SIR/XS Databases

This section of the documentation describes how to manage databases using SIR/XS. Anyone intending to create or change the definitions of a database or act as a database administrator should be familiar with this material. If you are simply going to access SIR/XS databases using applications developed by someone else, then the Database Overview provides a good introduction.

SIR/XS database management is covered under a number of topics:

- Introduction outlines the major database facilities in SIR/XS.
- Database Definition Commands describes the commands to define databases and records. Use these same commands to modify existing definitions together with some additional commands that only apply to updating existing definitions.
- File Input describes utilities that load data into a database from text files.
- Writing Data and Backing Up describes utilities that produce copies of a database and the verification and recovery utilities, including utilities to upload changes from another database, to merge data from another database and to reload a database.
- Tabfiles and Tables provides an overview of the various facilities available for the creation and maintenance of tabfiles and tables.
- Tuning and Efficiency discusses the internals of the SIR/XS database physical organisation.

Virtually all of the features are available through the menus and dialogs and this is the normal way to manage a database on a regular basis.

SIR/XS provides a concurrent environment if multiple people need to update a database at the same time across a network.

Overview

There can be any number of different databases for different applications. A SIR/XS session can have any number of databases connected at any one time and one of these is the current or *default* database. Any database operations happen on the default database. SQL, Forms, PQLForms and VisualPQL can access multiple databases.

A database is made up of one or more **record types**. Each record type contains **variables**. Each record type may have one or more variables defined as **keys**. Keys are used to identify each individual record.

A record may have secondary indexes allowing access by VisualPQL and PQLForms through values in non-key variables.

The definition of the database consists of overall information about the database plus definitions for each record type with definitions of each variable and any secondary indexes. This is collectively known as the *Schema*.

Names

All of the various SIR/XS entities such as records and variables have names that must conform to the same set of rules. A name can be a *standard name* that is up to 32 characters long, does not begin with a number and contains only letters, numbers and four characters (dollar sign $, hash sign #, at sign @, underscore _). Letters in standard names are translated to uppercase. A *non-standard* name can be used that does not conform to these rules, for example it might contain lower case characters or have embedded spaces. Non-standard names can be up to 30 characters and must be enclosed in curly brackets {...}. Non-standard names can contain any character (except curly brackets) and, where entities are listed by name, non-standard names are in the appropriate sort sequence position in the list.

Records and variables are retrieved by name and applications do not need to know anything about the physical organisation of data. i.e. Applications are independent of the physical structure of data.

Operating system files used by SIR/XS applications are assigned an internal name known as an Attribute. This is used in place of the full filename. If specifying full operating system filenames, it is best to enclose these in quotes.

Every application that accesses data from a database uses the schema, which means that the information is always verified and conforms to the schema definition. An application can access any record type within a database and can access multiple record types for complex processing. The schema can be defined through interactive dialogs or by creating a set of commands and running them in a similar way to a running a program. The resulting database is exactly the same regardless of the way that it is defined.

The database definition can be modified even after the data has been loaded without, in many cases, having to `UNLOAD` and `RELOAD` data.

There is a second method to store data known as **Tables**. A table holds a single record type and is independent of any database. Tables are held on **tabfiles**. A tabfile can hold many tables. An application can operate on many tables and many tabfiles.

CASE Structured Databases

SIR/XS is a true relational database and databases can be created that are simply sets of record types. However SIR/XS also allows databases that can take advantage of structure in the data:

Some databases have a natural structure known as a *Case* structure. For example, a medical database with information about patients could have a number of record types such as patient demographics, visits, procedures, followups, etc., most with multiple occurrences but all of the information about a single individual makes up a *case*. i.e. A case is a set of records that all refer to one single entity.

A case structure can be easier to use. Queries are simpler to write and less error prone within a case. A case structure can be faster. Since all of the information for any given case is available with a minimum of searching, access is very fast.

A case structure can be found in many applications. The following guidelines may help determine whether a particular set of data has a case structure.

- If most of the information belongs to some given single entity. For example; is the information all about Employees, or Prisoners, or Drill Holes, or Patients, or Samples ...?
- Is the information entered from forms and stored in a number of record types that all apply to a single entity? For example, with questionnaire data, the questionnaire number may be a case identifier.
- Is there some basic reporting unit in the application? If so, this unit might well be the case. Are results all about a given site or a particular experiment?
- Do most or all of the records have a hierarchical structure branching down from a single record.
- Is there a key variable in common across all record types that would qualify as a case identifier?

The Common Information Record

In a case structured database some information is maintained about each case. This is held in the **Common Information Record** or CIR. The CIR is a summary of data about the case. It holds the case identifier, counts of records belonging to the case and other specified common variables. It typically holds values that only occur once in the case although it can hold a copy of the last value entered for variables that occur multiple times in the case.

