Chapter 4

SQL

Essential reading

"An Introduction to Database Systems", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter3 – section 8 (pp. 65-68) and Chapter 8* (pp. 219-257).

Alternatively

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 13 and Chapter 14* (pp. 383–462).

Further reading

A clear and easy to read description of SQL is given in:

"LAN Times – Guide to SQL", by J. Groff and P. Weinberg, published by Osborne McGraw-Hill, 1994, [ISBN 0-07-882026-X]

For your implementation work, use the SQL reference manual(s) of the DBMS you have chosen / have access to.

Introduction

The most popular database language is, by far, SQL (Structured Query Language). Almost every extant commercial (relational) DBMS implements it in a form or another, so, it is a necessary requirement for you to know, if not in depth, at least some if its main features. For, you have to be aware that "[...] SQL is an enormous language. The standard document (ISO 1992) itself is over 600 pages long" (Date, 1995, p.219). In this chapter you can only be given a glimpse of what SQL is. Should you be interested in finding out more, refer to the reference manual of the SQL dialect you use for implementing the examples and activities of this chapter.

The authentic name for it is *International Standard Database Language SQL 1992* (ISO 1992), but it is often referred to as "SQL-92", "SQL/92" or "SQL2". The name is sometimes pronounced "sequel".

SQL-92 is a relational language in that it is founded on the relational model, more specifically on relational algebra. However, it is not a pure implementation of it; some features of the relational model were left aside, whereas some features not belonging to the relational model were added.

SQL-92 is not a specific implementation, but is "merely" an abstract specification of a relational database language. Specific DBMSs support specific implementation of SQL-92, called *dialects*, which, more or less, comply with the standard. In principle, there should be a hundred percent compliance with the standard, but unfortunately, this is not the case in practice.

Vendors have different views upon the importance of different aspects of SQL2. Accordingly, they only implement the "important" aspects, leaving aside the other ones. Also, the syntax of the language may present slight variations from the standard.

In parallel, vendors also try to empower their implementation with "extra" or "enhanced" (not specified by SQL2) features, called *extensions*. Therefore, even though

implemented in SQL, databases developed in different dialects of SQL might not be portable from a system to another.

The standard SQL has two main components: the data definition (DDL) and data manipulation (DML) component.

The DDL supports:

- definitions at the conceptual level;
  - *relations* are well supported (*base relations*, in fact);
  - there is little support for *domains*;
- definitions at the external level;
  - *views* – views, essentially, are relations defined on other relations, their extension being not explicitly stored in the database (it has to be computed every time the view is used);
- definitions at the internal level;
  - e.g. indexes; in this chapter there are no references to this component.

The DML is used for database query and maintenance (data insertion, deletion and modification / updating). The DML implements relational algebra, in that relational algebra expressions can be used for data manipulation. SQL also provides support for integrity constraints definition.

SQL is a fourth generation language. One of its big advantages over third generation languages, which makes it so powerful in terms of data management, is its ability to manipulate sets - its operators are *set level*, or *set at a time*. Moreover, because it is meant to be quite high level, no explicit support for the flow of control is provided (e.g. IF-THEN statements or WHILE loops) – i.e. it "lacks computational completeness" (Connolly 1999, p.736). Therefore, SQL has to allow for some of its statements to be *embedded* in programs written in third generation languages. That is, SQL provides primitives that can be accessed from inside such programs; it is said that "the standard SQL provides embedded SQL". If no complex manipulation of data is needed, all that is required being data definition, query and updating, SQL can be used *interactively*.

A different terminology, with regard to the relational model, is being used by the language: *table*, *row* and *column* are being used for relation, tuple, and attribute, respectively. Two important differences from the relational model should also be mentioned here: SQL allows duplicate tuples and the rows and columns of a table are ordered. Constants are called *literals* and they can be numeric or non-numeric. All non-numeric values have to be specified within quotes. SQL is *case insensitive*, but, for clarity, you should make use of upper and lower case (it is customarily that keywords are written in upper case letters and user defined words in lower case letters). To improve the readability of the statements, make use of *indentation* (alignment) and *comments* too. A comment starts with two dashes, i.e. "--" and continues to the end of the line.

In presenting the syntax of SQL we shall employ a BNF[1] (Backus Naur Form) notation which uses the following conventions:

- all SQL keywords are written in UPPERCASE letters;
- syntax elements (or place holders), are enclosed within "<" and ">";
- optional parts of a statement are enclosed within "[" and "]";
- alternative options are represented as "{ option1 | option2 | ...}";
- the symbol "::=" reads "is defined by" or simply "is".

SQL makes extensive use of a syntactical structure – list of elements (one or more) of the same kind, separated by a comma. This is represented in BNF as:

---

[1] In Chapter 5 you will study about the Boyce-Codd Normal Form, in short BCNF. Make sure you do not make confusions between BNF and BCNF.

```
<comma-list-of-elements> ::=
      <element> |
      <element> , <comma-list-of-elements>
```

To make the presentation of the syntax more concise, we shall also use the following convention:

- a list of elements (one or more) separated by comma is represented as "@<element>"; thus:

```
@<element> ::= <element> | <element> , @<element>
```

For illustrations purposes, the presentation of each issue is accompanied by examples. You are advised to try to devise your own set of examples to augment these. This is the best method of learning a new language: it will help you to better understand and to memorise.

It is not sufficient to devise examples with pen and paper. *You have to experiment.* Use an SQL dialect that is available for you (PostgreSQL, Oracle SQL, Microsoft Access SQL, etc.)

- use the generic SQL (in which the examples in this chapter are provided) to guide your experimentation;
- study the manuals of the dialect you are using to identify the differences (if any) from the standard SQL;
- adapt the examples in SQL-92 so that they can be executed by the DBMS you are using;
- *extend the provided examples* and *explore further*; this is the only suitable way of learning SQL.

SQL's DDL

The DDL component should support the creation / definition of the relational data objects – domains and relations. One of the big limitations of SQL is that domains are far from being properly supported. On the other hand, DDL allows the definition of certain integrity constraints that are not intrinsic to the relational model. Furthermore, the SQL's DDL is even more extensive (it allows the specification / definition of security rules, for instance), but these extensions are presented in following chapters of the second volume (under the heading "SQL support" for each topic in part). In this section we are only going to concentrate on domains, relations and integrity constraints.

Domains

SQL does not support domains as such; all it provides is a set of *basic data types*. There is a statement CREATE DOMAIN, but it is improperly named like this because all it does is to merely perform a *renaming* of a basic data type.

Data types

SQL provides the data types shown in Figure 1. They are called *basic, primitive, system-defined,* or *predefined* data types.

Type	KEYWORDS *for declarations*
character	CHAR, VARCHAR
bit	BIT, BIT VARYING
exact numeric	NUMERIC, DECIMAL, INTEGER, SMALLINT
approximate numeric	FLOAT, REAL, DOUBLE PRECISISION
date-time	DATE, TIME, TIMESTAMP
interval	INTERVAL

Figure 1: (Basic) Data types in SQL

Only two data types are described in the following paragraphs, namely character and exact numeric.

Activity: Since slight variations might exist from a dialect to another, investigate the data types provided by the dialect you are using.

For a more complete description of the standard SQL data types refer to (Connolly, 1999) pp. 427 – 431.

