1 SQL (*Structured Query Language*) ; an extensive introduction

Content of this SQL-reader:

1.1 Introduction to this SQL-*reader*

This *reader* is mainly based on an English '*manual'* from a rather obsolete SQL-database-system. A number of specific aspects of this RDBMS have been omitted. We also added a number of additions and observations.
The original *manual-RDBMS* was based on the SQL-*1*-version, which is mainly dedicated to the manipulating of data in a relational database; the *requesting of* data (by way of the SELECT …-command) was a very important aspect. In later versions of SQL, watching the mutual relationships between data in the database has been more elaborated.
Because in this reader we especially focus on the *requesting* possibilities of SQL, we can base ourselves unconcerned on SQL-*1*.
Some aspects of SQL-*2* are discussed in the last chapter. Further on in this course we'll also discuss SQL-*2* additional aspects about the enforcement of data constraints.

We give the following outline for a systematic approach to formulate an appropriate SQL-query:

➔ '*Top down'*- approach for drawing up SQL-queries:

For a systematic approach to formulate an appropriate SQL-query you always must start the analysis of an information demand with considering the following aspects (in the given sequence):

1. From which *table*(s) data have to be consulted/extracted?

2. In which *columns* do we find the data we need (in general: the data that have to be shown)?

3. What are the *criteria* to select the table-rules that have to be taken into account (eventually: join- or subquery-criteria)?

Later we will add some more refined questions like:
- Do we have to apply grouping of data (if yes: does there exist a grouping criterion)?
- Do we have to determine extreme values (minimum, maximum)?
- Is a '*correlated subquery'* necessary?
- And eventually a few additional aspects.

Not part of the *top down*-part of this approach, but very important is always the following aspect:

4. If you test your query, does it give an acceptable answer to the information demand?
   Do the testing process first by '*desk top testing'*, followed by testing with a real DBMS.

Always keep in mind this approach if you have to formulate an SQL-query.

Nijmegen, the Netherlands, October 2007,

Ger Paulussen

2 Basic concepts of SQL *(table, column, row, field, value)*

All information in the Relational Data Model and in SQL is in the form of tables. A table is usually called a relation in the Relational Data Model. In SQL, the preferred term is table.
All tables in SQL are two-dimensional, having a specific number of columns and a variable, arbitrary number of unordered rows.
In the Relational Data Model literature, columns are often called attributes, and rows are called tuples.

The next figure shows a table, called RECENT_PRESIDENTS. This table has six columns (named PRES_NAME, BIRTH_YR, YRS_SERV, DEATH_AGE, PARTY and STATE_BORN) and nine rows.

RECENT_PRESIDENTS:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri
Eisenhower D D	1890	8	79	Republican	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois

Names for tables and columns in SQL must begin with a letter, $, # or @ , and may contain up to 18 characters. Permitted characters are upper and lower case letters, numbers, $, #, @ and underscores. In general, names should not to be too short, (like A or BB) nor too long (like TERM_SERVED_IN_OFFICE).
Throughout this publication the underscore '_' will be used in table and column names to delimit words instead of blanks, which are illegal inside a name. This is done to improve readability, for example, YRS_SERV is clearer than YRSSERV.

This RECENT_PRESIDENTS table is used in the following text and examples.
The table contains five kinds of facts about presidents who were born after 1880:

- the president's birth year
- number of years in office
- his/her age at death if applicable [1]
- the party of the president at the time of first inauguration
- the state where the president was born

Before formulating a query on a database we need to know the existing table names and column names and their data types. It is useful to have a table template or relational schema diagram containing all the table names and column names.

For our example of RECENT_PRESIDENTS this template is given in the next figure.

TEMPLATE RECENT_PRESIDENTS (<= table name)

column name =>	PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
data type =>	CHAR (15)	SMALLINT	SMALLINT	SMALLINT	CHAR (12)	VARCHAR(14)

[1] Addition: this value cannot be 'mandatory' as of course for presidents that are still alive, their death-age cannot be known. In the case of *not recorded* values, we speak about '*NULL*'-values. Later on we will see, that such *NULL*-values are only possible if this is admitted by the table-definition. In generated output such NULL-values often are shown through a question mark ('?') or a <null> or something like that.

The first column, PRES_NAME, *identifies* the president. It contains the president's last name, separated by blanks.
We assume preliminarily that the number of columns in the table is fixed, and that the number of rows is variabIe, and grows every time a new president is electéd. However, a unique feature of SQL is the fact that the number or columns can be dynamically extended, as explained later.
The tabular data model of SQL is simple to understand. The elements of a table are: TABLE NAME, COLUMN NAME, COLUMN, ROW and FIELD.

3 The basic form of the retrieval command *(Select ...)*

The basic form of the retrieval command in SQL is:

```
SELECT      column-name , or list-of-column-names , or  *
FROM        table-name , or list-of-table-names
[ WHERE     search-condition ]
```

When setting up a retrieval command, we must first consider in which table or tables the information desired is contained. The names of these tables are listed in the FROM clause of the retrieval command.
If we want to retrieve all of the information contained in a table, we list the names of the columns of this table following the SELECT keyword, and omit the WHERE clause.
Alternatively, and more conveniently, we can enter:

```
SELECT      *
FROM        table-name
```

The symbol * means that all columns of the table named in the FROM clause will be retrieved.
If we do not want to retrieve all the information from a table, but rather a specific subset of a table or of several tables, we must specify this subset in the SELECT clause or in the WHERE clause. There are two kinds of subsetting.

- Vertical subsetting:
  If we are only interested in retrieving a subset of the available columns, then this subset is specified by listing the names of the columns we want to select in the SELECT clause.
- Horizontal subsetting:
  If we are only interested in retrieving a specific subset of the table rows, we specify this in the WHERE clause by using a search condition. Only those rows, which fulfill the search condition, will appear in the result.

Any combination of vertical and horizontal subsetting is possible.

A search condition is basically a comparison of fields, constants, and expressions. The following comparison operators may be used:

```
=       equal to
<>      not equal to              (or 'NOT  … = …'  or in some systems:  ^= )
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to
```

Addition: by way of a special test we also can check of from a table row a certain [field] value yes or not is filled in (so: yes or not 'NULL'). That test can be done via: '*IS [NOT] NULL'* .

Some examples of search conditions are:

```
YRS_SERV  >= 7   AND  PARTY = 'Democratic'
YRS_SERV  = 12   AND  DEATH_AGE  IS NOT NULL
```

A search condition may be any combination of (sub) search conditions linked by logical operators (*AND, OR , NOT*).

The arithmetic operators obey the usual algebraic precedence rules. Arithmetic operations are evaluated before Boolean operations. Operator precedence can be over-ruled by using parentheses.

The following expression can have several different meanings depending on the position of parentheses:

```
NOT YRS_ SERV * 10 > DEATH_AGE  AND  DEATH_AGE > 60 + BIRTH_YR / 100
```

One can introduce parentheses to this expression without affecting its meaning:

```
( NOT (YRS_SERV * 10 > DEATH_AGE ) ) AND ( DEATH_AGE > (60 + BIRTH_YR / 100 ) )
```

However, the meaning of the expression is changed by repositioning the parentheses:

```
NOT ( ( YRS_SERV * 10 > DEATH_ AGE ) AND  ( DEATH_AGE > 60 + BIRTH_YR /100 ) )
```

Quite often, the columns to be compared within a search condition are columns of the table(s) named in the FROM clause. This is the case of a simple search condition. There may also be the case where the comparison requires columns of a table or tables not named in the FROM clause. In this case we must specify a so-called subquery within the condition.
The subquery has precisely the same basic form as any other SELECT command (SELECT ... FROM ... [WHERE ...]). The square brackets around the keyword WHERE mean that this element is optional. It may, but need not, be used in the command.

For convenience of reading the result of a query, we may wish to have the columns and rows ordered in some specific way. This can be achieved quite easily. The order of the columns of the result is specified by the order of the column names in the SELECT clause.

This order need not conform with the order of the columns in the tables of the database. The order of the rows of the result can be specified by appending an ORDER BY clause to the retrieval command. There are many cases where we do not want to simply retrieve information from the database, but rather to derive information through calculations. For that purpose, we have the facility to apply specific arithmetic operators and built-in functions within a retrieval command.

The above mentioned concepts and features are already sufficient to formulate the majority of queries in an average database application. In the following sections we will treat these concepts and features in more detail, starting with the simplest, and proceeding to the more powerful possibilities.

4 Selecting All or Particular Columns from One Table *(Select ...From ...)*

The simplest case is to list a whole table. The format of this retrieval command is as follows:

```
SELECT      *
FROM        table-name
```

Many of the queries that follow will operate on the table of the RECENT_PRESIDENTS.

An example of a query which lists an entire table follows:

4.1.1 Question

List the names, birth years, years served, ages at death (if any), parties, and birth states of all recent presidents.

By investigating the database schema or table template, we find that we have precisely the same column ordering as described in the table format defined in our table template. Thus, the query can be formulated by using the format as given in the beginning of this chapter:

```
SELECT      *
FROM        RECENT_PRESIDENTS
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri
Eisenhower D D	1890	8	79	Republican	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois

(So exactly the same figure as the one given for the whole table RECENT_PRESIDENTS.)

4.2 ORDERING COLUMNS

We might wish to have the columns in a different order than provided in the table template in the database. We achieve this by specifying in the SELECT clause the column names in the order we require.

4.2.1 Question

List the names, birth years, ages at death (if any), years served, birth states and parties of all recent presidents.

```
SELECT      PRES_NAME, BIRTH_YR, DEATH_AGE, STATE_BORN, PARTY
FROM        RECENT_PRESIDENTS
```

Result:

PRES_NAME	BIRTH_YR	DEATH_AGE	STATE_BORN	PARTY
Roosevelt F D	1882	63	New York	Democratic
Truman H S	1884	88	Missouri	Democratic
Eisenhower D D	1890	79	Texas	Republican
Kennedy J F	1917	46	Massachusetts	Democratic
Johnson L B	1908	65	Texas	Democratic

Nixon R M	1913	?	California	Republican
Ford G R	1913	?	Nebraska	Republican
Carter J E	1924	?	Georgia	Democratic
Reagan R	1911	?	Illinois	Republican

4.3 ORDERING ROWS *(Order By …)*

We might also wish to have the rows in a specific order that is different from the order the rows were entered into the tables. To achieve this, we have to append an ORDER BY clause to the query. There can be one or several ordering criteria in an ORDER BY clause.
An ordering criterion is an item from the SELECT list with an indicator whether the ordering is done in an ascending or descending order of that item. The format of an ORDER BY clause is:

```
ORDER BY   column-specification   [ ASC | DESC ]
           [,column-specification [ ASC | DESC ] … ]
```

If DESC is present, the order is defined as descending, otherwise ascending.

4.3.1 Question

List all rows in the table named RECENT_PRESIDENTS, ordered by president names in ascending order.

```
SELECT      *
FROM        RECENT_PRESIDENTS
ORDER BY    PRES_NAME
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia
Eisenhower D D	1890	8	79	Republican	Texas
Ford G R	1913	2	?	Republican	Nebraska
Johnson L B	1908	5	65	Democratic	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Nixon R M	1913	5	?	Republican	California
Reagan R	1911	3	?	Republican	Illinois
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri

↑
ordering criterion

It might be noted that PRES_NAME is displayed first because it is the first column in the table, and not because of its appearance in the ORDER BY clause.
In case we ordered the rows of the table according to a column where the fields are not unique (that is, which is not a key), it might be useful to apply a second, third, etc., ordering criterion. While the first ordering criterion is applied to the whole table, the second ordering criterion is applied to each horizontal subset of the table where the fields specified by the first ordering criterion are equal. The third ordering criterion further refines the ordering of the second etc.

4.3.2 Question

List the table named RECENT_PRESIDENTS, ordered by years served in descending order, and within the same years served, ordered by birth state in ascending order.

```
SELECT      *
FROM        RECENT_PRESIDENTS
ORDER BY    YRS_SERV  DESC , STATE_BORN
```

The first ordering criterion is YRS_SERV. However, since there are several presidents with the same years-served figure, we decided to have a second ordering criterion, birth state.

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Eisenhower D D	1890	8	79	Republican	Texas
Truman H S	1884	7	88	Democratic	Missouri
Nixon R M	1913	5	?	Republican	California
Johnson L B	1908	5	65	Democratic	Texas
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois
Kennedy J F	1917	2	46	Democratic	Massachusetts
Ford G R	1913	2	?	Republican	Nebraska

ordering criterion *first:* ↑ *second:* ↑

The combination of years served and state born is unique in this case, but only by chance.
If they were not unique, the order of rows for a particular combination of YRS_SERV and STATE_BORN values would be undefined, since we did not specify a third ordering criterion.

Note: In the ORDER BY clause one can use either the column name or the position of the item in the SELECT list counting from left to right. This is most useful when the SELECT item is an expression rather than a column.

Thus, question 5.04 could also be formulated as follows:

```
SELECT      *
FROM        RECENT_PRESIDENTS
ORDER BY    3  DESC , 6
```

4.4 *VERTICAL* SUBSETTING *(by a list-of-column-names)*

If we wish only a vertical subset of a table, that is, only a subset of the available columns, we specify the columns we wish in the SELECT clause.

4.4.1 Question

List the names, birth years, and parties of all recent presidents. Order by parties and presidents' names, both in ascending alphabetical order.

```
SELECT      PRES_NAME, BIRTH_YR, PARTY
FROM        RECENT_PRESIDENTS
ORDER BY    PARTY, PRES_NAME
```

Result:

PRES_NAME	BIRTH_YR	PARTY
Carter J E	1924	Democratic
Johnson L B	1908	Democratic
Kennedy J F	1917	Democratic
Roosevelt F D	1882	Democratic
Truman H S	1884	Democratic
Eisenhower D D	1890	Republican
Ford G R	1913	Republican
Nixon R M	1913	Republican
Reagan R	1911	Republican

↑ *second* *first* ↑ *ordering criterion*

If we select columns whose combination is not unique within the table in the database, we find that duplicate rows appear in the display. If we want to avoid these duplicates, we use the DISTINCT operator immediately after the SELECT operator:

```
SELECT    DISTINCT column-name, or list-of-column-names
FROM      …
```

If we deliberately wish to have these duplicate rows in the resulting table, we can either omit the DISTINCT operator or we can replace it by the ALL operator:

```
SELECT    ALL  column-name, or list-of-column-names
FROM      …
```

4.4.2 Question

List all states where a recent president was born. Order by state (ascending order).

```
SELECT      STATE_BORN
FROM        RECENT_PRESIDENTS
ORDER BY    STATE_BORN
```

Result:

STATE_BORN
California
Georgia
Illinois
Massachusetts
Missouri
Nebraska
New York
Texas
Texas

We find in the result that duplicate rows are displayed, namely, the last two rows are both Texas. The same result could also have been achieved by using:

```
SELECT      ALL  STATE_BORN
FROM        RECENT_PRESIDENTS
ORDER BY    STATE_BORN
```

If we wish, however, *to avoid duplicates*, we have to use the DISTINCT operator, as in the following example.

4.4.3 Question

List all states where a recent president was born, and eliminate duplicates. Order by state.

```
SELECT      DISTINCT  STATE_BORN
FROM        RECENT_PRESIDENTS
ORDER BY    STATE_BORN
```

Result:

STATE_BORN
California
Georgia
Illinois
Massachusetts
Missouri
Nebraska
New York
Texas

5 Selecting Specified Rows of One Table *(Select ... From ...**Where** ...)*

For selecting specific rows of a table, that is for horizontal subsetting, we specify a WHERE clause after the table name. The condition in the WHERE clause is a comparison of fields, constants and expressions. To start, with, we will now consider only the simple case of using columns that are in one table.

The comparison operators are as follows:

=	equal to	
<>	not equal to	(or 'NOT … = …' or in some systems: ^=)
>	greater than	
>=	greater than or equal to	
<	less than	
<=	less than or equal to	

5.1.1 Question

List all facts available in the table named RECENT_PRESIDENTS about the president named Carter J E.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       PRES_NAME = 'Carter J E'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia

Note:

A constant containing characters (e.g. 'Carter J E') must be enclosed in single quotes.
Numeric constants are never enclosed in quotes.

In the above example, we used the PRES-NAME column for comparison using the operator =. This column is the key to the table, so the result can only be one row or empty. If we use a non-key column for comparison with =, the situation is different. The result could be empty, one or several rows.

In the cases of the other kinds of comparisons (<> , > , >= , < , <=), the result could be empty, one or several rows, independent of whether we use a key column or a non-key column for comparison.

5.1.2 Question

List all facts available in the table named RECENT_PRESIDENTS about all presidents born in Texas.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       STATE_BORN = 'Texas'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas

Remark:

Unlike conventional programming languages such as BASIC, C, COBOL, FORTRAN, PASCAL, PL/l etc. we have just seen that SQL is totally associative. This is a major advance from third generation techniques. In SQL there is only *one* data structure, namely the *table*. Concepts like pointer, array or

repeating group as in PASCAL, FORTRAN and COBOL have been excluded from SQL without any loss of functionality.
The result of these two basic differences is a drastic simplification in programming.

5.1.3 Question

List all facts available in the table named RECENT_PRESIDENTS about all presidents not born in Texas.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       STATE_BORN  <> 'Texas'          (or:  NOT  STATE_BORN = 'Texas')
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri
Kennedy J F	1917	2	46	Democratic	Massachusetts
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois

5.1.4 Question

List all facts available in the table named RECENT_PRESIDENTS about all presidents who served more than 4 years.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       YRS_SERV > 4
```

Note: *Numeric* constants are never enclosed in quotes.

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California

5.2 BOOLEAN OPERATORS *(AND, OR, NOT ,'=' , '>' , …)*

In a WHERE clause comparisons may be linked by the *logical operators AND* and *OR*, and possibly negated with the *NOT* operator.

5.2.1 Question

List all facts available in the table named RECENT_PRESIDENTS about all presidents who are Republican and were born in Texas.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       PARTY  =  'Republican'
      AND   STATE_BORN  =  'Texas'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Eisenhower D D	1890	8	79	Republican	Texas

5.2.2 Question

List all facts available in the table named RECENT_PRESIDENTS about all presidents who are Republican *or* were born in Texas.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       PARTY = 'Republican'
      OR    STATE_BORN = 'Texas'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Reagan R	1911	3	?	Republican	Illinois

5.2.3 Question

List all facts available in the table named RECENT_PRESIDENTS about all Republican presidents not born in Texas.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       PARTY = 'Republican'
   AND NOT  STATE_BORN = 'Texas'
```

Equivalent formulation with '<>' :

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       PARTY = 'Republican'
      AND   STATE_BORN <> 'Texas'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Reagan R	1911	3	?	Republican	Illinois

5.2.4 Question

List all facts available in the table named RECENT_PRESIDENTS about presidents who were born in Texas, and Republican presidents not born in California.

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       STATE_BORN = 'Texas'
      OR    ( PARTY = 'Republican'   AND NOT  STATE_BORN = 'California ' )
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas
Ford G R	1913	2	?	Republican	Nebraska
Reagan R	1911	3	?	Republican	Illinois

In the preceding examples, we were comparing fields with a single constant. If we have several constants, there are two possibilities. One is to split the comparison into several comparisons, so that each comparison has only one constant, and combine these comparisons using the logical operator OR. The other possibility is to use a set comparison operator together with a set of constants. The most frequently used set comparison operator is IN (or =ANY), which tests whether a given field value is contained in the set which is the second operand of the comparison.

5.2.5 The Set comparison operator *IN* (or: = ANY)

5.2.6 Question

List all facts available in the table named RECENT_PRESIDENTS about presidents born in Texas, California, Georgia or New York.

First possibility, without using a set comparison operator:

```
SELECT    *
FROM      RECENT_PRESIDENTS
WHERE     STATE_BORN = 'Texas'
    OR    STATE_BORN = 'California'
    OR    STATE_BORN = 'Georgia'
    OR    STATE_BORN= 'New York'
```

Second possibility, using the set comparison operator ***IN***:

```
SELECT    *
FROM      RECENT_PRESIDENTS
WHERE     STATE_BORN  IN  ( 'Texas' , 'California' , 'Georgia' , 'New York' )
```

If we use a set comparison operator like IN (or =ANY) with a list of constants, this list must contain at least two elements. A constant list is enclosed in parentheses, and constants are separated by commas.

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Eisenhower D D	1890	8	79	Republican	Texas
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Carter J E	1924	4	?	Democratic	Georgia

Set comparison operators are described in greater detail in a following chapter about subqueries.

5.2.7 BETWEEN

Some other operators are also available for the convenient formulation of comparisons. The first of these is the BETWEEN operator, which tests whether a value is within a specified range:

```
... expression-1  [NOT]  BETWEEN  expression-2  AND  expression-3
```

The condition is satisfied if the value of expression-l lies between the values of expression-2 and expression-3 (or lies outside this range if the NOT option is specified). The three expressions can be of arbitrary complexity, and may contain column names, constants, subqueries and arithmetic or other operators. The only restriction is that they must all have compatible types.

Note that the range expression-2 to expression-3 is *inclusive*. The two conditions below are equivalent:

```
A   BETWEEN  B  AND   C
A >= B   AND   A <= C
```

5.2.8 Question

List all facts available in the table named RECENT_PRESIDENTS about presidents who died at an age between 60 and 70 years.

Formulation *without* the BETWEEN operator:

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       DEATH_AGE  >=  60    AND   DEATH_AGE <= 70
```

Formulation *with* the BETWEEN operator:

```
SELECT      *
FROM        RECENT_PRESIDENTS
WHERE       DEATH_AGE  BETWEEN   60   AND   70
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Roosevelt F D	1882	12	63	Democratic	New York
Johnson L B	1908	5	65	Democratic	Texas

5.2.9 LIKE

Another comparison facility which can be useful in some cases is the ability to compare a character field with a pattern. For this purpose we use the ***LIKE*** operator:

```
... column-name [NOT]  LIKE  quoted-string.
```

The quoted string may contain arbitrary characters, however, special meanings are reserved for the characters _ and %. The character _ represents any single character, while the character % represents any string of zero, one or more characters. These two special characters may be used together with ordinary characters in the quoted string, in any combination.

Addition: in some SQL-systems (notably in *Microsoft's MS Access*) other '*wild card'*-symbols can be used. In MS Access a '?' is used as a *wild card* for just one character and '*' for *none, one or more* characters.

5.2.10 Question

List all facts available in the table named RECENT_PRESIDENTS about presidents whose name has tbe letter 'e' in the second position.

```
SELECT    *
FROM      RECENT_PRESIDENTS
WHERE     PRES_NAME  LIKE  '_e%'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Kennedy J F	1917	2	46	Democratic	Massachusetts
Reagan R	1911	3	?	Republican	Illinois

Note that the quoted string '_e' is not the same as '_e%'. This pattern would match only strings containing two characters with an 'e' in the second position.

5.2.11 Question

List all facts available in the table named RECENT_PRESIDENTS about presidents whose name has the letter 'e' in the second position, and not the letter 'R' in the first position.

