
- checking the result, committing (`commitDB`) or reporting errors (`showMessage`), if any.

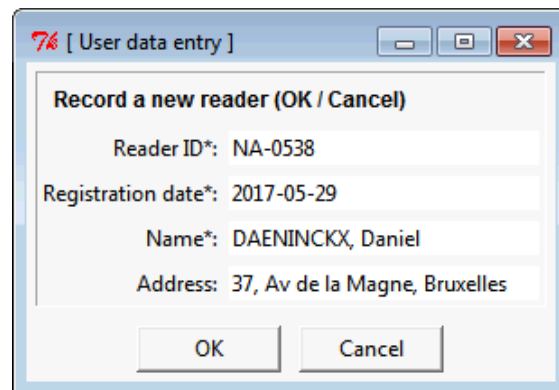


Figure 1.2 - Data entry box for a new reader

```

set rid,dat,nam,add = , $date$ , , ;
while (True);
  ask-u rid,dat,nam,add = [/bRecord a new reader (OK/Cancel)]
    Reader ID*:|Registration date*:|Name*:|Address:;
  if ('$DIALOGbutton$' = 'Cancel') return;
  if (trim('$rid$') <> '' and '$dat$' <> ''
    and trim('$nam$') <> '') exit;
endwhile;
insert into READER(ReaderID,RegDate,Name,Address)
  values('$rid$', '$dat$', '$nam$',
    case when '$add$' = '' then null else 'add$' end);
if ('$SQLdiag$' <> 'OK')
  showMessage Recording error ($SQLdiag$);

```

Script 1.3 - Code of the procedure to register a new reader [Script \_LIB-READER-NEW.sql]

## 1.6 Registering a new book

The procedure that registers a new book is quite similar to that of reader registering. The data entry box is shown in Figure 1.3. The value of BookID of a book is printed

Printed 5/6/23

as a barcode on a label stuck on the cover of the book. This way, a book can be identified by merely scanning its barcode.

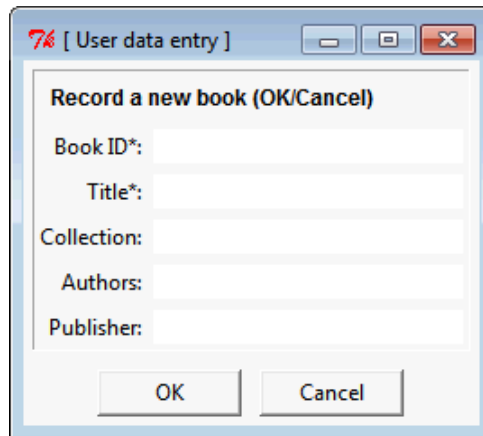


Figure 1.3 - Data entry box for a new book

## 1.7 Borrowing a book

The ID of the book and that of the reader are entered through the box shown in Figure 1.4. The book ID is collected through the codebar scanner while the reader is identified through a predefined list of reader names (actually ReaderID + Name).

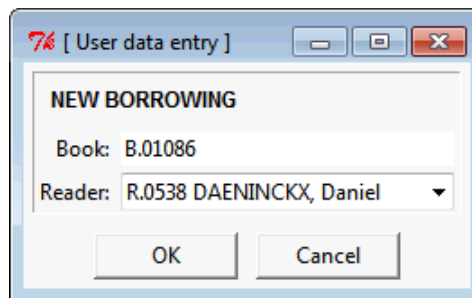


Figure 1.4 - Borrowing a book

The code of this operation is shown in Script 1.4. Statement **ask** returns a value of BookID and a value of ReaderID. These values can be considered valid due to the way they are collected: through barcode scanning and predefined value list. Therefore no validation code has been included.

However, we must ensure that the database records this book as being available, that is, there is no BORROW row for this book with a *null* ReturnDate value.

Searching the database for such a row must fail ('\$SQLdiag\$' = 'NONE'), in which case a new row describing the borrowing can be inserted and committed.