Character

Data of the type character is represented by string, e.g. "this is A String Of Characters". The legal set of characters varies from dialect to dialect (it is vendor specific). The syntax for declaring data objects of the character type is:

```
CHARACTER [VARYING] [<length>]
```

Note that

CHARACTER	can be replaced by	CHAR
CHARACTER VARYING	can be replaced by	VARCHAR

If VARYING is not specified then the string of characters has a constant length as specified by <length>. If a value with less characters is provided, for a constant length string, then the string is brought to the specified size by adding blanks; if the number of characters exceeds the specified size then the value is truncated to the right. E.g.

```
attribute CHAR(10);
attribute = "smaller";
--- the actual value of attribute is "smaller___"
--- the underscore was used to illustrate blanks
attribute = "this is bigger"
--- the actual value is "this_is_bi"
```

If VARYING is specified, then the string is of a variable length, but not exceeding the specified one. That is, if a value with less characters than the specified size is provided, the string will not be added any blanks, resulting in a smaller size. E.g.:

```
attribute VARCHAR(10);
attribute = "smaller";
--- the actual value is "smaller"
attribute = "this is bigger";
--- the actual value is "this_is_bi"
```

You know by now that a data type specifies also the scalar operators that can be applied to its data. Some of the SQL operators that are applied to data of CHARACTER type and produce results of CHARACTER type are shown in Figure 2.

Operator	Description	Example
CHAR_LENGTH	returns the length of a string in characters	CHAR_LENGTH("this_time") *returns 9*
||	concatenates two strings	"this" || "time" *returns "thistime"*
LOWER	converts upper-case letters to lower-case	LOWER("NOt GOod") *returns "not good"*
TRIM	removes (leading, trailing or both) characters from a string	TRIM (LEADING "X" FROM "XXXtitleXXX") *returns "titleXXX"*
POSITION	returns the position of one string within another	POSITION("a" IN "what") *returns 3*
SUBSTRING	returns the substring of a string	SUBSTRING("J._Smith", FROM 4 TO 8) *returns "Smith"*
CURRENT_USER USER	returns a string representing the current user name	

Figure 2: Some operators for the character data type

Exact numeric

Data of this type is represented by numbers with an exact representation, i.e. numbers guaranteed to have a certain *precision* and a certain *scale*. By precision it is meant the (total) number of relevant (or significant) digits, whereas by scale it is meant the number of decimal places. For instance 123.4567 has precision 7 and scale 4, whereas -0.012 has precision 4 and scale 3. As a special case of exact numbers, integers have inherently no decimal places (scale zero). They can be represented more economically (requiring no decimal places), therefore they are treated separately. The exact numeric data definition format (syntax for declaring the exact numeric data type) is:

```
NUMERIC [ <precision> [, <scale>]]
DECIMAL [ <precision> [, <scale>]]   --can be abbreviated to DEC
INTEGER                              --can be abbreviated to INT
SMALLINT
```

NUMERIC and DECIMAL specify decimal numbers, whereas INTEGER and SMALLINT specify integer numbers. SMALLINT represents integer values whose absolute maximum value is relatively small. The advantage of using this type is storage space economy. For example, the ID for the employees of a company could be a SMALLINT, whereas a current account balance in Pound Sterling could be a decimal number, precision 8, scale 2 (the biggest sum of money that this account can hold is 999,999.99 Pounds), as in:

```
Employee-ID     SMALLINT;
Balance         DECIMAL(8,2);
```

The most used numerical operators are addition "+", subtraction "-", multiplication "*" and division "/".

It is not necessary that the result of an expression is of the same type as its operands. An important case is represented by the truth values expressions; they can have operands of different types but they evaluate to a Boolean value (true or false). For instance, for the NUMERIC type, we have the comparison operators "<", ">", "=", etc.

Activity: Find out about other operators that result in a Boolean value (they are particularly useful in retrieving statements and integrity constraints).

Before moving to domains, another important operator must be mentioned: it is the operator that changes a data value of one type into another, the CAST operator. Its syntax is:

```
CAST (<values> AS <data type>)
```

For instance, the statement that changes the decimal value 12.3 into an integer value is:

```
CAST (12.3 AS INTEGER)
```

Domains

Domains are not supported in SQL-92. However, there exist a statement (a rather inappropriate name) CREATE DOMAIN which, in the main, solely creates a synonym to a basic data type. The syntax for creating a domain is:

```
CREATE DOMAIN   <domain name> <data type>
                [DEFAULT <default value>]
                [@<domain constraint>] ;
<domain constraint> ::= CONSTRAINT <name> CHECK <integrity constraint>
```

where

- <data type> must be one of the basic SQL data types
- <default value> specifies a scalar value, of the type <data type>; it can be a constant, a built-in function with of no arguments (e.g., USER) or NULL; the value of a function with no arguments depends on the context (e.g., the value of USER depends on who is the current user of the system and CURRENT_DATE changes every day);
  the value of a constant is independent on the context (e.g. "M. Smith");
  the use of default values is explained in the following section (but, generally, they are used when the value for an attribute is not provided);
- <integrity constraint> specifies a constraint that applies to any value of this type; it has to be a truth-valued expression; the standard definition of SQL does

not require this constraint to be specified only on the values of the domain, but it allows for this it to be a truth valued expression of *any complexity*[1]; a positive consequence of it is that more complex restrictions can be specified as domain constraints; the negative consequence of this liberty is that the programmer can develop a rather undisciplined code.

For instance, to define a domain for the number of hours per week that an employee can work (supposing that there is a minimum limit of 10 hours and a maximum limit of 50), you can use:

```
CREATE DOMAIN   Hours_Per_Week SMALLINT
                DEFAULT 40
                CONSTRAINT Legal_Hours
                        Value > 10 AND Value < 50 ;
```

A domain can be destroyed by means of DROP DOMAIN statement. The definition of a domain can be altered (change the default value or add / remove an integrity constraint) by means of a ALTER DOMAIN statement.

Some of the differences between the concept of domain, as defined by the relational model, and domain as defined by SQL have to be pointed out (in the following bullet point list the word "domain" is used to mean domain as defined by SQL):

- domains are, just another name for a basic data type (if the default value and constraint specification are disregarded); so, any definition that is made using domains, can be made without their use; this means that:
  - domains do not support the definition of user defined operators;
  - a definition of a domain cannot be made based on other domains;
  - there is no strong typing associated with domains; the only requirement for operations on scalar data is that they are made on the same *basic* type;
  - no features of inheritance are associated with domains.

In conclusion, an important concept of the relational model – the "domain" – is, in fact, not supported in SQL-92. .

Base Relations

Create a table

Relations are implemented as tables in SQL-92, by means of a CREATE TABLE statement. The syntax for this statement is (some "restricted" versions of it were presented in the previous chapter):

```
CREATE TABLE  <table name>
      (
      @<column definition>,
      <primary key definition>,
      [ @<candidate key definition>, ]
      [ @<foreign key definition>, ]
      [ @<table integrity constraint definition> ]
      ) ;

<column definition> ::=
      <column name>  {<basic data type> | <domain>}
      [DEFAULT <default value>] [NOT NULL] [UNIQUE]
      [<column constraint definition>]

<column constraint definition> ::=
      CHECK <truth valued expression>

<primary key definition> ::=
      PRIMARY KEY ( @<column name> )

<candidate key definition> ::=
      UNIQUE ( @<column name> )
```

[1] Refer to the "Integrity constraints" section in this chapter.

```
<foreign key definition> ::=
    FOREIGN KEY ( @<column name> )
    REFERENCES <base table> [ ( @<column name> ) ]
    [ ON UPDATE <option> ]
    [ ON DELETE <option> ]

<option> ::= {CASCADE | SET NULL | SET DEFAULT | NO ACTION}

<table integrity constraint definition> ::=
    [CONSTRAINT <constraint name>] CHECK ( <truth valued expression> )
```

Explanations:

