Overview

SQL stands for "Structured Query Language" and is an industry standard language which allows you to query existing data, modify that data and to define new tables, indexes and views.

The SQL module of SIR/XS implements the data retrieval, update and definition capabilities defined by "American National Standard X3.135-1986 Database Language SQL." In addition, SIR/XS has implemented many enhancements to simplify the interactive use of SQL and to take advantage of SIR/XS database structures.

The primary function of SQL is to select data from records and tables where particular conditions are true. The selected data always creates a new table. The data from a table can be then be displayed in a simple and straightforward manner. Tables can also be used by the other SIR/XS components such as VisualPQL to produce more complex analyses and outputs.

SQL creates and populates new tables and can also be used to create indexes and views to tables.

SQL can be used to update and modify databases and tables and can update whole sets of data which match particular conditions. While SQL also has some direct data entry functions, these are limited and data entry is better handled in other SIR/XS modules.

The SirSQL interface is a menu driven system, which, since SQL is a command based language, generates commands.

The main SQL command is SELECT, which selects data according to particular conditions, creating a new set of data on a new table. There are various formatting options for individual columns.

There are commands such as UPDATE which updates individual records or sets of records and CREATE which defines new tables.

The SQL settings such as the connected databases and tabfiles, the limits on reading and writing records, any synonym definitions, path definitions, etc. make up the *workspace*. The SAVE and GET commands save the workspace and get it again in subsequent sessions. SQL uses a default workspace file, called `SirSQL.wsp` which is loaded when you access SQL. You can modify and save the default workspace, or create and use any number of different workspace files.

A simple editor is used to enter commands into a command area. SQL commands can be created and saved for subsequent execution. Commands can be kept as members in the

procedure file of a database or as operating system files. Commands can be run as a batch process.

Online help is available for explanation and syntax of all commands, options and clauses.

Databases & Tabfiles

Databases

SQL can operate on multiple SIR/XS databases. Before working with a database, the database must be connected. The last database connected is the default database which is used whenever a particular database is not specified.

Queries are optimised to take advantage of database case structures and records with common keys, simplifying the required query specifications and making the retrieval more efficient. All quality controls, including security, defined in the database schema are applied when using SQL for data entry or modification.

SQL can operate concurrently with other SIR/XS modules when reading data and through Master for concurrent update.

Tables and Tabfiles

SQL can read, write and create tables. A table holds a single type of record and is equivalent to a database record. Tables can be used by VisualPQL and FORMS as well as by SQL. Tables can only be used for update by one user at once. Tables are held on a Tabfile.

A tabfile contains tables, indexes to the tables plus System tables which hold information about other tables. A single tabfile can contain multiple tables. Tabfiles can contain security controls and authorised users may grant or revoke permissions for operations on a tabfile or on specific tables within a tabfile.

SQL can operate on multiple tabfiles. Before working with a tabfile, the tabfile must be connected.

A temporary system tabfile ($SYSTEM) is the default used for creating temporary tables. This is created and deleted per session and is not normally listed on displays of tabfiles. The `$` as the start character in a name is used by the system to recognise system generated names and so should not be used except for this purpose.

The system expects a file extension of `.tbf` to be used for tabfiles.

Views

A View is a logical table created within SQL. A logical table is a table which does not physically exist; it is created 'on the fly' from other tables and records. Once defined, a

view may be used and referenced in the same way as other tables. Views can only be used by SQL.

Syntax Rules

SQL is primarily a command based system so there are rules for the syntax of the commands. The syntax is intended to be English like and as natural as possible. SQL translates all commands, keywords and names (unless a non-standard name) to uppercase.

A command normally ends at the end of a line. To specify that a command continues on the next line, put a hyphen (continuation character) after all the text on the line to be continued.

```
SELECT NAME SALARY -
FROM EMPLOYEE -
WHERE GENDER EQ 1
```

You can submit multiple commands at one time. To do this, start each command on a new line. The starting position of a command on a line is unimportant.

Names

Names are normal SIR/XS names. Standard names are 1 to 32 characters long with no spaces. The first character must be alphabetic. Characters can be letters, digits or four characters ($, #, @,_). Names are translated to upper case so uppercase and lowercase letters are equal. A name cannot be an SQL reserved word. The following are examples of valid names:

```
A     PART_NUMBER     NAME1
```

Non-standard names

If you wish to use a name which does not conform to these rules, enclose the name in curly braces {} as per other SIR/XS references to non-standard names. (Note: for compatibility with previous versions and with the SQL standard, the SIR/XS SQL module also supports the use of double quotes "" as delimiters on input for non-standard names, but these will be translated to curly brackets on output.)

A non-standard name can contain blanks or use lowercase letters. Non-standard names might be:

```
{On hand}     {SELECT}     {part*}
```

Suppose a column is created with a `SELECT` such as:

```
SELECT ...  max(salary)*1.1 .....
```

If this column is referred to in a subsequent `SELECT`, the column name is not a valid SQL name and must be enclosed in curly brackets. This name was not specified as a non-

standard name when originally created and was thus mapped to upper case, so the reference would be:

```
SELECT {MAX(SALARY)*1.1} .....
```

Note: It is possible to specify new column names with the FORMAT COLUMN NAME command and it is possible to refer to columns by a column number. This minimises the need to have non-standard column names, even where these have been derived from a calculation. For example:

```
SELECT id salary*1.3 FORMAT COLUMN 2 NAME newsal
```

Single v Double Quotes

Strings inside single quotes are constants; strings inside double quotes are names. (This is according to the SQL standard and is thus different from other SIR/XS modules standard behaviour.) When creating a table, it is possible to specify constants as columns, so you must use single or double quotes correctly. For example, suppose the following:

```
SELECT 'MAX(SALARY)*1.1' .....
```

Because this command has single quotes, it creates a character constant with the value `MAX(SALARY)*1.1` rather than looking for a column with this name.

Qualified Record and Table Names

Unless using the default database or tabfile, qualify record and table names with the appropriate database or tabfile name. Separate the names by a period. For example:

```
SELECT NAME FROM COMPANY.EMPLOYEE ON MYTFILE.TABLE1
```

Qualifying variable names

Variables within a record or table always have unique names. However, variables in different records may have the same name. If referring to two variables with the same name from different records or tables, qualify the name by preceding it with the record or table name. Separate the names by a period. For example:

```
SELECT EMPLOYEE.ID  DEPT.ID   FROM EMPLOYEE DEPT
```

Alias

Sometimes you may need to qualify the record or table by a database or tabfile name. However, qualification at two levels (A.B.C) is not a valid SQL format. In this case (or when joining a record to itself) specify an *alias*. Specify a single name after the qualified record or table name, optionally using the `AS` keyword and then specify this to qualify individual variable names used in the select, where, order by or other clauses. For example:

```
SELECT A.ID B.ID FROM COMPANY.EMPLOYEE AS A, OLD.EMPLOYEE AS B -
WHERE A.ID = B.ID
```

Filenames

When specifying filenames in SQL, use the standard short or long SIR/XS Filenames.

Numeric Constants

Specify integer constants as a series of digits without any embedded blanks or commas. They may be preceded by a + or - sign to indicate whether they are positive or negative. No sign is an indication of a positive number. For example:

```
1     134     999     +333     -9322
```

Specify real constants as a series of digits followed by a decimal point and another series of digits. They may be preceded by a + or - sign. Either the initial or terminal series of digits may be absent but not both. Specify a power of 10 exponent by suffixing the number by the character '**E**' followed by the power of ten. There can be no embedded blanks or commas in the number. For example:

```
1.     .44     +1.4     -.44433     123.456E3     333.432E-3
```

Character Strings

Specify character strings within single quote marks. To include a single quote within the character string, enter the single quote twice in succession for each occurrence required. For example:

```
'aaa'     'THIS is A string'     'Bill''s job'
```

Expressions

Expressions are a combination of variables and operators which produce a new value. There are numeric expressions and string expressions. For example, adding two numeric variables produces the sum; concatenating two string variables produces a longer string.

There are functions in SQL which convert numbers to strings and vice versa.

Dates, times and categorical variables can be either integers or strings and SQL decides which format to use from the expression that uses the variable. If a string is called for, the string is used; if a numeric value is called for, the integer value is used. For example, a time can be an integer (the number of seconds since midnight) or a character string such as '11.15AM'.

Numeric expressions consist of numeric variables, constants, all of the normal arithmetic operators (+, -, /, *, **), numeric functions, and parentheses. Expressions are evaluated according to normal precedence and parentheses can be used. Within equal precedence they are evaluated from left to right. Example numeric expressions are:

```
2 * 5 - 4     2 * ( 5 - 4 )     SALARY * 52 / 12
```

String expressions consist of string variables, string constants in single quotes, string functions, the concatenation operator (+) and parentheses. Example string expressions are:

```
'ABC' + 'DEF'     SBST('ABCDEF',4,3)     TIMEC(NOW(0),'HH:MM:SS')
```

Date Formats

Specify a date format for dates. See date formats for a complete description of date formats.

Time Formats

Specify a time format for times. See time formats for a complete description of time formats.

Functions

A function is a keyword followed by one or more arguments enclosed in parentheses. Arguments may be either variable names, constants or expressions. The function operates on the arguments and returns an appropriate value for each record selected. For example, `RND` is a function which returns a number rounded to the nearest whole integer;

```
SELECT NAME RND(SALARY*12/52) FROM EMPLOYEE
```

Comments

Comments can be included in commands using the exclamation character (**!**). When this character appears on a command line and is not enclosed in quotes, then that character and the remainder of the input line are ignored. The continuation character, must be the last character on the line.
Example:

```
SELECT NAME         ! get the full name -
RND(SALARY*12/52)   ! and weekly salary -
FROM EMPLOYEE       ! for each employee
```

SELECT

```
SELECT   [ DISTINCT ]  variable_list
FROM     [ tabfile.  ] table_name,     |
         [ tabfile.  ] view_name ,     |
         [ database.] record_name,     |
         [ database.] CIR ,            |
         path_name ,                   |
         [AS] alias_name,
```

Keywords:

```
         CASELIM n
         COMPILE_ONLY
         DBMS [ filename ]
         OUTPUT filename
         RECLIM n
         SAMPLE proportion [ , seed ]
         SELLIM n
```

Clauses:

FORMAT
GROUP BY
ON
ORDER BY
OUTER
UNION
WHERE

The `SELECT` command takes data from one or more existing records or rows and creates a new table with the selected data in it. The definition of data in the new table is copied from the existing definition. The number of rows in the new table depends on the number of rows in the input and the particular clauses specified. (See Output from SELECT.)

The variable list and the `FROM` clause are required. All other clauses are optional.

The result of the `SELECT` is a new table containing the selected variables. Tables created with the `SELECT` command are exactly the same as tables created in any other way. The `SELECT` command also controls the display format of the variables in the created table. The display format of the table can be modified with the display processor. Formatting can be specified on the `SELECT` with the `FORMAT` option.

The basic form of the command is:

```
SELECT variable_list FROM record_list WHERE condition
```

The variable list specifies the variables to be saved in the new table; the FROM clause, lists the records and tables which contain the input data and the WHERE clause, specifies

the conditions under which records or rows are selected.

Output

One occurrence of the set of selected variables is one row in the output table. The clauses and keywords specified in the SELECT affect the way in which the output table is constructed. If variables are selected from a single record or table, one output row is written for every individual data record or row which satisfies the WHERE clause.

If variables are selected from multiple records or rows, one output row is written for each individual data record or row which exists and which meets the selection criteria. For example the following results in one output row per combination of employee and review:

```
SELECT NAME POSITION RATING -
FROM EMPLOYEE REVIEW -
WHERE EMPLOYEE.ID = REVIEW.ID
```

This is sometimes referred to as an *inner* join. There is no output for instances where a record of one type exists but not the other. A join which produces an output row regardless of whether the joined record or row exists is known as an *outer* join. Specify the OUTER keyword following the FROM clause. Follow OUTER with one or more record or table names. For example;

```
SELECT ....  FROM A B OUTER B
SELECT ....  FROM A B OUTER A
SELECT ....  FROM A B OUTER A B
```

All these examples generate an output row for every combination of A and B; the first example adds all occurrences of A where B does not exist; the second example adds all occurrences of B where A does not exist; the third example takes both occurrences of A with no B and occurrences of B with no A. These are sometimes referred to as a RIGHT outer join, a LEFT outer join and a SYMMETRIC outer join.

DISTINCT

The DISTINCT keyword specifies that one output row is written for each distinct or unique set of values selected. For example;

```
SELECT DISTINCT SALARY FROM EMPLOYEE
```

lists each salary that one or more people are earning. There is only one output row per value of salary, regardless of how many people earn that salary. The keyword UNIQUE (abbreviation UNQ) is a synonym for DISTINCT.

Aggregation

Aggregation functions compute single values from multiple records and change the number of rows that are created. An aggregation function returns a single value to represent a calculation (such as an average) across multiple records. For example:

```
SELECT AVG(SALARY) FROM EMPLOYEE
```

This returns one value, the average salary of all the employees; correspondingly, only one row is created in the output table.

The following functions are aggregation functions and alter the number of output rows produced:

`SUM`
: Sum of values selected

`AVG`
: Average of values selected

`STD`
: Standard deviation of values selected

`MAX`
: Maximum value selected

`MIN`
: Minimum value selected

`FIRST`
: First non-missing value selected

`LAST`
: Last non-missing value selected

`COUNT`
: Count number of values

`SUM, AVG` and `STD` operate only on numeric data. Other aggregation functions operate on any type of data.

GROUP BY

The `GROUP BY` clause specifies that one output row is written for each unique value of the variables specified . For example,

```
SELECT .....  FROM EMPLOYEE GROUP BY CURRPOS
```

produces one output row per value in the `CURRPOS` variable. Since this is a grouping of data, individual data items cannot be selected; the only legal expressions that can be used in the variable list when `GROUP BY` is specified are aggregate data or the variables in the `GROUP BY` clause. For example, to produce a count of people and a total salary in each position.

```
SELECT CURRPOS COUNT(SALARY) SUM(SALARY) FROM EMPLOYEE -
```

```
GROUP BY CURRPOS
```

The table that results from aggregation has one entry per aggregation level. To produce a table with individual values for a column plus subtotals and totals (sums, averages, counts, or other statistics) , use the Format clauses.

Variable List

The variable list in the SELECT is a list of variable names and expressions. When a variable name is specified, the definition of that variable is copied from the existing definition. If variables have the same name on different records or tables, qualify the variable name by the name of the record or table. Qualified names are separated by a period.

An asterisk (*) specifies all of the variables in all of the records or rows referenced in the FROM clause. For example,

```
SELECT ID NAME FROM EMPLOYEE
SELECT * FROM REVIEW
SELECT EMPLOYEE.NAME .....  FROM EMPLOYEE .....
```

Computing New Values

Expressions may be specified in the variable list. For example, the following query computes the weekly salary by multiplying monthly salary (SALARY) by 12 and dividing by 52.

```
SELECT ID NAME SALARY*12/52 FROM EMPLOYEE
```

The name of the computed variable in the output table is the first 32 characters of the expression used to calculate it.

Numeric expressions may use numeric constants, numeric variables, the arithmetic operators (+ , - , / , * , **), SQL functions, and parentheses to denote the order of operations.

Character expressions may use quoted strings, variables which are strings, the "+" character to join strings, SQL string functions, and parentheses to denote the order of operation. Enclose strings in single quotes.

Functions

Functions are specified as a keyword followed by one or more arguments enclosed in parentheses. Arguments may be variable names, constants or expressions. The function operates on the arguments and returns a single value for each record selected.

For example, RND is a function which returns a number rounded to the nearest whole integer:

```
SELECT NAME RND(SALARY*12/52) FROM EMPLOYEE
```

FROM

The FROM clause is required on the SELECT and specifies the records, paths, views and tables to be accessed. Database records may be specified by name or number. If more than one record or table is specified on the FROM clause, a join is performed.

Referencing Multiple Tables

A join retrieves data from two or more records or tables with one query. A join is implied whenever the FROM clause references more than one source for the data.

There must be common values in some corresponding columns between the data sources in the FROM clause. For example, to create a table of employee name and position level for each position ever held by the employee:

```
SELECT NAME POSITION FROM EMPLOYEE OCCUP -
WHERE  EMPLOYEE.ID = OCCUP.ID
```

This uses ID to form the relationship between the records in the WHERE clause. Use the record name as prefix to differentiate between column names which are the same in different records or tables. That is, EMPLOYEE.ID refers to the value of ID in the Employee record, while OCCUP.ID refers to the value of ID in the Occup record.

In a case structured database with case mode enabled (the default), there is an automatic relationship between records. A query automatically joins records for the same case. The automatic relationship has the same effect as the WHERE clauses matching each record on the case ID. For example, referencing data from multiple records on the COMPANY database, has the following implicit WHERE clause:

```
SELECT ...  FROM  EMPLOYEE OCCUP REVIEW -
            WHERE EMPLOYEE.ID = OCCUP.ID -
            AND   EMPLOYEE.ID = REVIEW.ID
```

There is no need to specify the WHERE clause to join on the case identifier.

PATHS

A PATH links one record (or table) to another record (or table) and specifies the manner in which they are to be joined. System defined paths are automatically created by SQL between records in a database which have common keyfields. Common keyfields mean the same variable names of the same type in the same sequence in different records. For example, on the company database there is a system defined path between OCCUP and REVIEW which is equivalent to:

```
...  WHERE OCCUP.POSITION = REVIEW.POSITION
```

Each path has a name. Paths are explicitly invoked by naming a record (or table) and a path name on the FROM clause. The path is invoked implicitly by naming both records (or tables) in the FROM clause. If there are multiple paths between two record types, SQL uses the earliest defined path. SQL never automatically creates more than one path between any two records. The SHOW PATH command displays the path definitions in order.

Create paths with the CREATE PATH command. This names the path, the two records (or tables) to be joined and the variable(s) used to join them. For example,

```
CREATE PATH MYPATH -
FROM   COMPANY.OCCUP TO COMPANY.REVIEW -
VIA    POSITION
```

Use the path by naming the path rather than the second record on the FROM clause. For example,

```
SELECT ....  FROM OCCUP MYPATH
```

Path Mode and Case Mode

Records are joined whenever multiple tables or records are referenced in the FROM clause. The way in which records are joined is determined by the WHERE clause, by any PATHS that are referenced in the FROM clause and by the current settings of CASE mode and PATH mode.

If CASE mode is set, records are joined when the case identifier variable in one record is equal to the case identifier in another. This is equivalent to specifying a WHERE clause such as:

```
...  WHERE RECONE.CASEID = RECTWO.CASEID
```

If PATH mode is set, any paths between the records referenced in the FROM clause are automatically used. The system defined paths imply joins based on the case identifier and relationships implied in the keyfields.

For example, the default path joining the OCCUP and REVIEW records in the sample database is equivalent to the WHERE clause:

```
...  WHERE OCCUP.ID EQ REVIEW.ID AND OCCUP.POSITION EQ REVIEW.POSITION
```

The setting of `CASE` has no effect on joins if `PATH` mode is set. (However if you are using a subquery, case mode *must* be off). The setting of `CASE` does have an effect if `PATH`s are cleared (records are still joined within case).

If `PATHS` are explicitly referenced in the `FROM` clause, the setting of `CASE` and `PATH` have no effect, the `PATH` is always applied as defined.

Joins

The normal type of join and the automatic join performed by case and path mode is an *Equi-Join* because the comparison operator between the two columns in the two tables is an "Equal" (EQ or =).