CASELESS Databases

Records in a **caseless** database have no single common element that relates them. If a single database has various records types that are each independent from the other (e.g. Parts, Customers, Employees), then it might be a caseless database or perhaps the various entities might be held in individual case structured databases.

Keys

Each record in a SIR/XS database is unique and that uniqueness is defined in terms of the values of **keys**. A record may have one or more variables that make up the key. No two records in the database can have the same combination of record type and key values.

The main index that SIR/XS maintains to locate the records is built from the key and the key determines how the record is physically stored in the database.

Because the keys go into the index, they are relatively more "expensive" to maintain than non-key variables. Do not declare more keyfields than needed to make records unique or to define relationships. Whenever choosing variables to use for a key, use variables that are short and well defined. Use numeric codes wherever possible, particularly in place of long alphabetic strings. Using strings as keys leads to misspellings and possible confusions as well as being inefficient. Avoid floating point numbers as key fields.

Secondary indexes can be used to retrieve data in other sequences.

Order of Variables in Keys

In keys made up of multiple variables, the order in which the variables are specified in the key is important.

Specifying the complete key in a retrieval is the most efficient way to retrieve a single record. This can use the main index and directly retrieve the record.

A retrieval can locate records using just part of the key providing that it specifies the higher level key variables. Make higher level keys the ones more likely to be used to retrieve by.

Joining Records with Keyfields

Data from two record types can be joined by using a keyfield that is common to both record types. If keyfields in different record types have the same name and are in the same order, these record types are implicitly joined. A record type with one extra keyfield defines a hierarchy. For example, one record type may have `CUSTNO` as the key, a second record type `CUSTNO` and `ORDERNO`, a third record type `CUSTNO`, `ORDERNO`, `LINENO`, etc. These can go down through many levels of record types if needed, and the implicit joins make retrievals simpler and more efficient.

Defining Variables

Every variable in a database has a name and a structured definition. Whenever a variable is entered into the database, it is checked to ensure that it conforms to the definition. Variables are always referred to by their name. Descriptive names are usually best. For example, a variable called '`PHONE`' is easier to remember than one called '`VAR112`'.

Within a record type, variable names must be unique. Variables in different record types may have the same name. Beware of using the same name for different things in different record types in the same database. For example, it would be a mistake to use the name '`DATE`' for variables when they are different dates, such as Birth Date, Visit Date, etc.

Variable Data Types

Every variable is one of three basic data types:

`INTEGER`

Integers are the set of natural numbers both positive and negative including zero.

`FLOATING POINT`

Floating point or real numbers, are numbers that may have decimal characters. They may be either single or double precision.

`STRING`

String or character variables are variables whose values are alphanumeric.

Variable Extended Data Types

Although every variable is stored as one of the three basic types, there are several extended data types that translate information as they are entered or retrieved. These are:

`CATEGORICAL`

Categorical variables offer an efficient way to store predefined strings.

A categorical variable is a character string that has a limited number of values specified as an ordered list. When the variable is input as a string, it is compared to the list and the number that corresponds to the matching position in the list is stored instead of the value. This has the advantage that only valid entries are held and considerable space is saved. In programs and reports, the full string is displayed and retrieved.

For example, a categorical variable might be a list of the names of American states. If 'Alabama' were the first entry in the list, a variable entered as 'Alabama' results in '1' being stored.

The list is held in the data dictionary and is searched sequentially. It is a very simple and easy to use facility for short lists that are not updated very often. If there are hundreds or thousands of entries, or there is more information about each entry than just the name, or users need to modify the entries, use tables or record types with indexes to store this type of reference data.

`DATE`

A date variable is typically entered as a character string which is converted to an integer that is the number of days since the start of the Gregorian calendar. Day 1 is October 15, 1582. A date has a format such as `'MM/DD/YYYY'` that is used to interpret the input and to format the output. Within PQL programs, dates can be used as numbers for calculations or as character strings for display or input purposes. As dates are stored as the number of days since a predetermined point in the past, it is very simple to perform date based calculations and there are no potential problems at the end of centuries or millennia. There is currently an upper limit of Feb 28 3000 (day 517686) on the conversion of numeric days to/from the calendar.

The date format is a string, up to 32 characters, in quotes and describes both an input and an output format. The input format is used when data is read during batch data entry, or when a string value is assigned to the variable. The format is used to interpret the input data (see below). The output format is used as a default when the variable is written and the output date corresponds exactly to the format specified (this can be overridden by other date specifications at that point).

The date format consists of a combination of letters with special meanings and other characters used as separators. The letters M(month), D(day), Y(year), W(day of week/week number) and I (default separator) are special characters (upper/lower case can be used); all other characters in the format are treated as separators. The 'I' separator results in a blank separator on output. Like characters must be kept together, e.g. a format such as `'MYYYM'` is invalid.