- a table must have at least one column and the primary key has to be specified;
- column definition:
  - a column must have a name and a type associated with it;
  - optionally, a default value can be specified; this will be used when a row is inserted in the table but no value for the respective column is provided;
  - NOT NULL specifies that the respective column does not allow null values, i.e. a value must be provided for each row;
  - UNIQUE specifies that the respective column does not allow duplicate values (i.e. it is a (non-composite) candidate key)
  - a column constraint specifies an integrity constraint on the respective column; however, SQL allows the specification of a truth valued expression of *any complexity* as a column integrity constraint (similar to domains);
- a primary key is unique for a given table;
- candidate keys can be specified by means of the UNIQUE statement;
- foreign key definition:
  - the referred candidate key columns in the target table can be specified within brackets, after the specification of the target table's name;
  - the specification of foreign key rules is discretionary;
  - the CASCADE and NO ACTION foreign key options correspond to CASCADE and RESTRICT as discussed in Chapter 3;
  - SET NULL and SET DEFAULT set to NULL or to the default value, respectively, the values of the foreign key attributes for the tuples that refer to the tuple in the target relation that has been deleted or whose candidate key has been updated.
- a table integrity constraint specifies an integrity constraint upon the respective table and optionally can have a name; however, SQL allows the specification of a truth valued expression of *any complexity* as a table integrity constraint (similar to column integrity constraints).

For illustration, consider the creation a table "Student":

```
CREATE TABLE Student
    (
    student_id          Id_for_Students,        --domain
    first_name          CHARACTER(10)           NOT NULL,
    last_name           CHARACTER(10)           NOT NULL,
    date_of_birth       DATE                    NOT NULL,
    address             VARCHAR(50),
    course_id           CHARACTER(15)           DEFAULT 'not reg',
    department_id       CHARACTER(15)           DEFAULT 'not reg',
    PRIMARY KEY         (student_id),
    UNIQUE              (first_name, last_name, date_of_birth),
    FOREIGN KEY         (course_id) REFERENCES Courses
                        ON UPDATE NO ACTION
                        ON DELETE NO ACTION,
    FOREIGN KEY         (department_id) REFERENCES Departments
                        ON UPDATE CASCADE
                        ON DELETE NO ACTION,
    CONSTRAINT          valid_DOB
```

```
        CHECK (     EXTRACT (YEAR FROM date_of_birth) > 1934
                    AND
                    EXTRACT (YEAR FROM date_of_birth) < 1980 )
    ) ;
```

The student_id column is defined on a user defined domain and is used as a primary key. The first and last name and the date of birth have to be provided for each student (hence NOT NULL); moreover, they together constitute another candidate key for this relation (hence the UNIQUE statement). If no value is specified for the course or the department for which a student is registered, it is assumed that this registration did not yet take place, therefore the default value "not reg" - meaning not registered - is used. The course_id and department_id are foreign keys in the Courses and Departments tables, respectively.

Activity: Explain why the foreign key option rules have been specified as above.

A constraint, called "valid_DOB" was imposed on the date of birth attribute, namely that the year should be greater than 1934 and smaller than 1981 (i.e. a student, in 1999, should be younger than 65 and older than 18).

Alter the definition of a table

The definition of a table can also be altered according to the following possibilities[1]:

- adding / removing a column;
- adding / removing a table integrity constraint;
- setting / dropping a default value for a column.

The syntax for these statements is:

```
ALTER TABLE <name>
    [ADD [COLUMN] <column definition>]
    [DROP [COLUMN] <column name> [RESTRICT | CASCADE]]
    [ADD [CONSTRAINT <constraint name>] <constraint definition>]
    [DROP [CONSTRAINT] <constraint name> [RESTRICT | CASCADE]]
    [ALTER [COLUMN] SET DEFAULT <default value >]
    [ALTER [COLUMN] DROP DEFAULT] ;
```

The same specifiers used in the above description (ALTER TABLE) were used in the description of the CREATE TABLE statement. Therefore, their explanation is not repeated here. However, some small clarifications are needed:

- <constraint definition> subsumes key specifications, as well (e.g. PRIMARY KEY);
- the RESTRICT option, for DROP COLUMN, specifies that if there are (other) tables that refer to the column that is to be dropped (e.g. a foreign key, a view definition, etc.) then the DROP statement should not be allowed to be executed;
- the CASCADE option, for DROP COLUMN, specifies that the statement should be executed and all the columns, in all the tables of the database, that refer to the column that is to be dropped should be dropped as well; use it with care!
- RESTRICT and CASCADE work similarly for the DROP CONTRAINTS statement.

Destroying a table

A table is destroyed by means of a DROP statement:

```
DROP TABLE      <table name>      [CASCADE | RESTRICT] ;
```

If CASCADE is the option used, then all data objects that refer to the table to be dropped, and all data objects that refer to them, and so on, have to be dropped as well. In the case of RESTRICT, the statement is not allowed to be executed if there are other data objects that refer to the table to be dropped.

As a final remark, the tables created by the CREATE TABLE statement are called base tables (corresponding to base relations):

---

[1] It is possible that the dialect you are using provides a different (perhaps richer) syntax for altering a definition; refer to the manual.

- their definition is explicitly provided, in that it does not depend on the definition of other tables;
- they are physically stored in the database.

As we shall see in another section, there are also other kind of tables that can be created in SQL, such as views.

Activity: Choose an application (e.g. shop, hospital, library, school, etc.) and define in SQL-92 the relevant data objects you identified. Implement them in the dialect you have access to. Are there any differences? This activity is further developed in subsequent activities.

So far, you learnt how to define / create relational data objects (including certain integrity constraints, as well) in SQL. In other words you know how to define the schema of a database in SQL. The next section describes how data can be manipulated in SQL.

SQL's DML

The data manipulation language consists of two components. They define statements for retrieving data and statements for updating data, respectively. They all make use of relational algebra statements.

Retrieving operations

A database stores data. This data needs to be accessed by different applications / users for different purposes. It is essential to be able to retrieve data from the database.

Definition: A statement that specifies what data is to be retrieved (by the DBMS) from the database is called a query.

In SQL all queries are specified by means of a SELECT statement. The general format (syntax) of this statement is:

```
SELECT      [{DISTINCT | ALL}] { * | @<column name> }
FROM        @<table name>
[ WHERE     <where condition> ]
[ GROUP BY  @<column name> ]
[ HAVING    <having condition> ]
[ ORDER BY  @<column name> ] ;
```

SELECT is a very *powerful* and *versatile* statement. It can be used in conjunction with scalar operators and functions, with set operators, it can allow nested sub-statements, etc. A general description would be, somehow, inappropriate here, because it would have to be rather extensive. So, the power of the statement is going to be revealed little by little, by means of examples, from simple to more complex. We shall start firstly by illustrating how the relational algebra operators are implemented via the SELECT statement. Then we shall consider a series of queries in natural language (NL) and show how they can be implemented by means of the SELECT statement. Throughout this section we are going to use the miniature database from Figure 3, describing the information about the students of a university.

Students

S_id	S_name	DOB	Fees_paid	Department	Level	Average_mark

Modules

M_id	M_name	Type	Level	Value	Department

Registrations

S_id	M_id	Result

Figure 3: The tables used to exemplify the SQL's DML

Relational algebra operators

Three relational specific operators are implementable straightforwardly:

- *Restriction*

The relational algebra expression

```
<table name> WHERE <condition of restriction>
```

is implemented as

```
SELECT * FROM <table name> WHERE <condition of restriction> ;
```

where the symbol " * " specifies "all attributes / columns". For instance, the statement that selects all level 1 students is

```
SELECT * FROM Students WHERE Level = 1 ;
```

- *Projection*

The relational algebra expression

```
<table name> [ <list of columns> ]
```

is implemented as

```
SELECT <list of columns> FROM <table name> ;
```

For instance, the statement that selects the ID, name and address for all the students is:

```
SELECT S_id, S_name, Address FROM Students ;
```

- *Join*

The relational algebra expression

```
<table name 1> JOIN <table name 2>
```

is implemented as

```
SELECT * FROM <table name 1>, <table name 2>
WHERE <column in table name 1> = <column in table name 2>
        @[AND <column in table name 1> = <column in table name 2>];
```

For instance, the statement that provides all the information (including the modules for which they registered and the results they obtained) about all students is

```
SELECT * FROM Students, Registrations
WHERE Students.M_id = Registrations.M_id ;
```

Note that the scope (i.e. the table it refers to) had to be specified for each column name by means of the dot, ".", operator.

