Database Systems

P. Bartholdi[†*]

[†] *Observatory of Geneva*
CH-1290 Sauverny – Switzerland

Lecture given at the:
Second Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 20 October — 14 November 2003

LNS

[*] Paul.Bartholdi@obs.unige.ch

Abstract

In this chapter we look at Data bases as used in laboratories to organize data, keep them up-to-date and retrieve them in easy ways.

Only the relational model is considered, but two rather different implementations are described in some details.

A strong emphasis is put on public domain database systems that are readily available, powerful enough for most laboratory works, yet can be used under limited hardware facilities. Public software also assume more involvement from the user in testing, developping new facilities, catching and repairing errors etc.

The chapter end with JDBC , the Java connection to relational databases.

Many short examples illustrate the text, without to much theoretical considerations.

Keywords: SQL, JDBC, relational database, Unix file system, flat files.
PACS numbers:

Contents

1 Introduction

In general, we can consider a computer as a box with some input channels, some output channels, probably local storage devices and links to other computers.

Nowadays, operations are going very fast, even on small computers, and data storages (disk, CD, DVD) have huge capacities in the order of 10^{10} to 10^{11} Bytes. If we assume 10^4 characters on an A4 piece of paper (a rather full page), then a small 100 GB disk corresponds to 10^7 pages, or twenty metric tons of paper, all for about 60 US$.

The organisation of data in such a way that they can be classified and retrieved rapidly is not new. For centuries, librarians have been very good for this job. More recently, the punched cards were invented at the turn of last century to deal with the American census data. Pencil and paper were not sufficient anymore.

But this simple system of book catalogs or punched card sets is not sufficient, neither well adapted as data storage devices in computers.

From the early fifties until now, a lot of work on this has been done, many clever data organization systems developed and made available, even for small computers.

It is interesting to remember that in the mid fifties IBM thought they would eventually sell 10 copies of their first electronic computer. This number would be sufficient for decades... It had a few KB of memory, no disk, but tapes with a capacity of a few MB.

1.1 Why Data Base Systems?

A fundamental aspect of computers is their neutrality. Without changing anything in the hardware, they can be used for many different types of work, sometimes even at the same time. Data can be flags, numbers, words, phrases, codes, images, sounds etc. These data are not independent. Relations exist between many of them.

Think of music sound files. They are related to the composer, the CD editor, the interpreter, the place on your hard disk and in you book shelves, the date of purchase etc. What about finding the address of the interpreter, or a list of composers born in the same year as the one in your file?

What we want is a system that is sufficiently general to deal with all kinds of data, that is easy, fast and secure to use for storing new data, keeping them uptodate and retrieving them with their relations.

1.2 What is a Data Base System?

A Data Base System is a set of programs that can be used according to what is described in the previous paragraph. In many cases, they interact with the users through a single interface hiding their complexity effectively. Some even go as far as to replace entirely the operating system (the PIC relational computer

system or the R38 computer of IBM for example). At the other extreme, the system appears effectively as a large set of programs/commands that can be combined in the normal operating system (/rdb for example). We will see later (Section 3.5, page 12) the relative advantages of both approaches.

Data Base Systems can be local on a single machine, or distributed on many different ones, either in a hierarchical or egalitarian way. They can interact in real time or in pseudo batch mode (night update for example).

The fundamental aspects of the Database system are that it hides completely from the users the way the data are organized physically in the computers, and its ability to furnish a single and uniform way to interact with the data.

1.3 What is a Data Base ?

A Data Base is the set of all data that are somehow related.

A single Data Base system can contain many Databases that are not related. For example, my collection of books, with their authors, title, editors, date of publishing etc, has probably nothing to do with the meteorological data I keep for many years. They are two distinct Data Bases in my single Data Base System.

1.4 Database Models

As we have seen, a Database is not just data, but also the relations that exist between them.

In the world, we can recognize three or four types of relations:

hierarchical A company is made of divisions, divisions of workshop etc.

network A company has contractors and sales agents who are themselves. . .

relational A book is defined by its title, author, editor etc, and the author by its names, age, location etc.

object A melody is defined as a sheet of paper, its composer, author etc, and by all what can be done with it: singing, recording, printing. . .

They are all part of the reality, and Data Base Systems have been built in the past modeling more or less all four types. But today most work is done with relational model, Object (Relational) Databases being still in their infancy.

2 The Relational Model

2.1 Historical background

Historically, the Relational Model came up relatively late, mainly with the two marking papers of Codd (see section 10.1, page 40). Codd was working as a mathematician in IBM research laboratories. His goal was to setup a system

based on simple and mathematically coherent rules. It should avoid all the pitfalls of the hierarchical and network data bases. In particular, the system would be very easy to interrogate and resilient to all kinds of software and hardware problems.

The implementation of the concepts proposed by Codd was rather slow. They were too far advanced for the hardware available at that time. They were also given in a mathematically oriented vocabulary that discouraged many potential users or developers.

Among the cornerstones in the development of usable relational data bases, we should notice the definition of a single "structured query language" (SQL) by IBM, the hardware implementation of the concepts in the system R38 of IBM, the PIC operating system based entirely on the relational model and the work of Stonebraker (see section 10.1, page 40) in developing a full blown relational system on a mini computer. The last one became the Ingres system, soon joined by Oracle on the same ground.

Around 1985, a few people (see section 10.2, page 40) recognised that the relational model and the Unix pipes and redirections fitted very well together, permitting the elaboration of a completely open relational data base system. Sadly, this was never very popular outside a small circle of users, enthusiastic by its simplicity of usage.

Many, commercial or not, Relational Data Base Systems are now available, for example IBM DB2, Informix, Ingres, Oracle, Postgres, Sybase, /rdb, mSQL, MySQL, PostgreSQL . . .

2.2 The relational model

2.2.1 Vocabulary

The work of Codd is fundamental, but it uses a vocabulary more oriented toward mathematician than database users. So let us start with a small table of equivalent words :

Formal relational term	Informal equivalent
relation	table
tuple	record, row
attribute	field, column
primary key	unique identifier

2.3 The relational system

C. J. Date (see section 10.1, page 40) characterises a relational system by :

1. The data are perceived by the user as tables, and nothing but tables. The user should not necessarily be aware of the physical representation of the data.

2. The operators at the user's disposal are operations that generate new tables from old ones. The three main operators are **selection**, **projection** and **join**.

Codd himself goes into more details :

- represent all information as tables;
- keep the logical representation of data independent from its physical storage characteristics;
- use a high-level language for structuring, querying and changing the information in the data base;
- support the main relational operations (selection, projection and join), and set operations as union, intersection, difference and division;
- support views, which allow the user to specify alternative ways of looking at data in tables;
- differentiate between unknown (`NULL`) and zero or blank data;
- support mechanisms for security and authorization;
- protect data integrity through (atomic) transactions and recovery procedures.

2.4 Data entities and relations

Relations are contained in a homogeneous table, and each line conveys the information for one entity. As an example, take a table with all participants. It contains a set of columns like PIN, Name, FirstName, Address etc. For each participant, each entity, there is a line with his PIN, his name, first name, address etc. A data element, or value, is at the intersection of a row and a column. It can be "valid data", or "Unknown" or "Empty" (`NULL` value).

Each row must be identified by a **unique** identifier, called the **primary key**. In our case most probably the PIN, though on many occasions the Name or a combination of Name-FirstName could be easier to work with.

A database is made of a set of related tables. Most will contain data prepared by the user, but the system will also keep its information concerning the database in its own tables that can be accessed in the same way by the user; all information are treated as tables, as Codd says.

We will see later in this section how to organize the data into tables.

One more thing: Codd and Date insist rightfully on the independence of the physical implementation of the tables. We said that they can be considered as files, but in most relational systems the user will never be able to see them as such. The only exception is the Unix flat tables described in the next section. The logical design of tables should also be independent of the user's view of them.

2.5 The Data model

Here we will elaborate a little further on the organization of data into tables.

1. make a general overview of all the data you want to put into the database. Look for what is related though not originally intended to be part of the database.

2. make a list of entities with their properties or attributes; mark those properties that belong to more than one entity, those that are effectively transformations of another property.

3. make sure that each entity has an attribute (or a group of attributes) that you can use to uniquely identify any row in the future table. If the *logical* primary key (Name above for example) is possibly not unique, then look for a different attribute (PIN, ISBN number etc.) or create an artificial one (record number for example) if necessary.

4. consider the relationships between the entities. They can be of three forms:

   one-to-one: for each entry in a table there is a corresponding one in another table, and the converse is true;

   one-to-many: for each entry in a table there is a corresponding one in another table, but there are possibly many entries in the first table corresponding to one entry in the second one;

   many-to-many: for each entry in a table there are many entries in another table and the converse is also true.