A join condition can specify other relationships between columns, such as "greater than" (GT or >), "less than" (LT or <), etc. These are referred to as Non-Equi-Joins. For example, suppose a table is created of minimum and maximum starting salaries by division:

```
SELECT DIVISION MIN(STARTSAL) MAX(STARTSAL) -
FROM OCCUP -
ON DIVSAL -
GROUP BY DIVISION -
FORMAT COL 2 NAME MINSAL -
FORMAT COL 3 NAME MAXSAL
```

Then, select anyone whose current salary is greater than the maximum or less than the minimum starting salaries.

```
SELECT ID NAME SALARY FROM EMPLOYEE REVIEW DIVSAL -
WHERE SALARY GT MAXSAL OR SALARY LT MINSAL
```

Note that this non-equi-join is a join and not just a test on a column value. This means that a row is produced for every matching condition between the joined rows. Therefore, a row appears once for each time that salary is greater than one of the division maxsal columns or less than one of the division minsal columns.

Non-equi-joins on large tables can produce a tremendous number of rows and the `WHERE` conditions should be carefully examined to limit the output to the required combinations.

Alias

You can define an *Alias* name for a record or a table in the `FROM` clause. Use an alias to qualify a variable name when it cannot be done by using the unique record or table name. A record or table name is not unique if the same record or table is on two different databases or tables you are joining, or if you join a record or table to itself.

The alias is defined in the record or table specification in the `FROM` clause and is used wherever the table or record name would be used in other parts of the `SELECT` statement.

When an alias is defined, it follows the record name or the optional keyword `AS` and must be followed by a comma (,) to separate it from any other record names.

For example, suppose a genealogical database where everybody is in a `PERSON` record, and each person has an `ID`. In each person's record are the `ID`s of their father and mother. Because this involves joining a record type to itself, use an alias:

```
SELECT CHILD.ID FATHER.ID MOTHER.ID -
FROM   PERSON AS CHILD,  PERSON AS FATHER, PERSON AS MOTHER -
WHERE  CHILD.FATHERID EQ FATHER.ID AND -
       CHILD.MOTHERID EQ MOTHER.ID
```

Keywords

The following keywords can be specified on the SELECT command:

`CASELIM n`
: Specifies that the process stops after reading n cases and prompts whether to continue processing. If this is not specified, the default is used. The system default is 1000, which can be changed with the SET command.

`COMPILE_ONLY`
: Specifies that the query is compiled but not executed. This is used to check the syntax of a query without performing the retrieval.

`DBMS filename`
: Specifies that a VisualPQL version of the query is written. This option is intended as a debugging tool. Some modifications to the resultant program may have to be made before use.

`OUTPUT filename`
: Sets the print filename for subsequent `PRINT` commands. This can also be set by the `SET` command and by the `DISPLAY` command.

`RECLIM n`
: Specifies that the process stops after retrieving n records or rows and prompts whether to continue processing. If this is not specified, the default is used. The system default is 1000, which can be changed with the SET command.
Abbreviation: `RLIM`

`SAMPLE proportion [,seed]`
: Specifies that a random sample is produced. (0.0 < proportion < = 1.0) The optional seed parameter specifies an odd integer to be used as the seed for the random number generator. This permits the generation of different samples.

`SELLIM n`
: Specifies that the process stops after n rows have been selected and prompts whether to continue. If this is not specified, the default is used. The system default is 1000, which can be changed with the SET command.
Abbreviation: `SLIM`

FORMAT

```
FORMAT COLUMN column_list  column_format
```

Use `FORMAT` clause(s) on a SELECT command to specify the appearance of particular columns. When a table is created, the default settings are used to format columns unless specifically overridden with `FORMAT` clauses. Specify multiple format clauses on a single select command by repeating the complete `FORMAT` clause.

The first part of a format clause specifies one or more columns to be formatted. Specify the keyword `COLUMN` followed by the column_list. The next part of the FORMAT clause specifies the Column Format for the particular columns which is typically a keyword (e.g. `WIDTH`) and a setting.

You can specify multiple options on a single format clause where the options apply to the specified column(s). For example:

```
SELECT ID SALARY*1.1 FROM EMPLOYEE -
FORMAT COLUMN 1 NAME EMPLOYEE_ID WIDTH 12 -
FORMAT COLUMN 2 NAME NEWSALARY
```

Column References

Columns can be referenced by column number or by column name. Column number refers to the sequential (left to right) number of the column in the table. Column numbers remain the same regardless of any formatting commands. Each column has a name that can be altered by formatting commands. If the column name is changed then the new column name is used. A column label may be displayed, but the column is always referenced by its name or number.

Specify a list or range of consecutive columns by specifying the start and end columns separated by a colon (**:**).

Examples of column specifications might be as follows:

```
FORMAT COLUMN SALARY option
FORMAT COLUMN 1 option
FORMAT COLUMN 3:7 option
FORMAT COLUMN 1 4 8 option
FORMAT COLUMN SALARY option
FORMAT COLUMN BIRTHDAY option
```

Column Formats

Columns have a number of display characteristics which can be specified. The specification can be supplied when the table is created by a SELECT or a CREATE TABLE or can be amended by specific commands.

The column formats are specified as a keyword and a setting as follows:

```
[  DATE 'date_map' ]
[  DPLACES n ]
[  EXPONENT n ]
[  LABEL ON | OFF ]
[  MISSCHAR 'c' ]
[  NAME column_name ]
[  NULL 'string']
[  SEPARATOR 'string' | n BLANKS ]
[  TIME'time_map' ]
[  VALLAB ON | OFF ]
[  WIDTH n ]
[  ZEROS ON | OFF | 'string']
```

`DATE 'date map'`
: Sets the date map.

`DPLACES n`
: Sets the number of decimal places to display. Abbreviation: `DPL`

`EXPONENT n`
: Specifies that exponential notation is used for display or printing. The number specified is the number of decimal places and zero (0) is used to indicate that the field is not displayed in exponential format. For example:
1) To display the number 8,267 as 8.267E+003 specify `EXPONENT 3`.
2) To display the same number as 8.267000E+003 specify `EXPONENT 6`.

`LABEL`
: `ON` specifies that the variable label is used as the column name. `OFF` specifies the variable name is the column name and is the default. Abbreviation: `LAB`

`MISSCHAR`
: Sets the single character to display and fill the column when it contains missing values. Abbreviation: `MISS`

`NAME`
: Sets the column name. Names must obey the SQL name rules or be specified as a string constant in single quotes. For example:
`FORMAT col 3 NAME ANNUAL_SALARY`

`NULL`
: Sets the string to display if the column contains missing values. If the specified string is longer than can be displayed, it is truncated. `NULL` takes precedence over any `MISSCHAR` specified.

`SEPARATOR`

Sets the separator to the specified string. The separator precedes this column, separating it from the previous column and is typically a number of blanks. The separator may be set to a particular string or to a specified number of `BLANKS`. Abbreviations: `SEP` and `BL, BLANK`. For example:

```
FORMAT COLUMN salary SEPARATOR 4 BLANKS
```

`TIME`

Sets the time map. For example:

```
FORMAT COLUMN startime TIME 'HH:MM'
```

`VALLAB`

Specify `VALLAB ON` to display value labels instead of data values. Specify `VALLAB OFF` to display values. Use `SET` and `CLEAR` to change the `VALLAB` setting for all variables in the table. For example:

```
FORMAT COLUMN marstat VALLAB ON
```

`WIDTH`

Sets the column width to the specified number of characters. Column headings that do not fit are wrapped to as many lines as needed. Abbreviation: `WID`. For example:

```
FORMAT COLUMN salary WIDTH 12
```

`ZEROS`

Enables or disables the display of leading zeros for numeric variables. `ZEROS OFF` is the default. You can also specify a string to be displayed if the value is zero. For example:

```
FORMAT COLUMN salary ZEROS ON
```

GROUP BY

GROUP BY specifies that all sets of values selected are grouped together according to their unique values in the value list. This produces a summary table with one entry per group of records. For example, to calculate the average salary for male and female employees:

```
SELECT VALLAB(GENDER) AVG(SALARY) FROM EMPLOYEE -
GROUP BY GENDER
```

This produces a table with two entries:

```
VALLAB(GENDER)  AVG(SALARY)
Male              2745.83
Female            2831.25
```

The aggregation can be done at additional levels by adding further variables to the GROUP BY clause:

```
SELECT VALLAB(GENDER) VALLAB(EDUC) AVG(SALARY) -
FROM EMPLOYEE GROUP BY GENDER EDUC

VALLAB(GENDER)        VALLAB(EDUC)              AVG(SALARY)
Male                  Elementary                  2533.33
Male                  High School                 2500.00
Male                  Some University             2550.00
Male                  B.Sc.  or B.A.              3050.00
Male                  M.S.                        2625.00
Male                  Ph.D.                       3350.00
Female                High School                 2600.00
Female                Some University             2700.00
Female                B.Sc.  or B.A.              3533.33
Female                M.S.                        1650.00
Female                Ph.D.                       2400.00
```

Where there are no records for a level, no row is created in the retrieval. For example, there are no female employees with only elementary education.

Selecting Groups: HAVING Clause

Use the HAVING clause with the GROUP BY clause to select groups according to some condition. For example, to select groups with an average salary greater than 2500.

```
SELECT VALLAB(EDUC) VALLAB(GENDER) COUNT(SALARY) AVG(SALARY) -
FROM EMPLOYEE GROUP BY EDUC GENDER HAVING AVG(SALARY) > 2500
```

Case Aggregation

In a case structured database, by default SQL computes aggregate functions within cases. To compute aggregates across cases, turn off case mode with the CLEAR CASE command. This turns off case mode for the rest of the session, or until it is re-enabled. Use the SET CASE command to re-enable case mode. For example; with case mode on, to select the average starting salary for all the positions an employee has had:

```
SELECT ID AVG(STARTSAL) FROM OCCUP
```

The result is one row for each employee with any OCCUP records. This gives the average of the starting salaries for each position the employee has had. This same query with cases cleared gives a very different result:

```
CLEAR CASE
SELECT AVG(STARTSAL) FROM OCCUP
```

The result is one row, giving the average starting salary for all the positions held by all the employees in the company.

Case mode is equivalent to a GROUP BY clause on the case id for the aggregation functions.

Aggregations with Missing or Undefined Values

Missing or undefined values are ignored in the computation of aggregation functions. For example, if the value of SALARY is missing for an employee, the record is ignored in the computation of the average. The average is a true average of actual values.

COUNT

COUNT counts the number of values selected in a query. It is often used in conjunction with other aggregation functions to count how many records were used in computing the aggregated value.

For example, the following query computes the mean salary and the number of all male employees.

```
SELECT AVG(SALARY) COUNT(SALARY) FROM EMPLOYEE -
WHERE GENDER = 1
```

This counts only those records which have a non-missing salary. COUNT can also have the argument ***** which specifies a count of all selected records regardless of whether the values are valid, missing, or undefined.

ON

`ON` specifies the name of the new table to contain the results of the query.

If an output table is not specified with the `ON` clause, a table called `PREVIOUS_SELECT` is used.

All tables are on tabfiles. If a tabfile is not specified, the default is used. If a default tabfile has not been explicitly `SET`, a tabfile called `$SYSTEM` is used.

If all defaults are used, the results of a `SELECT` are stored on the table `$SYSTEM.PREVIOUS_SELECT`. This table is overwritten as necessary without prompting for confirmation. If you specify a table as output which already exists, you are prompted for confirmation that you want it overwritten before proceeding with the `SELECT`.

You cannot select `ON` to a table you are selecting `FROM`.

ORDER BY

Specify `ORDER BY` to sort the selected rows. The first variable in the `ORDER BY` list is the major sort key. Specify keys in sequence from major to minor. By default, variables sort in ascending sequence. Follow the variable with the `DESC` keyword to sort in descending sequence. This only applies to the variable immediately preceding `DESC`. Missing values sort to the beginning of a set of values regardless as to whether ascending or descending is specified. (The `MISS` function can be used to retrieve original values for missing values.)

The sort keys may contain variables and expressions. The sort key variables do not have to be in the variable list of the `SELECT`.

`DISPLAY` does not resequence rows and `ORDER BY` must be specified on the `SELECT` to create the table in a particular sequence if this differs from the source data. For example:

```
SELECT ID NAME SALARY FROM EMPLOYEE  -
ORDER BY SALARY NAME
```

Synonyms: `ORDER, SORT, SORT BY`

OUTER

When joining records on a case structured database, `OUTER` specifies that, if no matching record exist, the `SELECT` operates as if a record containing undefined values for all variables did exist.

Normally, a join operation creates a row for the resultant table if there exists a record for every record name in the `FROM` clause. The `OUTER` option allows the retrieval of some data even when some records may not exist. This operation is called an outer join. The keyword `OUTER` is specified after all records in the `FROM` clause and specifies all record names that the `OUTER` applies to.

```
SELECT NAME REVDATE FROM EMPLOYEE REVIEW  OUTER REVIEW
```

UNION

`UNION` adds the result of a second `SELECT` clause to the table created by the main `SELECT` command. With the `UNION` clause, each `SELECT` must result in the same number of output columns and each column must correspond in type. The first `SELECT` command determines the names and the types of the output columns. Numeric variables must correspond to numeric variables, string variables with string variables.

The main `SELECT` command defines the table that is produced when using the `UNION` statement. For example, assume two tables, one for current employees and one for ex-employees. A single output table for all employees could be produced with the `UNION` clause:

```
SELECT NAME SALARY FROM EMPLOYEE ON ALLEMPLOYEES -
       UNION  SELECT NAME SALARY FROM EXEMPLOYEE
```

This creates a new table with two columns and a row per employee.

As many `SELECT` clauses as required may be `UNIONed` together as long as the rules on number and type of variables are followed.

If sequence of the output table is important, use the `ORDER BY` clause to specify it. Any such `ORDER BY` should follow the last clause of the last `SELECT` in the command.

WHERE

WHERE specifies the logical conditions used to select records or rows. Only records or rows meeting the conditions are selected. The WHERE clause can reference any variables, regardless as to whether the variables are in the SELECT variable list or not.

The WHERE clause can reference expressions . Expressions are a combination of variables and operators which produce a new value.

The WHERE clause may contain compound conditions connected by the logical operators AND, OR, XOR and NOT.
AND means both expressions must be true;
OR means either expression must be true;
XOR means one expression must be true but not both;
NOT means the expression must not be true.

The WHERE clause is evaluated in the following order of precedence (parentheses ( ) can be used to denote an explicit order of evaluation):

1. Expressions
2. Relational Operators
3. NOT
4. AND
5. OR, XOR

Relational operators

The WHERE clause may include the following operators:

EQ or = or IS
: Equal to

NE or ><
: Not equal to

LT or <
: Less than

LE or <=
: Less than or equal to

GT or >
: Greater than

GE or >=
: Greater than or equal to

BETWEEN expr AND expr
: Between or equal to the values of two expressions

IN (expr,.....)
: Equal to one of the values in a list

LIKE

Matches a specified character pattern

`EQ NULL`

Is missing

`NE NULL`

Is not missing

The logical operator `NOT` can be used to test for the opposite of any condition.

EQ, NE, LT, LE, GT, GE & BETWEEN

These operators test the relationship between two values. If the specified condition is true, the data is selected. `NE` is provided as a convenient shorthand; it is identical to `NOT EQ`.

`BETWEEN` means equal to the end values or any value in between.

IN

Selects records or rows when the value that matches one or more values in a list. For example, to select data for employees 1, 5, and 7.

```
SELECT ...  FROM EMPLOYEE WHERE ID IN (1,5,7)
```

To select all records except those in the list use `NOT`:

```
SELECT .... WHERE  NOT ( ID IN (1, 5, 7))
```

LIKE Pattern Matching

A pattern is a partial string where symbols are used to indicate how that position is to be treated. The pattern consists of symbols plus the string you want to match.

Subqueries

Subqueries are used to select rows from a table based on data in other rows. The rows returned by one SELECT statement are used in the WHERE clause of another SELECT statement. The subquery executes first and returns one or more values which are then used by the main SELECT as if it were given a set of constant values. For example, to select the name, gender and education of all employees who have the same education as Mary Black.

```
SELECT NAME GENDER EDUC FROM EMPLOYEE -
WHERE EDUC = -
(SELECT EDUC FROM EMPLOYEE WHERE NAME = 'Mary Black')
```

The subquery (the one enclosed in parentheses) returns the value of Mary Black's education. This value is then used as the object of the WHERE clause for the main SELECT. The information selected by the main SELECT consists of the name and education for all employees with the same education as Mary. This set of employees naturally includes Mary.

The data comes from multiple cases and case structure must be off (CLEAR CASE) for this query to operate as required. With a case structured database and subqueries which retrieve data from one case which is used to SELECT other cases, CLEAR CASE mode.

Subqueries can be used wherever a WHERE can be specified.

The previous example shows the most basic use of a subquery - one that returns a single value. If a subquery can return more than one value, specify how the returned values are treated in the WHERE clause with the IN, ANY and ALL functions. ANY tests against any returned value; ALL tests against every returned value. These can be used with the relational operators (EQ, NE, LT, GT, LE, GE). IN tests a value to be equal to a value in the list of returned values and is equivalent to EQ ANY.

Where a subquery can return more than one value, this is equivalent to a list. For example, to select people whose salary is greater than anyone whose current position is in division 1:

```
SELECT ID NAME SALARY FROM EMPLOYEE -
WHERE SALARY GT ALL  -
(SELECT SALARY FROM EMPLOYEE OCCUP -
WHERE CURRPOS EQ POSITION AND DIVISION EQ 1)
```

The subquery finds the division in the OCCUP record which matches the current position and tests to be in division 1; the salary of all employees where this is true is retrieved and these are now equivalent to a list of salaries. The salary for each employee is tested against this list to be greater than all entries in the list. A subquery must only return one column to be used to construct the list.

A `WHERE` clause can contain a combination of conditions and subqueries. Any subquery either returns a single value or a list and can be treated as equivalent to a value or list specified by expressions which are not subqueries. The logical operators (`AND, OR, XOR, NOT`) are used to connect separate clauses in the `WHERE` expression.

The logical function `EXISTS` tests that the subquery returns at least one row. `EXISTS` returns "True" if at least one row exists, "False" if not. The test can be reversed with the `NOT` logical operator. For example:

```
SELECT ID NAME SALARY FROM 1 WHERE -
EXISTS (SELECT * FROM REVIEW WHERE RATING = 5)
```

When using the `EXISTS` function, the column returned by the subquery is irrelevant and must be specified as an asterisk (*).

Note: This query could have been performed more easily and more efficiently as a simple join although this would return multiple rows for people who had received multiple ratings of 5:

```
SELECT ID NAME SALARY FROM EMPLOYEE REVIEW -
          WHERE RATING EQ 5
```

Display

You can view the data in the current table created by SELECT. If you choose to display the data or `AUTODISPLAY` is `ON`, the table is listed.

The whole table is listed in your scrolled buffer. This has a limited size (64k bytes) but, if session logging is turned on, the output from the session is also written to the session log (SirSQL.slg).

Lines are as wide as necessary to hold all columns and no paging is done.

If you choose to alter the appearance of your table with the column formatting commands, re-issue the display command to see the altered display. The `DISPLAY` command displays a table and sets the table as the most recently displayed table. This table is then altered by any display commands. It is not necessary to do a `SELECT` command from a table before displaying it. If a specific table is not specified on the `DISPLAY` command, the last table used to store the result of a `SELECT` is displayed.