```
SELECT    *
FROM      RECENT_PRESIDENTS
WHERE     PRES_NAME  LIKE  '_e%'
      AND PRES_NAME  NOT LIKE  'R%'
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Kennedy J F	1917	2	46	Democratic	Massachusetts

5.2.12 Addition: remark I on '*NULL*'-values

NULL-values must be seen as an indication of '*at this moment unknown'*. Every (ex-) president sometime will die and only then can become a well-defined value of DEATH_AGE, but before decease that value is still unknown; so 'NULL'.
Therefore selection-comparisons in a WHERE-clause can have 3 possible values: 'true', 'false' and 'unknown'. *Only in the case of 'true' the concerning table rows will be shown.*
As a consequence, in the result generated by the next query:

```
SELECT    *
FROM      RECENT_PRESIDENTS
WHERE     DEATH_AGE = 90     OR     DEATH_AGE <> 90
```

no rows with NULL-values will appear!
To select the table rows with (or on the contrary: without) NULL-values, you must use explicitly the test '*IS NULL'* respectively '*IS NOT NULL'*.

6 Built-in '*aggregate*' Functions *(AVG, SUM, MIN, MAX, COUNT)*

There are five built-in '*aggregate'* functions: AVG, SUM, MIN, MAX and COUNT.

AVG can be applied *to any numeric column or expression*, and calculates the average of the values for that column or expression. Null values are ignored in this calculation. If all the values are null, or the set of values is empty, the result of the calculation is null.
The AVG function can be used with or without the keyword DISTINCT.
If AVG (column-name) is used, then the average is computed for all values in the column, including duplicates.
If *AVG (DISTINCT column-name)* is used, then the average is computed only for the different values in the column, i.e. *excluding duplicates*.
Example:

A
10
6
6
6

AVG (A) => 7 (10 + 6 + 6 + 6) / 4 = 7
AVG (*DISTINCT* A) => 8 (10 + 6) / 2 = 8

SUM can be applied *to any numeric column or expression*, and calculates the sum of the values. Null values are ignored. If all the values are null, or the set of values is empty, the result of the calculation is also null.
The SUM function can also be used with or without the keyword DISTINCT.

MIN can be applied *to any column or expression*; and determines the smallest value. If applied to non-numeric values, ASCII (American Standard Code for Information Interchange) ordering is assumed.

MAX can be applied *to any column or expression*, and determines the largest value. If applied to non-numeric values ASCII (American Standard Code for Information Interchange) ordering is assumed.

COUNT can be applied *to a table or a column*, and determines the number of rows in the table or field values for the column. There are two cases:

- **COUNT (*)** determines the number of rows in a table. If the table is empty, then the value of the COUNT function is zero.
- **COUNT (*DISTINCT* column-name)** determines the number of unique field values for a column, excluding null values. If no non-null values are found, the value of the COUNT function is null.

Note: The object of a function *must be enclosed in parentheses*.

From here on in this text our examples will make use of the Presidential database. This database consists of a number of tables, described in Appendix 1 and 2 of this manual.

6.1.1 Question

Show the average age at death of deceased presidents.

```
SELECT    AVG ( DEATH_AGE )
FROM      PRESIDENT
```

Result:

AVG (DEATH_AGE)
68.85

Notes:

- Since null values are ignored, only deceased presidents are considered.
- Some database systems will give the user a message if nulls are ignored in computation.
- Some database systems will show (if any) another *header*/column-name for the result.
- Some database systems will round or trunc the numeric result.

6.1.2 Question

Show the total number of marriages and the total of the number of children of *all* presidents.

```
SELECT    COUNT ( * ) ,  SUM ( NR_CHILDREN )
FROM      PRES_MARRIAGE
```

Result:

COUNT *(*)*	***SUM*** (NR_CHILDREN)
44	143

6.1.3 Question

Show the age at death of the Democratic president who died the oldest.

```
SELECT    MAX ( DEATH_AGE )
FROM      PRESIDENT
WHERE     PARTY = 'Democratic'
```

As anticipated, the system probably warns of the null condition being encountered with some message.

Result:

MAX (DEATH_AGE)
88

Aggregate functions can only be applied in the SELECT clause (or in the HAVING clause as we will see in a next chapter). If used in the SELECT clause, there must be an aggregate function applied to all items of the SELECT list, *unless* the grouping feature (see a following chapter) is used.

6.1.4 Example 1:

Look at the following (*incorrect*!) query.

```
SELECT    PARTY, COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Republican'
```

This is *incorrect syntax*, because PARTY is not used with a function. The system will come back with some error-message like the following:

```
AN SQL ERROR HAS OCCURED
AN ITEM IN A SELECT-CLAUSE OR IN A HAVING CLAUSE WAS NEITHER A BUILT-IN FUNCTION
NOR A COLUMN IN THE GROUP BY SPECIFICATION
```

This can be corrected by also applying a function to PARTY which can only be MIN or MAX, since it is a character field. Whether we choose MIN or MAX is in this case irrelevant, because only Republican presidents are selected, so both the maximum and the minimum value of the party-name will be ‘Republican’.

6.1.5 Question

```
SELECT    MIN ( PARTY ) , COUNT (* )
FROM      PRESIDENT
WHERE     PARTY = 'Republican'
```

Result:

MIN (PARTY)	COUNT (*)
Republican	16

6.1.6 Example 2: Question

Show the age at death of the president who died the youngest.

```
SELECT    MIN ( DEATH_AGE )
FROM      PRESIDENT
```

Note: Before giving the result the system can give a message about ignoring null values.

Result:

MIN (DEATH_AGE)
46

The temptation arises to ask which president this was. The novice user sometimes tries the following (*incorrect*!) SQL query:

```
SELECT    PRES_NAME, MIN ( DEATH_AGE )
FROM      PRESIDENT
```

The system will again come back with some error-message.
Indeed this was a mismatch. MIN (DEATH_AGE) is *a characteristic of a group* of presidents while PRES_NAME is the name *of a specific president*.
In this case it is not so easy to find a solution for this question. The solution to this problem needs a *subquery*-construction, which is treated in one of the following chapters (about subqueries).

6.1.7 Question

How many inaugurations were there altogether?

```
SELECT    COUNT (*)
FROM      ADMINISTRATION
```

Result:

COUNT (*)
58

6.1.8 Question

How many individual presidents were there?

```
SELECT    COUNT ( DISTINCT  PRES_NAME )
FROM      ADMINISTRATION
```

Result: 39

Note: We could have obtained the same result simply by counting the rows in the table named PRESIDENT by using SELECT COUNT (*) FROM PRESIDENT.

We chose however the table named ADMINISTRATION in order to show the use of the DISTINCT operator within a COUNT function.

6.1.9 Question

How many presidential marriages were there altogether?

```
SELECT    COUNT (*)
FROM      PRES_MARRIAGE
```

Result: 44

6.1.10 Question

How many married presidents were there altogether?

```
SELECT    COUNT ( DISTINCT PRES_NAME )
FROM      PRES_MARRIAGE
```

Result: 38

Observation: MicroSofts ***MS Access*** RDBMS can not get on with this ***COUNT(DISTINCT …)*** construction!

6.2 A dangerous extension in the SELECT-clause: assimilation of a complete query

Many SQL-systems support as an extension of the possible expressions in a SELECT-clause that [the result of] a complete query is included in a comma-list. A condition is that such a second query must be put in *round brackets* and that its result is only a *single value*.
A severe warning is given here, as in this way it is possible to generate absurd surveys without a realistic meaning.
E.g. the query:

```
SELECT    pres_name, (SELECT COUNT(*) FROM president)  AS  XX
FROM      president
WHERE     pres_name like "F%"
```

with the result:

PRES_NAME	XX
Fillmore M	39
Ford G R	39

but what does it mean? It looks like "just data, without any informational content…"

Or even worse:

```
SELECT    pres_name, (SELECT spouse_name FROM pres_marriage
                      WHERE pres_name LIKE "Monroe%")  AS  XX
FROM      president
WHERE     pres_name like "F%"
```

with its result:

PRES_NAME	XX
Fillmore M	Kortright E
Ford G R	Kortright E

for sure has no realistic meaning…

This extension mostly only delivers meaningful results if the second query can be *'correlated'* to the main query (see chapter 12: correlated [sub]queries).
So be careful with this extension! In general: don't use it if there are better structured constructs!

7 Calculations through 'scalar' operators *(+, -, *, / ...)*

The following arithmetic 'scalar' operators may be used in numeric expressions:

+	Addition
-	Subtraction
*	Multiplication
/	Division

They may be used in the SELECT item list as well as within the WHERE clause. The following examples illustrate the use of arithmetic operations.

Observations/additions:

- In some database systems the '/'-operator is implemented to be used as 'integer'-division, where *rounding off*, but also *truncation* of the numeric result can occur. In other systems the division-result can be a 'real'-number.
- The above given operators all act on *numeric* values. In most systems also exist operations on *string values*, like the *concatenation* of strings. Such *string concatenation* then can be realized (depending on the database system) via for instance an operator as '||', '+' or '&'.

7.1.1 Question

Show the average age at death of deceased presidents.

To calculate the average, not using the AVG function, but using the SUM and COUNT functions, we can write:

```
SELECT    SUM ( DEATH_AGE ) / COUNT (*)
FROM      PRESIDENT
```

The SQL system may give something like the following messages:

```
ARI5021 FOLLOWING SQL WARNING CONDITION ENCOUNTERED:TRUNCATION
ARI5021 FOLLOWING SQL WARNING CONDITION ENCOUNTERED:NULL IGNORED
```

Thereafter we get the result: 61

If we compare this result with the result of Q7.01 (68.85) we see that there is a substantial difference.

The query above did not give us the right result because the SUM runction is applied only to the non-null values of DEATH_AGE (deceased presidents), while the COUNT function is applied to all rows of the PRESIDENT table (deceased or alive presidents). To obtain the *right* result, we have to eliminate the rows where DEATH_AGE is null:

7.1.2 Question

```
SELECT    SUM ( DEATH_AGE ) / COUNT (*)
FROM      PRESIDENT
WHERE     DEATH_AGE  IS NOT NULL
```

The SQL system may give a message like:

```
FOLLOWING SQL WARNING CONDITION ENCOUNTERED:  TRUNCATION
```

Thereafter we get the result: 61

Since the division has integer operands, the result is truncated. Thus, we obtain a slightly different result from the case where we used the AVG function (see Question 7.01).
How you avoid truncation will be discussed under Question 8.05.

7.1.3 Question

Find those deceased politicians who served more than 10% of their lives as president. List their names and this percentage.

```
SELECT    PRES_NAME, 100 * YRS_SERV/DEATH_AGE
FROM      PRESIDENT
WHERE     (100 * YRS_SERV / DEATH_AGE) > 10
    AND   DEATH_AGE  IS NOT NULL
```

The system may give the following message:

```
ARI5021 FOLLOWING SQL WARNING CONDITION ENCOUNTERED: TRUNCATION
```

If we use a system that truncates the numeric results, then we get as:
Result:

PRES_NAME	100 * YRS_SERV/DEATH_AGE
Grant U S	12
Cleveland G	11
Roosevelt T	11
Wilson W	11
Roosevelt F D	19

Depending on the system it will be possible to change the column names to be more user-friendly.

7.1.4 Question

Give the name and age of president and spouse for those marriages where the president was at least 5 years older than the spouse and the spouse was less than 20 years old.

```
SELECT    PRES_NAME, SPOUSE_NAME, PR_AGE, SP_AGE
FROM      PRES_MARRIAGE
WHERE     PR_AGE > SP_AGE + 5
    AND   SP_AGE < 20
```

Result:

PRES_NAME	SPOUSE_NAME	PR_AGE	SP_AGE
Adams J	Smith A	28	19
Monroe J	Kortright E	27	17
Eisenhower D D	Doud G	25	19

7.1.5 Question

Which presidents were no more than 10% older than their spouse(s) at the time of marriage! List their name, their age at marriage, their spouse's age at marriage and the ratio of the president's age to his spouse's age as a decimal number.

Let us first look at the following SQL formulation:

7.1.6 Question (a)

```
SELECT    PRES_NAME, PR_AGE, SP_AGE, PR_AGE/SP_AGE
FROM      PRES_MARRIAGE
WHERE     PR_AGE / SP_AGE   BETWEEN  1  AND  1.1
```

If the system will apply truncation, it may give a message about that truncation and afterwards give (37) result-rows, but most of them are not useful to us.
There will be unexpected errors in the answer. Take for example Adams J. His age was 28 and that of his spouse was 19, and 28/19 = 1.47 which is clearly outside the range between 1 and 1.1. This unexpected result arises from the fact that the columns PR_AGE and SP_AGE are both integers, and integer division in SQL truncates fractional results.
Note that for example 11/10 = 1, 15/10 = 1, 19/10 = 1, 20/10 = 2, 24/10 = 2 etc.
One can easily avoid such problems and get the expected results by prefixing the division expression with a multiplication by a decimal number such as 1.0.
In our example the query in SQL then becomes:

7.1.7 Question (b)

```
SELECT    PRES_NAME, PR_AGE , SP_AGE , 1.0 * PR_AGE / SP_AGE
FROM      PRES_MARRIAGE
WHERE     1.0 * PR_AGE / SP_AGE   BETWEEN  1.0  AND  1.1
```

Now there is no warning because we have a division of decimal numbers and the exact result is displayed as follows:

PRES_NAME	PR_AGE	SP_AGE	1.0 * PR_AGE/SP_AGE
Jackson A	26	26	1.00
Van Buren M	24	23	1.04
Harrison W H	22	20	1.10
Tyler J	23	22	1.04
Pierce F	29	28	1.03
Garfield J A	26	26	1.00
Hoover H C	24	23	1.04
Truman H S	35	34	1.02

8 The Grouping Feature *(Group By … / Having …)*

Suppose we want to know the number of presidents in each party. With the SQL features discussed so far, this would require repeatedly inquiring about each party, as follows:

```
SELECT    COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Demo-Rep'                Result:   4

SELECT    COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Democratic'              Result:  13

SELECT    COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Federalist'              Result:   2

SELECT    COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Republican'              Result:  16

SELECT    COUNT (*)
FROM      PRESIDENT
WHERE     PARTY = 'Whig'                    Result:   4
```

It is obvious that this is a tedious procedure. Furthermore one has to know all the party names. This list could have been retrieved with the command SELECT DISTINCT PARTY FROM PRESIDENT, but then the user would still need to type a separate SELECT command for each party to find out the number or presidents in that party.
The operation we would actually want to do in such cases is to apply a built-in aggregate function (AVG, SUM, MIN, MAX, COUNT) or a calculation to each one or a number of specific subcategories of that column or table. The features we have introduced so far are not well suited to handle this kind of query. Such queries can however be handled easily by using the grouping feature, expressed by the GROUP BY clause.

```
SELECT    …
FROM      …
[WHERE    … ]
[GROUP BY column-name, or list-or-column-names ]
```

The effect of this GROUP BY clause on the query is that any calculation or function in the SELECT clause is applied to each individual group of elements specified in the GROUP BY clause; thus generating one single row in the result per each group.

The GROUP BY feature is very powerful and deserves additional explanation.

Let us look at the result of the following query:

```
SELECT    *
FROM      PRESIDENT
ORDER BY  PARTY
```

The result is presented in the next figure.

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN	Group nr
Adams J Q	1767	4	80	Demo-Rep	Massachusetts	
Monroe J	1758	8	73	Demo-Rep	Virginia	1
Jefferson T	1743	8	83	Demo-Rep	Virginia	
Madison J	1751	8	85	Demo-Rep	Virginia	
Kennedy J F	1917	2	46	Democratic	Massachusetts	
Truman H S	1884	7	88	Democratic	Missouri	
Pierce F	1804	4	64	Democratic	New Hampshire	
Cleveland G	1837	8	71	Democratic	New Jersey	
Roosevelt F D	1882	12	63	Democratic	New York	
Van Buren M	1782	4	79	Democratic	New York	2
Polk J K	1795	4	53	Democratic	North Carolina	
Johnson A	1808	3	66	Democratic	North Carolina	
Buchanan J	1791	4	77	Democratic	Pennsylvania	
Jackson A	1767	8	78	Democratic	South Carolina	
Johnson L B	1908	5	65	Democratic	Texas	
Wilson W	1856	8	67	Democratic	Virginia	
Carter J E	1924	4	?	Democratic	Georgia	
Adams J	1735	4	90	Federalist	Massachusetts	3
Washington G	1732	7	67	Federalist	Virginia	
Hoover H C	1874	4	90	Republican	Iowa	
Lincoln A	1809	4	56	Republican	Kentucky	
Roosevelt T	1858	7	60	Republican	New York	
Garfield J A	1831	0	49	Republican	Ohio	
Harding W G	1865	2	57	Republican	Ohio	
McKinley W	1843	4	58	Republican	Ohio	
Grant U S	1822	8	63	Republican	Ohio	
Harrison B	1833	4	67	Republican	Ohio	
Hayes R B	1822	4	70	Republican	Ohio	4
Taft W H	1857	4	72	Republican	Ohio	
Eisenhower D D	1890	8	79	Republican	Texas	
Arthur C A	1830	3	56	Republican	Vermont	
Coolidge C	1872	5	60	Republican	Vermont	
Nixon R M	1913	5	?	Republican	California	
Reagan R	1911	3	?	Republican	Illinois	
Ford G R	1913	2	?	Republican	Nebraska	
Fillmore M	1800	2	74	Whig	New York	
Taylor Z	1784	1	65	Whig	Virginia	
Harrison W H	1773	0	68	Whig	Virginia	5
Tyler J	1790	3	71	Whig	Virginia	

The count to the right as well as it's heading was done by the authors and is not displayed on the screen. We see that there are 5 groups of rows in the table PRESIDENT, corresponding to the 5 parties.

The GROUP BY clause has the following effect upon the operation of the SELECT: All functions in the SELECT clause operate on each group within a table, independent of the number of groups.
If we apply to our example, GROUP BY PARTY, then all functions specified in the SELECT clause will be computed once for each of the 5 groups of parties.

8.1.1 Question

For each party, count the number of presidents who belonged to this party. List the name of each party, together with this count.

To count the number of presidents irrespective of party we would issue the following SQL query:

```
SELECT    COUNT (*)
FROM      PRESIDENT
```

This query counts the number of presidents in the *entire* PRESIDENT table.

Result: 39

If we actually want to count the number of presidents which belonged to each party, we need to use the GROUP BY feature.
If we had to perform this task without SQL, we would take the table PRESIDENT and form as many groups of rows as there are parties. In this case, there are five parties, thus there are five corresponding groups of rows. We would then count the number of rows in each group of rows to give the required result.
In SQL we do basically the same. First, since we want to display the names of parties corresponding to the groups, we have to specify the column-name PARTY in the SELECT clause. Secondly we specify that we want the count of members in each party group.

```
SELECT      PARTY, COUNT (*)
FROM        PRESIDENT
GROUP BY    PARTY
ORDER BY    PARTY
```

Note that if the query contains a GROUP BY clause, then any column in the SELECT list must either be contained in the GROUP BY clause or have a built in aggregate function applied to it.

Result:

PARTY	COUNT (*)
Demo-Rep	4
Democratic	13
Federalist	2
Republican	16
Whig	4

8.1.2 Question

For each state, count the number of presidents born in that state. List the state name together with this count, and present in ascending order, and within the same count sort by ascending state name.

```
SELECT      STATE_BORN,  COUNT (*)
FROM        PRESIDENT
GROUP BY    STATE_BORN
ORDER BY    2, STATE_BORN
```

Note:
The number 2 in the ORDER BY clause refers to the second item in the SELECT list COUNT (*). This is necessary since only columns can be referred to by their name.

Result:

STATE_BORN	COUNT (*)
California	1
Georgia	1
Illinois	1
Iowa	1
Kentucky	1
Missouri	1
Nebraska	1
New Hampshire	1
New Jersey	1
Pennsylvania	1
South Carolina	1

North Carolina	2
Texas	2
Vermont	2
Massachusetts	3
New York	4
Ohio	7
Virginia	8

8.1.3 Question

For each party, calculate the total number of years served by presidents of that party, the number of presidents, and the average number of years served. List the party, total number of years served, number of presidents, and average number of years served.

```
SELECT      PARTY, SUM (YRS_SERV ), COUNT (*),  AVG ( YRS_SERV )
FROM        PRESIDENT
GROUP BY    PARTY
ORDER BY    PARTY
```

Result:

PARTY	SUM	COUNT	AVG
Demo-Rep	28	4	7
Democratic	73	13	6
Federalist	11	2	6
Republican	67	16	4
Whig	6	4	2

The list of column names in the GROUP BY clause is not restricted to a single column. Several column names mean that there are several criteria for grouping, or several dimensions of grouping. In the following example, we show a two-dimensional grouping. Let us look at the result of the following query:

```
SELECT      *
FROM        PRESIDENT
ORDER BY    PARTY, STATE_BORN, PRES_NAME
```

The result is presented in the next figure.
If we apply the clause GROUP BY PARTY, STATE_BORN to the PRESIDENT table, then all functions specified in the SELECT clause will be computed once for each of the 26 party / state-born combinations.

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN	*Group nr*
Adams J Q	1767	4	80	Demo-Rep	Massachusetts	1
Jefferson T	1743	8	83	Demo-Rep	Virginia	2
Madison J	1751	8	85	Demo-Rep	Virginia	
Monroe J	1758	8	73	Demo-Rep	Virginia	
Carter J E	1924	4	?	Democratic	Georgia	3
Kennedy J F	1917	2	46	Democratic	Massachusetts	4
Truman H S	1884	7	88	Democratic	Missouri	5
Pierce F	1804	4	64	Democratic	New Hampshire	6
Cleveland G	1837	8	71	Democratic	New Jersey	7
Roosevelt F D	1882	12	63	Democratic	New York	8
Van Buren M	1782	4	79	Democratic	New York	
Johnson A	1808	3	66	Democratic	North Carolina	9
Polk J K	1795	4	53	Democratic	North Carolina	
Buchanan J	1791	4	77	Democratic	Pennsylvania	10
Jackson A	1767	8	78	Democratic	South Carolina	11
Johnson L B	1908	5	65	Democratic	Texas	12
Wilson W	1856	8	67	Democratic	Virginia	13

Adams J	1735	4	90	Federalist	Massachusetts	14
Washington G	1732	7	67	Federalist	Virginia	15
Nixon R M	1913	5	?	Republican	California	16
Reagan R	1911	3	?	Republican	Illinois	17
Hoover H C	1874	4	90	Republican	Iowa	18
Lincoln A	1809	4	56	Republican	Kentucky	19
Ford G R	1913	2	?	Republican	Nebraska	20
Roosevelt T	1858	7	60	Republican	New York	21
Garfield J A	1831	0	49	Republican	Ohio	
Grant U S	1822	8	63	Republican	Ohio	
Harding W G	1865	2	57	Republican	Ohio	
Harrison B	1833	4	67	Republican	Ohio	22
Hayes R B	1822	4	70	Republican	Ohio	
McKinley W	1843	4	58	Republican	Ohio	
Taft W H	1857	4	72	Republican	Ohio	
Eisenhower D D	1890	8	79	Republican	Texas	23
Arthur C A	1830	3	56	Republican	Vermont	24
Coolidge C	1872	5	60	Republican	Vermont	
Fillmore M	1800	2	74	Whig	New York	25
Harrison W H	1773	0	68	Whig	Virginia	
Taylor Z	1784	1	65	Whig	Virginia	26
Tyler J	1790	3	71	Whig	Virginia	

8.1.4 Question

Count the presidents who were members of the same party and who were born in the same state. List party, state of birth, and this count.

```
SELECT      PARTY, STATE_BORN , COUNT (*)
FROM        PRESIDENT
GROUP BY    PARTY, STATE_BORN
ORDER BY    1 , 2
```

Result:

PARTY	STATE_BORN	COUNT(*)
Demo-Rep	Massachusetts	1
Demo-Rep	Virginia	3
Democratic	Georgia	1
Democratic	Massachusetts	1
Democratic	Missouri	1
Democratic	New Hampshire	1
Democratic	New Jersey	1
Democratic	New York	2
Democratic	North Carolina	2
Democratic	Pennsylvania	1
Democratic	South Carolina	1
Democratic	Texas	1
Democratic	Virginia	1
Federalist	Massachusetts	1
Federalist	Virginia	1
Republican	California	1
Republican	Illinois	1
Republican	Iowa	1
Republican	Kentucky	1
Republican	Nebraska	1
Republican	New York	1
Republican	Ohio	7
Republican	Texas	1
Republican	Vermont	2
Whig	New York	1
Whig	Virginia	3

The groups formed can be illustrated by a two dimensional matrix, where one dimension is PARTY, and the other is STATE_BORN:

STATE-BORN	PARTY Federalist	PARTY Demo-Rep	PARTY Whig	PARTY Democratic	PARTY Republican
Virginia	+	+++	+++	+	
Massachusetts	+	+		+	
South Carolina				+	
New York			+	++	+
North Carolina				++	
New Hampshire				+	
Pennsylvania				+	
Kentucky					+
Ohio					+++++++
Vermont					++
New Jersey				+	
Iowa					+
Missouri				+	
Texas				+	+
California					+
Nebraska					+
Georgia				+	
Illinois					+

Each field with at least one plus sign in the matrix represents a group. Each plus sign in the matrix represents a president. The COUNT function is applied to each one of these groups.