5. apply the normalisation rules to eliminate all one-to-one and many-to-many relationships.

6. verify the coherence of the system and its adequacy to the available data and foreseen queries.

2.6 Entity-Relationship diagrams

A convenient way to look at table organization is the entity-relationship (E-R) diagrams. The usual convention is to display each table as a box with columns listed inside. This could be done with paper and pencil during the early development phase, then automatically produced by the system.

Participants	**Countries**	**Cities**
Name	Country_Name	City_Name
FirstName	Capital	Country
Country	Male	Altitude
Year	Female	Longitude
Phone	Ratio	Latitude
Email	Growth	Population
	Population	

The second step is to mark (underline, using bold) the primary key for each table. On many occasions, it will consist of effectively adding a new column that is unique (the "Name" above is surely not a unique identifier, so we need to add a PIN column).

The third step is to mark with arrows the relationships between entities in different tables, without arrow head in the case of *one-to-one*, single arrow head for *one-to-many* and double arrow heads for the *many-to-many* case.

The fourth step consists of resolving the *one-to-one* and *many-to-many* relationships (see also the normalisation rules below). Both are unacceptable, and are an indication that something is wrong in the design, at least in the view of the relational model.

The *one-to-one* is easy to resolve. It is probably sufficient to merge the two tables into one as they relate to the same entities.

The *many-to-many* is a little more complex. Usually, it is necessary to create an extra table with only the two related columns. This new table is usually called *connecting* or *association* table.

In the case of the previous tables, since there are no *one-to-one* relations, there is no need to apply the merging steps above. If we assume that every participant can come only from one country, then we have a *one-to-many* relation (one country per participant, but many participants from each country). But if multiple nationalities are permitted, then we need to take out "Country" from the Participants table and crate a new table "Nationalities", with the two columns "Name" and "Country" only. Then, every multi-nationality will have only one entry in in the Participant table, but as many entries as nationalities in the third table. All in all, only *one-to-many* relations are left.

2.7 Normal forms

Codd himself suggested a set of rules or forms that must be verified by the table design. Four or five are usually recognised as mandatory, while the others are more often ignored. Each form imply that the requirements of the previous ones have been met. Following these normalisation guidelines, we will often decrease the number of columns and add new tables as we did just before.

2.7.1 First normal form

It requires that at each row-column intersection there must be one and only one atomic value. There must be no repeating group in a table. If participants were allowed to come back for many colleges, then the "Participant" table above would not be acceptable. It could be solved by creating a fourth table with "Name" (or "PIN") and "Year", dropping "Year" from "Participants".

2.7.2 Second normal form

This form concerns only tables where the primary key consists of many columns. Then every *non-key* column must depend on the entire primary key. A table must not contain a non-key column that pertains to only part of the composite primary key. In other words, no non-key column shall be a fact about a subset of the primary key.

2.7.3 Third normal form

This is a generalisation of the second form. It requires that no non-key column depends on another non-key column. Each non-key column must be a fact about the primary key only. In the "Countries" table above, the column "Ratio" depends numerically on the "Male" and "Female" columns, both non-key columns. In our case, this column (Ratio) should be dropped, as it can be recomputed easily at any time.

2.7.4 Fourth normal form

This form forbids independent one-to-many relationship between primary key columns and non-key columns. This situation is relatively rare. It is an indication that we are mixing together in the same relation things that are effectively independent and should appear in different tables.

2.7.5 Remarks

These rules or forms are rather abstract and the user may not see immediately why he should apply them.

If you are a stranger to their formal beauty, then there are very good reasons to apply them nevertheless. The keywords here are coherence and (no) redundancy. If the data base is to survive system crashes as well as operator errors, and stay coherent at the end, then the application of the four normalisation steps is necessary. First, there will be no redundancy; all information will appear only once. Then, because of this, all data will stay coherent, they cannot appear with different values at different places. The "Ratio" column discussed in the third normal form description, is a good example. If any value in the "Male" or "Female" column is changed, but the "Ratio" is left unchanged, then the table is incoherent, and there is no way to force them to be updated at the same time.

This was the main problem encountered with hierarchical and network data bases. They used either multiple copies of the same information, or had pointers all over the place to the same information. Any incident during an update was catastrophic, as different values for the same information were left behind, or pointers were pointing to data non existing anymore. Relational database systems can be built without using a single pointer.

Formal rules are very useful, but not always sufficient.

3 Unix flat tables

3.1 Using pipes and redirections

Among others, Unix is based on two things : 1) all files are unstructured strings of characters; 2) a large number of small programs, dedicated to a well defined task, that can be linked together in chains, each one getting data from its predecessor and giving results to its successor. Real files appear at both end of the chain, while pipes, virtual files (memory buffers ?), link the various modules (programs).

As a reminder, `prog < file1 > file2` means that `file1` is redirected to the standard input of `prog`, and the standard output is redirected to `file2`. Similarly, `prog1 | prog2` means that the standard output of `prog1` is redirected as standard input for `prog2`. These redirections and pipes can be combined as desired : `prog1 < file1 | prog2 | prog3 ... progN > file2`

This model is implicitly present in most, if not all, data base manipulation or interrogation if it is assumed that relations are represented by individual files (see the seminal work of Manis et al. 10.2, page 40).

As an example, we have a file containing the Name, FirstName, Country, Telephone, Email, No, Flag and Year for all participants of ICTP Colleges. Let us call this file `Participants`. We want to extract the Name and Email addresses of all participants of the colleges between 1997 and 1999. The result should be ordered alphabetically by Name. Here is a solution :

```
cat Participants | \
select ' Year > 1996 && Year < 2000 ' | \
column Name Email | \
sorttable Name > P1997-1999
```

3.2 Flat file format

A table represented as a Unix flat file is very simple: It is made of a two lines header and the body with the data rows.

The rows (lines) are ended with "EOL" characters.

The fields (columns) are separated with <TAB> characters.

All lines, including in the header, must have the same number of fields, but there are no limitation on the size of each fields. They can be empty (two consecutive <TAB>).

Similarly, the number of rows is limited only by the disk space available.

Here is the very first part of a typical Unix flat file table (though the real one contains more fields):

```
Name<TAB>FirstName<TAB>Country
----<TAB>---------<TAB>-------
Raich<TAB>Uli<TAB>Germany
Santamarina<TAB>Pablo<Chile
Verkerk<Tab>Rinus<TAB>France

...
```

3.3 Building Databases

Tables can be created with any editor (vi, crisp, emacs etc.), or from within any program in C, Fortran or Java, or from a shell script.

Here is a small example of a script that runs forever and records every 60 seconds the local time, the number of users and the three "loads" as measured with the `uptime` command.

The output from `uptime` is (interesting values are underlined):

```
  3:15pm  up 1 day,  5:38, 12 users,  load average: 0.06, 0.02, 0.01
  ------   --    ----  ----  ----
```

```
#!/usr/local/bin/ksh
# Create and load a table with time, nb of users and loads from uptime
echo "time\tusers\tload1\tload2\tload3"           > Load.rdb
echo "----\t-----\t-----\t-----\t-----"          >> Load.rdb
while true ; do
  uptime | tr -s " ,\t" "   " > /tmp/uptime
  exec 0< /tmp/uptime
  read Time dum dum Users dum dum dum Load1 Load2 Load3
  echo "$Time\t$Users\t$Load1\t$Load2\t$Load3"   >> Load.rdb
  sleep 60
done
rm /tmp/uptime
exit 0
```

The following is the beginning of the resulting file (the <TAB>s have been expanded as spaces) :

```
time    users   load1   load2   load3
----    -----   -----   -----   -----
10:22pm 0       0.17    0.15    0.10
10:23pm 0       0.12    0.14    0.09
10:24pm 0       0.09    0.12    0.09
```

Finally, here is a larger script that builds up a dictionary of all fields from the files in the current directory.