Union, *Intersection* and (set) *Difference* have, each, a corresponding operator implemented in SQL, namely UNION, INTERSECTION and EXCEPT, respectively. The SQL standard provides a special format of the SELECT statement for the Cartesian product, namely:

```
SELECT    [DISTINCT | ALL]  { * | @<column name> }
FROM      <table name 1> CROSS JOIN <table name 2> ;
```

Note that by this statement it is possible to achieve more than a Cartesian product. Strictly speaking, the Cartesian product is implemented by the following statement:

```
SELECT    DISTINCT  *
FROM      <table name 1> CROSS JOIN <table name 2> ;
```

A series of query requirements is presented in the rest of this section, in order to give a gist of the SQL's SELECT statement.

Eliminating duplicate rows

A table allows the existence of duplicate rows. If a table is not to contain any duplicate row then this fact must be explicitly stated by means of the DISTINCT keyword. For instance, the query

"Get all the departments that students have registered for and eliminate all duplicates"

is implemented as

```
SELECT DISTINCT  Department
FROM             Students ;
```

Ordering the rows

Since the rows in a table are ordered (as opposed to tuples in a relation), it is possible to require that the result of a query is ordered. There are two possibilities:

- ascending order – specified by "ASC"
- descending order – specified by "DESC"

For example, the query

"Get all the students ordered according to their descending average mark"

is implemented as

```
SELECT      S_id, S_name, Average_mark
FROM        Students
ORDER BY    Average_mark DESC ;
```

It is possible to have more than one ordering criteria, i.e. more than one *sort key*. The first sort key specified is called the major sort key and the others are called minor sort keys; the rows are ordered according to the major key, then within the groups that have the same major key value, the first minor key is considered, and so on. For example, the query

"Get all the students in ascending order according to their level of study, within that in descending order according to their average mark and, if they happen to have the same average mark then alphabetically in descending order"

is implemented as

```
SELECT      S_id, S_name, Average_mark
FROM        Students
ORDER BY    Level ASC, Average_mark DESC, S_name DESC ;
```

Note that the sort keys does not necessarily have to appear in the SELECT clause (e.g. Level, above).

The default order (i.e. when ASC or DESC are not stated) is ascending (ASC).

Check for non-specified values

I.e. check for nulls. In many situations, it is quite useful to be able to identify the tuples for which a certain value was not provided, i.e. the corresponding attribute is NULL (of course this attribute must not have the constraint NOT NULL). For instance, the following query

"Get all the students who are not registered with any department yet."

can be implemented as

```
SELECT    S_id, S_name
FROM      Students,
WHERE     Department IS NULL ;
```

Check for values in a set

A set of values is specifed within curly brackets, i.e. "{" and "}", and the set membership operator is "IN". For instance, the following query

"Get all the students who are registered with natural sciences departments (i.e. 'Maths', 'Physics', 'Chemistry' and 'Biology')."

can be implemented as

```
SELECT    S_id, S_name, Department
FROM      Students,
WHERE     Department IN {'Maths', 'Physics', 'Chemistry', 'Biology'}
ORDER BY  Department ;
```

Compound restriction condition

Rather than using a simple <where condition>, it is possible to combine more than one condition by means of logical operators, in truth valued expressions of any complexity. For instance, the following query

"Get all the students who are not registered with any department and paid some fees, have not paid any fees but are registered with a department, or have failed their year (i.e. Average_mark < 40)."

can be implemented as

```
SELECT    S_id, S_name
FROM      Students,
WHERE     (Department IS NULL  AND  Fees_paid > 0)  OR
```

```
        (Department IS NOT NULL   AND   Fees_paid = 0)  OR
        Average_mark < 40 ;
```

Activity: Investigate the set of comparison operators for each of the SQL's data types. They can be combined in an unlimited way (theoretically), by means of the logical operators AND, OR and NOT. Think of more complex queries (in terms of the where condition) and try to implement them in SQL. Experiment with the dialect you are using.

Calculated fields

It is possible to use the attributes of each tuple in an expression and, thus, to provide calculated fields as results. For instance, the following query

"Get for all the students the name, the year in which they were born and the rest of the fees they still have to pay (supposing that the fee per year is £5000)"

can be implemented as

```
SELECT    S_id, S_name,
          EXTRACT(YEAR FROM DOB) AS Year_of_bith,
          ( 5000 – Fees_paid ) AS Still_to_pay
FROM      Students ;
```

The keyword "AS" is used for naming the calculated fields.

Summaries - Aggregate functions

There are many real life applications in which summary information is required, such as the average weight, the maximum age, the minimum wage, etc. SQL provides a set of functions – *aggregate function* - for the computation of such summary information. Some of them are enumerated below:

Operator	Description
COUNT	counts the number of rows in a table
SUM	sums the numeric values of a certain field
AVG	computes the average of all the numeric values of a certain field
MIN	computes the minimum of all the values of a certain field
MAX	computes the maximum of all the values of a certain field

For instance, the query

"Get the total number of students, the total amount of fees paid and the average of their average marks"

can be implemented as

```
SELECT    COUNT(*) AS Total_no_of_students,
          SUM (Fees_paid) AS Total_fees_paid,
          AVG (Average_mark) AS Average_mark_for_all_students
FROM      Students ;
```

"COUNT (*)" above means "count all the tuples". COUNT can also be used in conjunction with DISCTINCT in order to count only the distinct elements of a column. For instance, the query

"How many types of modules are provided?"

can be implemented as

```
SELECT    COUNT (DISTINCT Type) AS No_of_types
FROM      Modules ;
```

Sub-summaries – grouping results

Sometimes we do not need to perform such operations upon the whole set of tuples. What we need is to be able to combine tuples in groups and apply aggregate functions to each of them. In other words we need to compute sub-summaries, rather than summaries. Such operations are implemented in conjunction with the GROUP BY clause. For instance, the query

"Get, for the students of each department, the minimum, average and maximum average mark"

can be implemented as

```
SELECT    Department,
          MIN(Average_mark) AS Min_AG,
          AVG (Average_mark) AS Avg_AG,
```

```
        MAX (Average_mark) AS Max_AG
FROM    Students
GROUP BY Department ;
```

The "GROUP BY Department" clause partitions the tuples of the Students relation in groups, each group consisting of students who belong to the same department. MIN, AVG and MAX are consequently applied to each of the constituted groups. The result is a table consisting of one tuple per department. Each tuple consists of the minimum and maximum mark of all the students of the corresponding department and of their average mark.

There are a few important restrictions that you have to be aware of:

- each item in the SELECT list has to be *single valued per group*;
- all the column names that appear in the SELECT clause, have to appear in the GROUP BY clause, unless they are parameters of aggregate functions; the converse is not true.

The following statement is *incorrect*

```
SELECT    Department, Level,
          AVG (Average_mark) AS Avg_AG
FROM      Students
GROUP BY  Department ;
```

because Level is not "single valued per group" (i.e. per department), students in a department can be in different levels, nor does it appear in the GROUP BY clause.