8.1.5 Question

For each birth state / party combination, count the presidents who were born in that state and who were members of that party. List state, party and this count.

```
SELECT      STATE_BORN, PARTY, COUNT (*)
FROM        PRESIDENT
GROUP BY    STATE_BORN, PARTY
ORDER BY    1, 2
```

Result:

STATE_BORN	PARTY	COUNT
California	Republican	1
Georgia	Democratic	1
Illinois	Republican	1
Iowa	Republican	1
Kentucky	Republican	1
Massachusetts	Demo-Rep	1
Massachusetts	Democratic	1
Massachusetts	Federalist	1
Missouri	Democratic	1
Nebraska	Republican	1
New Hampshire	Democratic	1
New Jersey	Democratic	1
New York	Democratic	2
New York	Republican	1
New York	Whig	1
North Carolina	Democratic	2
Ohio	Republican	7
Pennsylvania	Democratic	1
South Carolina	Democratic	1
Texas	Democratic	1
Texas	Republican	1
Vermont	Republican	2

Virginia	Demo-Rep	3
Virginia	Democratic	1
Virginia	Federalist	1
Virginia	Whig	3

Please note that the count is the same as in the previous question, but the results are presented in a different order. The sequence of column names in the GROUP BY clause is not relevant to the order of display, but the sequence of column names in the ORDER BY clause does affect the order.

It is also possible to apply a WHERE clause together with a GROUP BY clause. In such a case the WHERE clause acts as a sieve, removing those rows which do not satisfy the search condition. After unwanted rows are removed, the groups are formed and the built-in aggregate functions are applied to the rows in each group.

8.1.6 Question

For each party, list the party name and the number of presidents born after the year 1850.

```
SELECT      PARTY, COUNT (*)
FROM        PRESIDENT
WHERE       BIRTH_YR > 1850
GROUP BY    PARTY
```

Result:

PARTY	COUNT
Democratic	6
Republican	9

If we apply a WHERE clause together with a GROUP BY clause, we restrict the number of rows which form the various groups. If we wish rather to restrict the groups themselves, we have to specify a condition on the selection of groups. This is achieved with the HAVING clause:

```
SELECT      …
FROM        …
[WHERE      condition ]
[GROUP BY   column-name, or list-of-column-names ]
[HAVING     condition ]
```

The HAVING clause relates to the GROUP BY clause in the same way as the WHERE clause relates to the FROM clause. A *WHERE clause is applied to each row* before the groups are formed, while a *HAVING clause is applied to each group*.

The following examples illustrate the use of the HAVING clause.

8.1.7 Question

List the names of presidents and the number of their marriages for those presidents who married more than once.

We group the PRES_MARRIAGE table into groups having the same president (GROUP BY PRES_NAME). We then want to retain only those groups which have more than one marriage (HAVING COUNT (*) >1).

⇨ *With the HAVING clause one selects or rejects **groups**.*
⇨ *With the WHERE clause one selects or rejects **individual rows**.*

```
SELECT      PRES_NAME, COUNT (*)
FROM        PRES_MARRIAGE
GROUP BY    PRES_NAME
HAVING      COUNT (*) > 1
```

Result:

PRES_NAME	COUNT
Fillmore M	2
Harrison B	2
Reagan R	2
Roosevelt T	2
Tyler J	2
Wilson W	2

8.1.8 Question

For those parties which had more than 8 presidents born after 1850, list the names of the parties and the corresponding number of presidents born after 1850.

A typical ***erroneous*** (!) solution is as follows:

```
SELECT      PARTY, COUNT ( * )
FROM        PRESIDENT
WHERE       BIRTH YR > 1850
      AND   COUNT (*) > 8                    ( erroneous ! )
GROUP BY    PARTY
```

The SQL system will return something like the following error-message:

> AN SQL PROCESSING ERROR HAS OCCURRED. A BUILT-IN AGGREGATE FUNCTION SHOULD NOT OCCUR IN A WHERE-CLAUSE OR AS THE VALUE TO BE ASSIGNED TO A COLUMN IN A SET-CLAUSE OF AN UPDATE STATEMENT.

The error is that the condition that an individual president be born after 1850 is clearly a row condition and therefore belongs to the WHERE clause. However, COUNT (*) > 8 is a condition which cannot apply to a row because COUNT (*) deals with an entire table or group. Therefore COUNT (*) > 8 has to come in the HAVING clause.

Correct solution:

```
SELECT      PARTY, COUNT (*)
FROM        PRESIDENT
WHERE       BIRTH_YR > 1850
GROUP BY    PARTY
HAVING      COUNT (*) > 8
```

Result:

PARTY	COUNT
Republican	9

8.1.9 Question

Find those presidents who married at least twice and whose maximum number of children in any of their marriages exceeds their minimum number or children by at least 2. List their names, and the maximum and minimum number of children.

We find the relevant information in the table PRES_MARRIAGE. Since we consider all married presidents, we have no WHERE clause in this case. But since a president may occur in several rows (marriages) of the table, and since we want to apply various functions (COUNT, MIN, MAX) to all marriages of a president, we have to group by president.

That means that we have groups of rows in the table PRES_MARRIAGE where the field PRES_NAME is equal for each row of the group. But we do not consider all PRES_NAME groups: only those fulfilling a specific condition, which has to be specified in a HAVING clause. Altogether, the retrieval command is as follows:

```
SELECT    PRES_NAME , MAX (NR_CHILDREN) , MIN (NR_CHILDREN)
FROM      PRES_MARRIAGE
GROUP BY  PRES_NAME
HAVING    COUNT (*) >= 2  AND  MAX (NR_CHILDREN) >= MIN (NR_CHILDREN) + 2
```

Result:

PRES_NAME	MAX	MIN
Fillmore M	2	0
Roosevelt T	5	1
Wilson W	3	0

9 Selecting Columns and Rows From Several Tables *('joins')*

All the SQL queries discussed in the preceding chapters have retrieved information *from a single table*.
One of the most useful and powerful functions in SQL is the ability to retrieve with one SQL command, information from more than one table.
In the Relational Data Model literature this operation is known as a ***JOIN***. The practically unrestricted JOIN operation is a major difference between the powerful 4th generation database systems and third generation database systems

Even when used on several tables, the SELECT command retains the same basic format as we have seen previously. The SELECT clause specifies the names of the columns we wish to retrieve from the tables, and these tables are listed in the FROM clause.
Let us first look at a simple example, dealing with the two tables presented in next figures.

P_TABLE

	PRES_NAME	BIRTH_YR
1	Buchanan J	1791
2	Harrison B	1833
3	Nixon R M	1913
4	Reagan R	1911

M_TABLE

	PRES_NAME	SPOUSE_NAME
1	Harrison B	Scott C L
2	Harrison B	Dimmick M S L
3	Nixon R M	Ryan T C
4	Reagan R	Wyman J
5	Reagan R	Davis N

We have selected to operate on the smaller tables of these figures because of the lengthy results possible with unrestricted joins.

If we join both tables and include all columns, we would produce as the result a table with 4 columns as follows:

P_TABLE		M_TABLE	
PRES_NAME	BIRTH_YR	PRES_NAME	SPOUSE_NAME

In order to distinguish between the two columns with the name PRES_NAME we have to qualify each column name with a prefix which is the name of the table from which the column is retrieved. The format of the qualified column name is as follows:

table-name.column-name

Qualification is not necessary if the column names are distinct among the tables included in the join. In our example this means there is no need to prefix the columns BIRTH_YR and SPOUSE_NAME with the names of the tables they are retrieved from. When qualifying a column name, there must be a period and no other character or space between the table name and the column name.

9.1.1 Question

If we want to produce as a result a table with 4 columns, where each row consists of the president name of the P_TABLE and his birth year followed by the president name of M_TABLE and spouse name, for all possible combinations, we have to write the following SQL query:

```
SELECT      P_TABLE.PRES_NAME, BIRTH_YR, M_TABLE.PRES_NAME,
            SPOUSE_NAME
FROM        P_TABLE, M_TABLE
ORDER BY    P_TABLE.PRES_NAME
```

We get as result the table of the next figure with 4 columns and 20(!!) rows.

Result:

	PRES_NAME	BIRTH_YR	PRES_NAME	SPOUSE_NAME
1	Buchanan J	1791	Harrison B	Scott C L
2	Buchanan J	1791	Harrison B	Dimmick M S L
3	Buchanan J	1791	Nixon R M	Ryan T C
4	Buchanan J	1791	Reagan R	Wyman J
5	Buchanan J	1791	Reagan R	Davis N
6	Harrison B	1833	Harrison B	Scott C L
7	Harrison B	1833	Harrison B	Dimmick M S L
8	Harrison B	1833	Nixon R M	Ryan T C
9	Harrison B	1833	Reagan R	Wyman J
10	Harrison B	1833	Reagan R	Davis N
11	Nixon R M	1913	Harrison B	Scott C L
12	Nixon R M	1913	Harrison B	Dimmick M S L
13	Nixon R M	1913	Nixon R M	Ryan T C
14	Nixon R M	1913	Reagan R	Wyman J
15	Nixon R M	1913	Reagan R	Davis N
16	Reagan R	1911	Harrison B	Scott C L
17	Reagan R	1911	Harrison B	Dimmick M S L
18	Reagan R	1911	Nixon R M	Ryan T C
19	Reagan R	1911	Reagan R	Wyman J
20	Reagan R	1911	Reagan R	Davis N

In general, if there is more than one table listed in the FROM clause and there is no WHERE, GROUP BY, or HAVING clause, then the SQL system will produce as result a table in which the number of rows is equal to the product of the number of rows of the tables in the FROM clause. In other words, all possible combinations of rows from the tables listed in the FROM clause are listed in the result. Each row of each table is combined with each row of each other table in the FROM clause. Of course, this is a useful result only in exceptional cases, and it does tend to use enormous quantities of computer resources.

We want to emphasise that the result of specifying two or more table names in the FROM clause is again a table, and it is on this product table that the WHERE, GROUP BY, HAVING and ORDER BY clauses operate.
The join with all possible combinations of rows, i.e. a join with only the SELECT clause and the FROM clause is not normally a desirable result. However, it has shown to be an excellent concept to think of a join as if it was a normal SELECT on one table, namely the table with all columns from the tables specified in the FROM clause and all possible combinations or rows from these tables. The columns specified in the SELECT clause are the ones desired in the result and the WHERE clause specifies which rows are to be included in the result.

The part of the WHERE clause which specifies a condition between two columns in the originally separate tables is called a *join-condition*. All other parts of the WHERE clause are called *search-conditions*.
Let us now look at a more practical question, where we will retain only meaningful combinations of rows, and where we will not ask for the common column to be repeated in the result.

9.1.2 Question

List the name, birth year and spouse names of all married presidents in the tables of the figure with the P_TABLE and M_TABLE.

```
SELECT    P_TABLE.PRES_NAME, BIRTH_YR , SPOUSE_NAME
FROM      P_TABLE, M_TABLE
WHERE     P_TABLE.PRES_NAME = M_TABLE.PRES_NAME
```

Result:

PRES_NAME	BIRTH_YR	SPOUSE_NAME
Harrison B	1833	Scott C L
Harrison B	1833	Dimmick M S L
Nixon R M	1913	Ryan T C
Reagan R	1911	Wyman J
Reagan R	1911	Davis N

The WHERE clause operates on each of the 20 rows of the former figure. Only rows 6, 7, 13, 19 and 20 satisfy the WHERE clause.
An equality operator between two columns of different tables is the most common join condition, as well as the most efficient. When using other join conditions, keep in mind the potentially enormous volume or the product table that may have to be scanned.
Up to 16 tables may be listed in the FROM clause. For practical applications this has shown to be a very safe upper limit.

Let us now return to the sample tables of the presidential database in appendix B.

9.1.3 Question

List names, birth years, marriage years and spouses of all married presidents. Order by president name.

We find that the information we want to retrieve is contained in two tables, the PRESIDENT table and the PRES_MARRIAGE table. Thus we have to perform a join.
Therefore we have to combine each row of the PRESIDENT table with the corresponding row(s) of the PRES_MARRIAGE table where the fields in the column PRES_NAME of the PRES_MARRIAGE table are equal to the fields in the column PRES_NAME of the PRESIDENT table. The SELECT clause specifies which columns will be selected from the join of the two tables.

```
SELECT      PRESIDENT.PRES_NAME,  BIRTH_YR, MAR_YEAR,  SPOUSE_NAME
FROM        PRESIDENT, PRES_MARRIAGE
WHERE       PRESIDENT.PRES_NAME = PRES_MARRIAGE.PRES_NAME
ORDER BY    PRESIDENT.PRES_NAME
```

Note:
Each reference to a non-unique column name (appearing in more than one table in the FROM clause) must be prefixed with the table name, hence the ORDER BY (like the SELECT clause) must stipulate PRESIDENT.PRES_NAME.

Result:

PRES_NAME	BIRTH_YR	MAR_YEAR	SPOUSE_NAME
Adams J	1735	1764	Smith A
Adams J Q	1767	1797	Johnson L C
Arthur C A	1830	1859	Herndon E L
Carter J E	1924	1946	Smith R
Cleveland G	1837	1886	Folson F
Coolidge C	1872	1905	Goodhue G A
Eisenhower D D	1800	1916	Doud G
Fillmore M	1800	1826	Powers A
Fillmore M	1800	1858	McIntosh C C
Ford G R	1913	1948	Warren E B
Garfield J A	1831	1858	Rudolph L
Grant U S	1822	1848	Dent J B
Harding W G	1865	1891	De Wolfe F K

Harrison B	1833	1853	Scott C L
Harrison B	1833	1896	Dimmick M S L
Harrison W H	1773	1795	Symmes A T
Hayes R B	1822	1852	Webb L W
Hoover H C	1874	1899	Henry L
Jackson A	1767	1794	Robards R D
Jefferson T	1743	1772	Skelton M W
Johnson A	1808	1827	McCardle E
Johnson L B	1908	1934	Taylor C A
Kennedy J F	1917	1953	Bouvier J L
Lincoln A	1809	1842	Todd M
Madison J	1751	1794	Todd D D P
McKinley W	1843	1871	Saxton I
Monroe J	1758	1786	Kortright E
Nixon R M	1913	1940	Ryan T C
Pierce F	1804	1834	Appleton J M
Polk J K	1795	1824	Childress S
Reagan R	1911	1940	Wyman J
Reagan R	1911	1952	Davis N
Roosevelt F D	1882	1905	Roosevelt A E
Roosevelt T	1858	1880	Lee A H
Roosevelt T	1858	1886	Carow E K
Taft W H	1857	1886	Herron H
Taylor Z	1784	1810	Smith M M
Truman H S	1884	1919	Wallace E V
Tyler J	1790	1813	Christian L
Tyler J	1790	1844	Gardiner J
Van Buren M	1782	1807	Hoes H
Washington G	1732	1759	Custis M D
Wilson W	1856	1885	Axson E L
Wilson W	1856	1915	Galt E B

Note
Because the president Buchanan J is in the PRESIDENT table but not in the PRES_MARRIAGE table, he did not satisfy the WHERE clause and is therefore not in our result.

9.1.4 Question

List president name, birth year, the administrations served as president and the vice presidents in each administration, in order of administration number.

```
SELECT      PRESIDENT.PRES_NAME , BIRTH_YR , ADMIN_NR, VICE_PRES_NAME
FROM        PRESIDENT,  ADMIN_PR_VP
WHERE       PRESIDENT.PRES_NAME  =  ADMIN_PR_VP.PRES_NAME
ORDER BY    ADMIN_NR
```

Please note that the only condition in the WHERE clause is a *join-condition*.

Result:

PRES_NAME	BIRTH_YR	ADMIN_NR	VICE_PRES_NAME
Washington G	1732	1	Adams J
Washington G	1732	2	Adams J
Adams J	1735	3	Jefferson T
Jefferson T	1743	4	Burr A
Jefferson T	1743	5	Clinton G
Madison J	1751	6	Clinton G
Madison J	1751	7	Gerry E
Monroe J	1758	8	Tompkins D
Monroe J	1758	9	Tompkins D
Adams J Q	1767	10	Calhoun J
Jackson A	1767	11	Calhoun J

Jackson A	1767	12	Van Buren M
Van Buren M	1782	13	Johnson R M
Harrison W H	1773	14	Tyler J
Polk J K	1795	15	Dallas G M
Taylor Z	1784	16	Fillmore M
Pierce F	1804	17	De Vane King W R
Buchanan J	1791	18	Breckinridge J C
Lincoln A	1809	19	Hamlin H
Lincoln A	1809	20	Johnson A
Grant U S	1822	21	Colfax S
Grant U S	1822	22	Wilson H
Hayes R B	1822	23	Wheeler W
Garfield J A	1831	24	Arthur C A
Cleveland G	1837	25	Hendricks T A
Harrison B	1833	26	Morton L P
Cleveland G	1837	27	Stevenson A E
McKinley W	1843	28	Hobart G A
McKinley W	1843	29	Roosevelt T
Roosevelt T	1858	30	Fairbanks C W
Taft W H	1857	31	Sherman J S
Wilson W	1856	32	Marshall T R
Wilson W	1856	33	Marshall T R
Harding W G	1865	34	Coolidge C
Coolidge C	1872	35	Dawes C G
Hoover H C	1874	36	Curtis C
Roosevelt F D	1882	37	Garner J N
Roosevelt F D	1882	38	Garner J N
Roosevelt F D	1882	39	Wallace H A
Roosevelt F D	1882	40	Truman H S
Truman H S	1884	41	Barkley A W
Eisenhower D D	1890	42	Nixon R M
Eisenhower D D	1890	43	Nixon R M
Kennedy J F	1917	44	Johnson L B
Johnson L B	1908	45	Humphrey H H
Nixon R M	1913	46	Agnew S T
Nixon R M	1913	47	Ford G R
Nixon R M	1913	47	Agnew S T
Ford G R	1913	47	Rockefeller N
Carter J E	1924	48	Mondale W F
Reagan R	1911	49	Bush G

As in previous examples there may be conditions involved in the WHERE clause other than just the matching condition for the join. e.g. AND BIRTH_YR < 1800.

9.1.5 Question

List the names, birth years and hobbies of all presidents born before 1800. Order by birth year and president name.

In this case, we have to join the two tables PRESIDENT and PRES_HOBBY, such that the fields PRES_NAME are equal in both tables. This equality condition is called the *join-condition*. The condition that only those presidents who were born before 1800, is appended to the join condition in the WHERE clause with the logical operator AND, and is called a *search-condition*.

```
SELECT    PRESIDENT.PRES_NAME,  BIRTH_YR , HOBBY
FROM      PRESIDENT,  PRES_HOBBY
WHERE     PRESIDENT.PRES_NAME  =  PRES_HOBBY.PRES_NAME
     AND  BIRTH_YR < 1800
ORDER BY  BIRTH_YR,  PRESIDENT.PRES_NAME
```

Result:

PRES_NAME	BIRTH_YR	HOBBY
Washington G	1732	Fishing
Washington G	1732	Riding
Jefferson T	1743	Fishing
Jefferson T	1743	Riding
Adams J Q	1767	Walking
Adams J Q	1767	Billiards
Adams J Q	1767	Swimming
Jackson A	1767	Riding
Van Buren M	1782	Riding
Taylor Z	1784	Riding

9.1.6 Question

List name, birth year, marriage years and spouse names of those presidents who were born before 1776 and married before 1800. Order the list on president name in ascending order.

```
SELECT    PRESIDENT.PRES_NAME , BIRTH_YR , MAR_YEAR, SPOUSE_NAME
FROM      PRESIDENT, PRES_MARRIAGE
WHERE     PRESIDENT.PRES_NAME  =  PRES_MARRIAGE.PRES_NAME
    AND   BIRTH_YR  <  1776
    AND   MAR_YEAR <  1800
ORDER BY  PRESIDENT.PRES_NAME
```

Please note that the first condition in the WHERE clause is a *join-condition*, while the second and third are *search-conditions*.

Result:

PRES_NAME	BIRTH_YR	MAR_YEAR	SPOUSE_NAME
Adams J	1735	1764	Smith A
Adams J Q	1767	1797	Johnson L C
Harrison W H	1773	1795	Symmes A T
Jackson A	1767	1794	Robards R D
Jefferson T	1743	1772	Skelton M W
Madison J	1751	1794	Todd D D P
Monroe J	1758	1786	Kortright E
Washington G	1732	1759	Custis M D

9.1.7 Question

List the name, birth year, age at marriage, spouse's age at marriage and name, for all presidents who married when they were less than 20 years old, or who married a spouse less than 18 years of age. Order by age of president at marriage.

```
SELECT    PRESIDENT.PRES_NAME , BIRTH_YR , PR_AGE , SP_AGE,
          SPOUSE_NAME
FROM      PRESIDENT,  PRES_MARRIAGE
WHERE     PRESIDENT.PRES_NAME  =  PRES_MARRIAGE.PRES_NAME
    AND   ( PR_AGE  <  20   OR   SP_AGE  <  18 )
ORDER BY  PR_AGE
```

Result:

PRES_NAME	BIRTH_YR	PR_AGE	SP_AGE	SPOUSE_NAME
Johnson A	1808	18	16	McCardle E
Monroe J	1758	27	17	Kortright E

The powerful join operator can be used in connection with other SQL features, like the group-by operator, built-in (aggregate) functions or other calculations.