Notice that the /rdb `jointable` is undistinguishable from the other Unix commands

```
#!/usr/local/bin/ksh
# Build a dictionary of all fields from the /rdb files
# in the current directory
#
# First create a small dictionary for every table (*.rdb file)
for f in *rdb ; do
  dict=${f}_dict
  echo "field\t$f"    > $dict
  echo "-----\t----" >> $dict
  head -1q $f | \
    tr "\t" "\n" | \
    sort | \
    awk '{ print $1 "\tX" }' >> $dict
done
# Then join together all individual dictionary
First=1
for f in *_dict ; do
  if [[ $First -eq 1 ]] ; then
    First=0
    cp $f Dictionary
  else
    jointable -c < Dictionary field $f > /tmp/Dico
    mv /tmp/Dico ./Dictionary
  fi
done
# Add a comment field
awk ' NR==1 { print $0 "\tDescription" }
      NR==2 { print $0 "\t-----------" }
      NR>2  { print $0 "\t"        } ' < ./Dictionary \
                                       > /tmp/Dico
mv /tmp/Dico ./Dictionary
# Do some cleanup and exit
for f in *rdb ; do
  rm ${f}_dict
done
exit 0
```

and the resulting `Dictionary` :

```
field       College.rdb  Countries.rdb  Load.rdb  Description
----------  -----------  -------------  --------  --------------------
Capital                  X
```

```
Country     X           X
Email       X
Female                  X
FirstName   X
Flag        X
Groth                   X
Male                    X
Name        X
Nb          X
Population              X
Ratio                   X
Telephone   X
Year        X
load1                                   X
load2                                   X
load3                                   X
time                                    X
users                                   X
```

The Description field can now be edited by hand.

3.4 Unix commands

There are at least three or four different systems based on the previous ideas. The simplest (RDB) is entirely written in Perl. Because of this, the user can read the code and get an unusual feeling of what is done. The entire code is 8100 lines, for 20 commands. At the other end, starbase and /rdb, are written in C with access to awk and sed. /rdb contains about 130 different programs, including for commercial applications. starbase is almost as rich, but was developed with astronomical users in mind. Both are much faster than RDB.

Here is a commented list of the main programs of RDB.

indextbl builds an index for one or more columns from a table for fast searching (see `search` bellow);

jointbl joins two tables on a common column;

mergetbl adds the contents of a table to the end of another table (they must have exactly the same columns);

ptbl pretty prints the contents of a table;

rdbedit a window data entry/editing facility for tables;

reporttbl formats and prints an arbitrary style report, as specified in another file;

row extracts lines from a table according to some given criteria;

search fast searching using index file built with `indextbl`;

sorttbl sorts a table according to one or more columns in numerical or lexicographical order;

subtotal calculates subtotals for one or more numerical columns;

summ summarizes information from a table, as number of rows, number of unique values in some given columns as well as minimum, average, maximum and total.

uniqtbl removes adjacent rows that are identical in some columns;

valid check that a table is in "good" format, meaning all the rows correspond to the header of the table.

These programs have many options, including `-h` that will give a good description for each of them.

As said before, these Unix tables have nothing special and can be used or manipulated by any other program which is not part of the original Database System.

3.5 Advantages and limitations

The advantages of this approach are numerous, but its limitations are also important.

3.5.1 Advantages

- it is extremely simple and fast to create new tables, often *on the fly*, work with them, and destroy them when the job is finished;
- it is an open system, and so very easy to interface to other programs for data acquisition, backup, text processing, graphics, statistics etc.
- the files are very compact; no extra space is reserved in advance;
- the user has full control of what is happening; he can optimize complex operations using his knowledge of the data and query.
- considering the last rules of Codd (see section 2.3, page 4), the Unix system itself provides comfortable access security, though at the file level more than at the record level, and many solutions for recovery procedures. `rcs`, or better, `cvs`, can be used to keep track of all table modifications, including facilities for back and forward rolling. Atomic transactions can be done using lock files or semaphores.

3.5.2 Limitations

- its simplicity encourages going to fast development, bypassing the careful analysis needed for a good Data Base design;
- it is not standardised, neither in the exact format for the tables, nor in the command names;
- it is much better at data query than update due to the simple table format.
- there is no global view or description of the Data Base as such. All tables rest independent, only under the control of the user who has to keep them coherent.
- it is usually slower during execution, though the factor is smaller than could be expected for an interpreted system.

3.6 SQL equivalence

SQL is a *de facto* standard, in particular outside the Unix environment. It is thus useful to look at the equivalent commands on both side.

SQL	/rdb
`SELECT col1 col2 FROM table`	`column col1 col2 < table`
`WHERE column = expression`	`row 'column == expression'`
`COMPUTE column = expression`	`compute 'column = expression'`
`GROUP BY`	`subtotal`
`HAVING`	`row`
`ORDER BY column`	`sorttable col`
`UNIQUE`	`uniq`
`COUNT`	`wc -l`
`NESTING`	"pipes" |
`INSERT, UPDATE, DELETE`	"editors, form softwares" etc.

4 Using SQL Data Bases

4.1 Introduction to SQL

SQL is a language designed to resemble natural human expressions. It is not clear whether it succeeds in this respect. It contains verbs (like `CREATE`), objects (such as `TABLE` or `INDEX`), adjective phrase (eg. `participant_name CHAR(22)`). SQL contains a very small number of commands (verbs): `CREATE`, `DELETE`, `INSERT`, `SELECT` and `UPDATE`. But then there are a lot of qualifiers and modifiers that give SQL its power, and also difficulties, to its users.

SQL is insensitive to capitals/small letters, but it is a common usage to capitalize entirely all commands and use small letters for table and column names. Such code will be easier to read and understand.

All table and column names should start will a letter, though the underscore ('_') is permitted as a first character in some implementation. After the first character, it may contain any combination of letters, digits and '_', up to a length fixed at system installation, generally 31.

When combining columns from different tables, using the so called *join* operation, the name of the table is put in front of the column name, with a '.' in between, as in `participant.country`, where `participant` is the table name, and `country` is the name of the column.

In a similar manner, when more than one database can be used simultaneously, it is possible to put the database name in front of the table name, again with a '.' between the database and the table name, as in '`college.participant.country`'. This is called the fully qualified name of a table.

4.1.1 Client/server and distributed systems

Modern RDBMSs are closed systems, where the only interactions permitted are through SQL commands. It has the advantage that it is independent of the native operating system, but it also means that none of the usual Unix goodies are available. As for the PIC operating system, everything is a table, including the user login names, passwords, access rights etc. When starting a new database, the first necessary action will be to setup the user environment. Many things missing in Unix like: file and record locking, restricted access to subpart of a file (row or column), concurrent update of files etc, are provided.

While NFS or AFS provide a truly distributed file system without a master/slave relation, SQL servers are all of the client/server form, with or without a third tier. This has strong implication on the hardware. The main load is on the single or a small number of servers which need to be powerful, while the client can be very *thin*. The load is better distributed on many machines with NFS or AFS. The central servers are single points of failure, but the system is essentially insensitive to the client states, while for the distributed model, every system is important to the others. When one NFS machine is down, the part of the data on this machine becomes unavailable, and the whole system may be stopped though most of the machines are still running. This is particularly important during system updates which must be carefully synchronised.

4.1.2 Aliases

Long table or column names can be aliased, mostly in the `SELECT` command where the same name can appear repetitively, to short "local" nicknames.

This is done in following the long name by `AS nickname` (or in some systems by `=nickname`).

For example:

```
SELECT participant_name AS name, participant_first_name AS first_name, country
FROM ictp_college_on_realtime AS college, geography
```

```
WHERE college.year = 2001
```

4.2 CREATE command

The general form of this command is:

```
CREATE TABLE table_name ( col_name1 type [modifiers]
                      [ , col_name2 type [modifiers] ]
                        )
```

As we will see, the CREATE command is also used to create new database, indices and functions. Though everything is a table, special forms of CREATE are then used.

4.2.1 Data types

SQL support the following data types:

INT	Integer, usually between 2^{-31} and $2^{31}-1$
UNSIGNED INT	Unsigned integer, from 0 to $2^{32}-1$
TINYINT	Very small integer, between -128 and 127
BIGINT	Very large integers, up to $\pm 2^{64}-1$
FLOAT	Floating point value, single precision $\pm 10^{\pm 38}$
REAL	Floating point value, usually double precision $\pm 10^{\pm 308}$
CHAR(length)	Fixed length character string, the length of which is specified in the function parameter. Unused positions are filled with spaces
VARCHAR(length)	Variable length, up to 255 characters. *length* is an indication for packing.
TEXT(length)	Variable length character string, up to 64 K characters, *length* is an indication for packing.
DATE	Standard date value, good for past, current and any future date
TIME	Standard time value, independent of the date, in Unix should be independent of the longitude
DATETIME	Pack of DATE and TIME, not available everywhere.
ENUM('...', '...)	A (short) list of possible values, given as strings. May not be available everywhere.

Many other variants exist for different "word" lengths, from 1 to 16 bytes, but the specific documentation should be queried.

4.2.2 Modifiers

It is possible to specify further the qualification of a column:

`NULL`	the column will be filled with `NULL` entries when no data are available
`NOT NULL`	the column must be filled for all entries. This is certainly the case for the primary key.
`PRIMARY KEY`	unique column on which all others depend
`AUTO_INCREMENT`	only possible for a single column. During insertion of a new record, this column will be set to the highest previous value+1.