A correct version of the above statement could be

```
SELECT    Department, Level,
          AVERAGE (Average_mark) AS Avg_AG
FROM      Students
GROUP BY  Department, Level ;
```

Activity: What does this query do (how does it translate in natural language)?

Filtering groups

Sometimes we need to leave out certain groups (based on a (group) property they have) and provide sub-summary information only on the remaining ones. This can be achieved by means of a HAVING clause. For instance, the query

"Get the maximum of the average mark per department, for the students of each department, but only if it is greater than 70."

can be implemented as

```
SELECT    Department, MAX (Average_mark) AS Maximum_av_mark
FROM      Students
GROUP BY  Department
HAVING    MAX (Average_mark) > 70 ;
```

The filtering condition imposed in the HAVING clause does not have to be reflected in the SELECT clause (as the case with the maximum average mark above is). For example:

```
SELECT    Department, MAX (Average_mark) AS Maximum_av_mark
FROM      Students
GROUP BY  Department
HAVING    AVG (Average_mark) > 50 ;
```

selects the maximum of the average mark of each student per department, but only if the average per department is above 50.

Note that the condition imposed in the HAVING clause must be a group property, i.e. must characterise the group as a whole. For instance, the query

"Get the maximum of the average mark per department, for the students of each department, but only for the ones who paid their fees."

will not have the condition Fees_paid = 5000 in the HAVING clause, because this condition refers to each student in particular, thus it is not a group condition. This condition will have to go in the WHERE clause, as in:

```
SELECT      Department, MAX (Average_mark) AS Maximum_av_mark
FROM        Students
WHERE       Fees_paid = 5000
GROUP BY    Department ;
```

Activity: Describe the difference between

```
SELECT      Department, Level,
            MAX (Average_mark) AS Maximum_av_mark
FROM        Students
WHERE       Level = 2 OR Level = 3
GROUP BY    Department, Level ;
```

and

```
SELECT      Department,
            MAX (Average_mark) AS Maximum_av_mark
FROM        Students
WHERE       Level = 2 OR Level = 3
GROUP BY    Department ;
```

Subqueries

One of the big strengths of the SELECT statement is that it allows other SELECT statements to be embedded in themselves. This allows for intermediate or partial results, of an inner SELECT statement, to be used in an outer SELECT statement. For instance, the query

"Get all the students who have their average mark greater than the overall average mark (i.e. the average of the average mark of each student)."

can be implemented as

```
SELECT      S_id, S_name,
FROM        Students
WHERE       Average_mark > ( SELECT     AVG (Average_mark)
                             FROM       Students ) ;
```

In this case, the result of the inner query is a scalar value and not a table. Therefore SQL violates the relational closure property. Whenever the result of a SELECT statement should be (according to relational algebra) a table consisting of one column and one row, the actual result in SQL is a scalar value.

Subqueries can be used in conjunction with the IN operator. For instance, the query

"Get the student IDs for all the students who are registered for first level modules (i.e. for at least one)"

can be implemented as

```
SELECT      S_id
FROM        Registration
WHERE       M_id IN (
                    SELECT      DISTINCT M_id
                    FROM        Modules
                    WHERE       Level = 1 ) ;
```

It is possible to nest SELECT statements to an arbitrary depth. For instance, the query

"Get the name and the department for all the students who are registered for first level modules (i.e. for at least one)"

can be implemented as

```
SELECT      S_name, Department
FROM        Students
WHERE       S_id IN (
                    SELECT      S_id
                    FROM        Registration
                    WHERE       M_id IN (
                                        SELECT      DISTINCT M_id
                                        FROM        Modules
                                        WHERE       Level = 1 ) ) ;
```

Two other keywords (qualifiers) can be used in association with subqueries, namely SOME (or ANY) and ALL. They can only be used in conjunction with single column tables and, if SC_table is such a table then

- SOME SC_table – refers to at least one value from SC_table, whereas
- ALL SC_table – refers to all the values from SC_table.

For instance, the query

"Get the students whose average mark is greater than the average mark of at least one student from the Maths department."

can be implemented as

```
SELECT    S_id, S_name
FROM      Students
WHERE     Average_mark > SOME (
                              SELECT      Average_mark
                              FROM        Students
                              WHERE       Department = 'Maths' ) ;
```

Note that this query can also be implemented by means of the MIN aggregate function, as:

```
SELECT    S_id, S_name
FROM      Students
WHERE     Average_mark >  ( SELECT      MIN ( Average_mark )
                            FROM        Students
                            WHERE       Department = 'Maths' ) ;
```

If the query were

"Get the students whose average mark is greater than the average mark of all the students in the Maths department"

a possible implementation would have been

```
SELECT    S_id, S_name
FROM      Students
WHERE     Average_mark > ALL (
                              SELECT      Average_mark
                              FROM        Students
                              WHERE       Department = 'Maths' ) ;
```

Activity: How else this query would have been possible to be implemented?

Multi-table queries

The information in a database is spread across different tables. However, many queries (in real life applications) require information from more than one single tables. From relational algebra you know that two relations can be joined together by means of the JOIN operator. In SQL, two tables are joined together by specifying them both in the FROM clause and the join condition (i.e. equality between the corresponding columns) in the WHERE clause. For example, the following query

"Get the students (i.e. their IDs and names) and their results in all the modules they take"

can be implemented as

```
SELECT      St.S_id, S_name, M_id, Result
FROM        Students  St, Registrations  Reg
WHERE       St.S_id = Reg.S_id ;
```

A projection was performed as well, on the specified attributes of the SELECT clause.

Note that the tables can be renamed in the FROM clause, by providing a synonym besides each of their names (this is not compulsory). Also note that when confusions are possible (i.e. an attribute name occurs in at least two of the tables of the FROM clause), each attribute name has to be specified within its scope, by providing the name of the table it belongs to, in front of it, separated by a full stop "." (e.g. S_id could have belonged to either Students or Registrations, therefore it had to be qualified by the name "Students").

It is possible that more than two tables be joined by means of single SELECT statement. For instance, the following query

"Get the student's names, the modules they take (i.e. name), the value of each module and their result for each module."

can be implemented as

```
SELECT    Students.S_id, S_name, M_name, Value, Result
FROM      Students, Registrations, Modules
WHERE     Students.S_id = Registrations.S_id  AND
          Modules.M_id = Registrations.M_id ;
```

The previously described join operations are *inner* joins, in that only rows with matching attribute values are joined. However, SQL provides other kinds of join operations as well, namely *outer* joins. If you are interested refer to (Connolly 1999), pp. 415 – 417.

Checking existence

SQL-92 provides a very important predicate, EXISTS, to check the result returned by executing a query. "EXISTS <SELECT statement>" is *true* if there exists at least one row in the table representing the result of the query, and *false* otherwise. For instance, the query

"Get the student's ID, for all the students who take at least one optional module"

can be implemented as

```
SELECT    S_id
FROM      Registrations Reg
WHERE     EXISTS  (
                    SELECT    *
                    FROM      Modules Mod
                    WHERE     Reg.M_id = Mod.M_id AND
                              Type = 'Optional' ) ;
```

The EXISTS predicate is extensively used in defining integrity constraints (general or database integrity constraints, more precisely)[1].

Combining results

The results of different SELECT statements can be combined by means of set operators, UNION, INTERSECTION and EXCEPT (for difference). Remember, though (reference to relational algebra), that in order to apply the set operators, the tables involved have to be type compatible. For instance, the query

"Get all the students (ID and name) who have either not paid their fees or have failed the year"

can be implemented as

```
( SELECT    S_id, S_name
  FROM      Students
  WHERE     Fees_paid < 500 )
UNION
( SELECT    S_id, S_name
  FROM      Students
  WHERE     Average_mark < 40 )
```

The query

"Get all the students (ID and name) who have not paid their fees and also have failed the year"

can be implemented as

```
( SELECT    S_id, S_name
  FROM      Students
  WHERE     Fees_paid < 500 )
INTERSECTION
( SELECT    S_id, S_name
  FROM      Students
  WHERE     Average_mark < 40 )
```

The query

[1] See the (following) section about integrity constraints.