Date formats can be partial formats, without a day, month or year component. If the year is omitted, it is taken to be the current year. If the decade is omitted, it is taken to be the current decade. If the day or the month is omitted, they are taken to be 1. If there are more than two month characters, then English month names are used to the length specified. Names on output are in upper/lower case to match the case of the format. There are two special formats which do not have a month component. A date can be a year/day format (in either sequence) which must allow at least three characters for days (1-365). A date can be a year /week format (in either sequence) which should allow at least two characters for week numbers (1-53). (Week numbers follow the ISO standard; a week always begins on a Monday, and ends on a Sunday. The first week of a year is that week which contains the first Thursday of the year, or, equivalently, contains Jan-4.) The 'W' character, if not in a week number format, represents the English name for days of the week to the length specified. Names on output are in upper/lower case to match the case of the format. If month names or days of the week names are specified, on output all dates have the same length name. If the specific name is shorter than the format it is padded with blanks.

On input, if a date format has no separators and the input value has no separators, the input must correspond exactly to the format. e.g. format `'DDMMYYYY'` input must have leading zeroes when necessary such as '05062006'. If the input has separators, each component is taken to be variable length up to the separator or to the end of the input field regardless as to the specification of separators in the format. e.g. for format `'DD/MM/YYYY'` or format `'DDMMYYYY'` input could be '5/6/6'. Separators do not have to match specific characters. e.g. A '/' can be specified in the format and the input could contain a blank.

Months can be input as names (or partial names) or as month numbers regardless as to the format. Days of the week have no relevance on input and any text is skipped.

If the full four digits of the year are not entered, the input is tested against the century split parameter `CENY`. This is 1930 by default and can be set by the application. An input year greater than the century split is set to the specified century; years less than this are set to the next century. e.g. 99 becomes 1999; 7 becomes 2007. This calculation is not dependent on the current date in the system and therefore does not alter at any particular point in time.

For example, various date maps allow a sample of possible inputs and how the date is displayed as follows:

Format	Possible Inputs	Displayed Date
'mmddyyyy'	05312006	05312006
'mmddyyyy'	5 31 2006	05312006
'MMIDDIYY'	5/31/6 or 5 31 2006	05 31 06
'MM/DD/YYYY'	5-31-6 or 5 31 06	05/31/2006
'DD-MM-YY'	31/5/2006 or 31 May 6	31-05-06
'YYYY'	2006 or 6	2006
'MMM DD, YY'	As per other M/D/Y formats	MAY 31, 06
'WWW, DD MMM YYYY'	31/05/06 or xxx 31/5/2006	FRI, 31 MAY 2006
'Mmm/DD/YYYY'	As per other M/D/Y formats	May/31/2006
'Www, Mmm dddd'	xxxx 05/31	Fri, May 31st
'yyyy/ww	6/13	2006/13 (week number)
'ddd/yy'	85/2006	085/06 (day number)

If a datemap starts with the letter "E" then this is an *exact* date map and the value input into the date variable must match the map exactly. The E flag is not a part of the map but indicates that the following format is mandatory: digits must be entered for each M,D and Y in the map and their positions map exactly to the column positions in the datemap. Leading zeros must be entered. Numbers cannot be entered in columns that are not mapped to M, D or Y.

Format	Input	Value
'Emmddyyyy'	05312006	accepted
	5312006	rejected (needs leading zero)
	05 31 2006	rejected (misaligned columns)
'EMMIDDIYY'	05/31/06	accepted
	05031/06	rejected (number out of place)
'EMMMIDDIYYYY'	MAR 31 2006	rejected (numbers required)
	05 31 2006	rejected (need all leading zeros)
	005 31 2006	accepted

`TIME`

A time variable is typically entered as a character string and stored as an integer that is the number of seconds since midnight. A time has a time format such as `'HHMMSS'` that is used to interpret the input and format the output.

The time format is a string, up to 32 characters, in quotes and describes both an input and an output format. The input format is used when data is read during batch data entry, or when a string value is assigned to the variable. The format is used to interpret the input data (see below). The output format is used as a default when the variable is written and the output time corresponds exactly to the format specified (this can be overridden by other time specifications at that point).

The time format consists of a combination of letters with special meanings and other characters used as separators. The letters H(hour), M(minute), S(second), I(default separator) and P or A (AM/PM) (upper/lower case can be used); all other characters in the format are treated as separators. The 'I' separator results in a colon **:** separator on output. Like characters must be kept together, e.g. a format such as `'HMMH'` is invalid.

A time is normally taken to be a 24 hour time format. Specify 'PP' ('A' is a synonym for 'P') for a 12 hour format. On output 'AM' or 'PM' are written as appropriate. On input, a 'P' in the data indicates a 12 hour time after midday (a 24 hour time is still valid on input).