9.1.8 Question

For each president with more than three children, list their name, their birth year and the number of children from all marriages. Order by number of children in descending order, and then by name.

To formulate that query, we use a built-in function to retrieve the total number of each president's children, as six of the 39 presidents married more than once. We have to use the grouping feature to summarise by president name.
As each president has only one birth year we are able to apply a built-in function which will enable the grouping criteria to be met. Our choice is either MIN or MAX, either will suffice. In our query we have selected MIN.

```
SELECT      PRESIDENT.PRES_NAME , MIN (BIRTH_YR) , SUM (NR_CHILDREN)
FROM        PRESIDENT,  PRES_MARRIAGE
WHERE       PRESIDENT.PRES_NAME  =  PRES_MARRIAGE.PRES_NAME
GROUP BY    PRESIDENT.PRES_NAME
HAVING      SUM  ( NR_CHILDREN )  > 3
ORDER BY    3 DESC, 1
```

Note:
In the ORDER BY clause, one can use the column name or the position of the column in the SELECT list counting from left to right. In the case of a function, one has to use the position.

Result:

PRES_NAME	MIN	SUM
Tyler J	1790	15
Harrison W H	1773	10
Hayes R B	1822	8
Garfield J A	1831	7
Jefferson T	1743	6
Roosevelt F D	1882	6
Roosevelt T	1858	6
Taylor Z	1784	6
Adams J	1735	5
Cleveland G	1837	5
Johnson A	1808	5
Adams J Q	1767	4
Carter J E	1924	4
Ford G R	1913	4
Grant U S	1822	4
Lincoln A	1809	4
Reagan R	1911	4
Van Buren M	1782	4

Addition:

9.2 Join with *GROUP BY* in case of *N : M -relations* between tables

If we are grouping the results of a join of several tables, we have to watch very meticulously how the process of grouping evolves exactly.
Especially when joining tables where the join condition is valid for a number of records *of both tables* (shortly: in case of ***n:m****-relations* between those tables).

Example:

Show the married presidents who have more than 3 hobbies and show in the same view for each president as well the number of marriages as the number of hobbies, ordered by decreasing number of hobbies.

The next, '*raw defined'* query (formulated without analyzing in detail) looks very 'natural':

```
SELECT      M.Pres_Name,  COUNT ( Spouse_name ),  COUNT ( Hobby )
FROM        Pres_marriage M, Pres_hobby H
WHERE       M.Pres_name = H.Pres_name
GROUP BY    M.Pres_name                                    { incorrect ! }
HAVING      COUNT ( Hobby )  > 3
ORDER BY 3  DESC
```

with the (*incorrect* !) result:

PRES_NAME	COUNT	COUNT
Roosevelt T	14	14
Wilson W	6	6
Eisenhower D D	5	5
Coolidge C	5	5

As you can see a somewhat strange and definitely not meant result.

To discover what has gone wrong [2], we formulate a join query of which the result still not has been grouped, but where we have put as much as possible the same conditions to the result:

```
SELECT      M.Pres_name, Spouse_name, Hobby
FROM        Pres_marriage M, Pres_hobby H
WHERE       M.Pres_name = H.Pres_name
  AND       M.Pres_name   IN  (    SELECT  Pres_name
                                   FROM    Pres_Hobby
                                   GROUP BY Pres_name
                                   HAVING  COUNT ( * ) > 3 )
```

The result of this test query is:

PRES_NAME	SPOUSE_NAME	HOBBY
Roosevelt T	Lee A H	Boxing
Roosevelt T	Lee A H	Hunting
Roosevelt T	Lee A H	Jujitsu
Roosevelt T	Lee A H	Riding
Roosevelt T	Lee A H	Shooting
Roosevelt T	Lee A H	Tennis
Roosevelt T	Lee A H	Wrestling
Roosevelt T	Carow E K	Boxing
Roosevelt T	Carow E K	Hunting
Roosevelt T	Carow E K	Jujitsu
Roosevelt T	Carow E K	Riding
Roosevelt T	Carow E K	Shooting
Roosevelt T	Carow E K	Tennis
Roosevelt T	Carow E K	Wrestling
Coolidge C	Goodhue G A	Fishing
Coolidge C	Goodhue G A	Golf
Coolidge C	Goodhue G A	Indian Clubs
Coolidge C	Goodhue G A	Mechanical Horse
Coolidge C	Goodhue G A	Pitching Hay
Eisenhower D D	Doud G	Bridge
Eisenhower D D	Doud G	Golf
Eisenhower D D	Doud G	Hunting
Eisenhower D D	Doud G	Painting

[2] Formulating such a test query to make manifest the intermediate results, is anyhow a good strategy to test if a worked out query indeed gives the desired, correct result; see this as '*debugging* in SQL'.

Eisenhower D D	Doud G	Fishing

In this view we can see, that for president Roosevelt all 7 hobbies are combined with each of his two spouses successively.

This is a difficulty that can arise if as well in the first (Marriage-) table as in the other (Hobby-) table several records can appear for a single president and we thus touch on an *n:m-relation*. How do we get the desired, correct result (so for each president the number of his marriages and his hobbies)?

In this case, the most obvious way to get a correct result, is *not to count* the number of rows, but to count the number of *distinct* president marriages (via e.g. their different Spouse_name) as well as their distinct hobbies (through there hobby-name).
This most obvious way conducts us to the following query:

```
SELECT      M.Pres_Name, COUNT ( DISTINCT Spouse_name ) ,
                                        COUNT ( DISTINCT Hobby )
FROM        Pres_marriage M, Pres_hobby H
WHERE       M.Pres_name = H.Pres_name
GROUP BY    M.Pres_name
HAVING      COUNT ( DISTINCT Hobby )  > 3
ORDER BY 3  DESC
```

With the expected correct result:

PRES_NAME	COUNT	COUNT
Roosevelt T	2	7
Eisenhower D D	1	5
Coolidge C	1	5

10 Subqueries

The subquery is a particularly powerful SQL concept. *It permits an SQL user to phrase in one query a complicated question which would otherwise have required more than one query*. A subquery is simply a way of using the results of one select command inside another, without having to substitute the results by hand.
Let us first look at an example to introduce the concept.

10.1.1 Question

List all the facts of those presidential marriages which resulted in a number of children that is greater than the average number of children per presidential marriage.

With the SQL concepts so far described, we need to formulate two queries:

1. List the average number of children per presidential marriage.
   The result is 3.25.

2. List all facts about those marriages which have more than 3.25 children.

The first query in SQL is:

```
SELECT    AVG ( NR_CHILDREN )
FROM      PRES_MARRIAGE
```

This query results in a single number, 3.25, which can then be used in the second part:

```
SELECT    *
FROM      PRES_MARRIAGE
WHERE     NR_CHILDREN  > 3.25
```

With a subquery, this can be done *in one step*:

```
SELECT    *
FROM      PRES_MARRIAGE
WHERE     NR_CHILDREN  >  (  SELECT     AVG ( NR_CHILDREN )
                            FROM       PRES_MARRIAGE   )
```

Result:

PRES_NAME	SPOUSE_NAME	PR_AGE	SP_AGE	NR_CHILDREN	MAR_YEAR
Adams J	Smith A	28	19	5	1764
Jefferson T	Skelton M W	28	23	6	1772
Adams J Q	Johnson L C	30	22	4	1797
Van Buren M	Hoes H	24	23	4	1807
Harrison W H	Symmes A T	22	20	10	1795
Tyler J	Christian L	23	22	8	1813
Tyler J	Gardiner J	54	24	7	1844
Taylor Z	Smith M M	25	21	6	1810
Lincoln A	Todd M	33	23	4	1842
Johnson A	McCardle E	18	16	5	1827
Grant U S	Dent J B	26	22	4	1848
Hayes R B	Webb L W	30	21	8	1852
Garfield J A	Rudolph L	26	26	7	1858
Cleveland G	Folson F	49	21	5	1886
Roosevelt T	Carow E K	28	25	5	1886
Roosevelt F D	Roosevelt A E	23	20	6	1905
Ford G R	Warren E B	35	30	4	1948
Carter J E	Smith R	21	18	4	1946

This subquery must be enclosed in parentheses. The query in which the subquery is embedded is called the main-query or outer-level query.

A subquery has basically the same format as the main query, with a few restrictions:

- A subquery may only have a single column name or expression in its SELECT clause.
- A subquery may not have an ORDER BY clause.
- The result of a subquery must be of a type compatible with the other operand of the comparison.

A subquery in SQL can be used anywhere that a constant is allowed. This means that the contents of one table can be used freely in selecting from or updating other tables.

10.1.2 Question

Show the name and age of the president who died the youngest.

```
SELECT    PRES_NAME, DEATH_AGE
FROM      PRESIDENT
WHERE     DEATH_AGE  =  (  SELECT  MIN ( DEATH_AGE )
                           FROM    PRESIDENT   )
```

(Compare with 'Example 2 Question' in the chapter about *aggregate*-functions)

Result: (Depending on the system used, there can be a warning message like: NULL IGNORED)

PRES_NAME	DEATH_AGE
Kennedy J F	46

If a subquery returns more than one value, we cannot use the ordinary comparison operators, as in the case of ordinary conditions. The next example introduces *comparison operators between a value and a set of values*.

10.1.3 Question

List the hobbies and names of all those presidents who served 8 years or longer. Order by hobbies and presidents name.

The ***subquery*** necessary in this case is:

```
SELECT    PRES_NAME
FROM      PRESIDENT
WHERE     YRS_SERV >= 8
```

The result of this *subquery* is the following set of president names:

PRES_NAME
Jefferson T
Madison J
Monroe J
Jackson A
Grant U S
Cleveland G
Wilson W
Roosevelt F D
Eisenhower D D

We now have to select the hobbies of these presidents, using PRES_NAME from the PRES_HOBBY table. *For this comparison we may use the IN (or = ANY) operator* within the main query:

```
SELECT      HOBBY , PRES_NAME
FROM        PRES_HOBBY
WHERE       PRES_NAME   IN  (  SELECT      PRES_NAME
                               FROM        PRESIDENT
                               WHERE       YRS_SERV >= 8   )
ORDER BY    HOBBY, PRES_NAME
```

Result:

HOBBY	PRES_NAME
Bridge	Eisenhower D D
Fishing	Cleveland G
Fishing	Eisenhower D D
Fishing	Jefferson T
Fishing	Roosevelt F D
Golf	Eisenhower D D
Golf	Wilson W
Hunting	Eisenhower D D
Painting	Eisenhower D D
Riding	Jackson A
Riding	Jefferson T
Riding	Wilson W
Sailing	Roosevelt F D
Swimming	Roosevelt F D
Walking	Wilson W

Another solution *using the ANY modifier* is as follows:

```
SELECT      HOBBY, PRES_NAME
FROM        PRES_HOBBY
WHERE       PRES_NAME  = ANY ( SELECT      PRES_NAME
                                FROM        PRESIDENT
                                WHERE       YRS_SERV  >=  8  )
ORDER BY    HOBBY, PRES_NAME
```

Still another solution, using a join instead of a subquery is:

```
SELECT      HOBBY, PRESIDENT.PRES_NAME
FROM        PRESIDENT , PRES_HOBBY
WHERE       PRESIDENT.PRES_NAME  =  PRES_HOBBY.PRES_NAME
      AND   YRS_SERV  >=  8
ORDER BY    HOBBY,  PRESIDENT.PRES_NAME
```

10.1.4 Question

Which presidents never won an election?

A president who never won an election is one who is recorded in the PRESIDENT table but is not in the list of election winners in the ELECTION table.

To solve this problem we first make a list of *all the winners of elections* using the following subquery:

```
SELECT      DISTINCT  CANDIDATE
FROM        ELECTION
WHERE       WINNER_LOSER_INDIC  = 'W'
```

The result of this subquery is a list of all election winners.

We must now compare these election winning presidents against the PRESIDENT table. We may use either the '.. NOT = ALL ..' or the NOT IN operators within the main query (they are actually the same operator).

```
SELECT    PRES_NAME
FROM      PRESIDENT
WHERE     PRES_NAME  NOT IN  ( SELECT    CANDIDATE
                              FROM      ELECTION
                              WHERE     WINNER_LOSER_INDIC = 'W' )
```

Result:

PRES_NAME
Tyler J
Fillmore M
Johnson A
Arthur C A
Ford G R

10.1.5 Question

List the hobbies of presidents who served for

(a) 12 years or more,
(b) 8 years or more.

The subqueries for cases (a) and (b) are as follows:

```
SELECT    PRES_NAME
FROM      PRESIDENT
WHERE     YRS_SERV > = 12     (a)
WHERE     YRS_SERV >  = 8     (b)
```

Result: (a)

PRES_NAME
Roosevelt F D

Result: (b)

PRES_NAME
Jefferson T
Madison J
Monroe J
Jackson A
Grant U S
Cleveland G
Wilson W
Roosevelt F D
Eisenhower D D

```
SELECT    HOBBY
FROM      PRES_HOBBY
WHERE     PRES_NAME   =  (   SELECT    PRES_NAME
                             FROM      PRESIDENT
                             WHERE     YRS_SERV > = 12   )
```

Result:

HOBBY
Fishing
Sailing
Swimming

In case (a), the subquery returns *one* value, that is Roosevelt F D. Thus, the main query will return the hobbies of President Roosevelt F D.
In case (b) however, the subquery returns *several* values (a set of values): more than one president served 8 years or longer. Thus the main query returns an error indication.

It is, however, quite meaningful to have queries with subqueries which return more than one value. In order to avoid an error indication, we have to use a set comparison operator. A set comparison operator is an ordinary comparison operator, qualified by ANY or ALL:

comparison-operator **{ ANY | ALL }**

e.g.: **> ANY , = ALL , ...**

The qualification by ANY means that the condition is true if at least one value in the set of values specified by the subquery fulfills the comparison.
The qualification by ALL means that the condition is true if all values in the set of values specified by the subquery fulfill the comparison.

Examples:

Roosevelt F D = ANY (Jefferson T, Madison J, Monroe J, Jackson A, Grant U S, Wilson W, Roosevelt F D, Eisenhower D D)
=> *TRUE*, since Roosevelt F D is contained in the set.

Note:
The operator = ANY can be written also as IN , and should be read *'is contained in'*.

Ford G R ^= ALL (Reagan R, Carter J E, Nixon R M, Johnson L B, Kennedy J F)
=> *TRUE*, since Ford G R is not contained in the set

Note:
The operator ^= ALL can be written also as NOT IN and should be read *'is not contained in'*.

A = ALL (A, B, C)
=> *FALSE*, since A is not equal to some elements of the set, in this case B or C.

Note:
Such an expression can never be true, unless the set contains only one element.

A ^= ANY (A, B, C)
=> *TRUE*, since A is not equal to some elements of the set, in this case B or C.

Note:
Such an expression is always true, unless the set contains only one element.

Observation: the here used operator '^=' (with meaning '*not equal to'*) is *not* supported by all systems and is mostly replaced by '< >' (without blank between '<' and '>') or by 'NOT … = …' or 'NOT … IN …' .

3 < ANY (1, 2, 3, 4)
=> *TRUE*, since there is one element in the set which is greater than 3, that is 4.

3 < ALL (1, 2, 3, 4)
=> *FALSE*, since there are elements in the set which are less than 3, that is 1, 2.

3 > ALL (1, 2)
=> *TRUE*, since all elements in the set are less than 3.

10.2 Extreme (maximum / minimum) values *(>= ALL , <= ALL , …)*

Another kind of a frequently occurring query is the following one:

10.2.1 Question

Which state provided the largest number of presidents, and what is that number?

```
SELECT      STATE_BORN , COUNT (*)
FROM        PRESIDENT
GROUP BY    STATE_BORN
HAVING      COUNT (*)  >= ALL  ( SELECT COUNT ( * )
                                  FROM  PRESIDENT
                                  GROUP BY STATE_BORN )
```

{Where the number is greater or equal to all numbers in the following list}

The qualification ALL means that the condition is true if ALL values in the set of values specified by the subquery satisfy the comparison. In this case only the maximum count satisfies all, hence that is the only group which is retained in the main query.

Result:

STATE_BORN	COUNT (*)
Virginia	8

Note:
If we are sure that the subquery is going to return *only one value*, we can use an ordinary comparison operator. However, if the subquery will then return more than one value, the main query will give an error indication.

10.2.2 Question

Find those states which entered the union before President Washington was inaugurated.

The subquery that finds in which year Washington was inaugurated is as follows:

```
SELECT      YEAR_INAUGURATED
FROM        ADMINISTRATION
WHERE       PRES_NAME  =  'Washington G'
```

We find that this query returns two values:

YEAR_INAUGURATED
1789
1793

Thus our query was not clear. Do we mean those states which entered the union before President Washington was inaugurated first (a), or last (b)?

The situation is clarified by using the set comparison operators < ALL or < ANY, respectively:

10.2.3 Question (a)

```
SELECT      STATE_NAME
FROM        STATE
WHERE       YEAR_ENTERED  < ALL (   SELECT  YEAR_INAUGURATED
                                    FROM    ADMINISTRATION
                                    WHERE   PRES_NAME = 'Washington G' )
```

Result:

STATE_NAME
Massachusetts
Pennsylvania
Virginia
Connecticut
South Carolina
Maryland
New Jersey
Georgia
New Hampshire
Delaware
New York
North Carolina
Rhode Island

10.2.4 Question (b)

```
SELECT    STATE_NAME
FROM      STATE
WHERE     YEAR_ENTERED < ANY (  SELECT  YEAR_INAUGURATED
                                FROM    ADMINISTRATION
                                WHERE   PRES_NAME = 'Washington G' )
```

Result:

STATE_NAME
Massachusetts
Pennsylvania
Virginia
Connecticut
South Carolina
Maryland
New Jersey
Georgia
New Hampshire
Delaware
New York
North Carolina
Rhode Island
Vermont
Kentucky

It is possible that a subquery may contain in its WHERE clause or HAVING clause another subquery, a sub-subquery, so to speak (a nesting of subqueries).

10.2.5 Question

List all the facts available in the table PRESIDENT about presidents who were inaugurated after Hawaii entered the union.

```
SELECT    *
FROM      PRESIDENT
WHERE     PRES_NAME  = ANY ( SELECT  PRES_NAME
                              FROM    ADMINISTRATION
                              WHERE  YEAR_INAUGURATED  >
                                  ( SELECT YEAR_ENTERED
                                    FROM       STATE
                                    WHERE  STATE_NAME = 'Hawaii' ) )
```

Note:
Since the sub-subquery (the second subquery) returns only one value, the comparison operator > does not need to be qualified with ANY or ALL.
The first subquery however may return several values, thus the comparison operator = needs to be qualified, in this case with ANY.

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia
Ford G R	1913	2	?	Republican	Nebraska
Johnson L B	1908	5	65	Democratic	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Nixon R M	1913	5	?	Republican	California
Reagan R	1911	3	?	Republican	Illinois

This question may also be expressed with a JOIN and without a subquery as follows:

```
SELECT    DISTINCT PRESIDENT.PRES_NAME , BIRTH_YR , YRS_SERV,
          DEATH_AGE, PARTY, STATE_BORN
FROM      PRESIDENT, ADMINISTRATION, STATE
WHERE     PRESIDENT.PRES_NAME = ADMINISTRATION.PRES_NAME
    AND   YEAR_INAUGURATED  > YEAR_ENTERED
    AND   STATE_NAME = 'Hawaii'
```

A subquery may contain a GROUP BY or HAVING clause. In this case, the operator used in connection with the subquery should be a set comparison operator.

10.2.6 Question

List the names and ages at death of those presidents w ho were married more than once. Order by name.

The *subquery* for finding out who was married more than once is:

```
SELECT    PRES_NAME
FROM      PRES_MARRIAGE
GROUP BY  PRES_NAME
HAVING    COUNT ( * )  > 1
```

The result of this subquery is:

PRES NAME
Fillmore M
Harrison B
Reagan R
Roosevelt T
Tyler J
Wilson W

The main query then is:

```
SELECT    PRES_NAME, DEATH_AGE
FROM      PRESIDENT
WHERE     PRES_NAME  IN  (  SELECT     PRES_NAME
                            FROM       PRES_MARRIAGE
                            GROUP BY   PRES_NAME
                            HAVING     COUNT (*) > 1 )
ORDER BY PRES NAME
```

Result:

PRES_NAME	DEATH_AGE
Fillmore M	74
Harrison B	67
Reagan R	?
Roosevelt T	60
Tyler J	71
Wilson W	67

Many times a subquery may also be used in cases where the JOIN could be used.

10.2.7 Question

Find those states which entered the union the same year as President Eisenhower was born. Order by state name.

With the JOIN we get the following SQL command:

```
SELECT      STATE_NAME
FROM        STATE, PRESIDENT
WHERE       YEAR_ENTERED  = BIRTH_YR
      AND   PRES_NAME  =  'Eisenhower D D'
ORDER BY    STATE_NAME
```

Result:

STATE_NAME
Idaho
Wyoming

This request can also be expressed in SQL with the subquery construct. To find the states which qualify, we have to apply a condition which we can only evaluate by investigating the table PRESIDENT. The subquery is:

```
SELECT      BIRTH_YR
FROM        PRESIDENT
WHERE       PRES_NAME = 'Eisenhower D D'
```

Result:

BIRTH_YR
1890

Since every president has exactly one birth year, this query will always return one value, in our example 1890.

This value will be used *by the main query* which is:

```
SELECT      STATE_NAME
FROM        STATE
WHERE       YEAR_ENTERED  =  ( SELECT  BIRTH_YR
                               FROM    PRESIDENT
                               WHERE  PRES_NAME = 'Eisenhower D D' )
ORDER BY    STATE_NAME
```

Result:

STATE_NAME
Idaho
Wyoming

Addition: **Remark II on 'NULL'-values**
An initially unexpected effect occurs again with 'NULL'-values. So the query:

```
SELECT   *
FROM     ADMINISTRATION
WHERE    ( ADMIN_NR + 30 )  IN  (  SELECT  DEATH_AGE
                                   FROM    PRESIDENT  )
```

results in 26 rows, whereas the at first sight complementary query:

```
SELECT   *
FROM     ADMINISTRATION
WHERE    ( ADMIN_NR + 30 )  NOT IN  (   SELECT  DEATH_AGE
                                        FROM    PRESIDENT  )
```

does not give any result row. The test if a value is not in some column that (partly) may have an 'unknown' value, does not lead to a 'true', but always to an 'unknown' and therefore no single row will be shown!

However: the different RDBMS's do not apply this above-mentioned reasoning in a correct way. Therefore you always must thoroughly test especially those queries that in some or another way use (or can use) NULL-values in their conditions.

11 Use of More Than One Copy of a Table

In SQL it is possible to refer to a table by more than one name, and then assume that there are as many copies of the tables as there are names. There are several ways of using this feature.
We will first illustrate the use of more than one table copy for the same table with an example that can also be solved without this feature.

11.1.1 Question

For each president who was born in a year in which at least one other president was born, list his name and birth year.