"Get all the students (ID and name) who have not paid their fees but have passed the year"

can be implemented as

```
( SELECT   S_id, S_name
  FROM     Students
  WHERE    Fees_paid < 500 )
EXCEPT
( SELECT   S_id, S_name
  FROM     Students
  WHERE    Average_mark < 40 )
```

In conclusion, you are strongly advised to:

- implement the above examples in the SQL dialect you use;
- choose other situations and accordingly adapt the examples provided;
- elaborate more complex queries by combining the basic examples provided;
- further explore on your own aspects that have not been presented in this chapter.

Updating operations

The updating component is built around three basic operations:

- insertion;
- deletion;
- modification.

Insertion

An insertion operation has as result the insertion of some tuples in the specified table. It can use a set of user provided values or it can use some values already existing in the database. The syntax, for the first case, is:

```
INSERT INTO      <table name> [ @<column name> ]
VALUES           @<value> ;
```

If the <column name> list is not provided, then the <value> list has to match the columns as they are defined in the <table name>. On the other hand, if the <column name> list is provided, then the <value> list has to match them; in this case there is no restriction upon the order of the column names. For instance, a new row in the Students table can be inserted as either

```
INSERT INTO      Students
VALUES ("CIS0447", "M. Plank", "07/09/1975", 5000, "Computing", 1, 68) ;
```

or

```
INSERT INTO
  Students (S_id, S_name, DOB, Level, Average_mark, Department, Fees_paid)
VALUES ("CIS0447", "M. Plank", "07/09/1975", 1, 68, "Computing", 5000) ;
```

If null values are allowed by the definition of the column, then they can be used in an INSERT statement. For instance, considering that Average_mark allows null values (i.e. before the end of the exam period this value is unknown), then the following statement can be used to specify that M. Plank has not yet completed all his exams:

```
INSERT INTO      Students
VALUES ("CIS0447", "M. Plank", "07/09/1975", 5000, "Computing", 1, NULL) ;
```

If default values are specified for a certain attribute (either by defining it on a domain that provides a default value, or by specifying a default value in the column's definition), then this can be used in an INSERT statement by not specifying the names of these columns in the <column name> list. For instance, if the default values for Fees_paid is 0 (zero), for Department is "not yet chosen" and for Level is 1, then the following statement

```
INSERT INTO Students (S_id, S_name, DOB, Average_mark)
VALUES ("CIS0447", "M. Plank", "07/09/1975", NULL) ;
```

is equivalent to

```
INSERT INTO      Students
VALUES ("CIS0447", "M. Plank", "07/09/1975", 0, "not yet chosen", 1, NULL) ;
```

The syntax for using INSERT based on the values that already exist in the database, is

```
INSERT INTO      <table name>
<compatible select statement>
```

where the result of the <compatible select statement> is a table compatible with either the <table name>, in case the <column name> list is not specified, or with the columns listed in the <column name> list, if this is specified. For instance, in order to automatically register all level 1 students of a certain department for all level 1 compulsory modules of the respective department, the following statement can be issued:

```
INSERT INTO      Registrations
( SELECT   S_id, M_id
  FROM     Students St, Modules Mod
  WHERE    Mod.Level = 1                    AND
           Mod.Type = 'Compulsory'          AND
           St.Level = 1                     AND
           Mod.Department = St.Department       ) ;
```

Deletion

A delete operation deletes certain tuples from a table, according to a certain condition. Its general syntax is

```
DELETE FROM      <table name>
[ WHERE          <condition> ] ;
```

If no WHERE clause is specified then the statement deletes all the tuples in the specified relation. Otherwise, it only deletes the tuples that satisfy the specified condition. For instance, the request

"All the students who have not paid any fees at all have to be deregistered from the university (i.e. deleted from the Students table)"

can be implemented as

```
DELETE FROM      Students
WHERE            Fees_paid = 0 ;
```

Modification

Existing values in the database are modified by means of an UPDATE statement. Its general syntax is

```
UPDATE     <table name>
SET        @{<column name> = <value>}
[WHERE     <condition>] ;
```

The WHERE clause can specify which rows are going to be updated – i.e. those that satisfy the specified <condition>. Within the SET clause more than one field can be updated at a time. For instance, the request

"Promote all level 1 and level 2 students who have passed (i.e. average mark > 40) to the next year and set their paid fees to zero."

can be implemented as

```
UPDATE     Students
SET        Level = Level + 1, Fees_paid = 0
WHERE      (Level = 1 OR Level = 2) AND Average_mark > 40 ;
```

Activity: Reconsider the application you have previously chosen (reference to the application at the end of the previous section, p.9). Identify a set of NL queries and express them in SQL. If you attempted some complex queries, it is quite probable that some of them cannot be expressed in SQL? Do you know why? Can you modify the definition of the database (data objects) so that these queries would become expressible? Implement them in the dialect you use. Before this, you will have to populate (to provide an extension of) your database. For this, attempt first the activity referring to updating operations (next section).

Integrity constraints

The following types of integrity constraints were introduced together with the statements for data definition (CREATE DOMAIN and CREATE TABLE):

- *domain integrity constraints* – they are constraints that are applied to all values of any attribute / column defined on the respective domain; e.g.:

```
CREATE DOMAIN      Hours_Per_Week SMALLINT
                   DEFAULT        40
                   CHECK          Values > 10 AND Value < 50 ;
CREATE TABLE Employees (
        --- columns definition
        Working_hours          Hours_Per_Week;
        --- other definitions
) ;
```

  the value of Working_hours in each tuple is has to be greater than 10 and smaller than 50;

- *attribute / column integrity constraints* – somehow similar to the domain constraints, but specified in the definition of the table; e.g. the above constraint can be specified as a column constraint:

```
CREATE TABLE Employees (
        --- columns definition
        Working_hours          SMALLINT  DEFAULT 40
                               CHECK Working_hours > 10 AND
                                     Working_hours < 50,
        --- other definitions
) ;
```

- *base table integrity constraints* – specify constraints on the attributes of a table; e.g., if the maximum weekly salary for an employee who is paid by the hour is £800

```
CREATE TABLE Employees (
        --- columns definition
        Working_hours   Hours_Per_Week,
        Hourly_salary   DECIMAL(3,1) CHECK Hourly_salary > 3.5
        --- other definitions
        CONSTRAINT maximum_salary
                CHECK Hourly_salary * Working_hours < 800
) ;
```

Whenever an update operation (UPDATE or INSERT) is attempted, the DBMS checks the relevant integrity constraints. If at least one of them is violated, then the update operation is rejected. If all the constraints are satisfied then the update is performed.