The value of BorrowDate is that of **current\_date** while the return date (ReturnDate) will be left as *null* until the book is returned.

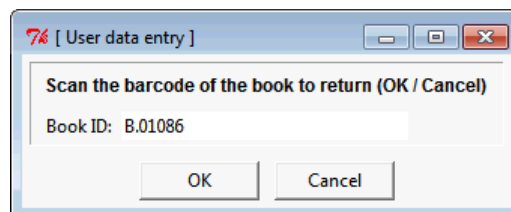
```
ask book,read = [/bNEW BORROWING] Book:
|Reader:[!select ReaderID||'|'||substr(Name,1,20),ReaderID
        from READER order by Name];
if ('$DIALOGbutton$' = 'Cancel') return;
extract r = select ReaderID from BORROW
        where BookID = '$book$' and ReturnDate is null;
if ('$SQLdiag$' = 'NONE');
    insert into BORROW(BookID,ReaderID,BorrowDate)
        values('$book$','$read$',current_date);
else;
    showMessage This book is currently borrowed by $r$;
endif;
```

**Script 1.4** - Code of the procedure to borrow a book [script \_LIB-BORROWING-BORROW.sql]

## 1.8 Returning a book

Returning a book translates into assigning the current date to column ReturnDate to the row describing the current borrowing of the book. This row then becomes a historical record for both the book and the reader.<sup>3</sup>

The box of Figure 1.5 comprises a single field, in which the value of BookID has to be entered through barcode scanning.



**Figure 1.5** - Returning a book

The code of the operation is shown in Script 1.5. Once the book ID has been collected (**ask**), the borrowing date (BorrowDate) of the book is extracted

3. In some countries, maintaining historical data on readers may be illegal for privacy reason.

(**extract**). Since columns {BookID, BorrowDate} form the primary key of the table, the values of variables **book** and **dat** can be used to identify the row in which the value of ReturnDate is set to the current date (**update**).

```
ask book = [/bScan the barcode of the book (OK/Cancel)]
           Book ID;;
if ('$DIALOGbutton$' = 'Cancel') return;
extract dat = select BorrowDate from BORROW
              where BookID = '$book$'
              and   ReturnDate is null;
if ('$SQLdiag$' = 'OK');
  update BORROW
  set   ReturnDate = current_date
  where BookID = '$book$' and BorrowDate = '$dat$';
else;
  showMessage The book $book$ is not currently borrowed;
endif;
```

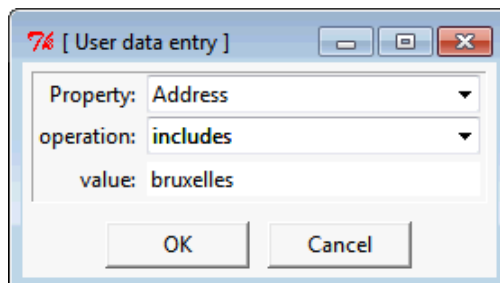
**Script 1.5** - Code of the procedure to return a book [script \_LIB-BORROWING-RETURN.sql]

## 1.9 Updating the data of a reader

Updating a reader is performed in two steps: first selecting a reader from a list of names, then presenting the data of this reader to let the user change some values (Script 1.6).

## 1.10 Querying reader and book data

We chose a procedure which is both simple and general. For both READER and BOOK tables, the user selects a *column* and an *operator*, then enters a *value*, considered case-insensitive (Figure 1.6).