One way to express this in SQL is:

Question a

```
SELECT    PRES_NAME, BIRTH_YR
FROM      PRESIDENT
WHERE     BIRTH_YR  IN  ( SELECT    BIRTH_YR
                         FROM      PRESIDENT
                         GROUP BY  BIRTH_YR
                         HAVING    COUNT (*) >1 )

                                   (instead of  IN  one may use  = ANY )
```

Result a:

PRES_NAME	BIRTH_YR
Adams J Q	1767
Jackson A	1767
Grant U S	1822
Hayes R B	1822
Nixon R M	1913
Ford G R	1913

One may also solve this problem by having two tables, say T1 and T2, which are both copies of the table PRESIDENT.
New copies of a table are declared in the FROM clause by placing the new name after the old name (separated by a blank).
For example, to introduce two new names T1 and T2 for the table PRESIDENT, our FROM clause should state:

```
FROM  PRESIDENT  T1,  PRESIDENT  T2
```

With the availability of two PRESIDENT tables we can compare a birth year of a given row in one table with a birth year of a row in the second table.
We can now write:

Question b

```
SELECT    T1.PRES_NAME, T1.BIRTH_YR , T2.PRES_NAME , T2.BIRTH_YR
FROM      PRESIDENT  T1,  PRESIDENT  T2
WHERE     T1.BIRTH_YR = T2.BIRTH_YR
```

Result b:

PRES_NAME	BIRTH_YR	PRES_NAME	BIRTH_YR
Washington G	1732	Washington G	1732
Adams J	1735	Adams J	1735
Jefferson T	1743	Jefferson T	1743
Madison J	1751	Madison J	1751

Monroe J	1758	Monroe J	1758
Adams J Q	1767	Adams J Q	1767
Jackson A	1767	Adams J Q	1767
Adams J Q	1767	Jackson A	1767
Jackson A	1767	Jackson A	1767
Harrison W H	1773	Harrison W H	1773
Van Buren M	1782	Van Buren M	1782
Taylor Z	1784	Taylor Z	1784
Tyler J	1790	Tyler J	1790
Buchanan J	1791	Buchanan J	1791
Polk J K	1795	Polk J K	1795
Fillmore M	1800	Fillmore M	1800
Pierce F	1804	Pierce F	1804
Johnson A	1808	Johnson A	1808
Lincoln A	1809	Lincoln A	1809
Grant U S	1822	Grant U S	1822
Hayes R B	1822	Grant U S	1822
Grant U S	1822	Hayes R B	1822
Hayes R B	1822	Hayes R B	1822
Arthur C A	1830	Arthur C A	1830
Garfield J A	1831	Garfield J A	1831
Harrison B	1833	Harrison B	1833
Cleveland G	1837	Cleveland G	1837
McKinley W	1843	McKinley W	1843
Wilson W	1856	Wilson W	1856
Taft W H	1857	Taft W H	1857
Roosevelt T	1858	Roosevelt T	1858
Harding W G	1865	Harding W G	1865
Coolidge C	1872	Coolidge C	1872
Hoover H C	1874	Hoover H C	1874
Roosevelt F D	1882	Roosevelt F D	1882
Truman H S	1884	Truman H S	1884
Eisenhower D D	1890	Eisenhower D D	1890
Johnson L B	1908	Johnson L B	1908
Reagan R	1911	Reagan R	1911
Ford G R	1913	Ford G R	1913
Nixon R M	1913	Ford G R	1913
Ford G R	1913	Nixon R M	1913
Nixon R M	1913	Nixon R M	1913
Kennedy J F	1917	Kennedy J F	1917
Carter J E	1924	Carter J E	1924

If we compare the result with the one of 'question a' we see that there are quite a few more rows in the result in 'question b'. This is essentially redundant or irrelevant information, caused by the simple fact that each president was born in the same year as himself and the doubling of each pair of presidents that fulfill the condition.
To avoid this kind of redundancy and irrelevancy we use the following SQL query:

Question c
List presidents born on same year. List each president only once.

We can eliminate unwanted rows by making sure that the match is between different presidents, and that this is the first time that this match has been found.

```
SELECT    T1.PRES_NAME, Tl.BIRTH_YR , T2.PRES_NAME , T2.BIRTH_YR
FROM      PRESIDENT  T1,  PRESIDENT  T2
WHERE     T1.BIRTH_YR  =  T2.BIRTH_YR
     AND  T1.PRES_NAME  <  T2.PRES_NAME
```

Note:

The last condition in the WHERE clause makes sure firstly that a president is not matched with himself, and secondly that a pair of presidents satisfying the equality condition is listed only once.

Result:

PRES_NAME	BIRTH_YR	PRES_NAME	BIRTH_YR
Adams J Q	1767	Jackson A	1767
Grant U S	1822	Hayes R B	1822
Ford G R	1913	Nixon R M	1913

Another useful application of having more than one table copy is the following:

11.1.2 Question

Using the table ELECTION for all elections after 1900, create a new table with the columns ELECTION YEAR, WINNER, WINNER VOTES, LOSERS, LOSERS VOTES.
To solve this task, we create two table labels W (winners) and L (losers) for the table ELECTION. We use SQL to distribute the rows of the ELECTION table over two new tables so the ELECTION W contains the rows with WINNER_LOSER_INDIC = 'W', and ELECTION L the rows with WINNER_LOSER_INDIC = 'L'. These two new tables are joined as follows:

```
SELECT      W.ELECTION_YEAR, W.CANDIDATE, W.VOTES, L.CANDIDATE, L.VOTES
FROM        ELECTION  W,  ELECTION  L
WHERE       W.ELECTION_YEAR = L.ELECTION_YEAR
      AND   W.ELECTION_YEAR > 1900
      AND   W.WINNER_LOSER_INDIC =  'W'
      AND   L.WINNER_LOSER_INDIC  =  'L'
ORDER BY W.ELECTION_YEAR, L.VOTES  DESC
```

Result:

ELECTION_YEAR	CANDIDATE	VOTES	CANDIDATE	VOTES
1904	Roosevelt T	336	Parker A B	140
1908	Taft W H	321	Bryan W J	162
1912	Wilson W	435	Roosevelt T	88
1912	Wilson W	435	Taft W H	8
1916	Wilson W	277	Hughes C E	254
1920	Harding W G	404	Cox W W	127
1924	Coolidge C	382	Davis J W	136
1924	Coolidge C	382	La Follette R M	13
1928	Hoover H C	444	Smith A E	87
1932	Roosevelt F D	472	Hoover H C	59
1936	Roosevelt F D	523	Landon A M	8
1940	Roosevelt F D	449	Wilkie W L	82
1944	Roosevelt F D	432	Dewey T E	99
1948	Truman H S	303	Dewey T E	189
1948	Truman H S	303	Thurmond J S	39
1952	Eisenhower D D	442	Stevenson A	89
1956	Eisenhower D D	457	Stevenson A	73
1956	Eisenhower D D	457	Jones W B	1
1960	Kennedy J F	303	Nixon R M	219
1960	Kennedy J F	303	Byrd	15
1964	Johnson L B	486	Goldwater B	52
1968	Nixon R M	301	Humphrey H H	191
1968	Nixon R M	301	Wallace G C	46
1972	Nixon R M	520	McGovern G S	17
1972	Nixon R M	520	Hospers J	1
1976	Carter J E	297	Ford G R	240
1980	Reagan R	489	Carter J	49

To illustrate the use of table labels we will introduce another example often published in the literature on relational databases.

11.1.3 Question

Which employee earns more than his manager?

With this query it is assumed that the following template exists (ENR is employee number).

EMPLOYEE TABLE:

ENR	NAME	MGR	SALARY

Let us introduce a small sample database:

EMPLOYEE

	ENR	NAME	MGR	SALARY
1	303	JOHNSON	-	95000
2	176	EDWARDS	303	65000
3	142	FRASER	303	70000
4	267	DIXON	303	42000
5	107	FRASER	176	42000
6	101	KELLY	176	55000
7	157	BAXTER	176	62000
8	147	BROWN	142	33000
9	144	KENT	142	63000
10	219	GREEN	101	37000
11	221	CODD	101	60000
12	297	SMITH	101	30000
13	387	WHITE	101	85000

How would we solve this problem without a computer?
We would go to the first row, look up the employee number of the manager of the employee of row 1, and because employee 303 has no manager we are finished with this row.
Next we go to row 2, which contains three facts about employee 176. We look in the third column and find that the manager of employee 176 is employee 303. We now take the salary of 176 which is 65000 and compare this with the salary of employee 303 (which is 95000). What we are doing is essentially making comparisons between values in two different rows. The WHERE clause of SQL does not permit comparisons between different rows in the same table.

Let us now suppose we had two tables, which are identical, one called EMPLOYEE and the other called MANAGER. We can now proceed as follows:
We go to row 2 in the table employee, find the employee number of the manager of this employee (176) in column 3 which is 303. We now go to the MANAGER table and look up the salary of employee 303 and compare this with the salary of 176.
This procedure is repeated for all rows in the table EMPLOYEE.

The reader may have already concluded that this will require a join and a comparison in SQL. The SQL query is:

```
SELECT      EMPLOYEE.ENR
FROM        EMPLOYEE, MANAGER
WHERE       EMPLOYEE.MGR = MANAGER.ENR
      AND   EMPLOYEE.SALARY > MANAGER.SALARY
```

Since there is no MANAGER table, but rather all managers are employees that manage other employees, we would actually have the following select:

```
SELECT    EMPLOYEE.ENR
FROM      EMPLOYEE, EMPLOYEE MANAGER
WHERE     EMPLOYEE.ENR = MANAGER.ENR
    AND   EMPLOYEE.SALARY > MANAGER.SALARY
```

Note that we have actually given the EMPLOYEE table two names: EMPLOYEE and MANAGER. We could also choose to use table labels on both tables, as in the following query:

```
SELECT    E.ENR
FROM      EMPLOYEE E, EMPLOYEE M
WHERE     E.MGR = M.ENR
    AND   E.SALARY > M.SALARY
```

Result:

ENR
387
221

12 Correlated Subqueries

In the previous subquery cases, we were able to obtain the result of a subquery independently (without referring to anything outside the subquery) and in particular without referring to the main query. There are however, queries where we cannot obtain the result of the subquery without referring to rows in the main query. A subquery which has a search condition which relates to the main query is called a correlated subquery.

Let us first look at an example that can be expressed in several ways in SQL.

12.1.1 Question

List the name and birth year of those presidents who were inaugurated at least once within 45 years of their birth year.

We deal in this case with two tables, PRESIDENT and ADMINISTRATION. Without the aid of a computer we, would probably have to take the two tables, both sorted by president name, and retain from the PRESIDENT table only two columns, namely PRES_NAME and BIRTH_YR and from the ADMINISTRATION table the columns PRES_NAME and YEAR_INAUGURATED for the first inauguration of each president.
These two tables are presented in the next figures.

PRES_NAME	BIRTH_YR
Adams J	1735
Adams J Q	1767
Arthur C A	1830
Buchanan J	1791
Carter J E	1924
Cleveland G	1837
Coolidge C	1872
Eisenhower D D	1890
Fillmore M	1800
Ford G R	1913
Garfield J A	1831
Grant U S	1822
Harding W G	1865
Harrison B	1833
Harrison W H	1773
Hayes R B	1822
Hoover H C	1874
Jackson A	1767
Jefferson T	1743
Johnson A	1808
Johnson L B	1908
Kennedy J F	1917
Lincoln A	1809
Madison J	1751
McKinley W	1843
Monroe J	1758
Nixon R M	1913
Pierce F	1804
Polk J K	1795
Reagan R	1911
Roosevelt F D	1882
Roosevelt T	1858
Taft W H	1857
Taylor Z	1784
Truman H S	1884
Tyler J	1790
Van Buren M	1782
Washington G	1732
Wilson W	1856

PRES_NAME	MIN (YEAR_INAUGURATED)
Adams J	1797
Adams J Q	1825
Arthur C A	1881
Buchanan J	1857
Carter J E	1977
Cleveland G	1885
Coolidge C	1923
Eisenhower D D	1953
Fillmore M	1850
Ford G R	1974
Garfield J A	1881
Grant U S	1869
Harding W G	1921
Harrison B	1889
Harrison W H	1841
Hayes R B	1877
Hoover H C	1929
Jackson A	1829
Jefferson T	1801
Johnson A	1865
Johnson L B	1963
Kennedy J F	1961
Lincoln A	1861
Madison J	1809
McKinley W	1897
Monroe J	1817
Nixon R M	1969
Pierce F	1853
Polk J K	1845
Reagan R	1981
Roosevelt F D	1933
Roosevelt T	1901
Taft W H	1909
Taylor Z	1849
Truman H S	1945
Tyler J	1841
Van Buren M	1837
Washington G	1789
Wilson W	1913

We would then go to the first row in the PRESIDENT table, take the birth year, go to the ADMINISTRATION table and take the row which has the same president name, and check whether the birth year plus 45 years is greater than the first inauguration year of that president. If this was true, we would list the president name in the result of the query.
This process would be repeated for every row in the PRESIDENT table.

It is important to repeat that we are comparing for each specific president, the birth year with 45 added to it with the first inauguration year of that president.

In principle, the SQL system works as shown in this example: it selects the first row in the PRESIDENT table and then evaluates the subquery for those rows in the ADMINISTRATION table which have the same president name as the first row in the PRESIDENT table. Then this procedure is repeated for the next row in the PRESIDENT table and so on until all rows of the PRESIDENT table have been dealt with.

The entire correlated query is:
(a)

```
SELECT    PRES_NAME,  BIRTH_YR
FROM      PRESIDENT
WHERE     BIRTH_YR + 45 >  (  SELECT  MIN ( YEAR_INAUGURATED )
                              FROM    ADMINISTRATION
                              WHERE   ADMINISTRATION.PRES_NAME =
                                      PRESIDENT.PRES_NAME )
```

The main aspect of the query can be explained as follows:
List the president name from the PRESIDENT table under the condition that the birth year + 45 is greater than the minimum year of inauguration ***of the president under consideration***.

Note:
The prefix ADMINISTRATION in the last line of the given subquery is *not* required, because there is a rule which specifies that with the prefix omitted, SQL assumes the table name(s) in the FROM clause *of the innermost subquery*.
The last line of the subquery restricts the results of the subquery to those rows of the ADMINISTRAT1ON table *where the president name is the same as in the current row of the PRESIDENT table under consideration in the main query*.

Result:

PRES_NAME	BIRTH_YR
Kennedy J F	1917
Roosevelt T	1858

PRESIDENT in the last line of this subquery may be referred to as a *correlation variable*.
We may prefer to stress this correlation variable aspect by creating appropriate table labels, say P in our example, and then writing the query as follows:

(b)

```
SELECT    DISTINCT PRES_NAME, BIRTH_YR
FROM      PRESIDENT  P
WHERE     BIRTH_YR + 45  >  (   SELECT  MIN ( YEAR_INAUGURATED )
                                FROM    ADMINISTRATION
                                WHERE   PRES_NAME =  P.PRES_NAME )
```

The result of Question (b) is the same as for Question (a)

The same question can be formulated with a JOIN as follows:
(c)

```
SELECT      DISTINCT PRESIDENT.PRES_NAME,  BIRTH_YR
FROM        PRESIDENT, ADMINISTRATION
WHERE       PRESIDENT.PRES_NAME = ADMINISTRATION.PRES_NAME
      AND   BIRTH_YR + 45 > YEAR_INAUGURATED
```

In this formulation SQL first forms the product of the tables PRESIDENT and ADMINISTRATION and then retains only those rows of the combined table which satisfy two conditions:

a. It must be about the same president
b. Year inaugurated is less than birth year plus 45.

Note that all three formulations gave the same result.
The use of the table label in the 'query b' was not required but permitted. However there are cases where we need to introduce additional table labels.
An example of a query where one needs to use table labels as well as a correlated subquery is the following:

12.1.2 Question

List the election year and winner of those elections in which the winner received more than 80% of the votes in that election.

How would we solve this problem without a computer?

For a given election, we take the winner and the number of votes he polled. We then compute the sum of the votes of all candidates for that election and compare whether the winner's votes are greater than 80% of the total.
Using a correlated subquery we get the following SQL formulation:

```
SELECT      CANDIDATE, ELECTION_YEAR
FROM        ELECTION  E
WHERE       WINNER_LOSER_INDIC = 'W'
      AND   VOTES  >  ( SELECT  0.8 * SUM (VOTES)
                        FROM    ELECTION
                        WHERE  ELECTION.ELECTION_YEAR = E.ELECTION_YEAR)
```

The SUM function operates on all rows in the ELECTION table which have the same election year as the election year of the row under consideration in the main query (E.ELECTION_YEAR). In this case we need the correlation variable or table label E in order to distinguish between rows of the table ELECTION in the main query and rows of the same table in the subquery.
Result:

CANDIDATE	ELECTION_YEAR
Jefferson T	1804
Monroe J	1816
Monroe J	1820
Pierce F	1852
Lincoln A	1864
Grant U S	1872
Wilson W	1912
Hoover H C	1928
Roosevelt F D	1932
Roosevelt F D	1936
Roosevelt F D	1940
Roosevelt F D	1944
Eisenhower D D	1952
Eisenhower D D	1956
Johnson L B	1964
Nixon R M	1972

Reagan R	1980

12.1.3 Question

Select the president's name, birth year and the year of inauguration of his first administration, and order the result in sequence of year of inauguration.

```
SELECT      PRESIDENT.PRES_NAME, BIRTH_YR, YEAR_INAUGURATED
FROM        PRESIDENT,  ADMINISTRATION
WHERE       PRESIDENT.PRES_NAME  = ADMINISTRATION.PRES_NAME
      AND   YEAR_INAUGURATED =  ( SELECT  MIN (YEAR_INAUGURATED)
                                  FROM    ADMINISTRATION
                                  WHERE  ADMINISTRATION.PRES_NAME =
                                                      PRESIDENT.PRES_NAME )
ORDER BY    YEAR_INAUGURATED
```

Result:

PRES_NAME	BIRTH_YR	YEAR_INAUGURATED
Washington G	1732	1789
Adams J	1735	1797
Jefferson T	1743	1801
Madison J	1751	1809
Monroe J	1758	1817
Adams J Q	1767	1825
Jackson A	1767	1829
Van Buren M	1782	1837
Harrison W H	1773	1841
Tyler J	1790	1841
Polk J K	1795	1845
Taylor Z	1784	1849
Fillmore M	1800	1850
Pierce F	1804	1853
Buchanan J	1791	1857
Lincoln A	1809	1861
Johnson A	1808	1865
Grant U S	1822	1869
Hayes R B	1822	1877
Arthur C A	1830	1881
Garfield J A	1831	1881
Cleveland G	1837	1885
Harrison B	1833	1889
McKinley W	1843	1897
Roosevelt T	1858	1901
Taft W H	1857	1909
Wilson W	1856	1913
Harding W G	1865	1921
Coolidge C	1872	1923
Hoover H C	1874	1929
Roosevelt F D	1882	1933
Truman H S	1884	1945
Eisenhower D D	1890	1953
Kennedy J F	1917	1961
Johnson L B	1908	1963
Nixon R M	1913	1969
Ford G R	1913	1974
Carter J E	1924	1977
Reagan R	1911	1981

Still another frequently occurring kind of query is exemplified in the following example:

12.1.4 Question

For those presidents whose number of marriages equals the number of administrations they served as president, list their name and this number.
Without SQL we could perform this as follows:
From the table ADMINISTRATION we would generate a table containing the president's name and the number of administrations he served as president. The select we could use to do the same work is as follows:

```
SELECT      PRES_NAME,  COUNT (*)
FROM        ADMINISTRATION
GROUP BY    PRES_NAME
```

Result (see table at the left).

From the table PRES_MARRIAGE we would generate a table containing the president's name and the number of marriages.

```
SELECT      PRES_NAME,  COUNT ( * )
FROM        PRES_MARRIAGE
GROUP BY    PRES_NAME
```

Result (see table at the right):

…FROM ADMINISTRATION:

PRES_NAME	COUNT (*)
Adams J	1
Adams J Q	1
Arthur C A	1
Buchanan J	1
Carter J E	1
Cleveland G	2
Coolidge C	2
Eisenhower D D	2
Fillmore M	1
Ford G R	1
Garfield J A	1
Grant U S	2
Harding W G	1
Harrison B	1
Harrison W H	1
Hayes R B	1
Hoover H C	1
Jackson A	2
Jefferson T	2
Johnson A	1
Johnson L B	2
Kennedy J F	1
Lincoln A	2
Madison J	2
McKinley W	2
Monroe J	2
Nixon R M	2
Pierce F	1
Polk J K	1
Reagan R	1
Roosevelt F D	4
Roosevelt T	2
Taft W H	1
Taylor Z	1
Truman H S	2
Tyler J	1
Van Buren M	1
Washington G	2
Wilson W	2

… FROM PRES_MARRIAGE:

PRES_NAME	COUNT (*)
Adams J	1
Adams J Q	1
Arthur C A	1
Carter J E	1
Cleveland G	1
Coolidge C	1
Eisenhower D D	1
Fillmore M	2
Ford G R	1
Garfield J A	1
Grant U S	1
Harding W G	1
Harrison B	2
Harrison W H	1
Hayes R B	1
Hoover H C	1
Jackson A	1
Jefferson T	1
Johnson A	1
Johnson L B	1
Kennedy J F	1
Lincoln A	1
Madison J	1
McKinley W	1
Monroe J	1
Nixon R M	1
Pierce F	1
Polk J K	1
Reagan R	2
Roosevelt F D	1
Roosevelt T	2
Taft W H	1
Taylor Z	1
Truman H S	1
Tyler J	2
Van Buren M	1
Washington G	1
Wilson W	2

We would then take the one row from each table with the same president's name, and compare the two counts seeking an equal value.

The entire query in SQL is:

```
SELECT      PRES_NAME, COUNT (*)
FROM        ADMINISTRATION
GROUP BY    PRES_NAME
HAVING      COUNT ( * ) =  (    SELECT  COUNT ( * )
                                FROM    PRES_MARRIAGE
                                WHERE   PRES_MARRIAGE.PRES_NAME =
                                        ADMINISTRATION.PRES_NAME )
```

Beware of the constructions that are used to connect on the one side *the same number* and on the other side *the same president name*.

Note: in the preceding result tables [shown side by side] in the right-hand table 'somewhere' (at the place where in the left-hand table the name 'Buchanan' appears) we inserted ("manually") a blank row to show matching president names at the same vertical level.

Some people add to the subquery:

```
GROUP BY    PRES_NAME
```

This is *redundant* because of the fact that the subquery will automatically select those groups from the PRES_MARRIAGE table having the same PRES_NAME as the current group in the main query. That is, the COUNT function of the subquery operates not on the whole table (PRES_MARRIAGE), but only on those rows of the table fulfilling the condition in the WHERE clause, that is on the corresponding groups defined by the GROUP BY clause of the main query.