(N.B. Re-issuing a DISPLAY command re-displays the last table created by a SELECT not the last table named on a previous DISPLAY command. If you have not just done a SELECT, and do not specify a table name, you may display the result of a table created by the system as part of running the menus.)

There are commands for formatting the currently displayed table. Any format changes are held as part of the table and are permanent changes. Formatting options include:

- including or excluding columns
- changing the display format of columns
- specifying headings or footings
- specifying totals and sub-totals

There are two commands to create output files:

> The PRINT command writes a file for subsequent printing. (Use an appropriate operating system command to output the file on a printer.)
>
> The WRITE command outputs a file without headings, which may be easier if it is to be used as input to another program.

EXCLUDE and INCLUDE

`EXCLUDE column_list`
Specifies columns to exclude from the display. `ALL BUT` specifies columns to include. As many column names or numbers can be specified in the column_list as necessary. Excluded columns can be brought back with the `INCLUDE` command. Abbreviation: `EXCL`

`INCLUDE column_list`
Includes columns that were previously excluded with the `EXCLUDE` command. If no columns are specified, all excluded columns are included. `ONLY` specifies that the named columns are included in the display and that all other columns are excluded. Abbreviation: `INCL`

Formatting Commands

The formatting options specify the appearance of the displayed table. (These are identical to column options on `FORMAT` clauses on a `SELECT` command.

You can specify the command as `FORMAT COLUMN` or just `COLUMN` followed by the column references. For example, suppose a table had been created with two columns `SALARY` and `NAME`. The following display commands alter the width of these:

```
COLUMN  SALARY   WIDTH 8
COLUMN  NAME     WIDTH 30
```

The following `SELECT` achieves the same results as the table is created:

```
SELECT SALARY NAME FROM EMPLOYEE -
       FORMAT COLUMN SALARY WIDTH 8 -
       FORMAT COLUMN NAME   WIDTH 30
```

Headings and Footings

The following commands set the titles and headings of the overall report.

```
HEADING [ RIGHT | CENTER | LEFT ] heading_specifications
FOOTING [ RIGHT | CENTER | LEFT ] footing_specifications
```

Specifies the heading or footing of the report. By default, the first 50 characters of the `SELECT` statement are used as the left-justified heading.

The text can be `RIGHT` justified, placed in the `CENTER` of the report or `LEFT` justified. `LEFT` is the default.

The heading or footing can have multiple character strings plus three predefined names and positioning characters. The predefined names are `DATE, TIME` and `PAGE`. The positioning characters are "X", "T" and "/":

`DATE`
: Specifies the current date in the heading or footing. The date format is determined by the setting of the system parameter `DATE`.

`TIME`
: Specifies the current time in the heading or footing. The time format is determined by the setting of the system parameter `TIME`.

`PAGE`
: Specifies the page number in the heading or footing. The page number is 1.

`nX`
: Skips **n** columns before the next print position.

`nT`
: Tabs to a particular column for the next print position.

`/`
: Skips to a new line.

Example: Heading specification

```
HEADING CENTER 'Report 1'  25T 'Produced on' 2X DATE 2X PAGE
```

There are two other commands which can also be used to alter the headings. These are:

```
BTITLE  'footing string' | ERASE
```

Sets the report footing to the specified string. The `ERASE` option clears the footing line. There is no default footing. Abbreviation: `BTI`

```
TTITLE  'heading string' | ERASE
```

Sets the report heading to the specified string. The `ERASE` option resets the heading back to the default . Abbreviation: `TT`

Grouping and Totalling

Grouping and totalling is controlled by a number of commands:

- `BREAK` defines the columns to be used for breakpoints.
- `GROUP` can be used as an alternative syntax to `BREAK` offering the same functionality.
- `OFF` and `ON` turn all totalling off, and back on again, without altering the definition of any of the totals.
- `SUBTOTAL` and `TOTAL` define the type of subtotalling and totalling to be performed.

BREAK

```
BREAK column, ....[([break_heading,] [C,] [G,] [L,] [P,])],....
```

Specify BREAK (abbreviation `BRE`) to control subtotalling. When a column is designated as a break column, a break occurs for each new value in the column. The table should have been produced sorted on the break columns in order to get meaningful results.

One line of subtotals is produced for a given break level with one value for any given numeric column. By default, a total (Sum) is produced for each numeric column at each break level. Specify the type of total with the `SUBTOTAL` and `TOTAL` commands. Grand totals are also produced.

`BREAK` clears any previous break settings, subtotalling, and totalling. It sets the specified columns as breaks in the order specified, major to minor.

Optionally specify a break heading and the codes `C, G, L` and `P`. Enclose the options for a column or a column list in parentheses and separate multiple options with a comma. An option applies to the preceding columns. Specify a minus sign "-" in front of an option to turn off that option if it is the default.

`break heading`
: Specifies a character string to display at the break point. The default is either the type of subtotal being produced, (Count, Sum, Max, etc.) or the column name. Column name is used if different statistics for different numeric columns at the same break level are specified.

`C`
: Specifies that the **C**olumn headings are re-printed when this column breaks. This is the default.

`G`

Specifies that the column is **G**rouped. When a column is grouped, the first line after a break contains the new value and all subsequent lines are blank until the next break. This is the default.

`L`

Specifies that the subtotals are displayed in the **L**eft margin.

`P`

Specifies that a **P**age eject is done when this column breaks.

```
BREAK educ ('Salary total',-C,L,P)
```

GROUP

```
GROUP [ EXCEPT ] [ ERASE ] ( column-list )
```

`GROUP`

Performs the same functions as `BREAK` with different syntax. The combination of `BREAK` and `SUBTOTAL` and of `GROUP` and `SUBTOTAL` can be used to provide identical functionality. `GROUP` is provided for compatibility with the SQL standard.

`EXCEPT`

Specifies that the break level grouping is removed on the specified columns.

`ERASE`

Specifies that the break level grouping is removed from all columns.

OFF | ON

```
OFF | ON
```

`OFF`

Suspends all grouping, subtotalling and totalling.

`ON`

Re-enables all grouping, subtotalling and totalling.

SUBTOTAL

```
SUBTOTAL [ COUNT | MAX | AVG | MIN | STD | SUM ]
         [ break column, ...  ] [(subtotal column, ...)]
         [ EXCEPT  ( column, ...  ) ]
         [ ERASE ]
```

SUBTOTAL (abbreviation STOT) sets subtotals for all numeric columns or for specified columns for a specified break column.

The settings for SUBTOTAL apply to the calculation and display of grand totals unless altered with the TOTAL command.

See the SET SPACES and SET SPACED system parameters for control of spacing around SUBTOTAL lines.

The type of subtotalling can be specified. There can only be one type of subtotalling on a given column for a break level. The options are:

COUNT
: A count of the non-missing values in the column. This can be used on non-numeric columns as well as numeric.

MAX
: The maximum value found in the column. This can be used on non-numeric columns as well as numeric.

AVG
: The average value for non-missing occurrences in the column. This applies to numeric columns only.

MIN
: The minimum value found in the column. This can be used on non-numeric columns as well as numeric.

STD
: The standard deviation of the column. This applies to numeric columns only.

SUM
: The total of values in the column. This applies to numeric columns only. This is the default subtotal.

Break and Subtotal Columns

Two types of columns can be specified on a `SUBTOTAL` command, and two formats are used to differentiate these. First, the break column(s) for which the subtotals are produced. Break columns are simply listed. Second, the column(s) that are to be subtotalled. Enclose the columns to be subtotalled in parentheses.

`EXCEPT`
: By default, subtotalling is on for all numeric columns. `EXCEPT` sets subtotalling off for specified columns. `EXCEPT` is cumulative. Set subtotalling back on for a column by specifying it as a subtotal column. `EXCEPT` applies to all break columns and cannot be specified for individual break columns.

```
SUBTOTAL EXCEPT (ID)
```

`ERASE`
: Sets subtotalling off for all columns.

Examples:

To display averages for all numeric columns when the break column `EDUC` changes:

```
SUBTOTAL AVG EDUC
```

To display the maximum of salary on any break:

```
SUBTOTAL MAX (SALARY)
```

To display the minimum of `SALARY` when the break column `EDUC` changes:

```
SUBTOTAL MIN EDUC (SALARY)
```

TOTAL

```
TOTAL    [ COUNT | MAX | AVG | MIN | STD | SUM ]
         [ total column, ...]
         [EXCEPT  ( column, ...  ) ]
         [ ERASE ]
```

`TOTAL` (abbreviation `TOT`) sets totals for all columns or for specified columns. Name the column(s) to total. Either enclose the column specifications in parentheses or simply list the columns. Use the `SET SPACET` system parameters to control spacing of total lines.

`COUNT`
: A count of the non-missing values in the column. This can be used on non-numeric columns as well as numeric.

`MAX`
: The maximum value found in the column. This can be used on non-numeric columns as well as numeric.

`AVG`
: The average value for non-missing occurrences in the column. This applies to numeric columns only.

`MIN`
: The minimum value found in the column. This can be used on non-numeric columns as well as numeric.

`STD`
: The standard deviation of the column. This applies to numeric columns only.

`SUM`
: The total of values in the column. This applies to numeric columns only. There can be one type of totalling on a given column. `SUM` is the default totalling.

`EXCEPT`
: Totalling is on for all numeric columns. `EXCEPT` sets totalling off for specified columns and resets any previous specification.

`ERASE`
: Sets totalling off for all columns.

Examples:

To produce the average salary from all records in the table:

```
TOTAL AVG SALARY
```

To produce totals for all numeric columns except `CURRPOS`:

```
TOTAL EXCEPT (CURRPOS)
```

PRINT and WRITE

```
PRINT
         [ OUTPUT filename ]
         [ LOWER | UPPER ]

WRITE
         [ OUTPUT filename ]
         [ LOWER | UPPER ]
```

These commands write a copy of the current displayed table to a file. `PRINT` writes a formatted file with column headings, underlines, blank lines, etc. This is more suitable for printing.

`WRITE` writes an unformatted file with a space between each column and displays a list of column positions used for each column. This may be more suitable as input to another program.

Use the Include/Exclude commands to vary the columns in the output.

`OUTPUT`

Abbreviation: `OUT`. Specifies the output filename. If `OUTPUT` is not specified, the current setting for the system parameter OUTPUT is used. An output file must be specified if a default has not been set.

```
PRINT OUTPUT 'TEST.LST'
```

`LOWER | UPPER`

`Abbreviations: LC, LOWERCASE`. `LOWER` specifies that the output uses upper and lowercase characters. This is the default. UPPER specifies that the output maps all characters to uppercase.

Setting Parameters

SirSQL contains a set of parameters, each identified by a name which affect the operation of SQL in a variety of ways. At the start of an SQL session, each parameter is set to a default value. These values can be updated and saved in a workspace file which can be re-used in subsequent sessions. Three commands control and display parameters:

`SET`
: Sets the parameter either to a specified value or simply to be `ON`.

`CLEAR`
: Resets the parameter either to the default or to be `OFF`.

`SHOW`
: Displays the current setting of the parameter.

Specify the name of the parameter and, optionally, any values with these commands. Certain parameters can also be set by options on the SQL execution statement.

The values of some settings are used to set column formats when a table is created. Altering these settings has no effect on existing tables. You can update the format of columns in existing tables with the column command.

Parameter List

`AUTODISP`

Controls whether display mode is automatically entered after a `AUTOSAVE`
Only used from the menu system. Controls whether a SAVE command is automatically executed when exiting the system.
Abbreviation: `AUTOD`
Default: `CLEAR`

`CASE`

Enables use of the database case structure during retrieval processing. When Case is `SET`, there is an implicit CIR record type added to the records in the `FROM` list of a `SELECT`. Joins and aggregations are done within each case rather than over all the records in the database. There is an implied *Join* by caseid. When aggregations are performed, there is an implied `GROUP BY` caseid clause on the `SELECT` specification. The setting of case has no effect on caseless databases.
`CLEAR` disables use of the database case structure during retrieval processing. In join operations, records are joined across all records in the database. System and user created paths may join records within case even with cases cleared. Aggregations occur within records or tables.
Default: `SET`

`CASELIM n`

Specifies a limit on processing cases in a `SELECT`. When the limit is reached, the process stops and prompts to continue with three options; continue and receive a further warning after another `CASELIM` number of rows are created; continue with a different `CASELIM`; stop the `SELECT`.
`CLEAR` disables the check.
Abbreviation: `CLIM`
Default: 1000

`CMPTRIM`

Specifies that trailing blanks are trimmed prior to string comparisons.
Default: `SET`

`CMPUPPER`

Specifies letters are translated to uppercase prior to string comparisons.
Default: `SET`

`COLHEAD`

Sets the printing of column headings at break points in the output of the `DISPLAY` command.
Default: `SET`

`CONTINUE '-'`

Sets the character to indicate a command is continued on the next line to the specified character. There must always be a continuation character, `CONTINUE` cannot be `CLEAR`ed though it may be set to ' ' (a blank).
Abbreviation: `CONT`
Default: A hyphen '`-`'.

`DATABASE`

Sets the default database if more than one is connected.
Default: The last database connected.

`DATE 'format'`

Sets the default date format. Any date column without a specific format, uses this format as a default.
`CLEAR` clears the default date map. If the column comes from a database, the format described in the schema is used as a default.
Default: '`Mmm DD, YYYY`'.

`DEFINE_SECURITY`

Specifies that tables created with `SELECT` have security definitions. When connecting to a tabfile with security definitions, specify group(.user) names to connect to the tabfile. Anyone with DBA permission at the tabfile level, has full permissions for all tables. With `DEFINE_SECURITY` set, you have full permissions for the new table created with `SELECT` but any other groups(.users) (except DBAs) have no permissions on that table. You (or a DBA) can grant permissions to other users. (see Permissions)
`CLEAR` specifies that created tables are Public access.
Default: `SET`

`DETAIL`

Controls whether detail lines are displayed when subtotalling is requested with the `BREAK` or `GROUP` commands. `CLEAR` suppresses the printing of the detail lines. Subtotal and total lines are printed.
Abbreviation: `DET, DTL`
Default: `SET`

`DOUBLE`

Displays a blank line between rows. `CLEAR` specifies that rows are displayed single-spaced, that is with no blank lines between them.
Abbreviation: `DBL`
Default: `CLEAR`

`DPLACES n`

Sets the number of decimal places used in printing decimal numbers to n. `CLEAR` resets to the default number of decimal places.
Abbreviation: `DPL`
Default: 2 decimal places.

`ECHO`

Specifies that any input from an alternate input file or from a call procedure is echoed to the screen.
Default: `CLEAR`

`EXEC`

Specifies that `SELECT` commands are translated and executed after the last line of the statement is entered. `CLEAR` prompts to translate and then execute the query.
Default: `SET`

`EXPONENT n`

Specifies that the non-integer numeric values are displayed in scientific notation. **n** is the number of decimals to display and zero (0) is used to indicate that values are displayed in normal decimal notation.
`CLEAR` specifies that decimal notation is used.
Default: `CLEAR`

`FAMILY family_name[/password]`

Specifies the default family and password for procedure file references. `CLEAR` clears the default family name and password.
Abbreviation: `FAM`
Default: Last family referenced

`GROUPING`

Specifies that if break levels are set, the break column value is printed the first time that the value changes. For the subsequent lines the break column value is suppressed. `CLEAR` specifies that if break levels are displayed, the break column values are printed on every line.

Abbreviation: GRP
Default: SET

GRPSIZE n

Specifies the size of the internal buffer used by the GROUP BY option on SELECT. If the number of different values grouped by, is less than or equal to this value, all processing is done in memory. If the number of entries exceeds this value, a disk sort is performed. CLEAR specifies that the GROUP BY in-core sort size is 1.
Default: 256.

INPUT filename [NOABORT]

Specifies that input commands are read from the specified file instead of interactively. This continues until an END command is encountered or end-of-file is detected. This is normally used for batch runs and is set as a parameter on the execution statement. Use this command interactively to run stored command sequences.

If an error occurs on an input file, the processing of that file stops at that point. NOABORT specifies that the command in error is written to the LOG file, and processing continues.

This can be used to import an SQL export file.
Default: Commands are read interactively.

LABEL

Specifies that column headings and break titles are labels instead of names. CLEAR specifies that variable names are used for column headings and break titles.
Abbreviation: LAB
Default: CLEAR

LOWER

Specifies that the report produced by the PRINT command has mixed uppercase and lowercase characters. This is identical to CLEAR UPPER . CLEAR LOWER specifies that the report produced by the PRINT command is in uppercase only and lowercase characters are translated to uppercase. This is identical to SET UPPER.
Abbreviation: LC
Default: SET

MASTER 'master_ip[:port]'

Sets the use of Master. If a name is not specified, sets the Master to be the previously used Master.

Equivalent execution statement parameter: MST=name
Default: Last master referenced

`MEMBER member_name [/password]`

Specifies the default member name and password. `CLEAR` specifies that there is no default member.
Abbreviation: `MEM`
Default: Last member referenced

`MISSCHAR 'c'`

Specifies the character used to display a column when it contains missing values. Specify one character which fills the column. See also `SET NULL`. `CLEAR` sets the character to blank. This is equivalent to specifying `SET MISSCHAR ' '`.
Abbreviation: `MISS`
Default: Asterisk `'*'`

`NULL 'string'`

Specifies a string displayed when a field contains missing values. `NULL` takes precedence over `MISSCHAR`. `CLEAR` clears the parameter; fields with missing values are filled with the `MISSCHAR` character.
Default: `CLEAR`

`OUTPUT filename`

Specifies the default print file. `CLEAR` specifies that there is no default output file and must be named on the `OUTPUT` clause on `PRINT` or `WRITE` commands.
Abbreviation: `OUT`
Default: `CLEAR`.

`PATH`

Paths are automatically used by `SELECT`. `CLEAR` specifies that paths are not automatically used by `SELECT` and you must name a path explicitly in a `SELECT` to use it.
`SHOW PATH [ path_name ]`
As well as showing the current path parameter setting, `SHOW PATH` shows the definitions of all paths or of the pathname specified.
Default: `SET`

`RECLIM n`

Specifies a limit on the number of records read by a `SELECT`. When the limit is reached, the process stops and prompts whether to continue. There are three options: continuing and receiving a further warning after another `RECLIM` number

of records is processed, continuing with a different `RECLIM`, and stopping the `SELECT` at that point. `CLEAR` disables the record limit option.
Abbreviation: `RLIM`
Default: 1000

`SELLIM n`

Specifies a limit on the number of new rows selected by a `SELECT`. When this number of new rows is reached, the process stops and prompts whether to continue with three options; continue and receive a further warning after another `SELLIM` number of rows are created; continue with a different `SELLIM`; stop the `SELECT`. `CLEAR` disables the select limit on new rows.
Abbreviation: `SLIM`
Default: 1000

`SELSIZE n`

Specifies the size of the internal buffer used for holding of the results of a sub-query on `SELECT`. If the number of different values selected in the sub-query, is less than or equal to this value, all processing is done in memory. If the number of entries exceeds this value, disk I/O is performed. This parameter controls the number of entries that are kept in memory. `CLEAR` specifies that the in-core size is 1.
Default: 256.

`SEPARATE 'string'`

Specifies the string used to separate columns on the display. `CLEAR` specifies that columns are not separated.
Abbreviation: `SEP`
Default: Two blanks.