On input, if a time format has no separators and the input has no separators, the input must correspond exactly to the format. e.g. format `'HHMM'` input must have leading zeroes when necessary such as '0805'. If the input has separators, each component is taken to be variable length up to the separator or to the end of the input field. e.g. for format `'HH:MM'` or `'HHMM'` input could be '8:5'. Separators do not have to match specific characters. e.g. A **:** can be specified in the format and the input could contain a blank.

For example, various time maps allow a sample of possible inputs and how the time is displayed as follows:

Format	Possible Inputs	Displayed Time
'hhmm'	2330 23-30	2330
'hhImm'	23 30 23-30	23:30
'HH:MM:SS'	23 30	23:30:00
'HH MM PP'	23 30 11:30 P	11 30 PM

SCALED

A scaled integer represent numbers multiplied by a power of ten (the power is the scale factor). These can include a decimal component (a negative scale factor) or can be a number with a set number of trailing zeros (a positive scale factor). This is efficient for data that has this characteristic of a fixed scale. For example, Money in dollars and cents would be scaled at -2.

The software handles any scaling issues involved. For example, if a scaled integer has a scaling factor of 2 (hundreds), and it is computed equal to 50 x 4, the database stores a 2. If the variable is printed, 200 is output. If used in another calculation, it would be 200. For all practical purposes it has the value 200, except it saves storage space.

The major limitation on scaled integers is that the maximum integer value is 2,147,483,643. If scaled to a negative power, this may not be large enough. For example, scaling to -2 for money, gives a maximum value of 21,474,836.43. (For larger money values, use whole cents in a double precision floating point variable:R*8)

Variable Formats

Variables can be defined in terms of an external format. These formats are:

`Aw`

A string w long. Strings are stored up to this length with trailing blanks compressed so very long fields (up to 4094) can be defined with little overhead. If using strings as key fields or as fields used in secondary indexes, make them as short as possible. If the variable is a categorical variable, the variable is stored as a number that varies in internal size depending on how many entries are in the list of allowed values.

`Iw`

An integer w long. For integers, the number of external digits (or the maximum size if specified as a range) determines the internal size. For example two digits holds up to 99, but this can be stored in one byte internally. Numbers with one or two external positions are stored in one byte, numbers with three or four external positions are stored in two bytes and numbers with five and over external positions are stored in 4 bytes.

Scaled integers with decimal places (negatively scaled) have external formats identical to floating point, although the internal storage is as an integer.

`Date 'format'`

A date. The external format is a string and the internal format is a 4 byte integer. See date formats for a complete description of external date formats.

`Time 'format'`

A time. The external format is a string and the internal format is a 4 byte integer. See time formats for a complete description of external time formats.

`Fw.n`

A single precision floating point number w long with an assumed n decimal places. Floating point numbers can be defined as single precision regardless of their external format. For example a floating point number might take 15 columns and be single-precision and it would be stored in 4 bytes at single precision accuracy.

`Dw.n`

A double precision floating point number w long with an assumed n decimal places.

When defining the external format of floating point numbers, specify the number of digits after the decimal point. This sets the format for printing and the **default** for input. If a physical decimal point is present in the input, this overrides the format i.e. the default is only used if there is no explicit decimal point.

Floating point numbers can be input in scientific notation, that is +n.nnnE+nnn where n.nnn is the number, E indicates exponentiation and +nnn is the power. For example, if the variable has four decimal places specify D(12) to hold the 12 columns of input required (+1.3456E+009).

Variable Size

The size of a variable is defined by either the external format or internal storage type:

- A variable can be defined in terms of the external characters or format it is input in or displayed as. For example, a date variable with a date format of 'dd/mm/yyyy' takes ten external characters.
- A variable can be defined in terms of the internal storage it takes. A date variable, regardless of date format, is stored as a four byte integer.

For most data definition purposes, specify how the variable looks externally and the appropriate internal format is created automatically. However it is as well to understand these transformations.

Variable Internal Formats

Variables are stored as:

`I* 1 | 2 | 4`

Integers of 1, 2 or 4 bytes.
I*1 have a range of -128 to +123;
I*2 have a range of -32,768 to +32,763;
I*4 have a range of -2,147,483,648 to 2,147,483,643.

`R* 4 | 8`

Floating Point of 4 bytes for single-precision; 8 bytes for double-precision.

To input a negative number (whether integer or floating point), simply precede the number with a minus sign (-).

`STRING*n`

Character Strings of up to 4094 bytes.

Variable Quality Control

Whenever variables are entered, they are checked to make sure that the input conforms to the defined data type, size and format. Several checks may be specified:

`Valid Values`

Valid Values are a list of values that are allowable for a given variable. Input that does not match the valid value list for a variable is rejected.

`Variable Ranges`

Variable Ranges are ranges of values that are allowable for a variable. Input that is outside the ranges is rejected.

`Missing Values`

Missing values define values that are held in the database but excluded in any calculations. Up to 3 missing values can be specified per variable.