In SQL, it is very important to distinguish the absence of a value from the `NULL` which means *No Valid Value* or *Unknown value*. It is more or less equivalent to the *NaN* for floating point values. An empty string is different from `NULL`, as is a zero in a numerical field.

It is easy to query for `NULL` values, but in general not for empty values.

4.2.3 Indices

For very fast access on a given column, it is worth associating with it an index, that will be used for fetching. It should be noted that indexing slows down the insertion of new records, but makes retrieving many orders of magnitude faster. With some systems, it is possible to specify the type of index (hash, binary, b-tree etc.).

Indices are extra tables that contain information for a single key and the position of the corresponding records. They cannot be accessed directly. The index table is usually much shorter that the main table and can be searched very rapidely.

During a `WHERE` clause, the index is searched for matching, and the cursor placed directly at the right place in the main table.

```
CREATE INDEX name ON table_name ( col1, col2, ...)
```

will create indices for each column *col1*, ...

The user who adds an index must have *INDEX* privilege on the table.

4.2.4 Other `CREATE` commands

The commands:

```
CREATE DATABASE name
CREATE FUNCTION name RETURNS values SONAME library
```

create an entirely new and empty database, or add a function into the current database. Usually, the creation of a new database is done with special administrative commands outside SQL. Similarly, the `FUNCTION` is used to give access to precompiled executable functions, not to SQL macros or procedures.

4.3 WHERE clause

The WHERE clause is used by many commands as INSERT, SELECT or UPDATE. It defines rows by specifying a value that must be matched by the given column. The WHERE clause comes usually at the end of these commands.

The form of the WHERE *clause* is very general. It may contain column names, literal values (number, strings etc.), with logical or arithmetic operators, functions etc. ( ) can be used to group sub-expressions as in usual mathematics.

The arithmetic operators are: ' + - * / % << >> '.

The logical operators are: ' || && ! OR AND NOT '.

The comparison operators are: ' = <> != < <= >= > '.

The interval/set operators are :

WHERE *value* BETWEEN *value1* AND *value2*
WHERE *value* IN (*value1*,*value2*,...)
WHERE *value* NOT IN (*value1*,*value2*,...)
WHERE *value* LIKE *value1* (*value1* can include wild cards etc)
WHERE *value* LIKE *value1* (*value1* is any extended regular expression (Unix style).

4.4 DELETE command

This is the simplest command:

```
DELETE FROM table [ WHERE clause ].
```

If the WHERE clause is missing, the table is emptied, but not destroyed. This can be useful to erase the contents of a table that has been corrupted, without changing the structure of the database.

The WHERE clause can be simple or very complex.
For example:

```
DELETE FROM participants WHERE country IN ( 'Switzerland', 'France' )
```

4.5 INSERT command

This command allows to fill data into tables, one row at a time at most.

```
INSERT INTO table ( col1, col2, ... )
           VALUES ( val1, val2, ... )
```

The *values* can be any number or strings between single quotes ('...'). The backslash '\' can be used as an escape character in front of the single quote inside a string if necessary.

For example:

```
INSERT INTO participants ( name, country )
              VALUES ( 'Aguir', 'Brazil' )
```

If data are available for all columns, then it is some times possible to drop entirely the column list.

For some RDBMSs, it is possible to enter at once bulk data set instead of using a new `INSERT` command for each row.

4.6 `SELECT` command

This now allows us to retrieve data from the database. `SELECT` looks at first very simple, and it can be so, as in the examples below. But it can also be extremely complex, as the `FROM` and `WHERE` clauses may contain operations on many tables (join) and even `SELECT` recursively.

```
SELECT col1, col2, col3, ...
[ INTO OUTFILE 'filename' delimiters ]
FROM table1, table2, table3, ...
[ WHERE clause ]
```

If all columns should be retrieved, then their names can be replaced by '*'.

The `INTO OUTFILE` option can be used to keep the results in a file for further processing. This will be particularly useful when debugging complex queries.

The `FROM` modifier can be just a simple list of files, or to the contrary a very complex expression used to join many tables together (see section 4.7, page 18).

In the simple case, the `WHERE` clause can be used to join tables together as in the following example:

```
SELECT name, first_name, country, population
FROM participant, geography
WHERE participant.country = geography.country
```

This is called *inner join*. Only rows that match in both tables are selected. The other joins permit to select all the rows from one or the other table, or both, even when there is no counterpart in the other table (see section 4.7, page 18).

4.7 `FROM` modifier

In the simplest form, the `FROM` is just followed by a list of tables we want to query, separated by ',' as for all lists.

The link between the columns is established by the conditions given in the `WHERE` clause. This is the `inner` join.

But more complex, not necessarily inner joins, can be done using the `FROM` clause.

The general form is: `FROM table1 JOIN table2 [ ON col1 op col2 ]`

where `JOIN` itself can be prefixed by `CROSS`, `INNER`, `NATURAL`, `OUTER`, `LEFT OUTER`, `RIGHT OUTER` and `FULL OUTER` qualifier.

The *op* can be any comparison operator (see section 4.3, page 17). The *col* expression will probably contain the table and column names, though the table name is not mandatory if there is no ambiguity.

The `CROSS JOIN` produces a Cartesian product, with every entry present in any table also present in the resulting file, even if it is not in the other tables.

The `NATURAL JOIN` works only if the two tables have a single identical column. The `WHERE` and `ON` are unnecessary.

The `ON` qualifier works in the same way as `WHERE`, but it cannot be used with the `CROSS` or `NATURAL` qualifiers.

The `OUTER JOIN` gives the join rows against the background of rows that did not meet the join conditions. In mathematical terms, it would be the complement in set theory.

The table `LEFT OUTER JOIN` will give all the rows of the left table, while the `RIGHT OUTER JOIN` will give all the rows of the second one.

Finally, the `FULL OUTER JOIN` returns all qualifying rows from both tables.

For all `OUTER JOIN`, missing values from one of the tables are replaced by `NULL` (see section 4.2.2, page 16).

4.8 `ORDER BY` modifier

One other modifier can be added to a `SELECT` command, to reorder the rows according to one or more columns, in a given direction (ascending or descending).

The syntax is: `ORDER BY col1 [ DESC ], col2...`

The *col* can be either column names, aliases, or arithmetic functions of columns. In principle, the list of columns or aliases should appear in the `SELECT` column definitions.

The `DESC` modifier indicates, as expected, that the ordering should be in reverse order (descending).

The first column indicates the primary sort key. The next ones are used only if two or more values are identical in the previous columns.

4.9 Combining `SELECT` commands

It is possible to combine the results of two `SELECT` commands with the `UNION`, `INTERSECT` and `MINUS` link commands.

It is used in the form:

```
select_command1
UNION
select_command2
```

The `UNION` returns a single copy of all the rows returned by both `SELECT` statements; duplicates are removed. In Unix, this is given by the `uniq` command.

The `INTERSECT` (in place of `UNION`) keeps only the rows common to both queries, and `MINUS` only the rows from the first query not present in the second.

4.10 `UPDATE` command

This command is used whenever some values have to be changed in the database.

The general syntax is:

```
UPDATE table
SET col1 = expression, col2 = expressions2, ...
[ WHERE search_conditions ]
```

Data can be changed in only one table at a time.

If the `WHERE` command is missing, all the rows of the table are changed. See section 4.3, page 17 for a description of the `WHERE` qualifier.

The changes will occur only if the new values do not violate the integrity constraints (data type etc).

The expressions can be literals or combinations of column values, as in:

```
UPDATE quotation
SET price = price * inflation
WHERE country = 'Switzerland'
```

4.11 Report command

While the `report` is a very powerful tool in /rdb, no equivalent exists in the standard SQL. Each RDBMS provides some sort of reporting facilities, and connection facilities to web clients.

4.12 Data dictionary

It is a good practice to keep in a table the names of all tables and columns with common information about them. The system itself maintains a table with the tables, columns, constraints, data types, maximum size, access rights etc.

The dictionary is less formal, but more verbose. It should help maintain a common “dialect” among developers and users as well.

5 Transactions and triggers

5.1 Transactions

When more than one user have access simultaneously to a database, the concurrence and protection problems need to be solved. From a logical point of

view, it is not possible for two persons to update the same information at the same time. In fact, the user update may be based on the content of some values which should not be changed until the update is committed.

One solution to this problem is for the user to *lock* the information at some level, until all the transaction is finished.

The *lock* mechanism can work at the database level, at the table, row or even single information in a row.

The lock can also be complete, no read, no write access for a while, or only the read or write access is denied until the user releases the lock.