`SINGLE`

Detail lines are single spaced on a display. There are no blank lines between consecutive detail lines. This parameter is changed by `SET DOUBLE` or `SET TRIPLE`. To see the value of all line spacing parameters, use `SHOW SPACING`.
Abbreviation: `SGL`
Default: `SET`.

`SPACEC n m`

Specifies that there are **n** blank lines between the page heading and the column headings and **m** blank lines between the column headings and the underline beneath the column headings on displays. `CLEAR` specifies that there are no blank lines.
Default: n = 1 , m = 0

`SPACED n m`

Specifies that there are **n** blank lines between the column heading underlining and the detail lines and **m** blank lines between the detail lines and the underline prior to the subtotals on displays. `CLEAR` specifies that there are no blank lines.
Default: `CLEAR`

`SPACES n m`

Specifies that there are **n** blank lines between the underline under the detail lines and the subtotal line and **m** blank lines between the subtotal lines and the next column headings on displays. `CLEAR` specifies that there are no blank lines.
Default: n = 0 ,m = 1

`SPACET n m`

Specifies that there are **n** blank lines between the subtotal lines and the grand total underline and **m** blank lines between the grand total underline and the grand total line on displays. `CLEAR` specifies that there are no blank lines.
Default: n = 0 , m = 1

`SRTSIZE n`

Specifies the number of entries in to sort. Sorting is done with the `ORDER BY` clause on `SELECT` for detail rows. `CLEAR` sets the sort size to the default of 1000.

`STATS`

Specifies that statistics are displayed automatically at the end of execution of a `SELECT`. These same statistics are displayed whenever a `SELECT` reaches a `CASELIM, RECLIM` or `SELLIM` and prompts for a decision to continue. `CLEAR` suppresses the display of statistics after a `SELECT`.
Default: `CLEAR`

`SUBTOTAL`

Specifies that subtotals are displayed when a `BREAK` or `GROUP` command is given for a report. `CLEAR` specifies that subtotals are not automatically produced for breaks or groups for a report.
Abbreviation: `STOT`
Default: `SET`

`TABFILE name`

Sets the default tabfile. `CONNECT`ing to a tabfile sets that tabfile to the default.
Abbreviation: `TF`
Default: `$SYSTEM`

`TABSIZE n`

A synonym for `SELSIZE`. The use of a sub-query on `SELECT` requires the holding of the results of the sub-query. This parameter controls the size of the internal buffer. If the number of different values selected in the sub-query, is less than or equal to this value, all processing is done in memory. If the number of entries exceeds this value, a disk I/O is performed. This parameter controls the number of entries that are kept in memory. `CLEAR` specifies that the in-core size is 1.
Default: 256.

`TIME 'format'`

Sets the default time format. `CLEAR` specifies that system default time format is used.
Default: '`HH:MM:SS`'

`TOTALS 'string'`

Specifies that totals are produced for displays with a break or group option. The `'string'` is a label displayed on the total line in the left margin. If not specified, the default label is `TOTAL`. `CLEAR` disables the total line display.
Abbreviations: `TOT, TOTAL`
Default: `CLEAR`

`TRANSFER_VALLAB`

Specifies whether value labels are transferred to the new table created by select wherever they exist. `CLEAR` specifies that value labels are not transferred to the new table.
Default: `SET`

`TRIPLE`

Specifies that the detail lines are displayed triple spaced, with two blank lines between each detail line. This parameter is altered with `SET SINGLE` and `SET DOUBLE`. To see the value of this parameter, use `SHOW SPACING`.
Abbreviation: `TPL`
Default: Detail lines are single spaced.

`UNDERCOL`

Specifies that the column headings are underlined. `CLEAR` specifies no underlining.
Abbreviations: `UCOL, UNDCOL`
Default: `SET`

`UNDHEAD`

Specifies that an underline is printed before a subtotal. `CLEAR` specifies no underlining.
Abbreviation: `UHEAD`
Default: `SET`

`UPPER`

Specifies that a report produced by `PRINT` is in uppercase. (Identical to `CLEAR LOWER`.)
`CLEAR UPPER` specifies that reports are in mixed lower and upper case. (Identical to `SET LOWER`.)
Abbreviation: `UC`
Default: `CLEAR`

`VALLAB`

Specifies that value labels are used in place of values, for columns with value labels defined. `CLEAR` specifies that values are used regardless of whether value labels exist.
Default: `CLEAR`

`VARCHAR n`

Specifies the display size for character columns. This does not effect the field size created on tables by `SELECT`. Field size is determined by the definition of the column or expression. `CLEAR` sets the display width of character fields to zero; they are not displayed.
Abbreviation: `VCHR`
Default: 20

`WIDTH {n | LABEL | MIN}`

Sets the width of output columns. This can be the specified number of characters (n); the width necessary to accommodate the column and value labels (`LABEL`); or, for numeric fields, to the minimum number of characters necessary to accommodate the widest number in the column (`MIN`). The `MIN` option can only be used at the table level in the display processor.
`CLEAR` sets the column widths to 8 for numeric data and to the string length specified in the schema for strings.
Default: `CLEAR`

`WORK filename [PASSWORD password]`

Specifies the default workspace for `SAVE` and `GET`. `CLEAR` specifies that there is no

default workspace.
Default: `SirSQL.wsp`

`ZEROS ON | OFF | 'string'`

Specifies the printing of leading zeros for numbers. `ON` means leading zeros are printed; `OFF` means they are suppressed. Also specifies a string to be printed if the value is zero. `CLEAR` is the same as `SET ZEROS OFF`.
Abbreviation: `ZERO`
Default: `CLEAR`

Control Commands

There are a number of ancillary commands which execute procedures, connect databases and tabfiles, etc. Most of these operations can be carried out more easily from the menus. The commands are as follows:

CALL

Executes commands stored as a procedure

CONNECT DATABASE

Connects databases

CONNECT TABFILE

Connects tabfiles

CREATE ATTRIBUTE

Creates a short internal name to represent a filename

CREATE SYNONYM

Defines a short word to stand for longer text

DISCONNECT

Disconnects a database or tabfile

DROP

Deletes a tabfile or elements such as paths or views which have been created

END

terminates session

GET

Restores the workspace from a file

SAVE

Saves the workspace to a file

CALL

```
CALL [database.] [family.] member [(parameter, ...)]
```

`CALL` executes a procedure from a SIR/XS procedure file. The procedure file must be connected. A procedure (or member) may be edited, amended and added to the procedure file from the SQL menus.

A procedure that is called in SQL should only contain SQL commands.

Procedures can be created which have parameters to specify particular conditions. Parameters are positional; that is, the stored procedure references parameters by number and these numbers are assigned in the order they are specified. A procedure parameter can be any sequence of text. Parameters are enclosed in parentheses and separated by commas. Null parameters are specified by a comma immediately following the previous comma.

- **Example:** Suppose the following were in the text of procedure `REP1`:

```
SELECT ID NAME <1> <2> FROM EMPLOYEE <3>
```

Call this procedure with:

```
CALL REP1 ( SALARY , GENDER , WHERE ( SALARY GT 2250 ) )
```

which results in the full command:

```
SELECT ID NAME SALARY GENDER FROM EMPLOYEE -
 WHERE ( SALARY GT 2250 )
```

CONNECT DATABASE

```
CONNECT DATABASE database_name
         [ PASSWORD  database_password ]
         [ PREFIX 'file_prefix' ]
         [ SECURITY  read_security,write_security ]
         [ SUPPRESS PATH ]
         [ SUPPRESS QPROFILE]
```

Abbreviation: `CON DB` Connects the named database. Supply the appropriate password and security passwords. If the database is not in the default directory, supply a prefix. Enclose the prefix in single quotes.

This database is made the default database. If other databases are connected, the `SET DATABASE` command alters the default. When using multiple databases, prefix non-unique record names with the database name. For example, in a database called `COMPANY` with a record named `EMPLOYEE`:

```
SELECT ....  FROM   COMPANY.EMPLOYEE
```

`PASSWORD`

Specifies the database password.
Abbreviation: `PW`

`PREFIX`

Specifies a directory other than the current default directory for database access. The directory used for any other files is unaffected. Specify the prefix in single quotes.
Abbreviation: `P`

For example:

```
PREFIX 'C:\MYFILES\'
```

`SUPPRESS PATH`

Suppresses the generation of paths. Paths are automatically generated when a SIR/XS database is connected. Turn off the use of paths with the `CLEAR PATH`

command. Disconnect and reconnect without suppressing paths to generate the paths.

`SUPPRESS QPROFILE`

Suppresses automatic execution of `SYSTEM.QPROFILE`. When a database is connected, SQL automatically executes this procedure on that database. The procedure can contain any SQL commands executed each time this database is connected. If the procedure does not exist, no member is executed.

`SECURITY`

Specifies the read and write security passwords for the database. Specify the read password first, then a comma, then the write password. If there is a write password but no read password, precede the write password with a comma.

```
CONNECT DATABASE ......  SECURITY HIGH,HIGH
```

CONNECT TABFILE

```
CONNECT TABFILE  name
    [ AUTO | READ | WRITE ]
    [ FILENAME  filename ]
    [ IDENT BY grpname/grppass.username/userpass ]
```

Abbreviation: `CON TF`

Connects the named, existing tabfile. The name used must be the name specified when the tabfile was created. This is an internal name and is independent of, and unaffected by, the operating system filename. All references to the tabfile are by this name. Tabfiles with the same name cannot be connected at the same time.

`AUTO`

> Specifies that the tabfile is opened and closed every time an SQL operation, such as a `SELECT`, is executed. This locks the tabfile for exclusive write for the minimum period of time. `AUTO` is the default. Specify `WRITE` for exclusive updating or `READ` if only doing queries.

`READ`

> Specifies that the tabfile is opened and remains open for shared read-only access until `DISCONNECT`ed.

`WRITE`

> Specifies that the tabfile is opened and remains open for exclusive write operations until `DISCONNECT`ed.

`FILENAME`

> Specifies the operating system filename for the tabfile. If a filename is not specified, the internal name of the tabfile plus ".tbf" is used as the operating system filename.

`IDENT BY`

> Specifies group name, group password, user name, and user password for access to this tabfile. If security controls were not defined when the tabfile was created,

these clauses are unnecessary. Depending on the access controls, you may need to specify a password for the group, a username and a password for the username.

CREATE ATTRIBUTE

```
CREATE ATTRIBUTE  name  FILENAME  'external filename'
```

Abbreviation: `CRE ATTRIB`

Associates an SQL internal name with an operating system filename specified in quotes. The SQL name can be used in subsequent commands wherever you need to specify a filename.

CREATE SYNONYM

```
CREATE SYNONYM  name text
```

Abbreviation : `CRE SYN` Creates a synonym. A synonym is a text replacement mechanism, typically used for long, repeated sets of names. Create the synonym and then use it wherever applicable. A synonym can be used at any point in any SQL command.

Do not enclose the text in quotes. The synonym name is a standard SQL name up to 32 characters. Do not use reserved words as synonym names.

```
CREATE SYNONYM MYSELECT SELECT ID NAME SALARY FROM EMPLOYEE
```

DISCONNECT

```
DISCONNECT DATABASE  database_name
DISCONNECT TABFILE   tablefile_name
```

Abbreviation: `DISCON DB, DIS DB, DISCON TF, DIS TF` Disconnects a database or tabfile. The database or tabfile is closed and all schema information is released.

DROP

```
DROP ATTRIB  attribute_name
DROP INDEX   index_name
DROP JOURNAL file_name
DROP PATH    path_name
DROP SYNONYM synonym_name
DROP TABFILE tabfile_name
DROP TABLE   table_name
DROP VIEW    view_name
```

Deletes the named entity. The entity no longer exists and must be re-created if required again.

END

```
END  [ CLEAR | SAVE [ workspace_filename ] ]
Synonyms: BYE, EXIT, QUIT, STOP, .
```

Terminates the SQL session from SQL.

If the workspace has been modified, you are prompted to save it. It will be saved as the default workspace file which is either the workspace file that has been restored (with a `GET`, the workspace file most recently saved or the default `SirSQL.wsp`. If the file is new, you are prompted for a password. If you specify a password, a user must specify the password when restoring the workspace. If you do not specify a password, you will not be prompted for it again.

GET

```
GET  [ filename ] [ PASSWORD  password ]
```

Restores a workspace from the specified file name. When starting an SQL session, the default workspace (SirSQL.wsp) is loaded unless the `WORK =` execution parameter is specified.

Use `GET` to load a previously saved workspace. The default workspace (set by `SET WORK` name), is loaded if a name is not specified on the `GET` command.

Specify the keyword `PASSWORD` and the required password if one is needed. If the password is not specified and one is required, you are prompted for it.

SAVE

```
SAVE workspace_filename  [ PASSWORD workspace_password ]
```

`SAVE` saves the current workspace on the specified file. If the file name is omitted, the workspace is saved on the default workspace file. The default for this is SirSQL.wsp.

Execution Statement

The parameters can be specified on the SQL execution statement. The parameters may be specified on the execution statement in any order and separated by a space. The parameters determine:

Batch Mode

Whether to run SQL in batch mode

Database

Specifies a database to connect automatically

Tabfile

Specifies a tabfile to connect automatically

Environment

Specifies the workspace to restore.

Batch Parameters

```
ABORT
BA
IN  = filename
OUT = filename
```

Use these to run SQL in batch mode.

`ABORT`

Specifies that batch processing is stopped if an error is detected. If `ABORT` is not specified, processing continues after an error. Specify `ABORT` if commands in the input file depend on the results of an earlier command.

`BA`

Specifies Batch Mode. SQL commands are read from an input file specified by the `IN` parameter.

`IN`

Specifies the input file containing SQL commands. Any SQL command can be included in the file. In batch mode `SELECT` does not automatically cause a display. To display the output, include a `DISPLAY` statement and an `END` command.

`OUT`

Specifies an output file for error and status messages.

Format File

SQL messages are held in a format file `sirsql.fmt`. This is a machine readable file produced by a batch run of SQL. This reads a text input file of messages and either creates a new format file or adds messages to a pre-existing file. This facility might be used to produce messages in a different language. Please contact SIR support if you require this facility.

```
PREPARE
FMT = format filename (output)
IN = input format text filename
OUT = output filename (messages)
```

`PREPARE`

Specifies that this is a batch run to prepare a format file.

`FMT`

Specifies the format file to create or added to if it already exists.

`IN`

Specifies the input text of the messages. This is in a predetermined format available from SIR.

`OUT`

Specifies the output file for any messages or errors.

Database Parameters

```
DB = database
P =  'prefix'
PW =  password
RS =  password
WS =  password
EX =  membername
SUPQ
```

Specify the following parameters to connect a default database.

`DB`

Specifies the name of a database to connect and use as the default.

`P`

Specifies a directory other than the current directory is the location of the database. This parameter has no affect on the directory used for any other files. For example: `P='C:\MYFILES\'`

`PW`

Specifies the database password. If a database name is not specified, any passwords are ignored.

`RS`

Specifies the database read security password.

`WS`

Specifies the database write security password.

`EX`

Specifies the procedure executed when the database is connected. The procedure is a member in the family `SYSTEM`. If this member does not exist, no member is executed. If this is not specified, the member `QPROFILE` (if it exists) is executed.

`SUPQ`

Specifies that `QPROFILE` is not executed when the database is connected.

Tabfile Parameters

```
TBL  (or TFL)     =        name
TBFN (or TFFN)    =        filename
GRP               =        name
GPW               =        password
USER              =        name
UPW               =        password
```

Specify the following parameters to connect a default tabfile.

`TFL`

Specifies a tabfile connected when SQL is executed.

`TBFN`

Specifies the operating system filename of the tabfile specified on the `TBL` parameter. If this is not specified, the filename is assumed to be the same as the tabfile name plus a prefix of `.tbf`.

`GRP`

Specifies the group name to access the tabfile.

`GPW`

Specifies the password for the group.

`USER`

Specifies the user name to access the tabfile.

`UPW`

Specifies the password for the user.

Environment Parameters

```
CENY    =       nnnn
WORK    =       workspace
WORKPW  =       password
```

Specify the following parameters as necessary to alter the default environment settings:

`CENY`

> Specifies the four digit year used for century calculations when converting dates with only two digit years. Specify the year below which dates are in the next century. The system setting is 1920. This means that years below 20 are assigned a century of 20xx. Years above this are assigned a century of 19xx. Valid dates run from 1582 to 2900.

`[NO]WORK`

> Specifies the name of the workspace file automatically loaded when SQL starts. If a workspace is not specified on the execution parameter, if `SirSQL.wsp` exists, it is automatically loaded unless `NOWORK` is specified. For example:
>
> ```
> SQL/WORK='SQLMYWS.wsp'
> ```

`WORKPW`

> Specifies the password for the workspace

SirSQL User Interface

SirSQL has a simple and easy to use graphical user interface. The main window contains a menu bar, a toolbar, output and input windows and a status bar.

The input window is the place to type SirSQL commands. You can load a file with SQL commands into this window using the menus or by dropping a file on the main window. You can save the contents of the input area into a text file. You can also store and retrieve sets of SQL commands as procedures (members). To execute the SQL commands from the input window, use the `File/Run` menu or the toolbar or the **Ctrl-R** key.

Results and messages are displayed in the output window. This is a relatively small window and is automatically rebuilt when necessary. This means that earlier output is discarded and lost unless it logged to the `SirSQL.slg` file. Logging depends on the *LogOutput* setting in the initialisation file.

The output window is as wide as necessary to display a complete line of the output. It is treated as a single page with headings at the beginning of a display of a table.

You can select part of the output to save, print or copy to the clipboard. You can use the appropriate INCLUDE/EXCLUDE commands to alter the output display of a table.

The status line displays the information about execution of the last command. If a command was unsuccessful, the status line indicates this and the command is retained in the input window to be edited and re-submitted.

SirSQL provides a set of menus and dialog boxes to perform many common operations. Some popular menu commands are also available on the toolbar.

The main menu consists of the following:

File

- Open
- Save As
- Run
- History
- Clear Output
- Save Output
- Print Output
- Exit

Database

- Connect
- Database
- List
- Members

Tabfile

- Connect
- Disconnect
- List
- Create

SQL

- Interactive Select

Utilities

- Verify
- Export
- Import
- Backup
- Restore

Options

- Settings
- Auto Display
- Save workspace

Help

File Menu

`Open` loads a text file (presumably containing SQL commands) into the input window.

`Save As` saves the contents of the input area into a text file.

`Run` executes the SQL commands from the input area.

`History` displays any previously entered commands and allows you to re-run them or load them into the input area for editing.

`Clear Output` clears the output area.

`Save Output` saves the output area as a named text file.

`Print Output` sends the output area to the current system printer.

`Exit` finishes the SQL session and exits the system. This also saves the workspace.

Connect database

Use this dialog to connect a database. Specify the database by its name. If the database is not located in the current working directory, prefix the database name with the appropriate path.

Specify the main password and read/write security passwords. A database can be protected by write and read passwords, only a write password or not have passwords at all;. it cannot have a read password without a write password. If you specify the read password without a write password, the system uses the same password for both read and write security.

The most recently connected database is selected as the default database.

See the CONNECT DATABASE command.

List of connected databases

Use this dialog to perform basic operations on databases. You can use it to change the default database. Select a database from the list and use the **OK** button to close the dialog. If you close the dialog using the **Cancel** button, the previous default is not changed (if this database is still connected).

You can add databases to the list using the **Connect** button. Other buttons act on the database selected in the list. You can disconnect databases using the **Disconnect** button. Use the **Structure** button to get the information about the tables and fields defined in the database. Use the **Members** button to get the list of families and members (stored procedures) associated with the selected database.