A *database (or general) integrity constraint* is an integrity constraint imposed on the tables of the database. The SQL syntax for creating and removing such an integrity constraint is:

```
CREATE ASSERTION <constraint name> ( <truth valued expression> ) ;
DROP ASSERTION <constraint name> ;
```

where <truth valued expression> is an arbitrary boolean expression, i.e. involving as many tables as needed. The expression usually involves aggregate functions and / or negated existentially-quantified relational algebra expressions (in the form of "*there exist no* tuples in relation R such that <condition>". To clarify, consider the following examples:

- the maximum weekly wage, for any employee, should be £800, for the following database:

Employees

E_ID	Dept_ID	Hours_per_week	

Departments

Dept_ID	Wage_per_hour	

this constraint can be implemented by means of an aggregate function:

```
CREATE ASSERTION Maximum_salary_Hourly_paid_work CHECK (
        ( SELECT MAX (Hours_per_week * Wage_per_hour)
          FROM Employees, Departments
          WHERE Employees.Dept_ID = Departments.Dept_ID ) < 800
) ;
```

or, the same constraint can be implemented by means of an existentially quantified relational algebra expression; for this case, it has to be rephrased as "there exists no employee that earns more than £800 by doing work by the hour";

```
CREATE ASSERTION Maximum_salary_Hourly_paid_work CHECK (
        NOT EXISTS
          (SELECT *
           FROM Employees, Departments
           WHERE Employees.Dept_ID = Departments.Dept_ID AND
                 Hours_per_week * Wage_per_hour > 800 )
) ;
```

- every student must be registered for at least one optional module, for the following database:

Students

S_ID	

Registrations

S_ID	M_ID	

Modules

M_ID	Type	

this constraint can be implemented by means of the set membership operator, "IN":

```
CREATE ASSERTION Optional_modules (
        NOT EXISTS
          ( SELECT *
            FROM Students
            WHERE S_ID NOT IN
                (SELECT S_ID
                 FROM Registrations, Modules
                 WHERE Registrations.M_ID = Modules.M_ID AND
                       Type = 'Optional' ) )
) ;
```

or, it can be implemented by means of the set difference operator, EXCEPT:

```
CREATE ASSERTION Optional_modules (
        NOT EXISTS (
          ( SELECT S_ID
            FROM Students )
                EXCEPT
          ( SELECT S_ID
            FROM Registrations, Modules
            WHERE Registratuions.M_ID = Modules.M_ID AND
                  Type = 'Optional' )
        )
) ;
```

Typically, in NL, the integrity constraints are expressed by means of the universal quantifier, as "all X must satisfy condition". Such statements can be restated preserving the meaning, by means of the existential quantifier and of negation, as "there exists no X such that X does not satisfy condition". The latter form is readily implementable as illustrated above.

Now we can make the clarification required about domain, column and table constraints. SQL allows them to contain arbitrary truth valued expressions. That means that they admit the same definitions as the ones used in database (general) constraints.

Activity: Identify a set of integrity constraints for the application you have chosen (p. 9 and p. 18), express them in SQL92 and implement them in the dialect you use.

Views

Most of the topics discussed in this section are applicable to the relational model in general. We are going to use phrases like "in SQL ..." where they are SQL specific.

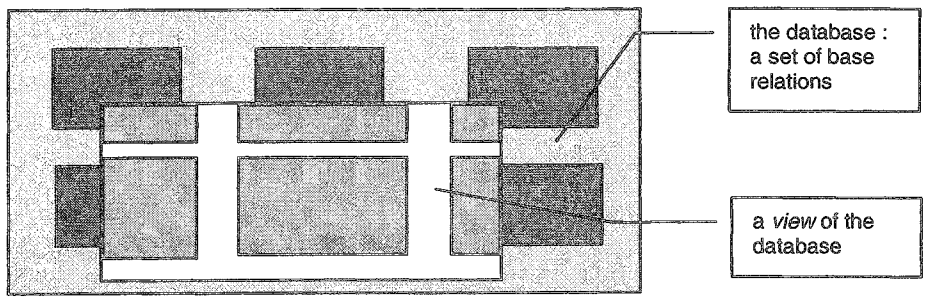

Figure 4: A view of a database

We have seen, thus far, how the relations making up a database can be defined by means of a DDL (more specifically, by means of SQL's DDL component). These relations are physically stored in the database. Together they represent the conceptual model – they define and contain *all* data that is stored in the database. However, as we previously mentioned (the ANSI/SPARC architecture of a database system), different users require different views of the database (at the external level). That is, they only need to see parts of (not necessarily disjoint) and not the entire database. Also, the data corresponding to these parts might need to be presented in different formats (Figure 1). So, data is only stored once in the database, and the views represent the way the users *perceive* the stored data.

Definition: A view is a named relational expression.

Since we have concentrated only on relational algebra, we shall consider a view as being a named expression of relational algebra. This will not reduce the generality of the statements we are going to make about views.

A view can be regarded as a *virtual relation*, because

- it defines a relation – the result of a relational algebra expression is a relation;
- it has no physical existence, therefore it is virtual.

The view mechanism is based on a substitution procedure. When a statement comprising a name of a view is executed, the name of the view is substituted with the view's definition. The relational closure property allows for this substitution procedure to be employed.

This substitution procedure can be applied recursively. therefore views can be defined on other views.

SQL provides support for views. The general syntax for creating a view is

```
CREATE VIEW     <name>  AS  <expression>
                [WITH CHECK OPTION] ;
```

and for destroying a view is:

```
DROP VIEW       <name> <option> ;
<option> ::=    RESTRICT | CASCADE ;
```

<name> represents the name of the view and <expression> is an SQL statement (SELECT) representing a relational algebra expression. WITH CHECK OPTION specifies the fact that the view cannot be updated if the UPDATE or INSERT statement violates the view's definition. We shall return on this issue later in the section.

Consider the tables Employees and Departments illustrated in Figure 5.

Employees

E_ID	Dept_ID	Hours_per_week	

Departments

Dept_ID	Wage_per_hour	

Figure 5: Two example tables

Suppose we need a view of the database that provides information about all the employees of the company working by the hour and their weekly wages. Its definition might be:

```
CREATE VIEW     Salaries_of_employees_by_the_hour
AS              SELECT  E_id, Dept_id,
                        Hours_per_week * Wage_per_hour AS Weekly_salary
                FROM    Employees Emp, Departments Dept
                WHERE   Emp.Dept_id = Dept.Dept_id ;
```

Now, Salaries_of_employees_by_the_hour can be used as if it were a base relation.

Advantages

Logical data independence

Views provide logical data independence, up to a certain degree. Logical data independence means that application programs are immune to changes in the logical structure of the database. Theoretically, this cannot be entirely achieved. For instance, if certain data is removed from the database then this data is simply not available anymore. The view mechanism provides logical data independence as long as the change in the structure of the database does not lead to loss of information.

Suppose a database DB. Suppose that DB' is a the new DB after a change in structure. If all the information that can be deduced from DB can also be deduced from DB' (the converse condition is not required), then it is possible to define a view of DB', say ViewDB', that will look identical with DB. Therefore, all the statements that worked on DB will also work on DB', through ViewDB'.

Consider a simple example. Suppose a database contains the relation Employees, as in (Figure 6 - a), which is then changed into NewEmployees as in (Figure 6 - b):

Employees

ID	Name	YearOfBirth	Dept	Salary

(a)

NewEmployees

ID	FirstName	Lastname	DOB	Dept	Salary	Position

(b)

Figure 6: Old (a) and new (b) table

A view can be defined on NewEmployees so that it will be equivalent to Employees:

```
CREATE VIEW     Employees
AS      SELECT  ID
                FirstName || LastName AS Name,
                EXTRACT(YEAR FROM DOB)  AS YearOfBirth
                Dept,
                Salary
        FROM    NewEmployees ;
```

The view Employees is equivalent to the original table Employees. Therefore, the application programs that used the original table Employees do not require any change in order to work with NewEmployees. All that is needed is the above view definition.

Different formats of data

Views represent a way of customising a database, because they allow users to perceive data according to their own requirements. E.g. some users might want to have access to the date of birth whereas other users might need to see the age of a person; some users might need to see the first and last name separately, whereas other users want to see it as a single attribute.