For example, a variable might have two valid values such as 'Y', and 'N' meaning 'Yes' and 'No'. A further three missing values might be defined such as 'X' for 'Not relevant', 'R' for 'Refused to Answer' and 'Z' for 'Invalid Answer'. When producing statistics on that variable, only the Y and N answers are included as the others are defined as missing values.

Blanks may be declared as a missing value. If a numeric field is blank on input and blank has been defined as a missing value, the variable is stored as missing. If blanks are not declared as a missing value for the variable, zero is stored when a numeric field is blank on input.

`Batch Data Input`

Certain checks may be specified that are applied by the Batch Data Input utilities. Consistency checks between variables in a record type can be specified. The values of data items can be computed before being stored in the database.

Variable Label

Variables on screen displays and printed output can be displayed with the variable name or with an optional, 78 character label.

Variable Documentation

The schema can hold documentation about a variable. There can be as many lines of documentation about a variable as necessary. This is simply stored with the rest of the definition of the variable and is listed as required.

Value Labels

Labels can be defined for specific values of variables. For instance, for a variable Gender, the value 1 may have the label 'Male' and the value 2 may have the label 'Female'. These labels may be used in reports, etc. but generally, internally within programs, the numeric values are used.

Decimal Points

When reading numbers from external files or specifying numbers in definitions or programs, an actual decimal point "." can be present in the number. In definitions and programs, if the decimal point is omitted, it is assumed at the right of the number. For example:

```
COMPUTE A = 1.2
COMPUTE B = 100
```

Some existing computer files may not have explicit decimal points but may have an implied decimal point at a given position. For example, a format of F4.1 or a scale of (-1) means that the number in these four positions has one decimal place **if an explicit decimal point** is not quoted. If this field contained "0012", it is read as "1.2", however if it contained 1.234, then 1.234 is the value that would be stored.

The decimal positions describe an implicit input format and an implicit output format. They **do not** describe the maximum number of decimal positions that can be stored in a field. If a number is defined as D10.0, it can still be computed equal to "0.12345" and .12345 is stored.

On some schema definition commands (`VALUE LABELS, VAR RANGES, MISSING VALUES` etc.), numbers can be specified. If the number has a decimal component, specify an explicit decimal point and the actual value regardless of any input definition of implied decimals. For example:

```
MISSING VALUES RESULT (99.99)
```

Schema Commands

There are commands to create a new database or to connect an existing database. Passwords can be supplied with the `PASSWORD` command. Attaching a database and supplying passwords is normally done through menus or with the Execution Parameters.

There are numerous commands to define the specific features of a database. A set of commands that corresponds to the definition of the attached database can be written to a file using `WRITE SCHEMA` and edited. The schema definition can then be executed as any other SIR/XS procedure.

A report on the definition of the attached database can be produced with the `SCHEMA LIST` command.

The complete database can be deleted. Individual record definitions can be deleted provided that there is no data for that record.

The format of schema commands is the same as other SIR/XS commands, that is new commands must begin at the start of a line and continuation lines for a command must have blanks at the start of the line. Individual clauses within a command may be separated by slashes for readability.

All changes to a database, including schema changes, are recorded on the journal file provided that journaling is on. Initial schema definition before any data is added to a database is not journaled. Once there is data in the database, each schema modification run increments the update level of the database.

Overall database commands precede record definition commands. There may be many sets of record definition commands (one set for each record type) and, within a record definition, there is an order for the various types of commands.

Secondary index definitions may follow a record definition. A secondary index is defined with a single `CREATE DBINDEX` command. These can be done before or after any initial data loading. Defining an index builds the index automatically if there is existing data in the record.

Overall Commands

The overall database commands specify whether there is a case structure, size estimates, security, and any documentary text. The commands are:

```
[NO] CASE ID
```

Defines whether this database has a case structure or is a caseless database. This is the only required command to define a database.

`DATA FILES`

Specifies a data file that is non-standard. This may have a different name, be in a different directory or may be split across multiple files.

`DATABASE LABEL`

Specifies a descriptive label of up to 78 characters for the database.

`DOCUMENT`

Stores text about the database in the data dictionary.

`ENCRYPT [ON|OFF}`

Defines whether data in this database is encrypted or not.

`MAX INPUT COLS`

Specifies the longest batch input record length.

`MAX KEY SIZE`

Specifies the maximum key size.

`MAX REC COUNT`

Specifies maximum number of records of one type.

`MAX REC TYPES`

Specifies maximum number of record types.

`N OF CASES`

Specifies maximum number of cases for case structured databases.

`N OF RECORDS`

Specifies maximum number of records for caseless databases.

`READ SECURITY`

Sets security passwords for read access levels.

`RECS PER CASE`

Specifies the average number of records per case.

`RECTYPE COLS`

Specifies the columns that contain the record type for Batch Data Input.

`SYSTEM SECURITY`

Sets security passwords for specific DBA utilities.

`SYSTEM SECURITY LEVEL`

Sets the security level for specific DBA utilities.