Database structure

Use this dialog to inspect the database structure. Select a table to get the list of its fields. Use the **Field Info** and **Table Info** buttons to get properties of individual tables and fields.

Create tabfile

Use this dialog to create a new tabfile. Specify the tabfile name. A tabfile name is up to 32 characters long and not case sensitive.

Specify the filename explicitly to place the physical file into a directory other than your current working directory or to have the file name other than the name of the tabfile suffixed by the `.tbf` extension.

Specify the journal file name if you want to have a journal for the tabfile. Specify group and user passwords if you need to control access to this tabfile.

Only specify block size when necessary.

The tabfile is created, automatically connected and selected as the default.

See the CREATE TABFILE command.

Connect tabfile

Use this dialog to connect a tabfile. Specify the tabfile by name. Specify the filename if the tabfile is not in the current working directory, or where the filename is not the tabfile name with a `.tbf` suffix. You can use the browse button to locate the file.

Specify group and user IDs and passwords if the tabfile is password protected.

The latest tabfile connected is set to be the default.

See CONNECT TABFILE command.

List of connected tabfiles

Use this dialog to perform basic operations on tabfiles.

You can change the default tabfile. Select a tabfile from the list and use the **OK** button to close the dialog. If you close the dialog using the **Cancel** button, the previous default will not be changed (if this tabfile is still connected).

You can add tabfiles to the list using the **Create** and **Connect** buttons. Other buttons act on the tabfile selected in the list. You can disconnect tabfiles using the **Disconnect** button. The **Drop** button disconnects and removes the tabfile.

Use the Structure button to get the information about the tables, fields and indexes defined in the tabfile. This also allows you to Create indexes for the table.

Tabfile structure

Use this dialog to inspect the tabfile structure. Select a table to get the list of its fields. Use the **Field Info** and **Table Info** buttons to get properties of individual tables and fields.

Use the **Drop Table** button to delete the table from the tabfile.

Use the **Indexes** button to inspect/add/remove the table's indexes.

Indexes

Select an index to get the list of its key fields. Use the **Add index** and **Drop Index** buttons to add/remove indexes.

Create index

To create a new index select the desired fields of the table and add them to the list of key fields. Define a name for the new index. Check the **Unique** checkbox if you create a unique index.

See the CREATE INDEX command.

Select

This dialog helps you to construct simple SELECT queries interactively. It does not provide all functionality of the SirSQL SELECT statement, but you can load the resulting statement into the input window and use it as the starting point for writing a more complex query.

The query under construction is displayed in the lower part of the dialog.

Start construction of the query by selecting some tables for the FROM clause on the **From** tab of the dialog. You can then activate other tabs. You can always return to the **From** tab later to modify the list of tables, but you need to have at least one table in this list to enable other tabs.

If you select the fields from a table and later delete this table from the table list, this results in an invalid query as references to fields are not deleted from the query.

Use these links for the information on the individual tabs:

- From
- What
- Where
- Order by
- Group by

See the SELECT command.

From

Use this tab to define the list of tables you want to run the query on. SirSQL allows you to run query on tables from multiple databases and tabfiles. You need to define aliases (use the **Add As** button) if you have the same table name on two databases/tabfiles or need to do a self-joint.

You need to specify at least one table to enable the other tabs. You can always return to this tab and modify the list of tables.

See the FROM clause of the SQL SELECT statement.

What

Use this tab to define the list of variables you're interested in. You can select the variables from any table in the FROM clause.

You can use **Add As** button and type any SQL expression (like aggregate function or arithmetic expression) if what you want to get is not just the value of the field.

Where

Use this tab to write the WHERE clause of your SELECT statement. Basically you just type it in, but you can select the variables from any table in the FROM clause and insert them into the text of your conditions.

See the WHERE clause of the SQL SELECT statement.

Order by

Use this tab to build the list of key fields used to sort the results of the SELECT statement. You can select the variables from any table in the FROM clause and use them as keys for ascending or descending sort.

See the ORDER BY clause of the SQL SELECT statement.

Group by

This tab is useful only when you do select on aggregate functions (you can do it using the **Add As** button on the **What** tab). Use the **Group by** tab to build the list of key fields used to group the results of the SELECT statement. Additionally you can write the conditions for the HAVING clause very much like for the WHERE clause. You can select the variables from any table in the FROM clause and insert them into the text of your conditions.

See the GROUP BY and HAVING clauses of the SQL SELECT statement.

Verify tabfile

Specify the tabfile to be verified. It doesn't need to be connected. You need to specify the file name if the tabfile uses non-standard name or located not in the current working directory. You can use the browse button to locate the tabfile.

See the VERIFY command.

Export tabfile

Use this dialog to export a tabfile as a text file. Specify the filename and select the tabfile to export from the list. You can export all the tables or individual tables and specify some other options here.

The resulting text file can be transferred on the different type of computer and imported by the **Utilities/Import** command.

See the EXPORT command.

Backup tabfile

Use this dialog to backup a tabfile. Specify the filename and select the tabfile to backup. You can backup the tabfile without indexes. In that case the indexes will be rebuild when you restore the tabfile.

To restore the tabfile use the **Utilities/Restore** command.

See the statement.

Restore tabfile

See the RESTORE command.

Data Entry and Modification

Data in SIR/XS databases or tables may be entered, modified or deleted with SQL. There are four commands available to do this. Three commands operate on sets of data in a similar way to `SELECT` with a `WHERE` clause. These are:

DELETE FROM

> Deletes rows or records.

INSERT INTO

> Inserts new records or rows into databases or tables. The new records or rows to insert are generated by a `SELECT` statement which is part of the command.

UPDATE

> Modifies values in existing rows and records

The fourth command creates single records or rows interactively:

ENTER INTO

> Prompts interactively for new records or rows data to add to the database or table.

DELETE FROM

```
DELETE FROM    [tabfile.] | [database.] name
               [WHERE condition]
               [COMPILE_ONLY]
```

Deletes rows or views from tables and records from databases.

Specify record name, record number or the keyword CIR as a database record name. If a database name is not specified, the default database is used. Specify a table or a view and the tabfile name. If the tabfile name is not specified, the default tabfile is used.

`WHERE`

WHERE defines the set of records to be deleted.

`COMPILE_ONLY`

Compiles the command and checks for syntax errors but does not perform the deletions.

ENTER INTO

```
ENTER INTO [tabfile.] | [database.] name
           [ LABELS ]
           [ LENGTH n ]
```

Prompts for data from the screen to enter new records or rows interactively. It is recommended that one of the other SIR/XS facilities such as FORMS or VisualPQL is used for interactive data entry for all but the most trivial instances.

Specify record name, record number or the keyword CIR as a database record name. If a database name is not specified, the default database is used. Specify a table or a view and the tabfile name. If the tabfile name is not specified, the default tabfile is used.

SQL prompts for the values of each variable. After the last value for one record or row, SQL prompts again for an entry into the first variable of the next record or row. Finish input at any time with the Cancel button. This cancels the current record and terminates the process.

`ENTER INTO` creates new records or rows. It does not allow the updating of existing data. If a record with the same key as an existing record is entered, it is rejected. If a row with the same values in a unique index as an existing row is entered, it is rejected. If a record or row is rejected the process is terminated.

`LABELS`

> Specifies that the variable label is used as the prompt in place of the variable name and data type.

`LENGTH`

> Specifies the length of the prompt in characters. Prompts are padded or truncated to the specified length. Maximum length is 32 characters.

INSERT INTO

```
INSERT INTO [tabfile.] | [database.] name
            [(variable list, ...)]
            VALUES (value_list) | SELECT statement
            [COMPILE_ONLY]
```

Inserts rows into tables and records into databases. The inserted rows or records are created by the SELECT clause.

Specify record name, record number or the keyword CIR as a database record name. If a database name is not specified, the default database is used. Specify a table or a view and the tabfile name. If the tabfile name is not specified, the default tabfile is used.

The variable list specifies the variables or columns. This must match the columns created by the SELECT. If the variable list is omitted, all variables defined for the record or row are expected.

VALUES

> The value list is a list of constants to insert. This creates one new record or row with the specified values. The values must correspond in type, length and order with the INSERT INTO variable list. Specify strings in single quotes.

SELECT

> SELECT retrieves data from other tables or record types and this data is then used to create the specified rows or records. Any valid SELECT can be used as the source of values for the insert operation. The variables that are in the SELECT list must match in type, length and order with the INSERT INTO variable list.

COMPILE_ONLY

> Causes the INSERT command to be compiled but not executed. It is used for checking syntax.

- **Example:** Suppose a table has been created from the employee database with the following SELECT, and some new employees are to be added:

```
SELECT * FROM EMPLOYEE ON MYTABLE WHERE GENDER EQ 1
INSERT INTO MYTABLE -
```

```
SELECT * FROM EMPLOYEE WHERE GENDER EQ 2
```

UPDATE

```
UPDATE [ tabfile.] | [ database.]  name
       SET variable_name = expression, ...
       [ WHERE logical condition]
       [ COMPILE_ONLY ]
```

Updates existing rows or records in a table or database. Specify record name, record number or the keyword CIR as a database record name. If a database name is not specified, the default database is used. Specify a table or a view and the tabfile name. If the tabfile name is not specified, the default tabfile is used.

The SET statement lists the variables to update. Separate each variable name or expression with a comma.

The WHERE clause specifies the rows or records to be updated. COMPILE_ONLY

Specify to compile the UPDATE command but not to execute it, for checking syntax before updating data.

- **Example:** To give everyone a 10% raise in Salary and to change date to today:

```
UPDATE MYTABLE SET SALARY = SALARY * 1.1, CURRDATE = TODAY(0)
```

Paths and Views

A *Path* is a logical connection between two records or tables which tells SQL how to join the two data sources when they are referenced in a FROM clause on a `SELECT`.

A *View* is a virtual table which is created from one or more records or tables and can then be referenced in the `FROM` clause as if it was a real table.

Create paths and views with the CREATE PATH and CREATE VIEW commands. List paths and views with the SHOW command. Delete paths and views with the DROP command.

Paths

Paths are implicitly invoked if the path is the only path between two records or tables named in the `FROM` clause of a `SELECT`. If there is more than one path defined between two records or tables, SQL uses the earliest defined path. When a SIR/XS database is connected, paths are automatically created between all records with matching keys.

Use the SHOW PATHS command to see the currently defined paths in the order that SQL deals with them.

Paths are explicitly invoked in the `FROM` clause of the `SELECT` statement by specifying the name of the first record type or table and the path name as the second name. For example, if there is a PATH called NEWPATH from RECX to RECY, invoke the path with:

```
SELECT RECXVARS RECYVARS FROM RECX NEWPATH
```

A path defined between two records on a case structured database operates within the same case unless the `USING` clause is specified on the path. The `USING` clause joins records belonging to different cases. The setting of `CASE` mode does not affect the operation of the path. Paths can be defined between the common information record (CIR) and other records or CIRs by specifying the keyword CIR.

A table name may be specified and the rows on the table are used in exactly the same way as records on the database. The `WHERE` clause in the path definition determines how the path operates and should normally reference indexed columns. `VIA` and `USING` clauses are not used if the `TO` clause specifies a table.

View

A View is a virtual table which does not physically exist. It is a mapping that retrieves data from tables and records and presents it as if it was a table. A view can provide a simpler presentation of a given subset of data. The view is dynamic and reflects the latest data contained in the base records and tables. Views provide additional security; permissions for various activities on the view may be granted to specific users; security on columns and rows may be controlled through the `SELECT` statement and the `WHERE` clause within the view.

A view definition resides in a tabfile and its columns are defined by a `SELECT` statement on the `CREATE VIEW` command. This is the same as any other `SELECT` except that the `DISTINCT, ORDER BY, FORMAT` and `UNION` clauses may not be used.

When a view is created, it is created on the tabfile in the same way as any other table. Views can be exported and imported on another machine. Security can be defined as for any other table. Permissions on views can be granted and revoked. (See Permissions).

Views can be used to `SELECT ... FROM ...` as any other table. `DISPLAY` does not access views. Views are used in SQL only; they cannot be used in other SIR/XS products such as VisualPQL or FORMS.

The underlying data in tables or records can be updated using the `VIEW` name on the update command with the following conditions:

- If the `FROM` clause in the `SELECT` statement of the `CREATE VIEW` command references more than one table or record, the view cannot be updated.

- If the `SELECT` statement implies any aggregation (`GROUP BY` or aggregation functions), the view cannot be updated.

- If the base table is a record or a table with a unique index, the view must include all the columns that compose the key or index in order to be updateable.

- If the `SELECT` statement contains a constant or an expression, then:

    - `INSERT` is not allowed on that column
    - `UPDATE` is not allowed on that column
    - `DELETE` is allowed

- If the view does not include all columns or variables from the table or record, then any unspecified columns in new rows or records are assigned undefined values.

CREATE PATH

```
CREATE PATH path_name
         FROM              [[database.]recordtype] | [[tabfile.] table]
         TO                [[database.]recordtype] | [[tabfile.] table]
         [WHERE boolean]
         [VIA    value list  |  USING   value list ]
         FIRST LAST OUTER REVERSE CIR
```

`CREATE PATH` names the path and specifies the two records or tables that the path joins. Paths may use Views as the `FROM` and `TO` references. The `WHERE, VIA,` and `USING` clauses link records and rows depending on the values of data items.

`FROM`

> `FROM` is required. It specifies the record, table or view which is to be joined to the `TO` record, table or view.

`TO`

> `TO` is required. It specifies the record, table or view which is to be joined to the `FROM` record, table or view.

`WHERE`

> The WHERE is a logical condition applied to individual occurrences of the data at execution time. If the condition is true, a row is returned.

`VIA`

> Specifies values for the key fields of the `TO` records. If one or more of the key fields are omitted, all records with the specified key fields are returned. The values can be an arithmetic expression, a constant or a variable name.
>
> An asterisk (*) can be substituted for explicit values to indicate that the key fields of the `FROM` record are to be passed. The asterisk can be used in combination with other values. The asterisk can only appear as the first item. An asterisk cannot be specified when a table is the `FROM` reference.

The case id is passed automatically when `VIA` is used. Do not specify the case id in the `VIA` clause.

`USING`

Specifies values for the case id and any other key fields for the `TO` records. `USING` allows the joining of records from different cases. The values can be constants, variables or arithmetic expressions. Only variables in the `FROM` record type can appear on the `USING` clause.

`FIRST`

Specifies that the path selects only the first record from the record type listed in the `TO` clause.

`LAST`

Specifies that the path selects only the last record from the record type listed in the `TO` clause.

`OUTER`

Specifies that if no occurrences of the `TO` record can be found to satisfy the path definition, then a dummy of all undefined values is used to complete the join.

`REVERSE`

Specifies that the records pointed to by the path definition are processed in reverse order. Abbreviation: `REV`

Examples:
1) To join employee records to occup records only where the employee is female and works in division 1, specify:

```
CREATE PATH MYPATH  -
         FROM EMPLOYEE TO OCCUP -
         WHERE GENDER EQ 2 AND DIVISION EQ 1
```

2) To create a path called `CURRENT` that joins `EMPLOYEE` and `OCCUP` records only for the current position, specify

```
CREATE PATH CURRENT -
          FROM EMPLOYEE TO OCCUP VIA CURRPOS
```

Note that `VIA` and `USING` are not specified together. Valid Specifications on the `VIA`, `VIA *`, and `USING` clauses are:

FROM	TO	VIA	VIA *	USING
table	recname	Yes	No	Yes
table	table	No	No	No
recname	recname	Yes	Yes	Yes
recname	table	No	No	No

CREATE VIEW

```
CREATE VIEW [tabfile.] viewname [(column list) ]
         AS SELECT     variable list ...
         FROM          [database.] rectype | [tabfile.]table , ...
         [GROUP BY     variable list, ... [HAVING expression]]
         [OUTER        [database.] rectype | [tabfile.]table ]
         [WHERE        expression]
         [WITH CHECK OPTION ]
```

`CREATE VIEW` creates a named view. The view is a `SELECT` from a number of records, tables or other views with particular conditions.

`viewname`

Every view has a name and its definition is stored in a tabfile. If the tabfile is not specified, the current default tabfile is used.

`column list`

The optional column name list renames the columns specified in the `SELECT` statement in the `CREATE VIEW` command. The column list must reference the same number of columns as the `SELECT` statement does. Renaming columns is useful when the `SELECT` creates columns with awkward names such as expressions or concatenations of table and column names.

`AS SELECT`

Specifies the columns in the view. The names are the variables and columns from the records and tables in the `FROM` clause. Specify expressions (constants, arithmetic expressions, functions) as in a `SELECT` command. If the view includes record(s) from a case structured database, the view automatically includes the case id and common variables.

`FROM`

Specifies the records, tables, views or paths which contain the specified variables or columns. Aliases may be assigned. If aliases are used, each `FROM` record or table must be separated by commas. If the optional database and tabfile names are not specified, the current defaults are used.

`GROUP BY`

Specifies that sets of values selected are grouped together according to their unique values in the value list. If this is a case structured database and the GROUP BY is intended to group records from more than one case, `CLEAR CASE` before using the view.

`HAVING`

The selection criteria is applied to each group selected with the `GROUP BY` clause. Only groups which satisfy the specified condition are included in the view.

`OUTER`

Specifies that if no occurrences of the specified record or table can be found, then a row is created with all variables or columns from the missing record or table set to undefined. If `OUTER` is not specified, a row is not returned.

`WHERE`

Specifies a condition that must be satisfied for a row to be included in the view. The `WHERE` clause can contain subqueries but these may not reference another view.

`WITH CHECK OPTION`

Specifies that, when a view is being used to update a base table or record, the new row must conform to the `WHERE` clause conditions. This means that the view can only add or modify rows which are part of the view. If this is not specified, rows can be added which the view could not retrieve.

Examples:
To create a view of `EMPLOYEES` with `ID, NAME` and `SALARY` for male employees only (including `ID` automatically because it is the case id):

```
CREATE VIEW MALES -
      AS SELECT NAME SALARY -
      FROM 1 WHERE GENDER EQ 1
```

To create a view of `EMPLOYEES` who have had low review ratings:

```
CREATE VIEW LOWRATING  -
         AS SELECT NAME SALARY POSITION DIVISION REVDATE RATING -
         FROM 1 2 3  -
         WHERE RATING LT 4
```

To create a dynamic summary of people by education level:

```
CREATE VIEW EDSUMM    -
         (LEVEL,NUMBER,WAGES,AVERAGE) -
```

```
AS SELECT EDUC COUNT(SALARY) SUM(SALARY) -
SUM(SALARY)/COUNT(SALARY) -
FROM 1  -
GROUP BY EDUC
```

RENAME VIEW COMMAND

```
RENAME VIEW  [ tabfile_name.  ] view_name TO view_name
```

Renames a view. If the tabfile is not specified, the default tabfile is used.

Tabfiles and Tables

SQL can create, define, populate, modify and retrieve data from tables stored in tabfiles.

A tabfile is a physical file on disk and is independent of all other tabfiles. A tabfile can hold multiple tables and is the largest unit that exists for security and access control.

An SQL session may be connected to multiple tabfiles and can retrieve data from tables in any connected tabfile.

Whenever tables are referenced, the tabfile can be specified or the default can be taken. One tabfile is always the default and can be any tabfile. If no other default is set, the `$SYSTEM` tabfile is the default.