Result:

PRES_NAME	COUNT (*)
Adams J	1
Adams J Q	1
Arthur C A	1
Carter J E	1
Ford G R	1
Garfield J A	1
Harding W G	1
Harrison W H	1
Hayes R B	1
Hoover H C	1
Johnson A	1
Kennedy J F	1
Pierce F	1
Polk J K	1
Roosevelt T	2
Taft W H	1
Taylor Z	1
Van Buren M	1
Wilson W	2

Please note that reversing the tables as in the following query results in the same answer.

```
SELECT      PRES_NAME, COUNT (*)
FROM        PRES_MARRIAGE
GROUP BY    PRES_NAME
HAVING      COUNT ( * ) = ( SELECT      COUNT (* )
                            FROM        ADMINISTRATION
                            WHERE       ADMINISTRATION.PRES_NAME =
                               PRES_MARRIAGE.PRES_NAME )
```

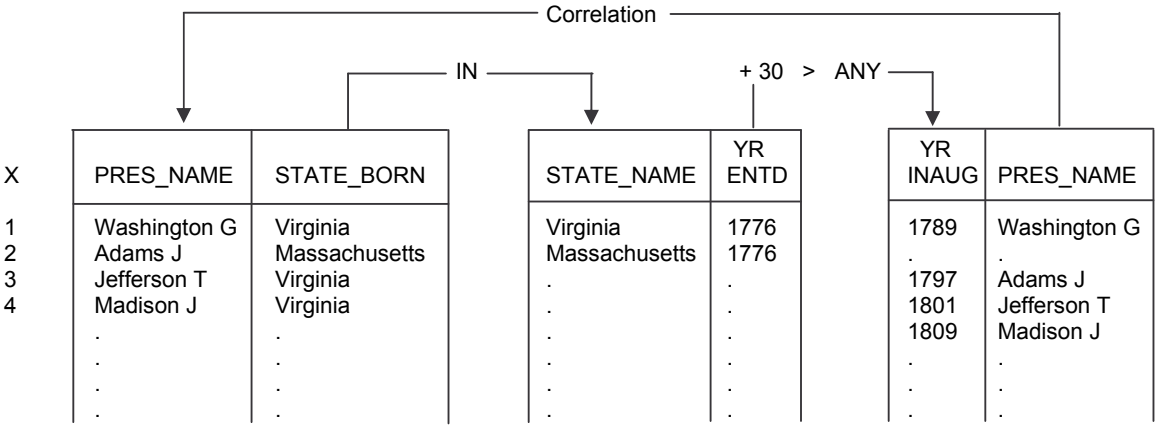
It is not necessary for the correlation to be between a main query and its direct subquery .
For example, there may be a correlation between a main query and one of its sub-subqueries.

12.1.5 Question

Find the names of those presidents born in a state which entered the union not more than 30 years earlier than their first inauguration year.

```
SELECT  PRES_NAME
FROM    PRESIDENT X
WHERE  STATE_BORN   IN   ( SELECT STATE_NAME
                           FROM STATE
                           WHERE YEAR_ENTERED + 30  >=  ANY
                                    (SELECT MIN ( YEAR_INAUGURATED )
                                    FROM     ADMINISTRATION
                                    WHERE  PRES_NAME = X.PRES_NAME ) )
```

We look at excerpts of the tables PRESIDENT, STATE, and ADMINISTRATION involved in this query:



X	PRES_NAME	STATE_BORN
1	Washington G	Virginia
2	Adams J	Massachusetts
3	Jefferson T	Virginia
4	Madison J	Virginia
	.	.
	.	.
	.	.
	.	.

STATE_NAME	YR ENTD
Virginia	1776
Massachusetts	1776
.	.
.	.
.	.
.	.
.	.

YR INAUG	PRES_NAME
1789	Washington G
.	.
1797	Adams J
1801	Jefferson T
1809	Madison J
.	.
.	.
.	.

The condition is true for X = 1, X = 2, X = 3, and not true for X = 4 (and the rest of the table PRESIDENT).

Thus, the result of the query is:

PRES_NAME
Washington G
Adams J
Jefferson T

12.1.6 Addition: correlation in case of *an assimilated query* in the SELECT-clause

Imagine, we want a survey from all presidents who died younger than 60 years and that we want a ranking where the youngest deceased president gets ranking number 1, the president who died a little older gets number 2 etcetera. Presidents with the same death-age get the same ranking number.
So:

NR	PRES_NAME	DEATH_AGE
1	Kennedy J F	46
2	Garfield J A	49
3	Polk J K	53
4	Lincoln A	56
4	Arthur C A	56
6	Harding W G	57
7	McKinley W	58

(There are 2 presidents with the same ranking '4', as both presidents died at the same age.)
How can we produce such a survey?
It must be clear that our ranking number is some derived value. If you realize that the ranking number of a president is de number (+ 1) of presidents who died younger than that president.
We can produce that extra ranking number by assimilating a complete (*correlated*) query in the SELECT-clause of the main query (see also paragraph 6.2 of this reader):

```
SELECT    (SELECT  COUNT(*)+1  FROM  president  WHERE death_age < P1.death_age)
          AS Nr,  pres_name, death_age
FROM      president  P1
WHERE     death_age < 60
ORDER BY 1
```

12.1.7 Addition: the *Join / GROUP BY* -problem of *N:M-relaties* in MS Access

In the chapter on '*Joins'* we discussed in the last section the Join/GROUP BY- difficulty of N:M-relations.
If we work with a RDMS-SQL-system (like ***MS Access***) where the use of a COUNT (DISTINCT ..) – construction is *not* possible, we will have to correct the results of the COUNT-'s in a much more laborious way, by dividing them by the number of values 'in the other table'.
So that can be (very laboriously...) as follows:

```
SELECT      M.Pres_Name, COUNT( Spouse_name) / (SELECT COUNT ( * )
                  FROM Pres_Hobby WHERE Pres_name = H.Pres_name ) ,
                  COUNT(Hobby) / ( SELECT COUNT ( * ) FROM Pres_Marriage
                  WHERE Pres_name = M.Pres_name )
FROM        Pres_marriage M,  Pres_hobby H
WHERE       M.Pres_name = H.Pres_name
GROUP BY    M.Pres_name
HAVING      (COUNT(Hobby) /  ( SELECT COUNT(*) FROM Pres_Marriage
                                    WHERE Pres_name = M.Pres_name ) ) >3
ORDER BY 3 DESC
```

And of course the result must be the same as earlier acquired through the COUNT-DISTINCT-variant:

PRES_NAME		
Roosevelt T	2	7
Eisenhower D D	1	5
Coolidge C	1	5

Take care on the necessary '*correlated query'*-constructions!

13 Test *for Existence* on Subqueries *(WHERE [NOT] Exists ...)*

Instead of using a comparison operator with a subquery, we may also use the EXISTS or NOT EXISTS operator within a WHERE clause.

. . . WHERE [NOT] **EXISTS** (subquery) . . .

If the EXISTS operator is used, the condition in the WHERE clause is satisfied if the subquery results in at least one row. If the NOT EXISTS operator is used, the condition is satisfied if the subquery returns no rows.

13.1.1 Question

List the names and ages at death of all presidents who were married.

The main query is on the table PRESIDENT, while the subquery is on the table PRES_MARRIAGE. However, it is not necessary in this case to investigate specific columns of this table. All that is needed is to check whether a value appearing in the column PRES_NAME of the table PRESIDENT does appear in the column PRES_NAME of the table PRES_MARRIAGE. This can be formulated using the EXISTS operator. It is also necessary in this case to correlate the column PRES_NAME in the table PRES_MARRIAGE with the column PRES_NAME in the table PRESIDENT.

```
SELECT    PRES_NAME, DEATH_AGE
FROM      PRESIDENT
WHERE     EXISTS  (   SELECT  *
                      FROM    PRES_MARRIAGE
                      WHERE   PRES_MARRIAGE.PRES_NAME =
                                        PRESIDENT.PRES_NAME )
```

Result:

PRES_NAME	DEATH_AGE
Washington G	67
Adams J	90
Jefferson T	83
Madison J	85
Monroe J	73
Adams J Q	80
Jackson A	78
Van Buren M	79
Harrison W H	68
Tyler J	71
Polk J K	53
Taylor Z	65
Fillmore M	74
Pierce F	64
Lincoln A	56
Johnson A	66
Grant U S	63
Hayes R B	70
Garfield J A	49
Arthur C A	56
Cleveland G	71
Harrison B	67
McKinley W	58
Roosevelt T	60
Taft W H	72
Wilson W	67
Harding W G	57

Coolidge C	60
Hoover H C	90
Roosevelt F D	63
Truman H S	88
Eisenhower D D	79
Kennedy J F	46
Johnson L B	65
Nixon R M	?
Ford G R	?
Carter J E	?
Reagan R	?

13.1.2 Question

List the names and ages at death of all presidents who were not married.

```
SELECT      PRES_NAME , DEATH_AGE
FROM        PRESIDENT   P
WHERE  NOT  EXISTS  (   SELECT  *
                        FROM    PRES_MARRIAGE   M
                        WHERE   M.PRES_NAME = P.PRES_NAME )
```

Note:
Please note the use of shorter table labels to save typing time.

Result:

PRES_NAME	DEATH_AGE
Buchanan J	77

13.1.3 Question

List the election year and name of the winning candidates who never became president.

```
SELECT      ELECTION_YEAR,  CANDIDATE
FROM        ELECTION   E
WHERE       WINNER_LOSER_INDIC = 'W'
      AND  NOT  EXISTS  (   SELECT  *
                            FROM    PRESIDENT
                            WHERE   PRES_NAME = E.CANDIDATE )
```

Result:

Apparently *none* of the winners of elections has *never* become a president. Depending on the used system there will be shown either nothing or a warning message that no rows satisfied the given conditions.

14 The UNION-operator *(... UNION [ALL] ...)*

It happens many times, that we want the results of two or more queries combined in a single survey. To generate such combinations SQL offers a possibility with the UNION-operator. An imposed condition for the UNION-operator is, that only *results with the same datatypes* can be combined.

14.1.1 Question

Show all the presidents, which had at most one child (the total number from all their marriages).

Note: we explicitly also want to see the presidents who never got married and thus (officially) had no children at all.

A query as:

```
SELECT      PRES_NAME
FROM        PRES_MARRIAGE
GROUP BY    PRES_NAME
HAVING      SUM ( NR_CHILDREN ) <= 1
```

only originates the names of presidents, which married officially (and got at most one child). By using the UNION-operator we also can display the names of the presidents, which never married:

```
SELECT      PRES_NAME
FROM        PRES_MARRIAGE
GROUP BY    PRES_NAME
HAVING  SUM ( NR_CHILDREN ) <= 1
UNION
SELECT      PRES_NAME
FROM        PRESIDENT
WHERE       PRES_NAME  NOT IN ( SELECT  PRES_NAME FROM PRES_MARRIAGE)
ORDER BY    PRES_NAME
```

As the result of this expansion also the name of president ‘Buchanan J’, who never married, will appear in the final result table:

PRES_NAME
Buchanan J
Harding W G
Jackson A
Madison J
Polk J K
Truman H S
Washington G

14.1.2 Question

Show all the Democratic presidents. In case such a president married, show also the name(s) of their spouse(s) and the number of children that arose out of each marriage.

Note: to combine through the UNION-operator the names of the presidents, which never married, with the names of presidents + names of spouse(s) + the number of children, we have to bear in mind that the UNION-operator only can combine similar results (of the same data type). Therefore we will have to append an additional string behind the name of an unmarried president; also to combine with the column ‘NR_CHILDREN’ we will have to add either the number ‘0’, or the typeless ‘NULL’.

Therefore the required query becomes:

```
SELECT      P.PRES_NAME, SPOUSE_NAME, NR_CHILDREN
FROM        PRESIDENT  P,  PRES_MARRIAGE  M
WHERE       P.PRES_NAME = M.PRES_NAME  AND  PARTY = 'Democratic'
UNION
SELECT      PRES_NAME,  'has never married!',  NULL
FROM        PRESIDENT
WHERE       PRES_NAME NOT IN  ( SELECT  PRES_NAME FROM PRES_MARRIAGE)
      AND   PARTY = 'Democratic'
```

With the result:

PRES_NAME	SPOUSE_NAME	NR_CHILDREN
Buchanan J	has never married!	
Carter J E	Smith R	4
Cleveland G	Folson F	5
Jackson A	Robards R D	0
Johnson A	McCardle E	5
Johnson L B	Taylor C A	2
Kennedy J F	Bouvier J L	3
Pierce F	Appleton J M	3
Polk J K	Childress S	0
Roosevelt F D	Roosevelt A E	6
Truman H S	Wallace E V	1
Van Buren M	Hoes H	4
Wilson W	Axson E L	3
Wilson W	Galt E B	0

Of course a concluding ORDER BY …-instruction will impose a required order in the result.

By default, the result of a UNION-construction does ***not*** contain any *duplicate* rows. If the option '*ALL*' is used, however, then duplicate rows (if existing…) will all be shown.

15 Creating and Manipulating Table Definitions *(Create, Alter+Add, Drop)*

The first stage in setting up an SQL database is *to define the tables*. Tables can only be created by users with DBA or RESOURCE authority (DBA: *DataBase Administrator*).

15.1 CREATE TABLE

The syntax of the CREATE TABLE command is:

```
CREATE TABLE table-name
(column-name-l data-type-l      [ NOT NULL ]
[ , column-name-2 data-type-2 [ NOT NULL ] ]...)
[ IN dbspace-name ]
```

table-name, column-name:

consist of up to 18 characters which can be upper or lower case letters, numbers, $, #, @ or underscore. The first character cannot be a number.

data-type:

may be any of the following:

1. SMALLINT	A whole number (integer) between –32,767 and +32,767.
2. INTEGER	A whole number (integer) between -2,147,483,647 and +2,147,483,647.
3. DECIMAL (m, n)	A decimal number consisting of m digits (maximum 18), an optional decimal marker located to the left of n digits and an optional sign (+ or -). In other words, a total of m digits, n of those after the decimal point and m-n before it, possibly with a numeric sign.
4. FLOAT	A floating point number. For example: 1, 1.5, 1.23E45, 1.23E-45. The maximum floating point value depends on the actual computer hardware being used.
5. CHAR (n)	A character string of fixed length (n) where n is at least 1 and at most 254.
6. VARCHAR (n)	A variable length character string of maximum length (n) where n is at least 1 and at most 254. SQL will only store the actual length.
7. LONG VARCHAR	A variable length character string of up to 32,767 characters.

Note: **N**owadays most RDBMS have additional data-types like DATE, TIME, and BINARY etc.

NOT NULL

specifies that the column cannot contain any null fields. In other words, each field of the column must always contain a value.

dbspace-name

is the name of the database file in which the table is to be placed. If the IN dbspace-name clause is omitted, the table will be placed according to the following rules:

- If you own one or more private dbspaces, the table will be created in the private dbspace you acquired first.
- If you do not own any private dbspaces and you have DBA authority, the table will be created in the main dbspace. This is the dbspace that is created when the database is initially created. Its name is the same as that of the database.

15.1.1 Example

Create a table which can hold all the facts about presidents as presented in the table PRESIDENT:

```
CREATE TABLE PRESIDENT
        ( PRES_NAME  CHAR (15)       NOT NULL,
          BIRTH_YR      SMALLINT     NOT NULL,
          YRS_SERV      SMALLINT     NOT NULL,
          DEATH_AGE     SMALLINT ,
          PARTY         CHAR (12)    NOT NULL,
          STATE_BORN VARCHAR (14) NOT NULL )
```

15.2 DROP TABLE

The syntax of the DROP TABLE command is as follows:

```
DROP  TABLE  table-name
```

When a table is dropped, the table definition, contents, indexes and views on the table, as well as all permissions to the table, are also dropped.

15.3 ALTER TABLE

A table definition can be modified *to include new columns*. The ALTER TABLE command appends a new column to the right hand side of an existing table. A column can be added irrespective of whether the table already contains rows of information, or is empty.
The syntax of the ALTER TABLE command is:

```
ALTER TABLE  table-name  ADD  column-name data-type
```

The arguments table-name, column-name, and data-type are the same as for the CREATE TABLE command.

15.3.1 Example

Add a new column to the table STATE for the population of the state at the time of entry to the union of states.

```
ALTER  TABLE  STATE  ADD   POP_ENTERED  SMALLINT
```

To see the effect of the ALTER TABLE command, suppose the query to display all states beginning with 'M' is issued.

```
SELECT      *
FROM        STATE
WHERE       STATE_NAME  LIKE  'M%'
```

Result:

STATE_NAME	ADMIN_ENTERED	YEAR_ENTERED	POP_ENTERED
Massachusetts	?	1776	?
Maryland	?	1776	?
Mississippi	8	1817	?
Maine	8	1820	?
Missouri	9	1831	?
Michigan	12	1837	?
Minnesota	18	1858	?
Montana	26	1889	?

Note:

- A column cannot be inserted amongst columns that have already been defined for a table. It can only be appended to the right hand side of the table.
- A column that is added to a table cannot be specified as NOT NULL, since there is no way of specifying a value for the column for each of the rows that might already exist in the table.

15.4 CREATE INDEX

Creating indexes for your tables is an extremely important part of setting up a database. Indexes can dramatically improve the performance of SQL queries, and also provide automatic checking for unwanted duplication of data.
Indexes provide a quick means of finding rows in a table. All SQL commands which have a WHERE clause use indexes if possible. This means that indexes can improve the performance of the SELECT, UPDATE and DELETE commands.

The use of indexes by SQL is invisible to the user. SQL knows which indexes are available and automatically determines the most efficient combination of indexes to answer your query. This is called *automatic optimization*.
Indexes can be created or dropped at any time during a table's lifetime. If a new index has been created to speed up the evaluation of a query, that index is used automatically in later processing of that query Furthermore, if an index is dropped, queries that previously used the index will use an alternative method to obtain the necessary result. In other words, routines, stored queries and commands entered interactively are all independent of the existence of indexes.

The syntax of the CREATE INDEX command is:

```
CREATE [ UNIQUE ] INDEX index-name ON [creator.]table-name
( column-name-1 [ ASC | DESC ] [,column-name-2 [ ASC | DESC ] ...)
```

UNIQUE

specifies that each field value (or combination of field values) for the columns specified in the CREATE INDEX command may appear only once in the table.

Note: Within a unique index, NULL is considered to be a value. This means that NULL can occur only once in a column which has a unique index defined on it.

ASC | DESC

specifies the ordering of the rows used for the internal implementation of the index.
ASC indexes are sometimes more efficient.

A table can have many indexes, and each index can reference up to 16 columns in the table.
An index can only apply to a single table.

Two examples of indexes are as follows:

```
CREATE INDEX STATE_NAME_INDEX ON STATE ( STATE_NAME )

CREATE INDEX STATE_ADMIN_INDEX ON STATE ( STATE_NAME, ADMIN_ENTERED)
```

We can represent these two indexes diagrammatically as follows:

```
STATE_NAME   |  ADMIN_ENTERED  |  YEAR_ENTERED
< ========== >                                  STATE_NAME_INDEX
< ============================== >              STATE_ADMIN_INDEX
```

15.5 Use of indexes for performance enhancement

To make the best use of indexes, you need to understand when they are used. The following SELECT command uses the index STATE_NAME_INDEX to produce the SELECT result of:

```
SELECT      *
FROM        STATE
WHERE       STATE_NAME = 'Texas'
```

Instead of having to scan the entire STATE table looking for rows where the state is Texas, SQL can use the index to go directly to the correct row.

The following WHERE clauses would also use STATE_NAME_INDEX:

```
WHERE STATE_NAME  > 'Idaho'
WHERE STATE_NAME  < 'Utah'
WHERE STATE_NAME  IN  ('Kansas', Oregon', 'Nevada')
WHERE STATE_NAME  BETWEEN  'Idaho'  AND  'Utah'
```

The LIKE operator is the only operator *that never uses an index*. For example, the following WHERE clause would *not* use STATE_NAME_INDEX:

```
WHERE STATE_NAME  LIKE  'M%'
```

However, in this case the restriction can be overcome by using the *BETWEEN* operator. The following WHERE clause could be used instead of the one containing the LIKE clause:

```
WHERE STATE_NAME  BETWEEN  'M'  AND  'MZ'
```

Multi-column indexes like STATE_ADMIN_INDEX are used to go directly to the rows which satisfy the WHERE clause if all columns covered by the index are used in the WHERE clause, and the comparisons are connected by AND.

For example, the following WHERE conditions use STATE_ADMIN_INDEX to find the correct rows:

```
WHERE STATE_NAME  >  'Idaho'  AND  ADMIN_ENTERED < 10

WHERE STATE_NAME  BETWEEN  'Idaho' AND 'Utah'
                              AND ADMIN_ENTERED  BETWEEN 10 AND 20
```

The rows which satisfy the WHERE clauses given above are obtained directly using the index STATE_ADMIN_INDEX. SQL does not have to scan any rows to determine whether they satisfy the conditions.
The exception is when one or more of the columns covered by the index is compared to a set of values. In other words, if IN, =ANY, =ALL or another set operation has been used, a multi-column index is used for one column only.

The following WHERE-condition does *not* use STATE_ADMIN_INDEX:

```
WHERE  STATE_NAME  IN  ( 'Idaho', 'Utah' )
                        AND  ADMIN_ENTERED  BETWEEN  10  AND  20
```

The WHERE clause above would use STATE_NAME_INDEX to help find all the state names in the specified set, and scan these for the ADMIN_ENTERED condition.
Multi-column indexes can also be used to answer queries if the WHERE clause refers to only the first column in the index, and there are no other indexes on this column which could be used to answer the query more quickly. This is less efficient than using an index which is defined only on the required column, but is usually more efficient than no index at all. If there is an index defined on the single column needed to answer the query, it will be used in preference to a multi-column index.

In our example, STATE_NAME_INDEX would be used to answer the queries on STATE_NAME, except when ADMIN_ENTERED is used with an AND, when STATE_ADMIN_INDEX is preferred. This is called *major key processing*. On the other hand, STATE_ADMIN_INDEX cannot be used to answer queries which only refer to ADMIN_ENTERED in their WHERE clause. Major key processing only works for the first columns in the index.

Indexes are aids solely to improve the performance of SQL in retrieving table rows, and are not necessary for accessing stored data. If no appropriate indexes can be found on a table when processing a particular SELECT command, the entire table will be scanned to find the rows which satisfy the conditions. This is much slower than using an index if the result is only a small proportion of the rows in a large table.

Indexes speed up retrieval of data but *slow down* operations involving inserting new data or updating existing data because of the overhead in maintaining the index. For example, when a new row is inserted into a table, indexes for that table have to be updated to include the new row.

You should think carefully about the kinds of queries that you expect to be asked most often, and define indexes which help to answer these queries.

15.6 Use of indexes as a *data integrity check*

In addition to speeding up evaluation of your queries, *unique* indexes provide a valuable means of checking the integrity of information entered in a table. SQL will not accept duplicates in columns on which a *unique* index is defined. For instance, SQL will not allow you to add an additional row to the STATE table with the name Texas if the unique index STATE_NAME_INDEX exists. SQL also prevents a row in the table from being changed so that there are duplicates in the columns which are covered by a unique index. This means that you do not have to worry about unwanted and possibly disastrous duplicate information being entered.

15.7 DROP INDEX

The syntax for the DROP INDEX command is: DROP INDEX index-name

16 Updating the database (*Insert, Update+Set, Delete*)

There are three methods of update available:

A) Add one or more new rows to a table within the database
B) Modify one or more existing rows of a table within the database
C) Delete one or more existing rows from a table within the database

16.1 Adding rows to a table: the *INSERT* command

The INSERT command has two formats:

Format 1 adds a single new row to a table.
Format 2 adds one or more existing rows (which are selected or computed from other tables) into a table.