In many SQL systems, some automatic locks are implemented for `DELETE`, `INSERT` and `UPDATE`. PostgreSQL offers a variety of locks, from prohibiting access share for the previous commands to granting exclusive access in any mode.

Another mechanism is the *transaction*. A *transaction* is a set of commands that are executed at once as an atomic operation. In practice, all the changes are accumulated during the transaction into a temporary buffer, which is visible only to the user. Then when everything is ready, the transaction is *committed* and made visible at once to the rest of the world. Transaction will use lock if necessary during the preparation phase.

Among the PD RDBMSs that I know, only PostgreSQL has the possibilities to start transactions and commit them.

Here are the main PostgreSQL commands for this:

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }

LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE

COMMIT
```

The `SET TRANSACTION` starts the transaction and gives the general visibility of changing records. All changes will be visible only after the `COMMIT` command has been executed.

With `READ COMMITTED`, a statement can only see rows committed before it began. It is the default.

With `SERIALIZABLE`, the current transaction can only see rows committed before the first statement in this transaction that changes some information. Two concurrent transactions will leave the database in the same state as if the two had been executed strictly one after the other, the order being irrelevant.

The `LOCK` command fixes more precisely the mode of locking.

The `ACCESS` ensures that the table structure cannot be modified or erased all together during queries.

The `SHARE` lock prevents the change of data during a query, while the `EXCLUSIVE` lock is used when the transaction itself is doing changes to the data. The `SHARE ROW EXCLUSIVE MODE` is the same as `EXCLUSIVE`, but permits `SHARE ROW` locks by others.

5.2 Triggers

Triggers are functions associated with certain conditions in the database. If, after a `DELETE`, `INSERT` or `UPDATE` command the conditions would have changed, then the function is activated automatically (either before or after the change itself).

In PostgreSQL, the triggering condition is simply any operation performed on the table. In other systems, it is possible to trigger the function on the new values for given columns etc.

The PostgreSQL syntax is the following:

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

A trigger can also be removed from a database:

```
DROP TRIGGER name ON table
```

6 Data Bases and the Web

6.1 The Client Server model

This model has already been presented (see section 4.1.1, page 14) while talking about SQL. One (or a few) central machine(s), called the server, contains the database(s) and all the procedures to access them. Client machines send requests in parallel to the server using a common protocol. Though it is not part of SQL itself, most if not all, SQL RDBMSs use this schema. The intermediate protocol, that encapsulates the SQL requests, is proprietary.

The servers are designed around two ideas: either with a multiple daemon receiving the requests, passing them to a central "processor" and sending the results back, or a unique but multi-threaded central "processor" which starts a new thread for every request.

Most `ftp` and `http` servers are of the first type. mSQL also works with multi-daemon. MySQL and PostgreSQL are both built on the multi-thread concept. which seems more appropriate when many short requests need to be processed in parallel. The time necessary to start a new thread is much shorter than the time to start a new program.

Usually, the clients do nothing but interacting with the users, choosing the databases, doing possibly simple verifications, but most of the burden is taken by the server. On some systems, the client can also try to pre-optimize the requests before sending them to the server.

6.2 The Three Tier model

The Three Tier model tends to split the work in three parts. The local user application, the centralised application services and the databases. The user has no more direct access to the databases.

The advantages are:

- the databases can be changed easily, without changes at the user level;
- the applications are centralised, independent of the user and databases implementations.

6.3 Access through the web, CGI programming

This is an example of the three tier model, if not of four tiers. The web browser is a (very) thin client with the user interface. All the requests are sent (see Figure1) to the web server which does all the page formatting preparation in one direction, and passes the requests to the cgi applications, which in turn, passes the requests to the database engines. SQL disappears at the user level and in the web server as well. The work is done in cgi applications, which can be programmed in many different ways. The one presented here is the simplest, but also the slowest. It has the advantage that the protocol is ultra simple and will work on any Unix environment. Better application interfaces can be built using *Fast CGI*, *HPP*, *C++* and now *Java*, *Java beans* etc.

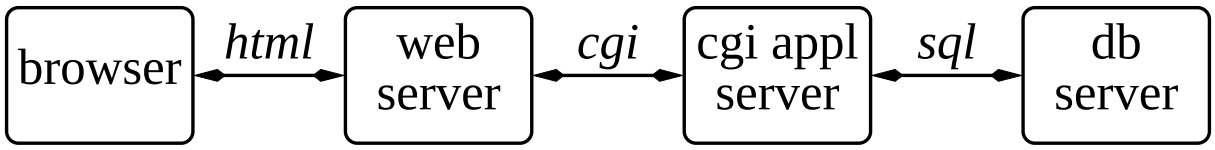

Figure 1: Communications between the user's browser and the database server

A few remarks are necessary concerning cgi programming.

1. The `http` protocol is stateless, when a transaction is finished, nothing is retained from the "conversation", all variables etc. are lost, with one small exception on the browser side.
2. The only data permanently available are the **cookies** which may, under user control, be stored on the user disk. The content of the cookies are very limited, mostly only the user ID and http addresses.
3. because of that, the cookies are accessible only from the user environment. With NFS, this may be all the machines inside the NFS domain, but if the user moves somewhere else, the cookies will definitely be inaccessible.

4. The cgi protocol itself is reasonably secure, but the programmer is responsible to check that the information provided are those intended. Consider the following trivial example: the returned parameter is the name of a file the user wants to print. After some transformations, the script will execute a command like '`cat $file_name`', where `$file_name` is the parameter passed.

Now, if the user gives some thing like '`MyFile;rm *`' as file name and the cgi script executes blindly the '`cat`' command above, all the files could be erased...

For further information, consult the books on SQL (see section 10.3, page 40). There is also an interesting web server:

`http://www.kmarshall.com/easy/cgi/`

Here is a commented cgi script using the normal `sh` of Unix. It will present all the information concerning one person, including his face, on the web page. Also, it does not access a database server as it uses the distributed model instead of the client/server one. Database access is done using the /rdb Unix commands.

```
#!/bin/sh
# IdCard.cgi , 20011203 , DM+Bdi
# Show on screen all information concerning one person,
# including his face.
#
# The $QUERY_STRING variable is given to this script,
# with the mon_id value for the person.
#
# This script should be called from an html document, as:
#
# HREF="http:// (the place of this file) /cgi-bin/carte.cgi?"
#
# The file must be executable (chmod 755)
#
REP=      # (where the database is)
RDBBIN=   # (directory for /rdb commands)

# 1. analyse the QUERY STRING.
# drop the double quotes,  translate the accents,
# and pipe into awk to create variables of the form QS_Key="Value"
# $QUERY_STRING contains all pairs Key1=Valu2,Key2=Value2,...
### This part can be copied unchanged for all simple cgi files.
eval ‘echo "$QUERY_STRING" |\
 sed -e ’s/’"’"’/_/g’ |\
 sed -e ’s/%C0//g’ | sed -e ’s/%E0//g’ |\
 sed -e ’s/%C2//g’ | sed -e ’s/%E2//g’ |\
 sed -e ’s/%C4//g’ | sed -e ’s/%E4//g’ |\
 sed -e ’s/%C7//g’ | sed -e ’s/%E7//g’ |\
 sed -e ’s/%C8//g’ | sed -e ’s/%E8//g’ |\
 sed -e ’s/%C9//g’ | sed -e ’s/%E9//g’ |\
 sed -e ’s/%CA//g’ | sed -e ’s/%EA//g’ |\
 sed -e ’s/%CB//g’ | sed -e ’s/%EB//g’ |\
 sed -e ’s/%CE//g’ | sed -e ’s/%EE//g’ |\
 sed -e ’s/%CF//g’ | sed -e ’s/%EF//g’ |\
 sed -e ’s/%D4//g’ | sed -e ’s/%F4//g’ |\
 sed -e ’s/%D6//g’ | sed -e ’s/%F6//g’ |\
 sed -e ’s/%D9//g’ | sed -e ’s/%F9//g’ |\
 sed -e ’s/%DB//g’ | sed -e ’s/%FB//g’ |\
 sed -e ’s/%DC//g’ | sed -e ’s/%FC//g’ |\
 sed -e ’s/+/ /g’ |\
 awk ’BEGIN{RS="&";FS="="}
```

```
      $1~/^[a-zA-Z][a-zA-Z0-9_]*$/ {
         printf "QS_%s=%c%s%c\n",$1,39,$2,39}''