`TEMP VARS`

Specifies temporary variables used during Batch Data Input.

`WRITE SECURITY`

Sets security passwords for write access levels.

Record Definition

The record definition commands specify the name and number of the record, the key fields and any documentary text together with the name, type, and size of each variable. A further set of information may be specified that relates to the Batch Data Input utilities. This specifies how data is loaded from serial files, including any computations and logical accept/reject clauses.

Sometimes the same data with the same coding scheme appears on multiple record types. For example standard drug codes or states in a country. Rather than repeating definitions in multiple record types, a *standard schema* can be defined that contains all of the descriptions and codes for the variable and it can then simply be included as a *standard var* in records as necessary.

In a case structured database, a set of variables can be held at the case level in the *CIR*. Specify variables that are in the CIR with a `RECORD SCHEMA 0 CIR` record definition.

Within a record definition, there is an order for commands. The example record definition shows some of the most commonly used record definition commands.

The following commands are used to specify records.

`ACCEPT REC IF`
Specifies acceptance tests for batch data input.

`CAT VARS`
Specifies variables are categorical.

`CHARACTER*n`
Defines new character variables and their length. n may be from 1 to 4094.

`COMPUTE`
Specifies computations in batch data input.

`CONTROL VARS`
Specifies that numeric variables are control variables not observation variables. By default, numeric variables defined with a set of `VALID VALUES` or `VALUE LABELS` are control variables. To specify other numeric variables as control variables, define a `VAR RANGE` for them.

`DATA LIST`
Defines the complete set of variables in the record with any appropriate external formats and positions for batch data input.

`DATE VARS`
Specifies variables are date variables and defines the external date format.

`DOCUMENT`
Stores text about the record in the data dictionary.

`END SCHEMA`
Specifies the end of this record definition.

`IF`
Computes values conditionally in batch data input.

`INPUT FORMAT`
Specifies the external format of variables defined by the `VARIABLE LIST`.

`INTEGER*n`
Defines new integer variables with an internal length of 1, 2 or 4.

`KEY FIELDS`
Specifies the variables that are keys for a record.

`MAX REC COUNT`
Specifies the number of records of this type that can be held.

`MISSING VALUES`
Specifies missing values for variables.

`OBSERVATION VARS`
Specifies that numeric variables which have valid values or value labels are observation variables instead of control variables.

`REAL*n`

Defines new floating point variables with an internal length of 4 or 8.

`REC SECURITY`

Sets default security levels for variables in this record.

`RECODE`

Specifies recodes performed by batch data input.

`RECORD SCHEMA`

Begins the record definition and names the record. It can include a record label.

`REJECT REC IF`

Specifies acceptance criteria for batch data input.

`SCALED VARS`

Specifies integer variables are scaled and defines the scaling factor.

`TIME VARS`

Specifies variables are time variables and defines their external format.

`VALID VALUES`

Defines valid values for variables.

`VALUE LABELS`

Defines value labels for variables.

`VARIABLE LIST`

Defines variables. Used with `INPUT FORMAT` as alternative to `DATA LIST`.

`VAR DOC`

Defines documentation for variables.

`VAR LABEL`

Defines a label for variables.

`VAR RANGES`

Defines ranges of valid values for variables.

`VAR SECURITY`

Defines read and write security levels for individual variables.

Modifying Database Definitions

The database definition can be modified through the menus. If using commands to modify the schema, you use the normal `RECORD SCHEMA` command to make specific changes. If a definition exists for a record and you do not submit a new `DATA LIST` command, then you are modifying the schema. If you do submit a new `DATA LIST` then you must re-submit the entire schema.

The `ADD VARS`, `MODIFY VARS` and `DELETE VARS` are equivalent to a `DATA LIST` when modifying a schema and have the same syntax.

If the `STANDARD SCHEMA` is modified, then all record types that have any `STANDARD VARS` are updated to reflect the changes. If a standard variable definition has been deleted, then standard variables that referenced that deleted variable are no longer standard variables.

There are a number of commands that are only applicable when modifying an existing record definition. Except as documented below, commands completely replace any existing definition.

`ADD VARS`

Adds new variables to the variable list.

`CLEAR BOOLEANS`

Clears all `ACCEPT/REJECT` conditions.

`CLEAR COMPUTES varname1,... | ALL`

Clears all computes for the specified variable(s).

`CLEAR RECODES varname1,... | ALL`

Clears all recodes of the specified variable(s).

`CLEAR VALUE LABELS varname1,... | ALL`

Clears all value labels for the specified variables.

`CLEAR VAR DOC varname1,... | ALL`

Clears all the lines of documentation for the specified variables.

`CLEAR VAR LABEL varname1,... | ALL`

Clears the label for the specified variables.

`COMPUTE`

Defines new compute definitions. These are added to the old definitions. Use the `CLEAR COMPUTES` command to delete old `COMPUTE` definitions.