The two formats are described below.

Format 1

```
INSERT [ INTO ] [ creator.] { table-name | view-name } [ (column-names) ]
VALUES ( data-items)
```

Data-items is a list of one or more values, separated by commas, to be inserted into the new row for the columns specified in column-names. The first data-item value is inserted in the first column-name specified; the second data-item value is inserted in the second column-name specified, etc. A null value is inserted into a field by typing the word NULL for the data-item value. The data items must be compatible in type to the matching columns.
Column names from the table that were not specified in the column-name list are given a null value; therefore, you must remember to include in the list all the NOT NULL columns in the table.

If no column names are specified, then a column list in the order of the table template is assumed.

Format 2

```
INSERT INTO [ creator.] { table-name | view-name } [ ( column-names) ]
select-statement
```

The select statement selects the data to be inserted from another table. The number of columns retrieved by the select statement must match the number of columns for the insert.
The columns selected must be type-convertible with the columns into which they will be inserted (numeric to numeric, character to character).

Notes:

1. Creator is not necessary for tables created by the current user.
2. Single quotes are required around character data values. A quote is represented in a string by two consecutive quotes.
3. The table specified in the FROM clause of the Select statement must not be the same as the table into which information is being inserted. You cannot use the INSERT command to duplicate rows or parts of rows in a single table.

Example:
To add a new row to the VERY_RECENT_PRES table (described in a previous chapter):

```
INSERT INTO VERY_RECENT_PRES
VALUES ( 'Nextpres I M', 1940, 2, NULL, 'Teaparty', 'Allstates' )
```

If we now enter the query:

```
SELECT      *
FROM        VERY_RECENT_PRES
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia
Ford G R	1913	2	?	Republican	Nebraska
Johnson L B	1908	5	65	Democratic	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Nixon R M	1913	5	?	Republican	California
Reagan R	1911	3	?	Republican	Illinois
Nextpres I M	1940	2	?	Teaparty	Allstates

Note that NULL has been entered to indicate the president has no death age.

Example 2:

```
INSERT INTO VERY_RECENT_PRES  ( PRES_NAME, BIRTH_YR )
        SELECT      CANDIDATE, ELECTION_YEAR
        FROM        ELECTION
        WHERE       WINNER_LOSER_INDIC = 'W'
                AND ELECTION_YEAR > 1990
```

After this command has been processed, SQL can display a message like:

```
ARI5001 SQL PROCESSING SUCCESSFUL.
SQLCODE = 0  ROWCOUNT = 2.
```

The table VERY_RECENT_PRES will now contain:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia
Ford G R	1913	2	?	Republican	Nebraska
Johnson LB	1908	5	65	Democratic	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Nixon R M	1913	5	?	Republican	California
Reagan R	1911	3	?	Republican	Illinois
Nextpres I M	1940	2	?	Teaparty	Allstates
Honour I	1995	?	?	?	?
Camelot M	1999	?	?	?	?

16.2 Updating rows of a table: the *UPDATE/SET* .. -command

The syntax of the UPDATE command is:

```
UPDATE    [creator.] table-name
SET       column-name-1 = expression-1
        [ , column-name-2 = expression-2 ]   . . .
[ WHERE   search-condition ]
```

column-name-1

is the name of the first column to be updated.

expression-1

is the new value to be placed in column-name-1. This expression may be any SQL expression, including a subquery. It may contain constants, NULL, column names and the arithmetic operators + , -, * , /.

column-name-2
is the name of the second column to be updated.

expression-2
is the new value to be placed in column-name-2.

search-condition
specifies the rows to be updated (same power as in the SELECT command).

Note:
All rows that satisfy the specified condition are updated; if no search condition is given, *all rows* in the named table are updated.

Example

```
UPDATE    VERY_RECENT_PRES
SET       STATE_BORN  = 'Texas'
WHERE     STATE_BORN  IS  NULL
```

Result (shown after submitting a Select command):

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Kennedy J F	1917	2	46	Democratic	Massachusetts
Johnson LB	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois
Nextpres I M	1940	2	?	Teaparty	Allstates
Honour I	1995	?	?	?	Texas
Camelot M	1999	?	?	?	Texas

The values in another table can be used to update the rows of a particular table. To do this, a subquery is used to set new column values.

Example

Set the party of all presidents whose party is unknown to the party which was in power when the state in which the president was born entered the union.

```
UPDATE  RECENT_PRESIDENTS
SET PARTY = ( SELECT  PARTY
              FROM    PRESIDENT
              WHERE  PRES_NAME =
                    ( SELECT  MIN ( PRES_NAME )
                      FROM    ADMINISTRATION
                      WHERE  ADMIN_NR =
                            ( SELECT  ADMIN_ENTERED
                              FROM    STATE
                              WHERE  STATE_NAME =
                                      RECENT_PRESIDENTS.STATE_BORN ) ) )
WHERE PARTY  IS NULL
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Kennedy J F	1917	2	46	Democratic	Massachusetts
Johnson LB	1908	5	65	Democratic	Texas

Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois
Nextpres I M	1940	2	?	Teaparty	Allstates
Honour I	1995	?	?	Democratic	Texas
Camelot M	1999	?	?	Democratic	Texas

Notes

- There must be only one column or expression in the subquery SELECT list. If you want to set several columns in a table to the values occurring in other tables, you will need a subquery for each column.
- The subquery must return a single value or none. If it returns no value, NULL is taken.
- The subquery is usually a correlated subquery. There is almost always a WHERE clause in the subquery. This is because you need to specify which rows from the source table are to be used to update which rows in the target table.
- If you only wish to update some rows of the target table, there should be a WHERE clause on the UPDATE command. In the example above, we only wished to update the rows for which we did not already know the party. The rows for which the party was already known were skipped by using the WHERE clause of the UPDATE command.

16.3 Deleting rows from a table: the *DELETE* -command

The DELETE command deletes all the rows of a named table which satisfy a specified search condition.
The syntax of the DELETE command is:

```
DELETE  [ FROM ]  [ creator.]  table-name
[ WHERE  search-condition ]
```

search-condition
specifies the row or rows to be deleted (same power as in the SELECT command).

Example:
In the previous examples for the INSERT command (employing Format 1), the president Nextpres I M was added to the VERY_RECENT_PRES table.
In example 2 for the INSERT command (employing Format 2), the two fictitious presidents 'Honour I' and 'Camelot M' were added to the VERY_RECENT_PRES table.
To delete the rows that were inserted in the above examples, the following DELETE statement could be used:

```
DELETE  FROM   VERY_RECENT_PRES
        WHERE  BIRTH_YR >= 1940
```

Note

INSERT and UPDATE *should not be confused* with ALTER TABLE (see previous chapter) which adds one *new column* on the right hand side of an existing table.

17 Views *(definitions for 'virtual tables')*

The VIEW feature of SQL provides the user with the ability to treat the result of a SELECT statement as if it were a real table, with some exceptions which are discussed later.
The major functions of a view are:

1. To provide a user with a mechanism to break down long and complex queries into more manageable pieces.
2. To provide a user with a table which contains only the information of interest, in the form required. For example, a view can be used to look upon a combination of several joined tables as one big table.
3. To provide the data administrator with a mechanism, in combination with the authorisation scheme, to be able to control access to a table or tables.

It is essential to know that a view is a virtual table. In other words a view does not have a real, permanent table population associated with it, but is recalculated each time that it is referenced by an SQL statement.

17.1 CREATE VIEW

A view is defined by a SELECT command.
The syntax is:

```
CREATE VIEW   view-name  [ (column-name-list) ]      [3]
AS            select-statement
```

As an example, to be able to query the table presented as 'RECENT_PRESIDENTS', with only 9 rows of the larger table PRESIDENT, we can define the following view:

```
CREATE VIEW  RECENT_PRESIDENTS  AS
                    SELECT   *
                    FROM     PRESIDENT
                    WHERE    BIRTH_YR > 1880
```

Views can be used to define new views as in the following example:

```
CREATE VIEW  VERY_RECENT_PRES  AS
                    SELECT  *
                    FROM    RECENT_PRESIDENTS
                    WHERE   BIRTH_YR > 1900
```

These views can be used as though they were real tables, as shown in the following SELECT command:

---

[3] Note: indicating the 'column-name-list' is:

- required**:** if one or more values from the SELECT-line are '*derived'*
  for example:
  ```
  CREATE VIEW  NUMBER_MARRIAGES ( PRES_NAME, NUMBER)
       AS      SELECT  PRES_NAME, COUNT (*)
               FROM    PRES_MARRIAGE  GROUP BY  PRES_NAME
  ```

- optional**:** if in the select-line just appear column names of an existing table and you those names just will be copied '*unmodified'*
  for example:
  ```
  CREATE VIEW  Recent_Presidents
       AS  SELECT * FROM PRESIDENT WHERE  BIRTH_YR > 1880
  ```

```
SELECT      *
FROM        VERY_RECENT_PRESIDENTS
ORDER BY    PRES_NAME
```

Result:

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Carter J E	1924	4	?	Democratic	Georgia
Ford G R	1913	2	?	Republican	Nebraska
Johnson L B	1908	5	65	Democratic	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Ni.xon R M	1913	5	?	Republican	California
Reagan R	1911	3	?	Republican	Illinois

Since a view is a virtual table, changes which do not apply directly to the table on which the view is defined cannot be processed. It is meaningless to update calculated quantities. Only real rows and columns can be updated.
Views can be treated as if they were real tables, with the following general exception: [4]

> *Operations which change existing rows or add new rows such as UPDATE, INSERT or DELETE, can only be used on views which have only one table in the FROM clause of the SELECT statement used to define the view.*

17.2 Addition: use of views to emulate the COUNT (DISTINCT …) - construct

In a RDBMS -as MSAccess- without the ***COUNT(DISTINCT …)***-possibility we can use a view definition to emulate that possibility.
Imagine that we want to know the number of [different] states where presidents are born.
The 'normal' SQL-query to find an answer to this question would be:

```
SELECT      COUNT ( DISTINCT state_born )
FROM        president
```

With result: 18

In systems like MSAccess, we can find an answer by first creating a view with only the distinct state_born names:

```
CREATE VIEW  DISTINCT_STATEBORN  AS
             SELECT      DISTINCT state_born
             FROM        president
```

followed by the query:

```
SELECT      COUNT ( * )
FROM        DISTINCT_STATEBORN
```

which of course will give us the same result: 18

Watch out: ***in MSAccess*** a view is not defined through a 'CREATE VIEW …'-command, but by just typing in the SELECT-query and saving that query with the choosen view-name (so saving it as '*distinct_stateborn*').

[4] Some RDBMS can have additional restrictions on the use of view definitions.

If in one query we not only want to ask for this number of different native states, but for instance also preceded by the number of presidents, then in a 'normal' RDBMS the query would be:

```
SELECT    COUNT(*), COUNT ( DISTINCT state_born )
FROM      president
```

With result: 39 18

In MSAccess we would have to query:

```
SELECT    COUNT( * ), (SELECT COUNT(*) FROM DISTINCT_STATEBORN)
FROM      president
```

with of course the same result.

17.3 View-definition with the *With Check Option*

We already remarked that in specific circumstances (supposing that a view is directly based on the data of just one table and thus for instance does not deliver '*derived'* values), a view can be used to change data in the underlying table through UPDATE-, INSERT- or DELETE-commands.

By means of the *With Check Option* in the CREATE VIEW-definition it will be possible to enforce restrictions on such changes, for example to prevent that (yes or no by accident) impossible values will end up in a database. By adding the *With Check Option* only those update-commands will be accepted, which, if executed, will cause an updated record that still is contained within the view-definition.
If for example you want to prevent that somebody can enter '*impossible values'* for birth_year (<=1700) or death_ages (<= 28), then you can agree (or even -see later- enforce through access rights), that users only can make changes on the PRESIDENT-table through the next view:

```
CREATE  VIEW   PRES_VIEW   AS
               SELECT       *
               FROM         PRESIDENT
               WHERE        BIRTH_YR >= 1700   AND   DEATH_AGE >= 28
               WITH CHECK OPTION
```

By using this *WITH CHECK OPTION* an unintentionally change through an UPDATE-command on this PRES_VIEW-view of a death_age of -93 (negative) for a president, should end in a situation where the involved record no longer would be shown through this view; so the *WITH CHECK OPTION* now will cause that such an update-attempt will generate an error message and that the negative death_age will not end up in the database.

Of course for such a database-protection we have to take care that nobody still can approach the underlying table to carry out *updates*. *Queries* better -i.e. *faster*- can act directly on the base tables.

18 Controlling the Execution of Commands *(Commit / Rollback Work)*

A command can be prevented from executing immediately, it can be started, cancelled or ended. There are a number of commands to be used for these and other purposes.

18.1 AUTOCOMMIT modes

Two modes of executing SQL commands exist:

- Mode **AUTOCOMMIT ON**:
  Every command that is entered commits the previously entered command. This means that every command is a transaction, and that we also have the option of entering ROLLBACK WORK to undo the last command.
- Mode **AUTOCOMMIT OFF:**
  Modifications of tables specified in a single SQL command, or in a series of consecutive SQL commands, are committed only upon encountering a COMMIT WORK command. This mode allows us to determine a logical unit of work, consisting, in general, of a series of SQL commands, which are executed as one transaction and are delimited by the command COMMIT WORK and ROLLBACK WORK (or CANCEL).

This mode can be specified using the SET AUTOCOMMIT command:

```
SET AUTOCommit { ON | OFF }
```

When the system is started, (generally) AUTOCOMMIT is ON by default.
If we detect an error in a command or logical unit of work, we may use the SQL commands ROLLBACK WORK or CANCEL to undo the current logical unit of work.

18.2 The ROLLBACK WORK command

ROLLBACK WORK ends a logical unit of work. If AUTOCOMMIT is OFF, all tables updated in the current logical unit of work will be restored to their appearance before the last COMMIT WORK command, or, if there was no such command, since the last AUTOCOMMIT SET OFF command. If AUTOCOMMIT is ON, and if we enter a ROLLBACK WORK immediately following an INSERT, DELETE or UPDATE command this table will be restored to its state prior to that command.

19 Granting and Revoking User Privileges

User privileges can be granted and revoked using the GRANT and the REVOKE commands respectively.

19.1 GRANT

GRANT is an SQL command which allows a user holding specific privilege(s), to grant them to one or more other users.
There are the following two formats for the GRANT command:

Format 1 For granting privileges on tables and views.
Format 2 For granting SPECIAL privileges.

Format 1 of the GRANT command is used to grant privileges on a table or view to other users. When using format 1 of the GRANT command, we must not only possess the privilege we want to grant to another user, but also the GRANT OPTION as well; the GRANT OPTION may or may not be passed along with the privilege.

The syntax of the GRANT command is as follows:

Format 1 (For granting privileges on tables and views)

```
GRANT  { list-of-privileges  |  ALL }
ON       [ creator.]  { table-name  |  view-name }
TO      { list-of-users   |  PUBLIC }
[ WITH GRANT OPTION ]
```

list-of-privileges

is one or more of the following privileges:

Privilege	Tables	Views
SELECT	X	X
INSERT	X	X
DELETE	X	X
UPDATE [(column-name-list)]	X	X
INDEX	X	
ALTER	X	

Notice that if UPDATE is to be granted, the option (column-name-list) can be used to grant privileges only on certain columns of the specified table or view. If multiple columns are specified, we separate them with commas. If column-name-list is not specified, the privilege(s) is granted for all columns.

ALL

specifies that we want to grant all privileges that we have on the specified table. ALL is *not* applicable to views.

creator

is the userid of the owner of the table or view on which the privilege{s) are to be granted, followed by a period. It is not necessary for our own tables or views. To grant privileges on another user's table, we must have been granted the privilege with the GRANT OPTION.

table-name | view-name

is the name or the table or view on which we want to grant privilege(s).

list-of-users

is a list of userids, separated by commas, of users to whom we want to grant privileges.

PUBLIC
specifies that we want to grant the privilege(s) specified to all users.

WITH GRANT OPTION
specifies that we want to allow the user(s) specified to be able to grant these privileges to someone else.

Example:
The following command gives the update privilege on the NR_CHILDREN column of the PRES_MARRIAGE table and the select privilege on the entire PRES_MARRIAGE table to user Reagan. It also allows Reagan to pass these privileges on to others.

```
GRANT UPDATE ( NR_CHILDREN ), SELECT
ON    PRES_MARRIAGE
TO    REAGAN
WITH GRANT OPTION
```

Format 2 of the GRANT command is used to grant special privileges to other users. In order to use format 2 of the GRANT command, you must have DBA Authority.

The special privileges which may be granted are:

DBA Authority
This is the highest level or authorisation provided, and implies CONNECT, RESOURCE and SCHEDULE authorities. A user with DBA authority can perform any operation on any table in the database.

CONNECT Authority
CONNECT authority is required to connect to SQL. Granting CONNECT authority in effect creates a new userid.

RESOURCE Authority
RESOURCE authority is required to acquire new database files (DBSPACES) and create tables in public DBSPACES.

SCHEDULE Authority
SCHEDULE authority allows connection to other userids without specifying a password.

The *syntax* of the format 2 of the GRANT command is as follows:
Format 2 (For granting special privileges)

```
GRANT {    CONNECT  TO  list-of-users  [ IDENTIFIED BY pswrd-list ]      |
           DBA   TO  list-of-users [ IDENTIFIED BY pswrd-list]           |
           SCHEDULE  TO  list-of-users [ IDENTIFIED BY pswrd-list ]      |
           RESOURCE TO  list-of-users   }
```

list-of-users
is a list of userids separated by commas. It identifies those users who are to receive the privilege(s).

pswrd-list

is a password for each userid listed in list-of-users (separated by commas). For granting DBA or schedule authority, these passwords may be omitted if they have previously been established for all of the users listed.

If DBA Authority is passed to a user, that user also receives connect, resource and schedule authority regardless of whether they are specified on the GRANT command. Furthermore, granting anyone of these privileges to a user without connect authority has the effect that this user automatically receives connect authority.

19.2 REVOKE

If we want to revoke a privilege from a user, we have to use a REVOKE command. A general rule is that a user can only revoke those privileges from another user, which he previously granted to that user. However, any privilege revoked from a user is also revoked from anyone to whom that user may have granted it.
To remove the GRANT OPTION granted by a previous GRANT command, the privilege itself must first be revoked, and then the same privilege without the GRANT OPTION must be re-granted.
The formats of the REVOKE command are very similar to the GRANT command and the parameters have the same meaning:

Format 1 (For revoking privileges on tables and views)

```
REVOKE  { list-of-privileges  |  ALL }
ON      [ creator.] { table-name  |  view-name }
FROM    { list-of-users  |  PUBLIC }
```

Format 2 (For revoking special privileges)

```
REVOKE  {  DBA          |
           CONNECT      |
           RESOURCE     |
           SCHEDULE  }   FROM  list-of-users
```

Format 2 of the REVOKE command can be applied only by a user with DBA authority.
Using this format, such a user can revoke any special privilege from any user, regardless of who originally granted this special privilege. However, resource or schedule authority cannot be revoked from a user with DBA authority. A user with DBA authority cannot revoke any privileges from himself, therefore ensuring that there is always one user with DBA authority for the database.

20 New possibilities with SQL-2 *(SQL 1992)*

Up until now we only considered possibilities of the (original) standard version of ***SQL*** ('SQL1'), which many times is also indicated as SQL/86. In 1989 this SQL/86 was extended to include an (optional) '*Integrity* Enhancement Feature', IEF, resulting in what was called SQL/89 (still indicated as SQL1).

A greatly expanded version ***SQL2*** was defined in 1992: SQL/92.
The most important aspects from SQL2 in comparison with SQL1 are that there exist more data-types, more join-types and more integrity/constraint-maintaining possibilities…
In 1999 the ***SQL3*** -standard was defined (with regular expression matching, recursive queries, triggers, non-scalar types and some object-oriented features) and nowadays people are working on a ***SQL4*** definition.

But: be aware that many current RDBMS do not fully comply with the SQL2-rules (or even not with all SQL1-rules; e.g. *COUNT (DISTINCT...)*).
We'll limit ourselves to a discussion of certain important aspects of the SQL2-standard.

20.1 New join-types in SQL2 *(especially the outer join)*

First of all: all studied query-constructs (like joins) from SQL1 are also valid and usable in SQL2!
But SQL2 gives us some new possibilities for joins.

SQL2 uses special syntax for various kinds of joins:

- cross join
- *qualified join:* *conditional join* ← **on** clause
  column-list join ← **using** clause
- natural join
- union join

Qualified and natural joins may be further classified into the following types:

- inner ← this is the default
- *outer:* *left*, *right*, *full*

Join type	***New syntax in SQL/92 (SQL2)***	***SQL1 syntax***
Cross	**select** * **from** A **cross join** B	**select** * **from** A, B
conditional	**select** * **from** A **join** B **on** *condition*	**select** * **from** A, B **where** *condition*
column-list	**select** * **from** A **join** B **using** (c1, …) - - c1, … are unqualified	**select** **A.c1, … , …** - - omit B.c1, … **from** A, B **where** A.c1 = B.c1 **and** … - - join columns are qualified

natural inner	**select *** **from** A **natural [inner] join** B - - join column in result is c	**select A.c, … , …** - - omit B.c, … **from** A, B **where** A.c = B.c - - join columns in result is A.c
left outer Also: right [..] join full [...] join	**select *** **from** A **left [outer] join** B **on** *condition* - nulls generated for nonmatches .. - eventually 'natural …'	**select A.c, …** { omit B.c } **from** A, B **where** A.c = B.c **union all** **select** c, …, **'?'**, .. **from** A **where** c **not in** (**select** c **from** B) - for composite c use exists with correlated subquery..

Some observations to the new SQL2 join types

- Natural joins: the RDBMS analyses if there are common column-names and uses the common names automatically as join-criterion-columns. If there are no common columns at all, a **A NATURAL JOIN B** degenerates to a **A CROSS JOIN B** .
- Most actual RDBMS's (including SQL Server) cannot handle *natural* joins nor the '*column-list*'-join (*using*...) ...
- A very interesting new join type is the so called [***left|right|full] outer join***.

20.1.1 Outer joins: show the row[part] of the (Left|Right|Full)-table, also if there is no connected row in the joined table

Lets consider the following examples:

```
SELECT    P.pres_name, death_age, hobby
FROM      president P  LEFT OUTER JOIN  pres_hobby H
      ON  P.pres_name = H.pres_name
WHERE     death_age > 83
```

Result:

PRES_NAME	*DEATH_AGE*	*HOBBY*
Adams J	90	(null)
Madison J	85	(null)
Hoover H C	90	Fishing
Hoover H C	90	Medicine Ball
Truman H S	88	Fishing
Truman H S	88	Poker
Truman H S	88	Walking

From the result table we may conclude, that a ***left outer join*** shows all allowed (by the WHERE-clause) rows ***from the left table*** with connected rows from the right table; if there is no connected row from the right table then the left row-data are still shown (with 'null' or blanks at the places reserved for data from the right table).
From the shown result table we can conclude that the presidents Adams J and Madison J did not have any hobby registered.