# create an alias
moni=$QS_id

# 2. start the real stuff, extracting the information for "moni"
#    into /tmp/carte.tmp (this is still a /rdb file)
${RDBBIN}/jointable ${REP}/addresse.r ${REP}/age.r | \
${RDBBIN}/jointable ${REP}/brev.r | \
${RDBBIN}/addcol affdate | \
${RDBBIN}/compute \
    'affdate=substr(naissance,4,2)"."substr(naissance,1,2)"."substr(naissance,7,2)' | \
${RDBBIN}/row "mon_id==\"$moni\"" > /tmp/carte.tmp

# 3. the next type declaration is MANDATORI, including the NewLine (\n)
echo 'Content-type: text/html\n'

# 4. send the header, fix colors etc
cat << EOT
<HTML>
<HEAD>
   <BASE HREF="http:// (where the pages are) ">
</HEAD>

<BODY BGCOLOR=#F0f8ff
TEXT=#000000
LINK=#D700
ALINK=#E4B5
VLINK=#0000>
EOT

# 5. Use report to format the information on screen
${RDBBIN}/report ${REP}/carte.html < /tmp/carte.tmp

# 6. send the footer for the page. And that's all !
cat << EOT
<HR WIDTH=95%>
<ADDRESS>
<A HREF="mailto: (email address) "> (your name in clear) </A><BR>
</ADDRESS>
</BODY>
</HTML>
EOT
```

Here is the `carte.html` file, used above to format the result of the query into html format. Names between < > which are not html commands, are replaced by the values of the corresponding columns, that is `<mon_id>`, `<affdate>`, `<number>`, `<street>`, `<town>`, `<telpriv>`, `<gsm>` and `<email>`. Notice that the `<affdate>` was just calculated in the previous `.cgi` file.

```
<! IdCard.report , 20011201 Bdi !>
<FONT SIZE=+4 BOLD><nom></FONT> 
<FONT SIZE=+4 BOLD><prenom></FONT><P>
<TABLE cellspacing=0 cellpadding=5>
  <TR VALIGN=TOP>
    <TD VALIGN=TOP><img height=200
                    src="/ (where the images are) /<mon_id>.jpg"></TD>
    <TD>
      <TABLE cellspacing=0 cellpadding=1>
     <TR VALIGN=TOP>
       <TD ALIGN=RIGHT><I>Born :</I></TD>
       <TD><affdate></TD>
     </TR>
```

```
    <TR>
      <TD ALIGN=RIGHT><I>Address :</I></TD>
      <TD><number>, <street></TD>
    </TR>
    <TR>
      <TD></TD>
      <TD><B><npa> <town></B></TD>
    </TR>
    <TR>
      <TD ALIGN=RIGHT><I>Tel. privat :</I></TD>
      <TD><telpriv></TD>
    </TR>
    <TR>
      <TD ALIGN=RIGHT><I>Tl. cell. :</I></TD>
      <TD><gsm></TD>
    </TR>
    <TR>
      <TD ALIGN=RIGHT><I>E-Mail :</I></TD>
      <TD><a href="mailto:<email>"><email></a></TD>
    </TR>
     </TABLE>
   </TD>
  </TR>
</TABLE>
```

Finally, the /rdb file `/tmp/carte/tmp` with the result of the extraction looks like:

```
mon_id affdate  number street  town   telpriv   gsm email
------ -------- ------ ------- ------ --------- --- --------------------------
mda54  19.01.54    117 5th Av. Geneva 776.17.09     Daniel.Megard@obs.unige.ch
```

Here, we have only one person extracted, and only one table was produced. If the `carte.tmp` had been larger, more tables would have been produced.

7 The Java JDBC API

7.1 Java + RDB : a strange marriage

Java talks about objects : data + methods;
RDB talks about relations + mathematical operations on them;
Two incompatible approaches !
JDBC provides:

- Embedding SQL statements as arguments to methods in JDBC interfaces
- Easy objects to relational mapping
- Database independence
- Distributed computing

7.2 Embedded SQL statements

Here is a small part of a SQL query from a Java application:

```
Statement stmt;
ResultSet rs;

stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_READ_ONLY);
rs = stmt.executeQuery("SELECT * FROM test ORDER BY test_id");

rs.afterLast();
while(rs.previous()) {
  Int a;
  String str;
  a = rs.getInt("test_id);
  str = rs.getString("test_val");
}
```

The first part sets the type of operation with the server and sends the SQL command string. The second part fetch the individual data from the returned ResulSet.

7.3 Database independence

JDBC is fully independent from any database system, but the SQL commands are not fully standardised.

The database system must furnish a runtime implementation of the interfaces, that route the SQL call to the database engine in its proprietary fashion.

These are available for all main SQL database systems, including the PD ones.

Nevertheless, most SQL database systems are not fully compliant with SQL 2. They differ in small details, as case sensitivity for column names, missing or extra options etc.

Full Database independence can be achieved only by carefully avoiding using the non-standard facilities.

7.4 Distributed computing

The clear distinction between the Java and the database parts allows to have them running on different computers.

JDBC is built with RMI from the Java Enterprise platform.

RMI allows applications to call methods in objects on remote machines as if these objects were located on the local machine.

The same Java method call syntax are used, whether on the same machine or across the network. They take care of of the TCP/IP connections etc.

This solves the problem of many users accessing the same data simultaneously.

Other solutions for remote computing include the "Common Object Request Broker Architecture" (CORBA) and the "Distributed Computing Environment" (DCE).

They are much more complex and not related specifically to Java. JDBC provides the full set of methods for the client and the connectivity to the server in one package.

7.5 JDBC Driver

This is the part on the client and server computers that receive the calls and translate them into specific commands for the database engine.

Sun defines 4 types of drivers:

- Type 1 is a gateway from JDBC to an ODBC driver (Windows) on the server
- Type 2 calls directly native C or C++ methods provided by the db system
- Type 3 is a client driver with generic API that is then translated into specific access on the server
- Type 4 talk directly to the database using Java sockets

Type 1 and 2 need usually a local driver on the client

Type 4 is a pure direct Java solution, but it uses a specific protocol rarely documented. They are provided by the database vendor.

7.6 Three steps to access the data base

7.6.1 Connecting to the database

`java.sql.DriverManager` match a driver implementation with the requested URL

`java.sql.Connection` control all the connection from the application to the database engine.

A series of SQL statements can be send through a single connection.

7.6.2 Database access

`java.sql.Statement` send a single SQL command to the database

`java.sql.ResultSet` fetch back the results from the previous statement.

Individual data have to be retrieved separately.

7.6.3 Cleaning up

`Connection`, `Statement` and `ResultSet` have `close()` method.

They may not be necessary, but are always useful to get rid of resources.

`Statement.close()` will close all pending `ResultSet`

`Connection.close()` will close all pending Statement

Any uncommitted transaction will be lost.

7.7 Exception handling

`SQLException` is a catch-all exception for database errors.

In addition to the common error informations, `SQLException` provides database specific informations, such as SQLState and database engine specific errors.

Use something as follows after most JDBC calls :

```
catch( SQLException e ) { e.printStackTrace(); }
```

A more detailed `SQLException` handling will be shown later (see 7.19, page 33).

7.8 Connecting to a database

For a connection, we need a driver and the URL for the specific database. We assume that the driver for MySQL has been installed.

First, we instantiate the corresponding driver, then we open the connection

"`rinus`" is the the MySQL user name, and "`NotMe`" his password. They are independent from Unix users and passwords.

```
Connection con = null;
try {
  String driver = "it.ictp.sql.mysql.MysqlDriver";
  String url = "jdbc:mysql://infolab03.ictp.it/";

  Class.forName(driver).newInstance();
  con = DriverManager.getConnection( url, "rinus", "NotMe" );
    /* some SQL  statements */
  con.close();
}
catch( SQLException e ) { e.printStackTrace(); }
```

7.9 Inserting/updating data

When the connection is open, we can send as many SQL statements as desired.

To insert or update rows, we use the `Statement.executeUpdate()` method:

```
Statement s = con.createStatement();

String composer = "Vivaldi";
String title = "La Folia";
int update_count =
  s.executeUpdate( "INSERT INTO music (composer, title) " +
                   "VALUES(" + composer + ", '" + title + "')" );

String composer = "Mozart";
String title = "Requiem";
int update_count +=
  s.executeUpdate( "INSERT INTO music (composer, title) " +
                   "VALUES(" + composer + ", '" + title + "')" );
...

System.out.println(update_count + " rows inserted.");
```

7.10 Retrieving data

To retrieve rows from the database, we use the `executeQuery()` and `ResultSet` objects:

The `ResultSet` is one or more rows returned by the database query.

The class provides a set of methods for retrieving columns from the successive rows.