Security

The view mechanism provides an implicit security feature, in that users see only the data that is relevant to them. As far as they are concerned, the other data in the database might as well be non-existent. This issue is discussed in greater detail Chapter 2 of Volume 2.

Macro-definitions

A view can also be seen as a way of defining a shorthand for a rather complex expression, therefore as a macro (programming) facility. Thus, the database programming activity can become more efficient.

Note again that views (in SQL) are virtual relations, in that only their definition is stored in the database. SQL also provides the possibility of defining relations (based on the existing ones) that are *actually* stored in the database. They are called *snapshots*.

Activity: Investigate on your own the issue of snapshots. Discuss the main differences between snapshots and views. For instance, suppose a base relation was updated. If a view, defined on it, is used, it would automatically reflect the change. However, in order to reflect the change, an equivalent (having the same definition) snapshot would have to be first recomputed. Elaborate. Identify and discuss other differences.

So far, we have seen how views can be defined (created). The next issue we look at is the way views are used. In particular, we look at:

- retrieving data by means of views;
- updating data by means of views.

Retrieving data

From the point of view of the user, views can be used exactly as if they were base relations (tables). With respect to SQL, whenever a SELECT statement referencing a view is to be executed, the DBMS substitutes the view's definition in the statement and then executes the resulting (equivalent) query. For instance, the query (see a previous example):

"Get all the employees (their ID and weekly salary) who earn less than £100 a week"

expressed as

```
SELECT   E_id, Weekly_salary
FROM     Salaries_of_employees_by_the_hour
WHERE    Weekly_salary < 100 ;
```

is transformed by the DBMS, by *substituting* the view's definition, into

```
SELECT   E_id, Hours_per_week * Wage_per_hour AS Weekly_salary
FROM     Employees Emp, Departments Dept
WHERE    Emp.Dept_id = Dept.Dept_id  AND
         Hours_per_week * Wage_per_hour < 100 ;
```

If a view is defined on another view then after its name was substituted with its definition the resulting query still contains a reference to a view. The solution is that the process of substitution continues recursively until the query is totally expressed on base relations (tables); i.e. each reference to a view (via its name) is substituted with the view's definition until all the referenced relations are base relations. Consider, for instance, the database in Figure 7.

Students

SID	Name	Course	Department	Address

Modules

MID	Name	Department	Type

Results

SID	MID	Mark

Figure 7: A part of the database of a university

Consider the following view definitions

```
CREATE VIEW   Detailed_results
AS            SELECT    Students.SID, Students.Name AS SName,
                        Students.Department,
                        Modules.Name AS MName, Mark
              FROM      Students, Modules, Results
              WHERE     Students.SID = Results.SID      AND
                        Results.MID = Modules.MID ;
CREATE VIEW   Overall_results
AS            SELECT    SID, SName, Department,
                        AVG (Mark) AS Avg
              FROM      Detailed_results
              GROUP BY  SID ;
CREATE VIEW   Good_students
AS            SELECT    SID, SName, Department, Avg
              FROM      Overall_results
              WHERE     Avg > 60 ;
```

The query

"Get all the good students from the CIS department"

can be implemented, using the Good_students view, as

```
SELECT    SID, SName
FROM      Good_students
WHERE     Department = 'CIS' ;
```

The query is firstly transformed (by the DBMS) into:

```
SELECT    SID, SName
FROM      Overall_results
WHERE     Avg > 60        AND
          Department = 'CIS' ;
```

Then Overall_results has to be substituted with its definition:

```
SELECT    SID, SName
FROM      Detailed_results
GROUP BY  SID
HAVING    AVG (Mark) > 60 AND
          Department = 'CIS' ;
```

Note that because both conditions Avg > 60 and Department = 'CIS' were imposed on a "subtotal" tuple (i.e. a tuple in the view Overall_results corresponds to the result of a group operation on the tuples of Detailed_results) they had to be expressed by means of a HAVING clause; they represent characteristics of groups of tuples (grouped by SID) in Detailed_results and not of individual tuples.

Eventually, Detailed_results has to be substituted with its definition. Therefore, the statement that the DBMS will eventually execute, i.e. the query after the process of substitution was completed is:

```
SELECT    Students.SID, Students.Name AS SName
FROM      Students, Modules, Results
WHERE     Students.SID = Results.SID      AND
          Results.MID = Modules.MID
GROUP BY  Students.SID
HAVING    AVG (Mark) > 60 AND
          Students.Department = 'CIS' ;
```

Updating data

Whereas the retrieving operations were relatively easy to understand, the updating operations do not work so straightforwardly. The main reason is that a view has no physical existence; a view is nothing more than mere definition. Therefore, the objects on which a view is defined (i.e. base relations) have to be updated in such a way that the resulting effect (of these updates), seen through the view, reflect perfectly (are equivalent to) the initial attempted update (on the view).

If we adopt a functional notation then this idea can be expressed a bit more formally. Suppose a view defined as

View = X (DB)

then, an update of the view is

Update (View) = Update (X (DB))

X(DB) does not physically exist, therefore Update(X(DB)) cannot be performed. However, we can try to find an Update' (that will be applied to the database) such that

X (Update' (DB)) = Update (X (DB))

If such an update can be found then it can be performed (for it is expressed on the database) and we have the equivalence

Update (View) = X (Update' (DB))

Every language (DBMS) implements a set of rules by means of which, given an update Update and a view definition View = X(DB), a corresponding set of update statements Update' can be worked out, such that

Update(X(DB)) = X(Update-dash(DB))

It is important that you know these rules, because they describe the behaviour of the system (DBMS). In other words, if you do not know them you might get unexpected behaviours.

The *general* rules (valid for any relational model) are quite complex. They are not presented here (therefore they are not going to be examined), but if you want to find out more about them refer to (Date 1995), Chapter 17, pp. 472-492.

However, in SQL-92 they have been drastically simplified due to a series of restrictions imposed upon the possibility of updating a view. Essentially, the only views that are updateable in SQL are those defined on a single base relation. More precisely, an SQL view is updateable, if and only if all of the following 8 conditions are satisfied (Date 1995, p. 491):

- the table expression is a single SELECT statement, i.e. it does not contain any of the following keywords: JOIN, UNION, INTERSECT, EXCEPT;
- the SELECT statement does not contain the word DISTINCT;
- every column specification in the SELECT clause consists of a simple reference to a column of the underlying table;
- the FROM clause contains only one table name;
- the table name in the FROM clause specifies either a base relation or an updatable view;
- the WHERE clause cannot contain a nested SELECT statement that contains a FROM clause referring to the same table as in the "main" (see above) FROM clause;
- no GROUP BY clause is allowed;
- no HAVING clause is allowed.

Conclusions

There is much more to SQL-92 than what it was presented in this chapter. As a matter of fact we only managed to touch on the basic aspects of the language. However, the overall view of the language that you have got by now will allow you to develop databases of a quite substantial level of complexity. Moreover, should you need to use other features offered by SQL-92, you will be able to find your way through in the reference book(s) of the SQL dialect you are using. In other words, the basis that we built in this chapter is sufficient for you to individually tackle a great deal of advanced topics.

If you do not feel confident with the topics you studied, return to them and exercise. Exercise, this is the best method of learning the topics of this chapter. It is crucial that

you used, in parallel, a DBMS that supports a dialect of SQL-92, and implemented all the examples you came across.

Hopefully, this chapter generated the desire for further exploration SQL-92.

Learning outcomes

On completion of this chapter you should be able to express in SQL-92 and implement in the SQL dialect you were using a complex database system. This means that you should be able to:

- define the data objects at the conceptual level;
- define integrity constraints;
- define views at the external level;
- implement natural language queries.

References

ISO, 1992. Database Language SQL (ISO 9075:1992(E)). International Organisation for Standardization.