With a ***right outer join***, the situation is comparable, albeit that all rows from the table at the right will be shown, if necessary combined with nulls or blanks.

```
SELECT    P.pres_name, state_born, state_name, year_entered
FROM      president P  RIGHT OUTER JOIN  state S
     ON   state_born = state_name
WHERE     year_entered  BETWEEN  1845  AND  1850
```

Result:

PRES_NAME	*STATE_BORN*	*STATE_NAME*	*YEAR_ENTERED*
(null)	(null)	Florida	1845
Eisenhower D D	Texas	Texas	1845
Johnson L B	Texas	Texas	1845
Hoover H C	Iowa	Iowa	1846
(null)	(null)	Wisconsin	1848
Nixon R M	California	California	1850

All states [allowed by the WHERE-clause] from the *right* [state] table are shown and if possible, combined with corresponding row-data from the *left* [president] table. We see that neither in Florida nor in Wisconsin was born any president.

20.1.2 Outer joins: offer a possibility to emulate a 'NOT IN …'-subquery

In SQL1 it was not possible to replace a 'NOT IN …'-subquery by a join.
In SQL2, however, the outer join-construct offers that possibility.
E.g. consider the next query and its result:

```
SELECT    P.pres_name, state_born, spouse_name
FROM      president P  LEFT OUTER JOIN  pres_marriage M
     ON   P.pres_name = M.pres_name
WHERE     spouse_name  IS NULL
```

Result:

PRES_NAME	*STATE_BORN*	*SPOUSE_NAME*
Buchanan J	Pennsylvania	(null)

We conclude that to get an answer to the question: "Which presidents never married (show also the name of their native state)?" can be answered in the next two ways:

By a (*NOT IN..)* subquery:

```
SELECT    pres_name, state_born
FROM      president
WHERE     pres_name NOT IN  (SELECT pres_name FROM pres_marriage)
```

Or by the next *outer join*:

```
SELECT    P.pres_name, state_born
FROM      president P  LEFT OUTER JOIN  pres_marriage M
     ON   P.pres_name = M.pres_name
WHERE     spouse_name IS NULL
```

Of course, both queries give the same result:

PRES_NAME	*STATE_BORN*
Buchanan J	Pennsylvania

20.2 SQL2: new DDL-possibilities to enforce data-integrity

At [almost] any cost the content of a database must be kept integer. We may not allow having conflicting data in our database.
SQL2 offers some nice new possibilities to enforce data-integrity.
For instance:

- ***primary key*** (col-list)
- ***foreign keys*** (col-list) ***references*** *tablename* (unique-collist)
- ***check*** (table-condition-on-same-row)

As an example we consider some tables from the presidential database with their SQL2-DDL:

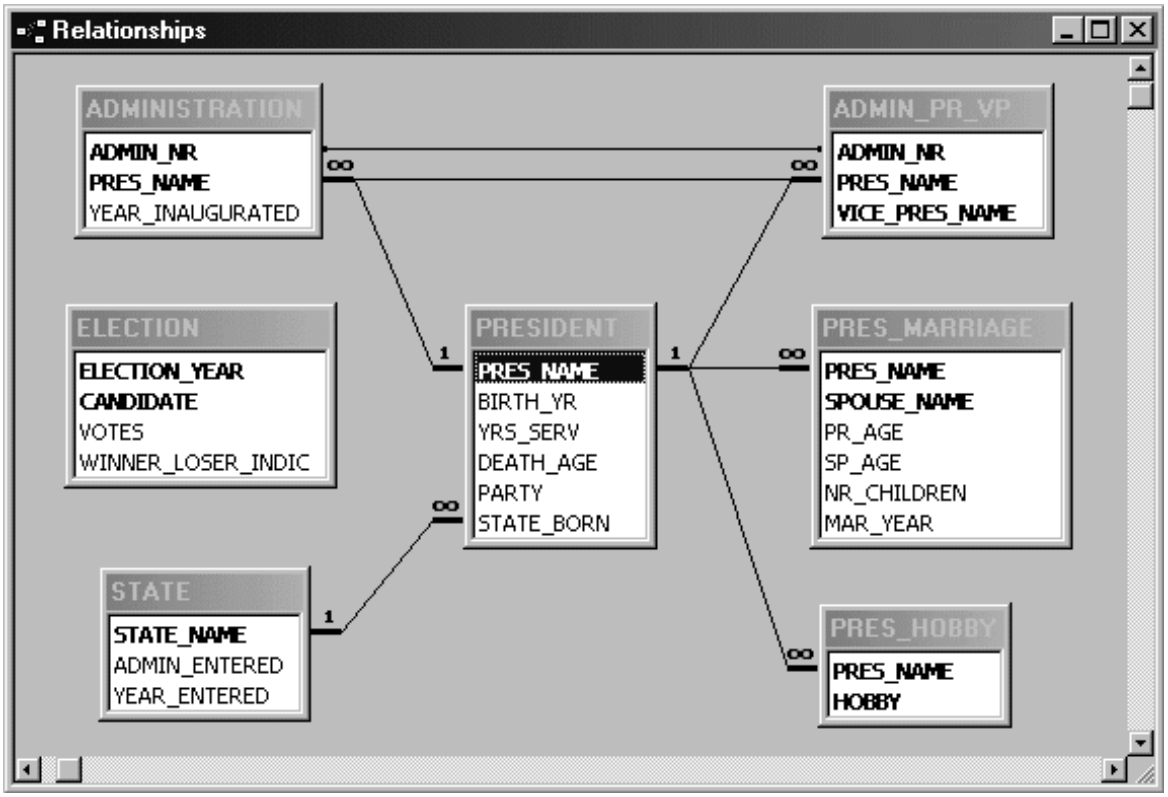
```
CREATE TABLE PRESIDENT
(
        PRES_NAME   CHAR(16)      NOT NULL,
        BIRTH_YR    SMALLINT      NOT NULL,
        YRS_SERV    SMALLINT      NOT NULL      CHECK BETWEEN 0 AND 12 ,
        DEATH_AGE   SMALLINT,
        PARTY       CHAR(11)      NOT NULL,
        STATE_BORN  CHAR(15)      NOT NULL,
        PRIMARY KEY (PRES_NAME)
)

CREATE TABLE PRES_HOBBY
(
        PRES_NAME   CHAR(16)      NOT NULL,
        HOBBY       CHAR(20)      NOT NULL,
        PRIMARY KEY (PRES_NAME, HOBBY)
)

ALTER TABLE PRES_HOBBY
        ADD FOREIGN KEY (PRES_NAME) REFERENCES PRESIDENT (PRES_NAME)
```

A ***primary key*** can be considered as an improved ‘unique index’; every table should have a primary key and its attributes must be [implicit or not] NOT NULL.
A ***foreign key*** points to a primary key in another table and means that a values in the attribute-fields of the table are only allowed if those values also [already] exist in the referenced primary key of that other table.
The ***check***-possibility enforces that new data or updates are only allowed if the new values obey to the conditions as imposed by the check-clause.

In a separate document we’ll discuss these (and other new) SQL2-possibilities for data-integrity enforcement in more detail.

21 Appendix A: Presidential database table relationships

We will show hereafter a 'relationships'-survey of the *presidential database*, with all the base tables in it. For each table is indicated (by **bold** column-names) which column or columns are the *primary key*-columns.
Through lines is sketched (as good as possible) the coherency between the tables (for instance a PRES_NAME from the PRES_MARRIAGE-table must also occur in the PRES_NAME-column of the PRES_MARRIAGE-table), so the *foreign keys*-columns.
Unfortunately we cannot conclude from this survey which columns are *optional*.



22 Appendix B : The presidential database

22.1 PRESIDENT

PRES_NAME	BIRTH_YR	YRS_SERV	DEATH_AGE	PARTY	STATE_BORN
Washington G	1732	7	67	Federalist	Virginia
Adams J	1735	4	90	Federalist	Massachusetts
Jefferson T	1743	8	83	Demo-Rep	Virginia
Madison J	1751	8	85	Demo-Rep	Virginia
Monroe J	1758	8	73	Demo-Rep	Virginia
Adams J Q	1767	4	80	Demo-Rep	Massachusetts
Jackson A	1767	8	78	Democratic	South Carolina
Van Buren M	1782	4	79	Democratic	New York
Harrison W H	1773	0	68	Whig	Virginia
Tyler J	1790	3	71	Whig	Virginia
Polk J K	1795	4	53	Democratic	North Carolina
Taylor Z	1784	1	65	Whig	Virginia
Fillmore M	1800	2	74	Whig	New York
Pierce F	1804	4	64	Democratic	New Hampshire
Buchanan J	1791	4	77	Democratic	Pennsylvania
Lincoln A	1809	4	56	Republican	Kentucky
Johnson A	1808	3	66	Democratic	North Carolina
Grant U S	1822	8	63	Republican	Ohio
Hayes R B	1822	4	70	Republican	Ohio
Garfield J A	1831	0	49	Republican	Ohio
Arthur C A	1830	3	56	Republican	Vermont
Cleveland G	1837	8	71	Democratic	New Jersey
Harrison B	1833	4	67	Republican	Ohio
McKinley W	1843	4	58	Republican	Ohio
Roosevelt T	1858	7	60	Republican	New York
Taft W H	1857	4	72	Republican	Ohio
Wilson W	1856	8	67	Democratic	Virginia
Harding W G	1865	2	57	Republican	Ohio
Coolidge C	1872	5	60	Republican	Vermont
Hoover H C	1874	4	90	Republican	Iowa
Roosevelt F D	1882	12	63	Democratic	New York
Truman H S	1884	7	88	Democratic	Missouri
Eisenhower D D	1890	8	79	Republican	Texas
Kennedy J F	1917	2	46	Democratic	Massachusetts
Johnson L B	1908	5	65	Democratic	Texas
Nixon R M	1913	5	?	Republican	California
Ford G R	1913	2	?	Republican	Nebraska
Carter J E	1924	4	?	Democratic	Georgia
Reagan R	1911	3	?	Republican	Illinois

22.2 PRES_MARRIAGE

PRES_NAME	SPOUSE_NAME	PR_AGE	SP_AGE	NR_CHILDREN	MAR_YEAR
Washington G	Custis M D	26	27	0	1759
Adams J	Smith A	28	19	5	1764
Jefferson T	Skelton M W	28	23	6	1772
Madison J	Todd D D P	43	26	0	1794
Monroe J	Kortright E	27	17	3	1786
Adams J Q	Johnson L C	30	22	4	1797
Jackson A	Robards R D	26	26	0	1794
Van Buren M	Hoes H	24	23	4	1807
Harrison W H	Symmes A T	22	20	10	1795
Tyler J	Christian L	23	22	8	1813
Tyler J	Gardiner J	54	24	7	1844
Polk J K	Childress S	28	20	0	1824
Taylor Z	Smith M M	25	21	6	1810
Fillmore M	Powers A	26	27	2	1826
Fillmore M	McIntosh C C	58	44	0	1858
Pierce F	Appleton J M	29	28	3	1834
Lincoln A	Todd M	33	23	4	1842
Johnson A	McCardle E	18	16	5	1827
Grant U S	Dent J B	26	22	4	1848
Hayes R B	Webb L W	30	21	8	1852
Garfield J A	Rudolph L	26	26	7	1858
Arthur C A	Herndon E L	29	22	3	1859
Cleveland G	Folson F	49	21	5	1886
Harrison B	Scott C L	20	21	2	1853
Harrison B	Dimmick M S L	62	37	1	1896
McKinley W	Saxton I	27	23	2	1871
Roosevelt T	Lee A H	22	19	1	1880
Roosevelt T	Carow E K	28	25	5	1886
Taft W H	Herron H	28	25	3	1886
Wilson W	Axson E L	28	25	3	1885
Wilson W	Galt E B	58	43	0	1915
Harding W G	De Wolfe F K	25	30	0	1891
Coolidge C	Goodhue G A	33	26	2	1905
Hoover H C	Henry L	24	23	2	1899
Roosevelt F D	Roosevelt A E	23	20	6	1905
Truman H S	Wallace E V	35	34	1	1919
Eisenhower D D	Doud G	25	19	2	1916
Kennedy J F	Bouvier J L	36	24	3	1953
Johnson L B	Taylor C A	26	21	2	1934
Nixon R M	Ryan T C	27	28	2	1940
Ford G R	Warren E B	35	30	4	1948
Carter J E	Smith R	21	18	4	1946
Reagan R	Wyman J	28	25	2	1940
Reagan R	Davis N	41	28	2	1952

22.3 PRES_HOBBY

PRES_NAME	HOBBY
Adams J Q	Billiards
	Swimming
	Walking
Arthur C A	Fishing
Cleveland G	Fishing
Coolidge C	Fishing
	Golf
	Indian Clubs
	Mechanical Horse
	Pitching Hay
Eisenhower D D	Bridge
	Golf
	Hunting
	Painting
	Fishing
Garfield J A	Billiards
Harding W G	Golf
	Poker
	Riding
Harrison B	Hunting
Hayes R B	Croquet
	Driving
	Shooting
Hoover H C	Fishing
	Medicine Ball
Jackson A	Riding
Jefferson T	Fishing
	Riding
Johnson L B	Riding
Kennedy J F	Sailing
	Swimming
	Touch Football
Lincoln A	Walking
McKinley W	Riding
	Swimming
	Walking
Nixon R M	Golf
Roosevelt F D	Fishing
	Sailing
	Swimming
Roosevelt T	Boxing
	Hunting
	Jujitsu
	Riding
	Shooting
	Tennis
	Wrestling
Taft W H	Golf
	Riding
Taylor Z	Riding
Truman H S	Fishing
	Poker
	Walking
Van Buren M	Riding
Washington G	Fishing
	Riding
Wilson W	Golf
	Riding
	Walking

22.4 ADMINISTRATION

ADMIN_NR	PRES_NAME	YEAR_INAUGURATED
1	Washington G	1789
2	Washington G	1793
3	Adams J	1797
4	Jefferson T	1801
5	Jefferson T	1805
6	Madison J	1809
7	Madison J	1813
8	Monroe J	1817
9	Monroe J	1821
10	Adams J Q	1825
11	Jackson A	1829
12	Jackson A	1833
13	Van Buren M	1837
14	Harrison W H	1841
14	Tyler J	1841
15	Polk J K	1845
16	Taylor Z	1849
16	Fillmore M	1850
17	Pierce F	1853
18	Buchanan J	1857
19	Lincoln A	1861
20	Lincoln A	1865
20	Johnson A	1865
21	Grant U S	1869
22	Grant U S	1873
23	Hayes R B	1877
24	Garfield J A	1881
24	Arthur C A	1881
25	Cleveland G	1885
26	Harrison B	1889
27	Cleveland G	1893
28	McKinley W	1897
29	McKinley W	1901
29	Roosevelt T	1901
30	Roosevelt T	1905
31	Taft W H	1909
32	Wilson W	1913
33	Wilson W	1917
34	Harding W G	1921
34	Coolidge C	1923
35	Coolidge C	1925
36	Hoover H C	1929
37	Roosevelt F D	1933
38	Roosevelt F D	1937
39	Roosevelt F D	1941
40	Roosevelt F D	1945
40	Truman H S	1945
41	Truman H S	1949
42	Eisenhower D D	1953
43	Eisenhower D D	1957
44	Kennedy J F	1961
44	Johnson L B	1963
45	Johnson L B	1965
46	Nixon R M	1969
47	Nixon R M	1973
47	Ford G R	1974
48	Carter J E	1977
49	Reagan R	1981

22.5 ADMIN_PR_VP

ADMIN_NR	PRES_NAME	VICE_PRES_NAME
1	Washington G	Adams J
2	Washington G	Adams J
3	Adams J	Jefferson T
4	Jefferson T	Burr A
5	Jefferson T	Clinton G
6	Madison J	Clinton G
7	Madison J	Gerry E
8	Monroe J	Tompkins D
9	Monroe J	Tompkins D
10	Adams J Q	Calhoun J
11	Jackson A	Calhoun J
12	Jackson A	Van Buren M
13	Van Buren M	Johnson R M
14	Harrison W H	Tyler J
15	Polk J K	Dallas G M
16	Taylor Z	Fillmore M
17	Pierce F	De Vane King
18	Buchanan J	Breckinridge
19	Lincoln A	Hamlin H
20	Lincoln A	Johnson A
21	Grant U S	Colfax S
22	Grant U S	Wilson H
23	Hayes R B	Wheeler W
24	Garfield J A	Arthur C A
25	Cleveland G	Hendricks T A
26	Harrison B	Morton L P
27	Cleveland G	Stevenson A E
28	McKinley W	Hobart G A
29	McKinley W	Roosevelt T
30	Roosevelt T	Fairbanks C W
31	Taft W H	Sherman J S
32	Wilson W	Marshall T R
33	Wilson W	Marshall T R
34	Harding W G	Coolidge C
35	Coolidge C	Dawes C G
36	Hoover H C	Curtis C
37	Roosevelt F D	Garner J N
38	Roosevelt F D	Garner J N
39	Roosevelt F D	Wallace H A
40	Roosevelt F D	Truman H S
41	Truman H S	Barkley A W
42	Eisenhower D D	Nixon R M
43	Eisenhower D D	Nixon R M
44	Kennedy J F	Johnson L B
45	Johnson L B	Humphrey H H
46	Nixon R M	Agnew S T
47	Nixon R M	Agnew S T
47	Nixon R M	Ford G R
47	Ford G R	Rockefeller N
48	Carter J E	Mondale W F
49	Reagan R	Bush G

22.6 STATE

STATE_NAME	ADMIN_ENTERED	YEAR_ENTERED
Massachusetts	?	1776
Pensylvania	?	1776
Virginia	?	1776
Connecticut	?	1776
South Carolina	?	1776
Maryland	?	1776
New Jersey	?	1776
Georgia	?	1776
New Hampshire	?	1776
Delaware	?	1776
New York	?	1776
North Carolina	?	1776
Rhode Island	?	1776
Vermont	1	1791
Kentucky	1	1792
Tennessee	2	1796
Ohio	4	1803
Louisianna	6	1812
Indiana	7	1816
Mississippi	8	1817
Illinois	8	1818
Alabama	8	1819
Maine	8	1820
Missouri	9	1821
Arkansas	12	1836
Michigan	12	1837
Florida	14	1845
Texas	15	1845
Iowa	15	1846
Wisconsin	15	1848
California	16	1850
Minnesota	18	1858
Oregon	18	1859
Kansas	18	1861
West Virginia	19	1863
Nevada	19	1864
Nebraska	20	1867
Colorado	22	1876
North Dakota	26	1889
South Dakota	26	1889
Montana	26	1889
Washington	26	1889
Idaho	26	1890
Wyoming	26	1890
Utah	27	1896
Oklahoma	30	1907
New Mexico	31	1912
Arizona	31	1912
Alaska	43	1959
Hawaii	43	1959

22.7 ELECTION

ELECTION_YEAR	CANDIDATE	VOTES	WINNER_LOSER_INDIC
1789	Washington G	69	W
	Adams J	34	L
	Jay J	9	L
	Harrison R H	6	L
	Rutledge J	6	L
	Hancock J	4	L
	Clinton G	3	L
	Huntington S	2	L
	Milton J	2	L
	Armstrong J	1	L
	Lincoln B	1	L
	Telfair E	1	L
1792	Washington G	132	W
	Adams J	77	L
	Clinton G	50	L
	Jefferson T	4	L
	Burr A	1	L
1796	Adams J	71	W
	Jefferson T	68	L
	Pinckney T	59	L
	Burr A	30	L
	Adams S	15	L
	Ellsworth O	11	L
	Clinton G	7	L
	Jay J	5	L
	Iredell J	3	L
	Henry J	2	L
	Johnson S	2	L
	Washington G	2	L
	Pinckney C C	1	L
1800	Jefferson T	73	W
	Burr A	73	L
	Adams J	65	L
	Pinckney C C	64	L
	Jay J	1	L
1804	Jefferson T	162	W
	Pinckney C C	14	L
1808	Madison J	122	W
	Pinckney C C	47	L
	Clinton G	6	L
1812	Madison J	128	W
	Clinton G	89	L
1816	Monroe J	183	W
	King R	34	L
1820	Monroe J	231	W
	Adams J Q	1	L
1824	Adams J Q	84	W
	Jackson A	99	L
	Crawford W H	41	L
	Clay H	37	L
1828	Jackson A	178	W
	Adams J	83	L
1832	Jackson A	219	W
	Clay H	49	L
	Floyd J	11	L
	Wirt W	7	L
1836	Van Buren M	170	W
	Harrison W H	73	L
	White H L	26	L
	Webster D	14	L
	Mangum W P	11	L

1840	Harrison W H	234	W
	Van Buren M	60	L
1844	Polk J K	170	W
	Clay H	105	L
1848	Taylor Z	163	W
	Cass L	127	L
1852	Pierce F	254	W
	Scott W	42	L
1856	Buchanan J	174	W
	Fremont J C	114	L
	Fillmore M	8	L
1860	Lincoln A	180	W
	Breckinridge J	72	L
	Bell J	39	L
	Douglas S	12	L
1864	Lincoln A	212	W
	McClellan G B	21	L
1868	Grant U S	214	W
	Seymour	80	L
1872	Grant U S	286	W
	Hendricks T A	42	L
	Brown B G	18	L
	Jenkins C J	2	L
	Davis D	1	L
1876	Hayes R B	185	W
	Tilden S J	184	L
1880	Garfield J A	214	W
	Hancock W S	155	L
1884	Cleveland G	219	W
	Blaine J G	182	L
1888	Harrison B	233	W
	Cleveland G	168	L
1892	Cleveland G	277	W
	Harrison B	145	L
	Weaver J B	22	L
1896	McKinley W	271	W
	Bryan W J	176	L
1900	McKinley W	292	W
	Bryan W J	155	L
1904	Roosevelt T	336	W
	Parker A B	140	L
1908	Taft W H	321	W
	Bryan W J	162	L
1912	Wilson W	435	W
	Roosevelt T	88	L
	Taft W H	8	L
1916	Wilson W	277	W
	Hughes C E	254	L
1920	Harding W G	404	W
	Cox W W	127	L
1924	Coolidge C	382	W
	Davis J W	136	L
	La Follette R M	13	L
1928	Hoover H C	444	W
	Smith A E	87	L
1932	Roosevelt F D	472	W
	Hoover H C	59	L
1936	Roosevelt F D	523	W
	Landon A M	8	L
1940	Roosevelt F D	449	W
	Wilkie W L	82	L
1944	Roosevelt F D	432	W
	Dewey T E	99	L
1948	Truman H S	303	W

1948	Dewey T E	189	L
	Thurmond J S	39	L
1952	Eisenhower D D	442	W
	Stevenson A	89	L
1956	Eisenhower D D	457	W
	Stevenson A	73	L
	Jones W B	1	L
1960	Kennedy J F	303	W
	Nixon R M	219	L
	Byrd	15	L
1964	Johnson L B	486	W
	Goldwater B	52	L
1968	Nixon R M	301	W
	Humphrey H H	191	L
	Wallace G C	46	L
1972	Nixon R M	520	W
	McGovern G S	17	L
	Hospers J	1	L
1976	Carter J E	297	W
	Ford G R	240	L
1980	Reagan R	489	W
	Carter J	49	L