```
Statement select = con.createStatement();
ResultSet result = select.executeQuery
                    ("SELECT composer, title FROM music");

System.out.println("Got results:");
```

7.11 Retrieving individual data

The methods for retrieving a column are all of the same form:

```
ResultSet.get<type>( <Int>|<string> )
```

where `<type>` is any Java data type (`Int, String, ...`) corresponding to a SQL type. The parameter itself can be either an `Int` representing the column number, or a `String` with the column name.

For Example:

```
int key = result.getInt(1);
String val = result.getString(2);
```

Notice that using column number instead of column name is a rather bad idea, that goes against the physical independence required by Codd. The user should not be aware of the the order of the columns in the tables. In particular, if a column is deleted in the middle of a table, then all other columns on the right will change their positions, and all statements using position for them will return wrong results.

7.12 Moving forward and backward through the ResultSet

Only one row is available at once with get<Type>() methods. It is possible to move forward to the next one, or backward to the previous. Just after the query statement, the cursor is positioned before the first row.

next() move to the next row, and return "true" if one is available for processing.

previous() move to the previous row, and return "true" if one is available for processing.

afterLast() move the cursor after the last row.

beforeFirst(), first(), last() move the cursor according to the method's name.

absolute(), relative() move the cursor in the same way as above, but with an Int parameter specifying the place of the next row.

A negative param for absolute() indicates a row relative to the last one.

7.13 Where are we ?

Finally, we have methods to retrieve the current position of the cursor:

isFirst(), isLast(), isBeforeFirst(), isAfterLast(), getRow()

Except getRow(), they return a boolean.

getRow() returns the current row number as an integer.

7.14 Scrollable ResultSet

createStamenent() can have two parameters: scrolling and concurrency modes.

7.14.1 Scrolling mode

```
ResultSet.TYPE_FORWARD_ONLY,
         .TYPE_SCROLL_SENSITIVE,
         .TYPE_SCROLL_INSENSITIVE
```

With TYPE_FORWARD_ONLY, next() is the only method available.

With TYPE_SCROLL_SENSITIVE, any change made locally to the ResultSet becomes visible while scrolling back to the modified rows.

7.14.2 Concurrency mode

```
ResultSet.CONCUR_READ_ONLY, .CONCUR_UPDATABLE
```

With .CONCUR_UPDATABLE, rows from the ResultSet can be modified in place.

The default is (ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY)

7.15 Helping the driver with Scrollable `ResultSet`

A few methods are available to help the driver, in particular when the `ResulSet` is expected to be large:

```
setFetchDirection()
```

with the possible parameters:

```
ResultSet.FETCH_FORWARD
ResultSet.FETCH_REVERSE
ResultSet.FETCH_UNKNOWN
```

An other method fixes the maximum size of the returned `ResulSet`:

```
setFetchSize()
```

The default parameter is ”0” and is ignored, and the driver is expected to make its best.

If you expect to use only the very first or last rows from a large returned Set, then this method can speed up considerably the process.

7.16 Retrieval example (reading backward)

```
Statement stmt;
ResultSet rs;

stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
rs = stmt.executeQuery("SELECT * from music ORDER BY composer");

rs.afterLast();
while(rs.previous()) {
  int a= rs.getInt("test_id");
  String strc = rs.getString("composer");
  String strt = rs.getString("title");

  System.out.print( "test_id= " + a );
  System.out.println( "Composer = " + strc + "Title = " + strt );
}
```

7.17 SQL and Java Datatypes

SQL Type	Java Type	SQL Type	Java Type
BIT	boolean	TINYINT	byte
SMALLINT	short	INTEGER	int
BIGINT	long	REAL	float
FLOAT	double	DOUBLE	double
DECIMAL	math.BigDecimal	NUMERIC	math.BigDecimal
CHAR	lang.String	VARCHAR	lang.String
LONGVARCHAR	lang.String	DATE	sql.Date
TIME	sql.Time	TIMESTAMP	sql.Timestamp
BINARY	byte[]	VARBINARY	byte[]
LONGVARBINARY	byte[]		

The following definitions are available only with SQL 3

BLOB	sql.Blob	CLOB	sql.Clob
ARRAY	sql.Array	REF	sql.Ref
STRUCT	sql.Struct		

7.18 SQL and Java "null"

In the relational model, the "NULL" concept is important. It represents the absence of value, an indication that the value is unknown. It is different from a count of "0", or from an on purpose empty field.

Java has no way to represent directly this concept.

The method `wasNull()` after any retrieval will return "true" if it was a SQL "NULL"

Usually, `getInt()` itself will return "0" and `getString()` an empty string, which cannot be distinguished from "NULL".

`wasNull()` should be used after each `getXXX()` on column that can return a "NULL".

7.19 SQL Exception (2)

SQLException class extends the general lang.Exception class with informations about database errors. This includes:

- the SQL_State string describing the error according to the XOPEN SQL_State conventions
- the database specific error code, usually a number. The `Exception` class `getMessage()` can turn it into more readable form.
- a chain of exceptions leading to the last one.

Here is a small code to handle these exceptions:

```
catch( SQLException e ) {
  while( e != null ) {
    System.err.println( " SQLState: " + e.getSQLState() );
    System.err.println( "     Code: " + e.getErrorCode() );
    System.err.println( "  Message: " );
    System.err.println( e.getMessage() );
    e = e.getNextException() ;
  }
}
```

7.20 Warnings

The class SQLWarning is an extension to the `SQLException` class.

A warning is an indication that something could be wrong, but is not fatal at this point.

The method `getWarnings()` call be called repeatedly until it returns a null.

7.21 Time representation

Time representation is an other very loosely defined concept in SQL .

`java.sql.Date. sql.Time`, `sql.Timestamp` extend the functionality of other Java objects and provide a common representation of their SQL counterparts independently of the idiosyncrasies of the database system.

The granularity is the day for `Date`, the second for `Time`, and the nanosecond for `Timestamp`.

Various methods are available to retrieve them, compare them and put them into more readable form.

The internal value of all three time representation is ”long”.

8 SQL Database Systems in the PD

Many relational database systems exist for Linux, some sold for various prices, but some freely available. IBM offers a Linux version of their flagship DB2 rdbms, essentially the same as for the large computers. Borland has also brought out recently a rdbms for Linux. Oracle has both 32 and 64 bits versions for Linux single and clustered computers. This is also true for Ingres. But we will concentrate more on three products typical of the free software, where users are also supposed to be part of the debugging and development. I will describe them shortly, and end with a rapid comparison.

8.1 mSQL

mSQLwas the first of its kind of cheap, lightweight and fast rdbms designed from scratch with modern tools. It has a good interface with C, Java (with JDBC API), Perl and Python, but it lacks some rare features of the standard SQL definitions. While the client/server model is well supported, as are the connections with the web, it is not intended for very large databases, or for too many simultaneous users. It is not multi-threaded, but built on multi daemons for fast access by many users.

8.2 MySQL

MySQLcame out about a year after mSQL, for similar reasons. Commercial rdbms were too complex, too heavy, and above all too expensive for a small company. While simplicity was a main moto for mSQL, speed was important for the development of MySQL. It does not support yet the full SQL standard, but is nearer than mSQL. It is multi-threaded, and intended for large databases and supports a large number of simultaneous users.

MySQL has been ported to many other operating systems, not just Linux, but its interface to other languages is not as wide.

8.3 PostgreSQL

PostgreSQL had a very different development. When Stonebraker left the university development of Ingres to a more commercial team, he started looking for new ideas, in particular to integrate objects into rdbms.

Out of this project came the first Postgres and its query language PostQuel. Stonebraker was never convinced about SQL as it was defined and always preferred his own language (contradicting Codd rules). Postgres almost disappeared a few years ago, mostly because every one wanted to use SQL . Then two graduate students of Professor Stonebraker designed an SQL frontend for Postgres, and many more users became interested. PostgreSQL is an object-relational database system, that supports transactions, triggers and subselect, all missing in the previous rdbms. This comes at some price; it is heavier and more complex than the other ones.

8.4 Comparison

Any comparison is difficult in this field because the new developments are done by many users worldwide in a not too much structured way (the “bazaar”); it is difficult to follow all the novelties and improvements of each of them. For a while, the order set above corresponded to their increasing complexity and intended size of use. PostgreSQL was also the slowest. But all three are improving full speed,

and show no intention to get out of the way. More recently, tests indicated that PostgreSQL was now the fastest.

Although benchmarks tend to mimic as well as possible the normal expected work load of a rdbms, they are probably not equivalent to your intended use (if you are able to define it). It should also be noted that the differences in speed are usually small compared to differences between two computers bought 6 months apart.

Finally, the comparisons with commercial rdbms indicate that, if the three rdbms cited above are not so rich in subtle commands and less ambitious toward very large applications, they are usually faster, sometimes much faster than the commercial ones. They are also easier to install and administer.

8.5 Where to look for informations

Here is a set of Internet links of interest with minimal comments.

For more information, look at meta-servers like *google* or *altavista*.

http://www.hughes.com.au/ for mSQL. Sources only.

http://www.mysql.com/ for MySQL, though many other sites have uptodate `.tar` files.

http://www.postgresql.org for PostgreSQL. They prefer to send a CD than provide `.tar` files.

http://www.ispras.ru/~kml/gss/index.html for the GNU SQL project. This work is currently done at the Institute for System Programming of the Russian Academy of Science. Supposed to cover the full SQL standard.

http://www.beaglesql.org another project to cover the full SQL , with options from the work done on objects by the PostgreSQL group.

http://www.mysql.com/crash-me-choose.html A set of tests to verify the stability of a rdbms.

http:/people.ee.ethz.ch/~oetiker/webtools/rrdtool/ Round Robin multi-resolution databases.

http:/www.rsw.com/ This is the original /rdbpackage to manipulate flat ASCII databases from shell scripts. It is a commercial product, but very fast and efficient. *C* interface available as well. See the book by Manis et al (10.2, page 40).

http://cfa-www.harvard.edu/~john/starbase/starbase.html another implementation of the same concept, optimised for astronomical applications, also very fast, but some versions were not very robust. *C* interface available as well.

http://www.isi.edu/~johnh/SOFTWARE/JDB/ Another rdb package with the same concept.

http://www.linux.it/~carlos/nosql/ Still another rdb package with the same concept, all written in *perl*.

http://www.cs.uu.nl/pub/PERL/ still another one, also in *perl*.

http://www.kmarshall.com/easy/cgi/ gives useful information concerning cgi and database access.

9 Other Data Base models

9.1 Object Oriented Data Base Systems

There is a clear tendency to move rapidly towards object programming in all aspects of program developments, and databases being at the heart of most of them, many projects on bringing together the two concepts have emerged. In most cases, it consists of putting an object layer over the relational model, but some developments exist for very large databases (in the Peta Bytes region) where the original relational model seems to reach its limits and only new, pure object databases would be realistic.

9.2 Round Robin Data Base Systems

On many occasions, we are interested to keep historical data with various time resolutions, very high for the last minutes or hours, intermediate over days or weeks and lower over month or years. Round Robin databases have been created to solve automatically just this problem. To my knowledge, they were first introduced for controlling network activity, but they can be used for other tasks where the time enter as a sort of primary key.

During the database creation phase, different sub-databases are defined, all with the same elements, but with various time resolutions and validity. Fields have also two characteristics: type gauge or counter and attribute Min, Average or Max. This fixes from the beginning the total size of the database.

Then data can be entered at any time and rate, hopefully frequently enough to fill all the slots at the highest time resolution with at least one value.

If more than one value is entered in one time slot then the Minimum, Average or Maximum during this time slot are recorded. The same mechanism works for all other resolutions. When old data expire the validity at a given resolution, they are automatically replaced by the new ones. When no value is entered during a time slot, then either a *NaN* is recorded, or the value is interpolated between the previous and next slot. The maximum size of the gap for interpolation can be fixed during the definition phase.

For further information, downloading etc. (see section 8.5, page 36), look at: `http:/people.ee.ethz.ch/~oetiker/webtools/rrdtool/`

9.2.1 Monitoring temperatures

Here is a practical example of using a round robin database (rrd): We want to record the chip temperature in our computer over long term (the life of the computer) but a weekly indication is then sufficient. On the other hand, it would be useful to keep track of the temperatures over the last 48 hours with a resolution of 5 minutes and over a month with a resolution of 6 hours.