**Figure 1.6** - Consulting the readers whose **address** **includes** the word '**Bruxelles**'



```

ask rid = [/bSelect a reader to modify]
          Reader ID:[!select ReaderID||' - '||Name, ReaderID
                        from READER order by Name];
if ('$DIALOGbutton$' = 'Cancel') return;
extract dat,nam,add = select RegDate,Name,Address from READER
                      where ReaderID = '$rid$';
while (True);
  ask-u dat,nam,add = [/bModify the reader (OK/Cancel)]
                    Reader ID*:|Registration date*:|Name*:;
  if ('$DIALOGbutton$' = 'Cancel') return;
  if (trim('$dat$') <> '' and trim('$nam$') <> '') exit;
endwhile;
update READER
set RegDate = '$rid$', Name = '$nam$',
    Address = case when '$add$' = '' then null else '$add$' end
where ReaderID = '$rid$';
if ('$SQLdiag$' <> 'OK')
  showMessage Recording error ($SQLdiag$);

```

**Script 1.6** - Code of the procedure to modify the data of a reader [script \_LIB-READER-MODIFY.sql]

The list of operators to select from comprises the usual comparison operators (<, >, <=, >=, <>) + includes, a reduced version of SQL predicate like.

A first version of the function produced the result of Figure 1.7.

ReaderID	RegDate	Name	Address
R.0538	2016-11-13	SMITH Bernard	Place Bockstael, Bruxelles
R.0572	2016-11-21	JONES F. J.	108, Sq de Meeus, Bruxelles
R.0668	2016-12-21	TRAVIS K. K.	9, Rue H. Stockel, Bruxelles

**Figure 1.7** - The readers living in Brussels

It was considered insufficient since the main consultation needs were related to the past and current borrowings of each reader or of each book. Something like Figure 1.8, reporting on each reader and on their borrowings (same for books) would be better.

The code of both procedures is exactly the same, except for the *name of the table* and the list of *column names*. Therefore, we split the code into three procedures: \_LIB-READER-CONSULT, \_LIB-BOOK-CONSULT and \_LIB-CONSULT. The latter does all the work while the first two just prepare the specific arguments (script 1.7 for consulting readers).

ReaderID	RegDate	Name	Address	Book	BorrowDate	ReturnDate
R.0538	2016...	SMITH..	...	B.01196 B.01377	2017-05-08 2017-05-10	2017-05-14 2017-05-14
R.0572	2016...	JONES..	...	B.01086	2017-05-20	--
R.0668	2016...	TRAVIS..	...	B.01196 B.01208 B.01196	2017-05-20 2017-05-26 2017-05-01	-- -- 2017-05-08

**Figure 1.8** - The readers living in Brussels and the history of their borrowings

```

set table = READER;
set fields = ReaderID,RegDate,Name,Address;
execSQL LIBRARY/_LIB-CONSULT.sql;

```

**Script 1.7** - Code of the procedure to consult readers [script \_LIB-READER-CONSULT.sql]

Procedure \_LIB-CONSULT is a bit more complex (Script 1.8). From the data collected by the dialogue box (**ask**), the procedure builds the selection condition (**set condition = ...**) that will be inserted in the SQL **select** query.

This query comprises two subqueries: the first one extracts the data of READER rows that satisfy the condition (the blue one in Figure 1.8) and the second one extracts the data of BORROW rows that depend on each READER row selected (the red ones in Figure 1.8).

### READER (or BOOK) subquery

Let us examine the first subquery (in blue in Script 1.8). The **from** and **where** clauses are straightforward:

```

from $table$
where $condition$

```

The **select** clause comprises two lists of column names. The first one is the list of column names of table READER (or BOOK) while the second one acts as a *padding* for the columns of BORROW, that must be empty. The number and names of the latter columns are known and are the same for tables READER and BOOK. The **select** list comprises an additional column (**Sort**) that will be explained later.