Table

A table is all of the individual instances (or rows) of a single type of record. A record is a set of columns (or variables). The definition of the individual columns includes the column name, format, data type, missing values and value labels. There are no key columns in a table; key processing is done through indexes.

Index

An index is a way of accessing the rows in a table using the values of particular column(s) as the key. Indexes can be defined on any column or combination of columns. An index may specify that rows must have unique values or may allow many rows with the same value. Indexes can be used to process tables randomly given a particular index value as the key, or sequentially in index order. If a table is processed without an index, it is retrieved sequentially in the order in which it was created. When an index is defined, it is built from any existing data and is automatically maintained as the table is updated.

Commands

The SQL commands which create tabfiles, create tables, and create indexes are CREATE TABFILE, CREATE TABLE and CREATE INDEX.

The SELECT command also creates tables which are exactly the same as tables created in any other way. `SELECT` copies data definitions and populates the table and can be a much more convenient way to define new tables than using explicit commands.

You can also use VisualPQL or the SirSQL menus and SirDBMS menus to create tabfiles and tables.

There are five utilities which can be used with tabfiles. These are:

EXPORT which creates a text version of the tabfile or individual tables which can then be used by SQL to re-create the table. This can be used to move the tabfile from one machine to another.

VERIFY which checks tabfiles for possible corruptions.

BACKUP TABFILE which takes a sequential file copy of a tabfile .

RESTORE TABFILE which rebuilds a tabfile from the sequential copy and applies changes logged to a journal.

DISPLAY JOURNAL which lists the contents of a journal file.

CREATE TABFILE

```
CREATE   TABFILE   tabfile-name
         [FILENAME filename ]
         [IDENTIFIED BY grpname [/grppass] [.username[/userpass]]]
         [JOURNAL  filename]
         [BLOCKS n]
```

`CREATE TABFILE` creates a tabfile to store one or more tables. The tabfile name is the name by which this is referenced in all other commands. This name must be used to CONNECT to this file in subsequent sessions. A tabfile is automatically connected after being created. For example:

```
CREATE TABFILE MYFILE FILENAME 'MYFILE.TBF'
```

`FILENAME`
Specifies a filename for the tabfile. If this parameter is not specified, the tabfile name plus a suffix of .tbf is used as the filename. (This must therefore be a valid filename).

`IDENTIFIED BY`
Starts to create security definitions for access to the tabfile. The group name and optional group password specifies a group name who has DBA permission for the tabfile. If the DBA wishes other groups to access this tabfile, the DBA gives permissions with the GRANT command. The user name and optional password further restrict original DBA access to the tabfile to a second level of name and password. For example to leave `MYFILE` available to everyone to connect to it:

```
CREATE TABFILE MYFILE.TBF
```

To require that the group name `SURGEON` password `BYPASS` and user name `JONES` password `INTERN` are needed when connecting to this tabfile as a DBA in subsequent sessions:

```
CREATE TABFILE MYFILE.TBF IDENTIFIED BY -
          SURGEON/BYPASS.JONES/INTERN
```

If the `IDENTIFIED BY` clause is not specified, any user can access the tabfile with all permissions and this cannot be changed subequently without rebuilding the tabfile, for example exporting and editing the export file to have the `IDENTIFIED BY` clause.

`JOURNAL`
Specifies that journaling is turned on for this tabfile and names the operating system file which is to be used. If the journal file is not there when the tabfile is updated, a new journal is created, otherwise new journal data is added to the end of the file.

`BLOCKS`
Specifies the number of the blocks that are used to create a physical block. The default is 1. The actual block size is 2k bytes. A specification of 2 would give 4k bytes and so on. The number must be a positive integer.

In general the default is adequate. There is one circumstance where the block size must be specified. A block must be able to hold the largest physical row. If you plan to define very large rows, specify a `BLOCKS` clause to create a block large enough to accommodate this.

The `BLOCKS` clause may be specified for performance reasons. Larger blocks are more efficient for serial processing but take more memory. Small blocks are more efficient for random processing through indexes where each I/O probably accesses a different block.

CREATE TABLE

```
CREATE TABLE  [tabfile.]table
              (column-name data-type [options] , ...  )
              [optional-table-clauses]
```

Creates a definition of a new table. A SELECT does this automatically.

Specify the name of the table and a list of the columns that make up the table. A maximum of 250 columns can be specified for a table. The table and one column with its data type are the only required clauses.

Each column must have a name and a data type. Other column specifications are optional. Enclose the complete column list in parentheses. Optional specifications that are not explicitly defined take the default values `SET` in the current session.

`tabfile`
Specifies the tabfile where the table is located. A single tabfile can contain any number of tables. The table is stored in the default tabfile if a tabfile is not specified.

`table`
The table name is required. The table name must be unique within the tabfile.

`column_name`
Specify a name for each column. Column names must be unique within the table.

The Data Type controls whether a column is numeric, date or string, etc. and cannot be altered once the table is created.

The Column Options clauses control how the column is created and stored and cannot be altered once the table is created. The Display Format control the appearance of a column and these can be altered after the table is created.

Specify as many optional clauses as needed for any column (provided that the clauses used are compatible with the data type specified for the column).

There are further optional clauses which can be specified after all of the column specifications:

```
[ FORMAT clauses ]
[ PCTFREE (n)]
CONSTRAINTS UNIQUE column-list ]
```

`FORMAT`
Specifies how SQL displays output. These clauses only effect the SQL display process. Any of the formatting clauses that do not describe individual columns can be specified. See Format.

`PCTFREE (n)`
Specifies the percentage of free space that is reserved for future expansion in each data block of a table. The only purpose of allowing room for growth in this clause, is for existing rows to be modified where the modified records takes more space. Specify an integer from 1 to 99. Rows grow when column values increase in size. The default is 10 (percent).

`CONSTRAINTS UNIQUE (column list)`
Specifies a list of columns where combinations of the values in the columns must be unique for a number of rows. If an attempt is made to add a row where the combination is not unique, the row is rejected with an error message. The columns must also be specified as `NOTNULL`.

`CONSTRAINTS UNIQUE` creates a unique index for the table which is given a system generated name: `'&UNIQUE_MULTIPLE_INDEX_I_n'` where **n** is the number of the index.

Specify the column names enclosed within parentheses. Repeat the clause for as many combinations of columns as required. For example:

```
CONSTRAINTS UNIQUE (Name, Sex, Birthday).
```

The same result could also be achieved with the CREATE UNIQUE INDEX command.

Column Data types

There are a number of possible specifications for column data types, some of which are synonyms for others.

```
CATEGORICAL [(n)]
CHARACTER | STRING [(n)]
DATE [('date_map')]
DEC | NUMERIC [(length,decimals)]
FLOAT [(n)] | REAL | DOUBLE]
TINYINT | SMALLINT | INT |
TIME [('time_map')]
```

`CATEGORICAL`
Defines the column as character which contains one value from a pre-specified list of values. The input data is checked against the list. The position in the list that corresponds to the input data is stored in the table rather than the value of the entry. Typically this is used for a list of names (e.g. Names of States), where it is more efficient to store a code rather than a value.
n is the number of entries in the list. Values for the strings are then defined in the `VALID VALUES` clause.

`CHARACTER`
Defines the column as character. `STRING` is a synonym for `CHARACTER`.
n specifies the maximum string length. The default is 254 which is also the maximum. Strings are held as variable length unless the optional clause `FIXED` is specified. `CHARACTER` can be abbreviated to `CHAR`.

`DATE`
Defines the column as a date which is displayed or entered according to the date map. Internally, the date is held as a number of days since the start of the Gregorian calendar. Externally, the date is input and output in accordance with the date map. If a date map clause is not specified, the current system date map is used.

`DEC`
Defines the column as a scaled integer number. `NUMERIC` is a synonym for `DEC`. Length specifies the total length of the number. Decimals specifies how many of those digits are to come after the decimal point. For example: `DEC (10,2)` means that the integer is 10

digits long, with 2 digits to the right of the decimal point. This is equivalent to defining an integer type and the optional `SCALE` clause with a value of -2.

`FLOAT n | REAL | DOUBLE .`
Defines the column as a floating-point number. `FLOAT n` is either 4 or 8 and the default is 8. `FLOAT (4)` or `REAL` gives single precision. `FLOAT (8)` or `DOUBLE` gives double precision.

`INT | SMALLINT | TINYINT`
Defines the column as a fixed length integer.
`INT` is a 4 byte integer; `SMALLINT` is a 2 byte integer; `TINYINT` is a 1 byte integer;

`TIME`
Defines the column as a time which is displayed or entered according to the time map. Internally, the time is held as a number of seconds since midnight. Externally, the time is input and output in accordance with the time map. If a time map clause is not specified, the current system map is used.

Column Options

The clauses controlling how the column is stored are :

```
[ BIAS (n) ]
[ FIXED  ]
[ MISSING  value 'label')]
[ NOTNULL [ UNIQUE ]]
[ PRESET  ]
[ SCALE  ]
[ VALID   (value) ]
[ VALUE LABELS  ]
[ VARYING  ]
```

`BIAS`
Specifies an integer constant that is added to an integer before it is stored. For example, this might be used in a study with questionnaires from multiple sources, each of which was numbered from 1. To avoid multiple questionnaires numbered the same, `BIAS` the question number in each table by a different amount such that one table had questionnaires 1 - 99, the next 101 - 199, 201 - 299, etc.

With `SCALED` integers, the `BIAS` is done before scaling, so express the bias as the unscaled number. For example, if the scale is 2 (hundreds) a `BIAS` of 1, results in a bias of 100.

`FIXED`
Specifies that the string column is stored as fixed length. Strings are variable length by default. `FIXED` may result in faster processing, but may use space inefficiently if there is a wide variation in lengths of values. For example, a string for social security number, which is always present and always the same length, could be specified as `FIXED`. If a string may vary considerably in length, let it default to variable length.

Numeric columns are always `FIXED`.

`MISSING`
Specifies the column's missing values and can associate a label with each value or range of values. Specify single missing values or ranges of missing values with optional labels for these values. The value may be a value that corresponds to the data type of the column or may be the keyword `BLANK` or `UNDEFINED` which are allowed for any data type. If the data type is a string, enclose the value in quotes.

The command may be specified as `MISSING VALUES` or `MISSING RANGES` as documentation but this has no effect on the specification.

Defining a value as `MISSING` is an implicit definition that it is a `VALID` value. `MISSING` and `VALID` values are stored as a single list which is searched serially. Once a match is

found, the search stops. The list is checked for overlapping ranges which are reported as an error. Separate each entry with a comma. Enclose the whole list in parentheses.

Except for undefined or `BLANK` numeric variables, the actual value entered is still held in the table. However when the value is retrieved, it is flagged as a missing value. The `MISS` function can be used to retrieve values which include original values that would otherwise be missing.

For example:

```
MISSING  (8 'Refused to Answer',
          9 'Not Applicable',
          BLANK 'No Answer Coded')
MISSING  ('N/A' 'Not Applicable',
            BLANK 'No Answer Coded')
```

Specify a range of values and associate a single value label with any value entered in that range. A range is a pair of values, separated by a colon, which correspond to the data type of the column. To separate ranges for readability, use the square brackets [ ].

Parentheses specify ranges where the end points are not missing. When using parentheses, the keywords `LOWEST` and `HIGHEST` can be used to specify end points.

For example:

```
MISSING (0:18 'Under Age', 66:99 'Retired')
MISSING ([1:18],[50:59],[90:99])
MISSING  ( (LOWEST:18)  'Too Young',(65:HIGHEST) 'Too Old')
```

The second example creates ranges without labels. The third example specifies that those younger than 18 and older than 65 are missing. The parentheses specifies that the actual quoted value is not missing but that all values from that point on are. This is useful for real numbers, where it may be impossible to specify the actual end of a range.

`NOTNULL`

Specifies that the column cannot be missing. An attempt to insert a row that contains a missing value for this column fails.

`NOTNULL UNIQUE`

Specifies that no two rows can have the same value for the column. An attempt to insert the same value twice fails, and an error message is issued. The `CONSTRAINTS UNIQUE` clause specifies combinations of columns to be unique.

`PRESET (value)`
Specifies a value that is stored if no value is explicitly given. The value must agree with the type, length, and map specifications for the column. Enclose string values in quotes. By default, columns are set to `UNDEFINED` and there is no need to specify this.

`SCALE (n)`
Specifies the power of 10 that a number is multiplied by as it is placed in storage as an integer. This provides efficient storage of large or small integers where the accuracy level is only required at the scaling factor.

For example, for a result in kilovolts, where the calculations are in volts, specify a scale of 3 for kvolts data, and the conversions are handled properly.

A scale of **-n**, specifies that there are **n** decimal positions. For example, decimal money can be held at a scale of -2.

Any calculations which refer to a scaled integer, should express the number as the external normal value of the number; the software deals with any internal scaling. For example, to select rows where an amount of money is greater than 100 dollars:

```
SELECT ...  WHERE  AMOUNT  GT 100
```

Scaled integers, by definition, cannot hold data at less than their scale. Any computation is rounded. For example, setting `KVOLTS` (scale 3) to any number which is not a round thousand, results in the number being rounded to the nearest thousand.

```
VALID (range list).
```

Specifies the valid list of values or ranges of values that are allowed in the column. The syntax rules for the range list are the identical to the range list in the `MISSING` clause. A specification of a value as missing, means that this is a legal value to be input.

The command may be specified as `VALID VALUES` or `VALID RANGES` as documentation but this has no effect on the specification.

All missing and valid values are stored as a list which is searched serially. Once a match is found, the search stops. The list is checked for overlapping ranges which are reported as an error.

`CATEGORICAL` variables must have individual valid values not ranges.

```
VALUE LABELS (value label list)
```

Specifies labels for particular values that occur in the column. Each entry consists of the value followed by the label. Separate multiple entries with commas and enclose the whole list in parentheses. For example:

```
VALUE LABELS (   'AL' 'Alabama',
                 'AK' 'Alaska',
                 ......,
                 'WY' 'Wyoming')
```

This associates the full state name with the abbreviation. When referencing `CATEGORICAL` variables, specify the equivalent integer

`VARYING`
Specifies that the column is variable length. This clause is documentary only. Strings are variable length by default and this clause has no effect on other data types.

CREATE INDEX

```
CREATE [UNIQUE] INDEX index-name ON [tabfile.]table
       (column [ASC|DESC], ...)
       [PCTFREE integer value]
```

`CREATE INDEX` creates an index for a table providing direct access to a subset of records. Index usage is automatic in SQL once the index is defined.

`UNIQUE`
Specifies that two rows cannot have the same index value. Rows with a value the same as an existing row are rejected. When creating an index for an existing table, if existing rows contain identical values, the index is not built and an error message is issued.

`index name`
Specifies the name of the index. Index names must be unique on the tabfile.

`ON`
Specifies the table to index. If a tabfile is not specified, the default tabfile is used.

`column`
Specifies the column(s) to index in major to minor sequence. Specify `DESC` for any columns in descending order. For example: a specification of `(Sex, Name)` gives all males by name, then all females by name. A specification of `(Name, Sex)` gives everyone with the same name together, males preceding females.

`PCTFREE`
Specifies the percentage of free space to leave in the index blocks. This is used as new index entries are made. If the table is updated on a regular basis, take the 50% default. If the table is static and the index is not going to be updated, specify a low figure. For example:

```
CREATE UNIQUE INDEX XID ON MYFILE.EMPLOYEE (ID)
CREATE INDEX XNAME          ON MYFILE.EMPLOYEE  (LASTNAME , FIRSTNAME)
CREATE INDEX XREVIEW_DATE   ON MYFILE.EMPLOYEE  (REVDATE DESC)
```

Permissions

The ability to perform various types of operations (such as the ability to use a tabfile or a table, to update a table, create a new table, etc) may be restricted to specific groups of users or to individual users. Users are allowed to perform particular activities through a set of permissions.

Permissions may apply to the tabfile as a whole, to individual tables and views and to individual columns. Permissions start with the creator of a tabfile. When creating a tabfile, specify the `IDENTIFIED BY` clause to restrict access to the named group or group.user. If the `IDENTIFIED BY` clause is not specified, you cannot restrict access to operations on the tabfile and anyone can do anything to any table. The group(.user) named on the `IDENTIFIED BY` clause has DataBase Administrator (DBA) permission for the tabfile and can assign permissions to other groups(.users).

Permissions may be granted to and revoked from groups of users and individual users within a group. A group is a set of users who are allowed to do the same operations. A group has a name and may have a password.

There is no commonality between tabfiles for groups or users; groups and users only exist within a tabfile. To use standard names for groups to access multiple tabfiles, create naming standards and conventions which are then used for each individual tabfile.

Create a group by naming it on a the `IDENTIFIED BY` clause on the `CREATE TABFILE` or on a `GRANT` command. An individual user in a group may be granted permission to do additional operations. Permissions may be granted and revoked by users, who may only grant the permissions which they have been granted.

GRANT

```
GRANT    {permission,...| ALL | ALL BUT permission,...}
         TO grpname[/grppass][.username[/userpass]],...
         ON tabfile | [tabfile.]table_name
         [ WITH GRANT OPTION ]
```

GRANT gives permissions on tabfiles, tables, views or individual columns to specified groups or users.

DBA, CONNECT, and CREATE are only applicable at the Tabfile level. All other permissions may be granted at the individual table or view level. Permission may be granted on individual columns for SELECT and UPDATE which restricts access to those specific columns. To give permissions for particular columns, specify a list of column names on the permission clause:

```
GRANT SELECT [ ( varname, varname... ) ] TO ... ON ...
GRANT UPDATE [ ( varname, varname... ) ] TO ... ON ...
```

permission
Specifies the permission(s) being granted. The following permissions may be granted:

ALL - all permissions are granted.

ALL BUT - all permissions except the specified permission(s) are granted.

DBA - permission to do anything to a tabfile. This is required for certain utilities, in particular EXPORT. There are no activities on an individual table that require DBA permission.

CONNECT - permission to connect to a tabfile.

CREATE - permission to create tables.

SELECT - permission to SELECT from tables.

DELETE - permission to DELETE rows from a table.

UPDATE - permission to UPDATE rows in a table.

INSERT - permission to INSERT rows in a table.

`DROP` - permission to DROP or delete tables.

`ADDCOL*` - permission to add columns to tables.

`MODCOL*` - permission to modify columns in tables.

`DELCOL*` - permission to delete columns from tables.

`INDEX` - permission to create or drop an index on table.

* These permissions are provided for the future implementation of the `ALTER TABLE` command. These currently do not have any effect.

Permissions exist at two levels, the tabfile level and, possibly, at the individual table level. When a user connects the tabfile, the appropriate set of permissions are retrieved and the user is only permitted to perform allowed activities. For example, a user must be allowed to `SELECT` at the tabfile level before being able to `SELECT` from a specific table. If a group(.user) has DBA permission at the tabfile level, they have all permissions on all tables, regardless as to how the tables were created or permissions assigned.

If a user is allowed an activity at the tabfile level, whether they are allowed that activity on any table on the tabfile depends on how the table is created and can be modified by specific `GRANT/REVOKE` commands. If the table is a *PUBLIC* table, then no further checking is done.

If the table is created in SQL with a `SELECT`, then a Public table can be created in SQL by clearing the Define_Security setting before doing the select. Otherwise the group(.user) that created the table has all permissions on that table and is the only group(.user) allowed any activities on that table until further permissions are granted.

If the table is created in SQL with the `CREATE TABLE` command, the group(.user) that created the table has all permissions on that table and is the only group(.user) allowed any activities on that table until further permissions are granted.