`DELETE VARS`

Deletes variables from the variable list.

`EDIT LABELS`

Edits the value label list for the specified variables. This command has the same format as the `VALUE LABELS` command. Any new values are added to the list, any existing values that are referenced are updated. Existing value labels that are not referenced are not altered.

`MODIFY VARS`

Modifies the type, external format or batch data input position of existing variables.

```
RENAME VARS existing_variable_list {AS new_variable_list | PREFIX
'text' | SUFFIX 'text'}
```

Renames one or more variables while keeping all existing definitions of the renamed variable(s). The variables to be renamed are specified as a list and this can use the `ALL` or `TO` keywords. The new names can be specified individually as a list (which can also use the `TO` keyword), in which case there must be the same number of variable names in both lists. Alternatively the new names can be constructed by appending a prefix or suffix. If a prefix or suffix is specified, enclose the text in quotes. Note that this text is used exactly as specified so ensure that the correct case is used. Appending a prefix or suffix can result in non-standard names. The resulting names must fit within the 32 character limit on names.

Examples:

```
RECORD SCHEMA 1
EDIT LABELS JOBCODE  (21) Salesperson
                     (22) Senior Salesperson
                     (23) Sales Manager/
VAR RANGES JOBCODE (1 23)
```

The following example adds two variables to the Employee record type.

```
RECORD SCHEMA 1, EMPLOYEE
ADD VARS    ETYPE 70  (I)/
            PHONE 71 - 80 (A)
VAR RANGES  ETYPE (1 3)
VAR LABEL   ETYPE 'Employee Type'/
            PHONE 'Home Phone Number'/
END SCHEMA
```

The following example modifies the variable label for Position and the variable label for value 5 of Rating in the Review record type.

```
RECORD SCHEMA 3, REVIEW
VAR LABEL    POSITION   'Job Code'
EDIT LABELS  RATING (5) 'Excellent'
END SCHEMA
```

Format of Commands

Start new schema commands in column 1. Continue commands by leaving column 1 blank. Comments and general listing control statements can appear between any commands but not between the clauses of a single command.

When the same command applies to several variables, you can specify multiple variables and definitions on a single command. You can optionally delimit the specifications for each variable with a slash for readability. For example,

```
VALID VALUES CODE1 (1 2 3)
             CODE2 (1 2)
VALUE LABELS CODE1 (1) 'Tested'
                   (2) 'Preliminary'
                   (3) 'Passed'
             CODE2 (1) 'Domestic'
                   (2) 'Overseas'
```

TO Lists

When defining variables, you can define a set of variables by using a pair of identical variable names with a numeric suffix in ascending sequence separated by the keyword word `TO`. For example, to define ten variables named `VAR01, VAR02, ... VAR10`:

```
VARIABLE LIST VAR01 TO VAR10
```

Once variables have been defined, they can be referenced as a list in other commands by using a pair of variable names separated by the word `TO`, regardless of the format of the variable names. The sequence of the variables included in the list is determined by the sequence in which the variables were defined. The `TO` list is inclusive and backwards references are not allowed. For example, suppose the following variables were defined on a variable list:

```
VARIABLE LIST ID EMPNO NAME STATUS1 TO STATUS3 GENDER
```

A reference on another command (such as `MISSING VALUES`) might be `ID TO GENDER` to include all the variables, or `NAME TO STATUS3` to include `NAME` and the three `STATUS` fields. For example:

`MISSING VALUES EMPNO TO GENDER (BLANK)` The keyword `ALL` can be used to reference all of the variables in a record type.

Order of Commands

Record definition commands follow database definition commands. The record definition commands of each record occur together. You do not have to define records in any particular order however define any standard schema as part of the database definition, immediately followed by `RECORD SCHEMA 0 CIR` for the common vars on a case structured database.

The `KEY FIELDS` command and the `DATA LIST` or `INPUT FORMAT / VARIABLE LIST` can only occur once per record. Other commands may occur multiple times.

Within each record, there is a general order that must be followed. The commands to do with the overall record structure come first, then the definition of the variables and then additional specifications referring back to the defined variables. It is normal for the definition of all variables to precede the optional specifications for all variables, but this is not required. It is required that the definition of a particular variable precedes the specification for that variable.

Group	Commands
1	RECORD SCHEMA * DOCUMENT KEYFIELDS/SORT IDS MAX REC COUNT REC SECURITY
2	DATA LIST or VARIABLE LIST/ INPUT FORMAT* INTEGER VARS REAL VARS CHARACTER VARS
3	CAT VARS CONTROL VARS DATE VARS MISSING VALUES OBSERVATION VARS TIME VARS SCALED VARS STANDARD VARS VALID VALUES VALUE LABELS VAR DOC VAR LABEL

	VAR RANGES VAR SECURITY
4	ACCEPT REC IF COMPUTE IF RECODE REJECT REC IF
5	END SCHEMA

The * commands (`RECORD SCHEMA` and either `DATA LIST` or `VARIABLE LIST/INPUT FORMAT`) are mandatory and must be supplied. All other commands are optional.