```
rrdtool create lab01.rrd      \
  --start 1002190000          \  beginning time
  --step         600          \  time slot width
  DS:Die:GAUGE:1200:0:100     \  chip temp, max interpolation over 10 min
  DS:Amb:GAUGE:1200:0:100     \  same for ambient temp
  RRA:AVERAGE:0.5:1:576       \  average over 1 time slot (5 min) for 48 h
  RRA:AVERAGE:0.5:72:124      \              72            6 h        31 d
  RRA:AVERAGE:0.5:2016:1460   \            2016            1 w         4 y
  RRA:MAX:0.5:1:576           \  same for Max temp
  RRA:MAX:0.5:72:124          \
  RRA:MAX:0.5:2016:1460
```

A `cron` command was recording the temperature every minute. The command looks like (time is given in Unix seconds):

```
time=‘date +%s‘
rrdtool update lab01.rrd $time:$die:$amb
```

Here is the results of fetching the data back with different resolutions. The command is given only once, and the results have been reformatted.

```
rrdtool fetch --start now-10h lab01.rrd AVERAGE

  date      time       Die     Amb
  --------  --------   -----   -----
  20011120  13:00:00   53.17   37.00
  20011120  13:05:00   53.50   37.00
  20011120  13:10:00   53.64   37.00
  20011120  13:15:00   53.67   37.00

  date      time       Die     Amb
  --------  --------   -----   -----
  20011112  21:00:00   54.17   37.50
  20011113  03:00:00   53.49   37.00
  20011113  09:00:00   53.46   37.00
  20011113  15:00:00   53.50   37.00

  date      time       Die     Amb
  --------  --------   -----   -----
  20011006  02:00:00   55.12   39.00
  20011013  02:00:00   54.67   39.00
  20011020  02:00:00   54.67   39.00
  20011027  02:00:00   54.98   39.00
```

The `rrdtool` package contains options for creating, updating, fetching, graphing databases, as well as moving data from/to ascii external and binary internal

formats. It is very easy to transform the results into /rdb format or simpler tables for SuperMongo or Octave.

`rrdtool` is also at the heart of `mrtg`, a package widely used to monitor with various time resolutions the traffic going through nodes in a network. It is available from the same place as the `rrdtool`.

10 Bibliography

The following list is not an up-to-date bibliography on the subject. New books and articles are appearing every day on database, SQL or JDBC.

They are a personal choice, based on my experience. Most are much shorter that current publications. They all bring and emphasize some important aspects on the subject. A few are here also for historical reasons.

10.1 Relational Database

- **Codd E. F.** A Relational Model of Data for large Shared Data Banks. *CACM,* **13, No 6, 377-387, June 1970**

- **Codd E. F.** Relational Database: A Practical Foundation for Productivity. *CACM,* **25, No 2, 109-117, February 1982**

- **Date C. J.** Relational Data Bases: Selected Writing. *Addison-Wesley 1986*

- **Date C. J.** A Guide to Ingres. *Addison-Wesley 1987*

- **Parsaye K., Chignell, M., Khoshafian, S., Wong, H.** Intelligent Databases. *Wiley 1989*

- **Stonebraker M** The Ingres Papers: Anatomy of a Relational Database System. *Addison-Wesley 1985*

10.2 Unix Flat Tables

- **Manis R., Schaffer E., Jørgensen** Unix Relational Database Management, Application Development in the Unix Environment. *Prentice-Hall 1988*

10.3 SQL

- **Bowman J. S., Emerson S. L. and Darnovsky M.** The Practical SQL . *Addison-Wesley 2001*

10.4 CGI programming

- **Scott, G., Gundavaram, S. and Birznieks, G.** CGI Programming with Perl. *O'Reilly 200*

10.5 JDBC

- **White S., Fisher M, Cattell R., Hamilton G. and Hapner M.** JDBC API Tutorial and Reference, Second Edition. Universal Data Access for the Java 2 Platform, The Java Series (SUN microsystems), *Addison Wesley (1999)*, ISBN 0-201-43328-1

- **Reese G.** Database programming with JDBC and Java , Second Edition. *O'Reilly (2000)*, ISBN 1-56592-616-1

10.6 Public Domain

- **Yarger R. J., Reese G. and King T.** MySQL& mSQL. *O'Reilly 1999*

- **Reese, G., Yarger, R. J. and King T.** Managing and Using MySQL. *O'Reilly 2002*, ISBN 0-596-00211-4

- **Reese, G.** MySQL Pocket Reference. *O'Reilly 2003*, ISBN 0-596-00446-X

10.7 Other

- **Bentley Jon** Programming Pearls. *Addison-Wesley 1989*

- **Bentley Jon** More Programming Pearls, Confessions of a Coder. *Addison-Wesley 1988*

- **Wall L., Christiansen, T, Schwartz, R. L.** Programming Perl *O'Reilly & Associates 1996*

- **Burke E. M. and Coyner B. M.** Java Extreme Programming Cookbook. *O'Reilly 2003*, ISBN 0-596-00387-0

Index