If the table is created by the VisualPQL procedure `SAVE TABLE`, the procedure can optionally specify groups(.users) that have full permissions on the table. The group name used to connect the tabfile is always granted all permissions on the table. The procedure can specify a group of Public to create a public access table.

`TO`
Specifies the group names to receive the permissions. If the group does not already exist for this tabfile, it is created. An optional group password may be specified. An optional username and user password may also be specified. Permissions granted to a group are granted to all members of the group. There is no need to specify individual users in a group unless you need to allow someone specific additional permissions. You cannot specify a user to be a member of a group and restrict them from any group permissions. If

you specify a group and user, and the group does not exist, the group and group password are created but group level permissions are not set up automatically and you cannot use just a group name until permissions are set for the group.

Group names, user names and passwords are checked when a tabfile is connected. To update a group or user password, connect to the tabfile as that group/user and use the `GRANT` command with the special keyword `PASSWORD`. e.g. `GRANT PASSWORD TO GROUP1/NEWPASSWORD`. If a password is forgotten, a DBA can export the tabfile with the security option and the export file will contain `GRANT` commands with readable passwords.

If the group already exists, there is no need to respecify the group password. If the group/user combination already exists, there is no need to respecify the user password. In either of these cases, the original password is kept and any password specified on the command is ignored.

`ON`
Specifies the tabfile name or table name to which the permissions apply.

`WITH GRANT OPTION`
Specifies that the group or user can grant these permissions, or a subset of these permissions, to other groups or users. Without this clause, these permissions cannot be granted by this user to other users.

Examples:
Suppose you want to create a tabfile called MYFILE and restrict DBA authority to group DBAS and they have a password DBASPASS. Specify

```
CREATE TABFILE MYFILE IDENTIFIED BY DBAS/DBASPASS
```

Then, when people try to connect to that tabfile, they either specify `CONNECT TABFILE MYFILE IDENTIFIED BY DBAS/DBASPASS` and get full permissions or they will not be allowed even to connect to the tabfile. To allow connection, connect as a DBA and specify, for example, `GRANT CONNECT, SELECT TO RESEARCH ON MYFILE`. If you as DBA create a table called TABLE1 then you might `GRANT SELECT TO RESEARCH ON MYFILE.TABLE1` to allow that group to select from Table1. Similarly if you had created a table EMPLOYEE on tabfile PERSONNEL and granted Connect and Select permissions to a Personnel group, you might then `GRANT SELECT (ID, NAME, SEX) TO PERSONNEL`

`ON CURRENT.EMPLOYEE WITH GRANT OPTION` which allows them access to specific columns on that table with the ability to create other groups(.users) with those permissions.

REVOKE

```
REVOKE   {permission,...| ALL | ALL BUT permission,...}
         TO grpname[/grppass][.username[/userpass]],...
         ON tabfile | [tabfile.]table_name
```

REVOKE revokes permissions on tabfiles or on individual tables for the specified groups or users. REVOKE is the opposite of GRANT and has identical keywords and syntax.

You can revoke permissions only if you granted them. You do not have to REVOKE all permissions originally granted, you can revoke a subset.

If permissions are revoked, that user no longer has the rights accorded by those permission. This carries down to groups and users who have been granted permissions by that user. For example, if USERA has granted a permission to update a certain table to USERB, USERB is not able to update that table if update permission is revoked from USERA.

Permissions revoked from a group are revoked from all members of the group.

To revoke permissions you must be connected to the tabfile. The group(.user) specified at connection time is the *grantor*. The original creator of a tabfile has a special set of permissions with a special System grantor and you cannot revoke any permissions from that group(.user). Similarly, the original creator of a table using SQL has the same special grantor and you cannot revoke permissions from that group(.user) for that table.

If you are connected as a group(.user) you can revoke permissions from other group(.users) that you created directly. If those users have granted permissions to other users, and those permissions are affected by your revoke, then the revoking is carried down the hierarchy. However you cannot revoke permissions directly from group(.users) that you did not grant to directly. If you grant to groupa and they grant to groupb, you cannot revoke groupb even if you are the overall DBA for the tabfile. You also cannot revoke from yourself.

If you revoke all permissions from a group(.user) at the tabfile level, the group(.user) is deleted from the file.

DEFINE_SECURITY

> Specifies that tables created with SELECT have security definitions. When connecting to a tabfile with security definitions, specify group(.user) names to connect to the tabfile. Anyone with DBA permission at the tabfile level, has full permissions for all tables. With DEFINE_SECURITY set, you have full permissions

for the new table created with `SELECT` but any other groups(.users) (except DBAs) have no permissions on that table. You (or a DBA) can grant permissions to other users. (see Permissions)
`CLEAR` specifies that created tables are Public access.
Default: `SET`

EXPORT

```
EXPORT [ FILENAME ] filename
       [ RECSIZE n ]
       [ NOSECURITY ]
       [ NOINDEXES ]
       [ NODATA ]
       [ NOTABFILE ]
       [ NOWORKSPACE ]
       tabfile [(table [(column, ......)], ......)]
```

Exports tabfiles, tables, or selected columns from tables. `EXPORT` creates a text file from which the exported elements can be imported on any machine running SIR/XS.

`filename`
Specifies the file to contain the exported data. The filename must be the first clause in the export. The keyword `FILENAME` can be specified for readability.

Follow the filename with any keywords:

`RECSIZE`
Specifies the record length for the named file. The minimum recsize is 80. This is the default. Records are fixed length.

`NOSECURITY`
Prevents the security privileges from being written to the exported file.

`NOINDEXES`
Prevents indexes from being written to the export file. By default, all indexes are written to the export file.

`NODATA`
Prevents the data from being written to the export file. Only the schema is written. By default, the export includes the data for each table.

`NOTABFILE`
Prevents information specific to the overall tabfile from being exported. When the export file is imported all of the tables are placed on the default tabfile of the user performing the import.

`NOWORKSPACE`
Prevents information about the current settings from being exported.

`tabfile`
Specifies the tabfile to export. If a tabfile is not specified, all connected tabfiles are exported.

`table`
Specifies the table(s) to export. If no tables are specified, all the tables on the tabfile are exported. The entire table or selected columns of that table can be exported. More than one table can be specified. The entire table is exported when columns are not specified.

`(column, ...)`
Specifies individual column(s) to export.

Example:

```
EXPORT 'EXPORT.DAT'  MYTABFILE (MYTABLE (COL1 COL2) MYTABLE2 )
```

You can only export tables and columns for which you have read security.

An SQL export file is simply a set of SQL commands and data in textual form. It can be imported in three ways:

> It can be used on the `IN` parameter when executing SQL to import the file in batch mode.
>
> It can be imported through the utilities menu option.
>
> You can read the file into the input area (with cut and paste or by 'Opening' it) and execute the statements directly. If you do this, first delete the initial `BEGIN IMPORT` line.

VERIFY

```
VERIFY tabfile  [ ON filename ]
```

Checks all of the tables on the specified tabfile. If a table or tables are found to be corrupt, SQL issues a notice of the affected tables and purges the corrupted tables. Any tabfile can be verified, it does not have to be connected.

If a tabfile is corrupt, there may be difficulty connecting to it. `CONNECT` to a corrupt tabfile with `READ` access only. If the `$PASSWORD` or `$SECURITY` system tables are corrupted, then all users have DBA permissions on the tabfile.

Specify the `ON` clause to identify the physical file where the name of the physical file differs from the internal tabfile name.

BACKUP TABFILE

```
BACKUP TABFILE  tabfile_name FILENAME filename [ FULL | DATA ]
```

Backs up a tabfile to an operating system sequential file. Specify the keywords `BACKUP TABFILE` and `FILENAME`. The filename is the name of the file being created as the backup.

`FULL`
Specifies that each block of the tabfile is compressed and written to the output file. When it is restored, the tabfile is the exact size as before (no pointer restructuring of the indexes is done).

`DATA`
Specifies that only the physical data records and definitions of the index(es) are written to the backup file. The backup file is smaller but indexes have to be rebuilt when the file is restored.

RESTORE TABFILE

```
RESTORE TABFILE tabfile_name [ FILENAME filename ]
         [ FROM   filename ]
         [ JOURNAL        filename ]
         [ APPLY  filename , ...  ]
```

Restores a tabfile from a backup file and/or applies journalised updates. `RESTORE TABFILE` does not overwrite existing tabfiles. The `FROM` clause specifies the name of the backup file; the `FILENAME` clause specifies the operating system name of the restored tabfile if the tabfile name is not the operating system filename.

Specify `JOURNAL` to assign a new journal file to this tabfile. If `JOURNAL` is not specified, the original journal file is used for journaling.

`APPLY` applies journal files changes to the tabfile. Specify the journal file to be used. All journals applied must be in order with no gaps.

Example:

```
BACKUP TABFILE mytabfile FILENAME 'MYTAB.BAK'
RESTORE TABFILE mytabfile FROM 'MYTAB.BAK'
```

DISPLAY JOURNAL

```
DISPLAY JOURNAL filename          FILENAME  filename
         [ HEADER         [ TF | TABLE | INDEX ]...  ]
         [ DETAILED       [ TF | TABLE | INDEX | ROW ]...  ]
```

Lists information about the contents of a specified journal file. The tabfile must be connected to display the journal. The `FILENAME` clause specifies an output file for the listing. Specify both the journal filename and the output filename. By default, tabfile headers, table and index entries are listed (headers are one line of information). Specify detailed information on the tabfile (TF), a table, index or rows with the `DETAILED` clause.

SQL Functions

All SQL functions return a single value that is either a number or a string. Functions may be used anywhere that a value or expression is appropriate. There are two types of functions:

`Standard`

: These return a value from an expression.

`Aggregation`

: These compute a value from a number of rows and thus alter the number of rows produced in the output table.

The `EXISTS` function tests whether a row is returned by a subquery.

Standard Functions

The expressions in the arguments must be of the correct type (numeric or string) for the function being used. Expressions can contain constants, variables, computations and other functions. Functions can be nested as necessary. Enclose string constants in single quotes. Date, time and categorical variables can be treated as an integer or as a string depending on the context. A specification of one of these variables retains the original data type. However, if these are referred to in a numeric function, the integer value is returned; if referred to in a string function, the string value is returned. Once a variable has been used in a computation or expression in this way, the resulting column is an integer or a string rather than being a date, time or categorical column. SQL eliminates null computations, for example `TODAY(0)+0`, creating an expression which is equivalent to a simple specification of the variable. For example;

```
SELECT BIRTHDAY FROM EMPLOYEE
```

This creates a `BIRTHDAY` column which is a date:

```
SELECT BIRTHDAY+365 FROM EMPLOYEE
```

This creates a "`BIRTHDAY+365`" column which is an integer. The SQL functions are:

`ABS`

Returns the absolute value of the numeric expression.

```
num = ABS( expression )
```

`ALL`

Tests against the values in the value_list. Returns 1 (true) if matches every item. Equivalent to individual tests against each item in the list joined by `AND` operators. `ALL` is used particularly to test against sets of values returned by subqueries.

```
value = ALL ( value_list | subquery )
```

`ANY`

Tests against the values in the value_list. Returns 1 (true) if matches any item. Equivalent to individual tests against each item in the list joined by `OR` operators. `ANY` is used particularly to test against sets of values returned by subqueries.

```
value = ANY ( value_list | subquery )
```

`CDATE`

Returns the date integer for the date specified by the date string in the first argument. The second argument is the date map. If the second argument is omitted, the current value of the system parameter `DATE` is used as the date map.

```
num = CDATE ( date-string [, date_map])
```

`CTIME`

Returns the time integer for the time specified by the time string in the first argument. The second argument is the time map. If the second argument is omitted, the current value of the system parameter `TIME` is used as the time map.

```
num = CTIME ( time_string [, time_map ])
```

`DATEC`

Returns a date string for the date integer in the first argument. The second argument is the date conversion map. If the second argument is omitted, the current value of the system parameter `DATE` is used as the date map.

```
str = DATEC ( date_integer, date_map)
```

`EXISTS`

Tests whether a subquery returns any rows. Returns 1 (true) if one or more rows are selected in the subquery. `NOT` can be used to test for the opposite condition. Specify an asterisk in the subquery as the returned variables when using the `EXISTS` function .

Note that this function is different from the VisualPQL `EXISTS` function. The SQL `EXISTS` function is the ANSI standard function.

```
SELECT .... WHERE EXISTS ( SELECT * ....  )
SELECT .... WHERE NOT EXISTS( SELECT * ....)
```

`INT`

Returns the truncated integer value for the numeric expression.

```
num = INT ( expression )
```

`LEN`

Returns the number of characters (including trailing blanks) in the string expression.

```
num = LEN( string )
```

`LOWER`

Returns the string with all uppercase letters converted to their lowercase equivalent.

```
str = LOWER( string )
```

`MAXIMUM`

Returns the maximum of the two values supplied.

```
num = MAXIMUM ( value1, value2 )
```

`MINIMUM`

Returns the minimum of the two values supplied.

```
num = MINIMUM ( value1, value2 )
```

`MISS`

Returns the original value of the named variable even if this would otherwise be flagged as a missing value. If the variable contains undefined, then missing is returned. This function can be used with all types of database or tabfile variables; string values are returned for character, categoricals, dates and times. (`MISSING` is a synonym.)

```
num = MISS (column)
```

`MOD`

Returns the remainder of the integer division of the number by the divisor.

```
num = MOD ( number, divisor )
```

`NOW`

Returns the current wall clock time as a time integer. The argument is a dummy argument.

```
num = NOW (0)
```

`NUM`

Returns the numeric equivalent of the number stored in the specified string.

```
num = NUM ( string )
```

`RECCOUNT`

Returns a count of occurrences of this record in this case. The rectype can be the record name or record number.

```
num = RECCOUNT ( rectype )
```

`RND`

Returns the integer value rounded to the nearest integer. The optional second argument specifies the number of decimal places to be rounded to instead of the nearest integer.

```
num = RND ( number [, digits ] )
```

`SBST`

Returns the substring of the specified string starting at a particular position and continuing for the specified length.

```
str = SBST ( string, start, length )
```

`SIGN`

Returns the sign of the second argument times the absolute value of the first argument.

```
num = SIGN ( number, sign )
```

`TIMEC`

Returns a time string for the time specified by the first argument. The second argument is the time map. (Defaults to the system parameter `TIME`.)

```
str = TIMEC ( time_integer, time_map )
```

`TODAY`

Returns the current calendar date as a Julian integer. The argument is a dummy argument.

```
num = TODAY (0)
```

`TRIM`

Returns the specified string with trailing blanks removed.

```
str = TRIM ( string )
```

`UPPER`

Returns the string specified with all lowercase letters converted to their uppercase equivalent.

```
str = UPPER ( string )
```

`VALLAB`

Returns the value label associated with the current value of the specified column_name.

```
str = VALLAB ( column_name )
```

Aggregation functions

Aggregation functions return a single value for all of the relevant rows processed. See SELECT for the effect aggregation functions have on the `SELECT` process. The aggregation functions are:

`AVG ([UNIQUE] numeric_col)`

Returns the average or mean value of the non-missing values for numeric columns. If `UNIQUE` is specified then only unique values are used to calculate the mean.

`COUNT ( [ UNIQUE ] col | * )`

Returns the number of non-missing values encountered. If `UNIQUE` is specified, then only the unique values add to the count. An asterisk as the argument returns the number of all rows selected regardless of whether the values are valid, missing or undefined.

`FIRST ( col)`

Returns the first non-missing value encountered. The type of variable returned corresponds to the type of the variable being referenced.

`LAST (col)`

Returns the last non-missing value encountered. The type of variable returned corresponds to the type of the variable being referenced.

`MAX (col)`

Returns the maximum non-missing value encountered. The type of variable returned corresponds to the type of the variable being referenced.

`MIN (col)`

Returns the minimum value of the non-missing values. The type of variable returned corresponds to the type of the variable being referenced.

`STD ( [ UNIQUE ] numeric_col)`

Returns the standard deviation of the non-missing numeric values. If `UNIQUE` is specified, then only the unique values are used in calculating the standard deviation.

`SUM ( [ UNIQUE ] numeric_col )`

Returns the sum of the non-missing values. If `UNIQUE` is specified, then only the unique values are summed.

System Tables

System tables contain information about the database(s) and tabfile(s) currently connected. `SELECT` can be used to retrieve information from these in the same manner as from any other table or view though the menus can be used to access much of the same information in a more convenient manner.

As an example of using these tables, the following three `SELECT`s use the $REC system view which describes records. The first query retrieves all the information about each record in the default database; the second retrieves all the information about each record in the `COMPANY` database; the third retrieves all the information about the `OCCUP` record in the `COMPANY` database.

```
SELECT * FROM $REC
SELECT * FROM COMPANY.$REC
SELECT * FROM COMPANY.$REC WHERE RECNAME EQ 'OCCUP'
```

Most of the system tables are views. That is, they are not physical tables but are representations of the data presented by SQL as tables.

Database System Tables

Database system tables are all views and access information for a single database at a time. These views can be referenced on the `FROM` clause of the `SELECT` statement as: [database.]viewname

$DBCASE

Case schema information.

$DBDOC

Case and record document text.

$DBSTATS

Database general information.

$REC

Record schema information.

$SORTID

Record sort-ids or keys.

$VALLABEL

Variable value labels.

$VALVALUE

Variable valid values.

$VAR

Variable schema information.

$VARLABEL

Variable labels.

Tabfile Views and Tables

`$PASSWORD, $SECURITY` and `$VALUE_LABEL` are tables; all the others are views. The tabfile views and tables can be referenced on the `FROM` clause of the `SELECT` as: [tabfile.] viewname

$COL

Column schema information.

$INDEX

Index definitions.

$INDEXCOL

Index column definitions.

$PASSWORD

Group and user names and passwords.

$SECURITY

Tabfile and table permissions.

$TAB

Table schema information.

$TFSTATS

Tabfile general information.

$TRANGE

Column valid and missing ranges.

$VALUE_LABEL

Value labels.

$COL - Table Columns Schema

Contains one row for each column (or variable) in each table of the tabfile. The columns are:

`TABFILE`

Tabfile name.

`TABLE`

Table name.

`VARNAME`

Variable name.

`VARTYPE`

Variable type.

`VARLEN`

Variable length.

`VARLABEL`

Variable label.

`SCALE`

Variable scaling factor.

`BIAS`

Integer bias factor.

`NRANGES`

Number of missing or valid ranges.

`MAP`

Variable display map.

`DECIMAL`

Number of decimal places displayed.

`FILL`

Fill character for display.

`LZERO`

Leading zero character.

`LNEG`

Leading negative character for display.

`LPOS`

Leading positive character for display.

`MISSING`

Missing value character.

`NULL`

Missing value string.

`FORMAT`

Format for printing number.

`SEPARATOR`

Separator character to left of value.

`THOUSANDS`

Thousands separation character.

`TNEG`

Trailing negative character.

`TPOS`

Trailing positive character.

`VALLABS`

Value labels defined (Yes/No).

ZERO

String printed for hard zero.

$DBCASE - Database Case Schema

Contains only one row with general database parameters. The columns are:

`DBNAME`

Database name.

`UPLEVEL`

Update level.

`CASEID`

CASEID variable name.

`CASEIDOR`

`CASEID` order (ascending or descending).

`CASEIDTY`

`CASEID` variable type.

`NCASES`

Number of cases.

`NRECS`

Number of records.