### BORROW subquery

The second subquery (in red in Script 1.8) generates the depending rows of BORROW. Here, the *padding* columns are the first ones. They comprises an empty column for each column of READER (or of BOOK). Their **select** sublist is built in variable **empty** by statement **compute**, that works as follows:

- variable **fields** contains the list of column names, separated by commas
- function **itemLen**(*S*, *sep*) returns the number of items in list **S** with separator **sep**; so, **itemLen**('fields\$',',') computes the number of column names in list **fields**
- function **repeat**(*S*, *n*) returns a string formed by **n** instances of substring **S**
- for table **READER**, that comprises four columns, variable **empty** contains ' ', ' ', ' ', ' ', that is, **4** instances (= number of columns) of substring ' ',
- the arguments of **compute** comply with the SQL syntax for character constants, according to which a constant must be quoted and each internal quote must be doubled; therefore, substring ' ', must be coded ' ', ' ', ' ', ' '.<sup>4</sup>

Now, the structure of the **compute** statement should be clearer!

The subquery is a join between table **BORROW**, from which data are extracted, and table **READER** (or **BOOK**), reduced to its selected rows (**\$condition\$**).

### Sorting rows

The two subsets of rows are simply *unioned* in the **from** clause. The order of the rows resulting from this union is undefined and does not produce the nice hierarchical ordering shown in Figure 1.8, in which each **READER** row is directly followed by its dependent **BORROW** rows (same for **BOOK**).

We must force the rows to appear in this definite order. In order to do so, we build an artificial column, called **Sort**, comprising column **ReaderID** (more generally **\$table\$ID**) plus suffix **-0** for **READER** (or **BOOK**) and **-1** for table **BORROW**. In this way, all the rows with the same value of **ReaderID** are grouped, and, in each such group, the row of **READER** appears in the first position.

The particular structure of the **from** clause, as the union of two subqueries, is necessary to hide column **Sort**, which is meaningless for users.

Finally, we decided to send the resulting data in a text window. Those data are first stored in variable **result**, which is then displayed through statement **showText**.

## 1.11 Conclusion

The librarian claimed to be quite happy with this small application. Despite its unpolished interface and simplistic logic, it allowed him to keep his activity alive and to encode the backlog of recent transactions written on paper.

Two weeks later, on the basis of his comments, some improvement was carried out to the application, that, from then, has been used for about one year, until the data were migrated to a more professional software.

---

4. For any complaint, contact [www.iso.org](http://www.iso.org)

```

ask key,op,val = Property:[$fields$]
                        |operation:[(=,<,>,<=,>=,<>,includes)]|value:;
if ('$DIALOGbutton$' = 'Cancel') return;
if ('$op$' in ('=','<','>','<=','>=','<>'))
    set condition = lower($key$) $op$ lower('$val$');
if ('$op$' = 'includes')
    set condition = lower($key$) like '%'||lower('$val$')||'%';
compute empty = repeat('','',itemLen('$fields$',','));
outputOpen result.var;
select $fields$,RID as Reader,BID as Book,
        BorrowDate,ReturnDate
from
(select $fields$, '' as RID, '' as BID,
    '' as BorrowDate, '' as ReturnDate,$table$ID||'-0' as Sort
 from $table$ where $condition$
 union
 select $empty$ B.ReaderID as RID,B.BookID as BID,
        BorrowDate,ReturnDate,B.$table$ID||'-1' as Sort
 from BORROW B, $table$
 where B.$table$ID = $table$.$table$ID and $condition$
 )
order by Sort,ReturnDate;
outputAppend window;
showText result = [/w0/x100/y20];

```

**Script 1.8** - Code of the procedure to consult a reader or a book [script \_LIB-CONSULT.sql]

Though data recovery took some time (about 12 hours), due to the lack of documentation, writing the complete application just cost:

- less than 70 instructions
- 4 hours.<sup>5</sup>

Of course, the context in which application LIBRARY has been developed is just a minor issue. Actually, it was a nice opportunity to experiment with the concept of *rapid application development (aka RAD)* and *application prototyping*.

Case study *The human factor* tackles a similar application (library management) in which, due to the wider variety of users, the way they interact with the program leads to new problems.

---

5. Probably twice as much as what a professional application programmer would have spent!

**A last remark**

The scripts developed in this document 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 develop their own applications.