`NTEMPS`

Number of temporary variables.

`NVARS`

Number of variables.

`MAXCASES`

Maximum number of cases.

`MAXRECS`

Maximum number of records.

`MAXRECTY`

Maximum number of defined records. (Different record definitions not individual records in the database.)

$DBDOC - Database Documentation

Contains one row for each line of documentary text describing the database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`LINENUM`

Line number of text line.

`LINE`

Line of documentary text.

$DBSTATS - Database Statistics

Contains only one row with database statistics. The columns are:

`DBNAME`

Database name.

`UPLEVEL`

Update level.

`CREDATE`

Creation date.

`CRETIME`

Creation time.

`CHNGDATE`

Date of most recent update.

`CHNGTIME`

Time of most recent update.

`NCASES`

Number of cases.

`NRECS`

Number of records.

`NVARS`

Number of variables.

`NTEMPS`

Number of temporary variables.

`AVGRECS`

Average number of records per case.

`CASEIDSZ`

Size of CASEID in bytes.

`CIRLEN`

Length of CIR in SIR/XS words.

`KEYSIZE`

Key size in bytes.

`ACTDATB`

Number of active data blocks.

`INADATB`

Number of inactive data blocks.

`DATBLKSZ`

Data block size.

`MINDATSZ`

Minimum size of data record in SIR/XS words.

`MAXDATSZ`

Maximum size of data record in SIR/XS words.

`ACTINDB`

Number of active index blocks.

`INAINDB`

Number of inactive data blocks.

`INDBLKSZ`

Size of index block in SIR/XS words.

`INDEXLEN`

Length of index in SIR/XS words.

`MAXINENT`

Maximum number of index entries in one block.

`MAXRECVR`

Maximum number of variables in any one record.

$INDEX - Tabfile Index Definitions

Contains one row for each index in the specified tabfile. The columns are:

`TABFILE`

Tabfile name.

`TABLE`

Table indexed.

`INDEX`

Name of index on table.

$INDEXCOL - Tabfile Index Column Definitions

Contains one row for each column in each index on tables in the specified tabfile. The columns are:

`TABFILE`

Tabfile name.

`TABLE`

Table indexed.

`INDEX`

Name of index on table.

`COL`

Name of the column in index.

$PASSWORD - Group User Names

Contains one row for each group and group-user name. Only a DBA has the authority to view this table. The columns are:

`GRPNAME`

Group name.

`USERNAME`

Name of the user within a group.

$REC - Database Record Schema

Contains one row for each defined type of record in the database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`COUNT`

Number of records of this type in the database.

`IDCNT`

Number of sort ids (including the case id).

`LENGTH`

Length of records in SIR/XS words.

`LOCK`

Lock status (YES/NO).

`MAX`

Max number of records of this type per case.

`VARCNT`

Number of record variables in this record.

$SECURITY - Tabfile and Table Permissions

Contains a row for the tabfile, and a row per table and user permission. Only a DBA has the authority to view this table. Each column which refers to a permission holds one of three values, N, Y or G. These mean respectively, no permission, permission and permission with the ability to grant this permission to others. The columns are:

`GRPNAME`

Group name holding these permissions.

`USERNAME`

User holding these permissions.

`TABLE`

Table name.

`COLADD`

Able to add columns.

`TABFILECONNECT`

Able to connect tabfile.

`TABCREATE`

Able to create tables.

`DBA`

Able to act as Database administrator.

`COLDELETE`

Able to delete columns.

`ROWDELETE`

Able to delete rows.

`TABLEDROP`

Able to drop table.

`INDEXCREATE`

Able to create or drop an index.

`ROWADD`

Able to add rows.

`COLMOD`

Able to modify columns.

`SELECT`

Able to select rows.

`ROWMOD`

Able to modify rows.

`GRANTERGRPNAME`

Group granting permission to this group or user.

`GRANTERUSERNAME`

User granting permission to this group or user.

`COLPERM`

Columns permissions if a table permission entry in column order.

$SORTID - Sort Id Variables

Contains one row for each keyfield (sort id) of every defined record in the database. The columns are:

`RECNUMB`

Record number.

`RECNAME`

Record name.

`VARNAME`

Variable name.

`ORDER`

Sort order (Ascending or Descending) of the sort id variable.

`TYPE`

Variable type.

$TAB - Tables

Contains one row for each table on the tabfile. The columns are:

`TABFILE`

Tabfile name.

`TABLE`

Table name.

`UPLEVEL`

Update level.

`DATECREATE`

Date of creation.

`TIMECREATE`

Time of creation.

`DATEUPDATE`

Date of last update.

`TIMEUPDATE`

Time of last update.

`NROWS`

Number of rows in table.

`NCOL`

Number of columns in table.

`NINDEX`

Number of indexes associated with table.

`MAXROWS`

Maximum length of each row in bytes.

`LENGTH`

Length of fixed area of each row in bytes.

`NBLOCKS`

Number of data blocks in the table.

`NROWDELETE`

Number of rows deleted.

`PADDING`

Padding percentage (1-99).

$TFSTATS - Tabfile Statistics

Contains one row for each tabfile currently connected. The columns are:

`TABFILE`

Tabfile name.

`DATECREATE`

Date of creation.

`TIMECREATE`

Time of creation.

`DATEUPDATE`

Date of most recent update.

`TIMEUPDATE`

Time of most recent update.

`NTABLES`

Number of tables on tabfile.

`TABFILELDI`

Tabfile filename (logical dataset identifier).

`BLOCKSZ`

Tabfile block size.

`JOURNAL`

Internal filename of journal file if defined.

$TRANGE - Tabfile Column Ranges

Contains one row for each missing or valid range for any column in every table of the specified tabfile. The columns are:

`TABFILE`

Tabfile name.

`TABLE`

Table name.

`VARNAME`

Variable (or Column) name.

`RANGETYPE`

Type of range
'Valid' or 'Missing' for values
'Valid [,]' or 'Missing [,]' for ranges.

`LOW`

Minimum value.

`HIGH`

Maximum value.

$VALLABEL - Database Value Labels

Contains one row for each value label of each variable of every defined record for the database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`VARNAME`

Variable name.

`NVAL`

Numeric value.

`SVAL`

String value.

`LABEL`

Value label.

$VALUE_LABEL - Tabfile Value Labels

Contains one row for each value label in tables of the specified tabfile. The columns are:

`TABLE`

Table name.

`VARNAME`

Column (or variable) name.

`NVALUE`

Value if the column is numeric.

`SVALUE`

Value if the column is a string.

`LABEL`

Value label.

`MVALUE`

Reserved for future use.

`REFCOUNT`

Reserved for future use.

$VALVALUE - Database Valid Values

Contains one row for each valid value of each variable of every defined record for the current database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`VARNAME`

Variable name.

`NVAL`

Numeric value.

`SVAL`

String value.

$VAR - Database Variables

Contains one row for each variable of every record type of the current database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`VARNAME`

Variable name.

`LABEL`

Variable label.

`TYPE`

Variable type.

`LENGTH`

Variable length in bytes.

`NMIN`

Minimum numeric value.

`NMAX`

Maximum numeric value.

`SMIN`

Minimum string value.

`SMAX`

Maximum string value.

`MISS`

Number of missing values.

`NMISS1`

Numeric missing value 1.

`NMISS2`

Numeric missing value 2.

`NMISS3`

Numeric missing value 3.

`SMISS1`

String missing value 1.

`SMISS2`

String missing value 2.

`SMISS3`

String missing value 3.

`MAP`

Date or time map.

`SCALE`

Scaling factor.

`VALLABS`

Whether value labels have been defined.

`VVALS`

Whether valid values have been defined.

$VARLABEL - Database Variable Labels

Contains one row for each line of label information for each variable of every defined record for the database. The columns are:

`RECNUM`

Record number.

`RECNAME`

Record name.

`VARNAME`

Variable name.

`LINENO`

Line number.

`LABEL`

Label text.

Reserved Keywords

Certain keywords are reserved for use by SQL. Avoid using them out of context. If you must use one of these words as a column or file name, enclose the word with quotation marks:

```
SELECT varname "keyword" FROM table_name
```

The longest form of each reserved word is shown; avoid direct contractions of these words or plurals as these are also reserved.

ADDCOL	DISTINCT	LOWER	READ	TIMEC
AGGREGAT	DOUBLE	LOWERCASE	REAL	TINYINT
ALL	DPL	LPOS	RECALL	TNEG
ALTER	DPLACES	LPOSSIGN	RECCOUNT	TNEGSIGN
AS	DROP	LRECL	RECFM	TO
ASC	DTL	MAX	RECLIM	TOTALS
ATTACH	ECHO	MAXIMUM	RECSIZE	TPL
ATTRIBUTE	EDIT	MEM	RELATION	TPOS
AUTO	EJECT	MEMBER	REM	TPOSSIGN
AUTODISP	ENTER	MEMORY	REMARKS	TREE
BACK	ERASE	MINIMUM	RENAME	TRIPLE
BACKWARD	ESCAPE	MISS	REPORTS	TT
BATCH	EXCEPT	MISSCHAR	REV REVERSE	TTITLE
BL	EXCL	MISSING	REVOKE	UC
BLANKS	EXEC	MOD	RIGHT	UCOL
BLKSIZE	EXPERT	MODCOL	RLIM	UHEAD
BLOCKS	EXPONENT	MODES	RT	UNDCOL
BOLD	EXPORT	MODHIST	SAMPLE	UNDEFINE
BR	FAM	MODIFY	SAVE	UNDERCOL
BREAK	FAMILY	MODS	SCALE	UNDERHEAD
BT	FILENAME	MONITOR	SCR	UNDHEAD
BUFFERS	FILES	NAME	SCRATCH	UNION
BUFNO	FILL	NEG	SCREEN	UNIQUE
BUT	FIRST	NOCOUNTS	SECURITY	UNQ
BY	FIXED	NOLABELS	SEL	UPDATE
BYE	FLOAT	NOSECURITY	SELECT	UPPER
CALL	FMT	NOTABFILE	SELLIM	UPPERCAS
CASE	FOOT	NOTNULL	SEP	USING
CASELESS	FOOTING	NOVICE	SEPARATE	USRTSIZ
CASELIM	FOR	NOWAIT	SEPARATOR	VALLAB
CATALOG	FORMAT	NOWORKSPACE	SET	VALUES
CATEGORICAL	FROM	NULL	SETTING	VARCHAR

CENTER
CHARACTER
CLEAR
CLIM
CLR
CMD
CMDINCR
CMPTRIM
CMPUPPER
COL
COLHEAD
COLUMNS
COM
COMMANDS
CONTINUE
CONTROLS
CREATE
CTR
DATABASE
DATE
DATEC
DB
DBA
DBL
DBMS
DEBUG
DEC
DECIMAL
DECSIGN
DELCOL
DELETE
DESC
DET
DETAIL
DIFF
DISCONNECT
DISPLAY
DIST
GET
GRANT
GROUP
GROUPING
GRP
GRPSIZE
HAVING
HCTR
HEAD
HEADCENT
HEADCTR
HEADING
HOLD
HOST
IN
INCL
INCLUDE
INDENT
INDEX
INDICES
INPUT
INSERT
INTEGER
INTER
INTO
JOURNAL
L
LAB
LABEL
LAST
LC
LEFT
LIMITS
LININCR
LIST
LNEGSIGN
LOCAL
LOG
NUMERIC
OBSERVATION
OFF
ON
ONLY
ORDER
OUT
OUTER
OUTLINE
OUTPUT
P
PAGE
PAGEHEAD
PAGELIM
PAGES
PAGESIZE
PAGING
PASSWORD
PATH
PATHLESS
PATHS
PCTFREE
PERMHIST
PG PGH
PGHEAD
PGLIM
PGS
PGSIZE
PREFIX
PRESET
PRINT
PROC
PROCEDURE
PROCS
PROMPT
PS
PW
R
SGL
SHOW
SINGLE
SLIM
SMALLINT
SORT
SPACE
SPACEC
SPACED
SPACES
SPACET
SPARSE
SPLIT
SPSS
SRLEN
SRTSIZE
START
STATS
STORE
STOT
STRUCTURE
SUBFILE
SUBTOTAL
SUPPRESS
SYN
SYNONYMS
SYNS
SYSTABS
T
TAB
TABFILE
TABLE
TABSIZE
TEXCLUDE
TF
THOUSAND
THOUSIGN
TIME
VARIABLE
VCHR
VIA
VIEW
VOL
VOLUME
WEIGHT
WHERE
WIDTH
WORK
WRAP
WRITE
X
ZERO

Pattern Matching

Pattern matching applies to the use of the LIKE keyword in a WHERE clause. This feature enables the finding of text strings with particular characteristics such as all starting with the same character.

Patterns are described by the use of symbol characters together with ordinary characters which are to be matched in the string being searched.

Characters in a pattern are taken literally unless they are one of the pattern matching symbols described below. For example,

```
WHERE  ADDRESS  LIKE  'Ave'
```

finds all values of ADDRESS containing the string "Ave". The string "Ave" may appear anywhere. This is different to the EQ relational operator, as in:

```
WHERE  ADDRESS  EQ  'Ave'
```

This condition will only be true when ADDRESS is exactly equal to the string "Ave".

For example, to find the name of everyone whose first name starts with "B" and second name starts with "L":

```
SELECT ID  NAME  CURRPOS  -
FROM    EMPLOYEE   -
WHERE   NAME   LIKE '%B?*L'
```

searches for and finds all rows in the EMPLOYEE table in which NAME starts with the letter "B" followed by zero, one, or more intervening characters followed by the letter "L".

Trim and Upper

Pattern matching is affected by system parameters CMPTRIM and CMPUPPER. CMPTRIM causes trailing blanks of strings to be trimmed before they are compared. CMPUPPER maps all strings to upper case before the comparison takes place. Generally, when using the LIKE function, SET CMPTRIM and CLEAR CMPUPPER. By default, both parameters are SET.

Symbols

The symbols are:

`%`

beginning of line

`$`

end of line

`?`

match any single character

`[`

start a character class

`-`

range of characters

`]`

end a character class

`!`

negate a character class

`*`

closure, zero or more occurrences

`+`

closure, one or more occurrences

Beginning of the Line %

The % character specifies searching for patterns at the beginning of a string variable.

To find all rows that have a string variable which begin with the word PROCESS, use:

```
... WHERE variable LIKE '%PROCESS'
```

This returns only those rows that begin with the string "PROCESS". It does not return rows containing "PROCESS" in the middle of the variable such as "END PROCESS" or "EXIT PROCESS".

End of the Line $

The $ character specifies searching for patterns at the end of a string variable.

To search for all records in which the NAME column ends in "smith" :

```
SELECT  ID  NAME   FROM  EMPLOYEE  -
         WHERE   NAME  LIKE 'smith$'
```

Match Anything Character ?

The character ? matches any single character. For example,

```
... WHERE NAME LIKE 'A?e'
```

finds names containing strings such as:

```
Aae  Abe  Ace   .......   Axe  Aye  Aze
```

and also:

```
A+e  A-e  A*e  A/e  A,e  A.e  A(e  A)e  A'e  A"e
```

The match anything character can appear more than once in a pattern. The next example, selects all records in which the customer identifier begins with the letters AC followed by any three characters followed by a 9. Notice the use of two symbols, the % and the ?.

```
SELECT  CUSTID  CUSTNAME ADDRESS PHONE  -
         FROM  CUSTFILE  -
         WHERE   CUSTID  LIKE  '%AC???9'
```

Classes of Characters [...]

Search for a class or set of characters by enclosing them in square brackets. Some examples of character classes are:

```
[12]
```

match all instances of "1" or "2" or both

`[123]`

match all possible combinations of "1", "2", and "3"

`[a-z]`

match lowercase letters

`[A-Z]`

match uppercase letters

`[0-9]`

match decimal digits

`[J-Q]`

match uppercase letters "J" through "Q"

`[A-Za-z]`

match uppercase and lowercase letters

For example, to locate information on 2005 accounts. The account identifiers for 2005 begin with an uppercase letter followed by the string "2005". The Where clause might be:

```
... WHERE  ACCTID  LIKE  '%[A-Z]2005'
```

Negated Character Class [!...]

To match all lines except those containing the members of a character class, place the negation character ! at the beginning of the class inside the square brackets. For examples:

`[!12]`

match all characters except "1" and "2"

`[!a-z]`

match all characters except lower case letters

`[!A-Z]`

match all characters except upper case letters

`[!0-9]`

match all characters except decimal digits

`[!J-Q]`

match all characters except upper case letters "J" through "Q"

`[!A-Za-z]`

match all characters except upper and lower case letters

For example, to search and delete all rows in which the value of DEPTNUM is not composed entirely of digits.

```
DELETE  FROM  TCOMPANY.TAB1  -
         WHERE   DEPTNUM  LIKE '[!0-9]'
```

Closure Character (Zero or More Occurrences) *

To search for strings or patterns of characters that occur an indefinite number of times (known as a closure) specify the closure character "*" after the required pattern.

Some examples of closure patterns are:

`a*`

zero or more occurrences of lowercase a

`[A-Z]*`

zero or more uppercase letters

`[Q3x]*`

zero or more occurrences of "Q" or "3" or "x"

`[a-zA-Z]*`

zero or more letters, upper or lower case
this pattern matches a word of text or a null string

For example, to search for all text that appears inside a pair of parentheses :

```
... WHERE STRING LIKE '(?*)'
```

The pattern requests all lines that contain "(" followed by zero or more occurrences of any character followed by ")".

Similarly, to search for all Illinois accounts. These are identified in the ACCTID when characters 3 and 4 are "IL" and the last two (verification) digits are "13". The ACCTID may be 10 to 17 characters long and therefore the last two digits may appear in positions 9-10, 10-11, ..., 16-17. The closure character takes care of this problem.

```
SELECT  *  FROM  ACCOUNTS  -
         WHERE   ACCTID  LIKE  '%??IL?*13$'
```

Note the beginning and end of line characters. The % character followed by ??IL makes sure that "IL" appears in position 3 and 4. The $ character preceded by 13 makes sure that "13" appears as the last two digits. The ?* notation means that any number of characters can appear between "IL" and "13".

Closure Character (One or More Occurrences)+

This specifies a search for one or more occurrences of a pattern instead of zero or more occurrences. For example, the following command:

```
... WHERE STRING LIKE ' [aehrt]+ '
```

searches for complete words made up of the letters a,e,h,r,t.

The search pattern requests strings containing a blank followed by one or more occurrences of the letters a,e,h,r,t followed by another blank. The lines listed contain words such as "a", "are", "at", "here", "rather", "that", "the", "there", "three", etc.

Escape Character @

The symbols are instructions sent to the pattern matching routine. Occurrences of these symbols cannot be searched for in the normal way. A search for question marks in a field cannot be specified as:

```
... WHERE string LIKE '?'
```

because this command will match every character.

The escape character @ is provided to handle this situation. Precede any symbol character with @, and the character is treated literally. Thus to search for question marks enter:

```
.... WHERE string LIKE '@?'
```

In addition, symbols lose their meaning when they appear out of context (i.e. the escape character should not be used) as follows:

`%`

: when not at the beginning of the pattern

`$`

: when not at the end of the pattern

`*`

: at the beginning of the pattern

`+`

: at the beginning of the pattern

`!`

: not at the beginning of a character class

`-`

: at the beginning or end of a character class

Symbols do not apply in the specification of a character class except for:

`!`

: at the beginning of the character class

`-`

: in the middle of the character class

`@@`

: anywhere in the character class