- Group 1 commands are concerned with the overall record.
- Group 2 are data definition commands and create new variable names in the database.
- Group 3 further specify the variables created in group 2.
- Group 4 is concerned only with batch data input processing. Note that new variables can be created with `COMPUTE, IF` and `RECODE` commands or these commands can use previously defined variables. If the batch data input utilities are not relevant, this group of commands can be ignored.
- Group 5 is simply the end of the command set.

Example Record Specification

The typical definition of a record with no batch data input processing consists of :

RECORD SCHEMA
KEY FIELDS
DATA LIST or VARIABLE LIST/INPUT FORMAT
DATE VARS & TIME VARS
MISSING VALUES
VALID VALUES
VALUE LABELS
VAR LABEL
END SCHEMA.

For example:

```
RECORD SCHEMA 1,PATIENT
KEY FIELDS     ID
VARIABLE LIST  ID,LNAME,FNAME,DOB,SEX,STATUS
INPUT FORMAT   (I4,A40,A40,DATE'MM/DD/YYYY',I1,I1)
MISSING VALUES  LNAME to STATUS (BLANK)
VALID VALUES    SEX    (1,2)
                STATUS (1,2,3)
VALUE LABELS    SEX       (1) 'Male'
                          (2) 'Female'
                STATUS    (1) 'Inpatient'
                          (2) 'Outpatient'
                          (3) 'No Longer Attending'
VAR LABEL       ID        'Patient Id'
                LNAME     'Last Name'
                FNAME     'First Name'
                DOB       'Date of Birth'
                STATUS    'Current Status'
END SCHEMA
```

In the above example the following commands were used:

`RECORD SCHEMA`
Specifies the record number and name, and begins the specification of a record type.
`KEY FIELDS`
Specifies the key field variables for the record type in order from major to minor key. This immediately follows the `RECORD SCHEMA` command.
`VARIABLE LIST`
`INPUT FORMAT`
The `VARIABLE LIST` names the variables; the `INPUT FORMAT` defines the data type, size and format. The variable names are simply listed in the order they are to

appear in the database, separated by blanks or commas. Variables are held in the order they are defined. The format specification is enclosed in parentheses and commas are used as separators. "I" specifies integers, "A" specifies alphanumeric. Date formats consist of the word `DATE` followed by a date format. See date formats for a complete description. Time formats consist of the word `TIME` followed by a time format. See time formats for a complete description.

`MISSING VALUES`

Up to three missing values may be specified for any variable. The keyword `BLANK` specifies that blank input is treated as missing. When the same value is to be assigned to a set of variables the variable list format can be used. This consists of the two variable names that define the start and end of the list and the word "to".

`VALID VALUES`

Valid values or ranges of valid values may be specified for variables. Values that do not match these and are not a valid missing value can never appear in the data. Attempts to store an invalid value result in the system missing value (undefined).

`VAR LABEL`

A variable label may be up to 78 characters and can be used instead of the name of the variable on screens, column headings on reports, etc.

`VALUE LABELS`

Value labels associate a label with a given value. The label can be displayed in place of the value. Value labels are up to 78 characters long. The specification for each variable may be separated by a slash for readability.

`END SCHEMA`

Ends a record schema definition set of commands.

CREATE DATABASE

```
CREATE DATABASE database_name
        [JOURNAL  = {ON | OFF}]
        [PASSWORD = database_password]
        [PREFIX   = database_directory]
```

Creates a new database. (`NEW FILE` is a synonym.) A database name must be a valid SIR/XS a name. To connect to an existing database, use the `CONNECT DATABASE` command.

`JOURNAL`
Controls whether journaling is performed for the new database. The default is `ON`. Journaling can be turned on or off with the `JOURNAL` command

`PASSWORD`
Defines the database password for the new database. The password must be a valid SIR/XS name. Specifying this keyword lists the password in the output listing. The database password can be supplied with the `PASSWORD` command that does not list the password in the output listing. If a password is not defined for a database, the password is set to blank. If a database has no password, future connections to the database need not specify a password. The `UNLOAD FILE` utility can change the database name and password.

`PREFIX`
Specifies the database directory. If not specified, the database is created in the current directory.

CONNECT DATABASE

```
CONNECT DATABASE database_name
        [JOURNAL  = {ON | OFF}]
        [PASSWORD = db_password]
        [PREFIX   = database_directory]
        [SECURITY = {read_pw | *} [{write_pw | *}]]
        [CREATE]
```

Connects the specified existing database. (`OLD FILE` is a synonym.) A database must be connected before it can be used. The last connected database is the *default database*. Al processes and utilities run on the default database.

A pre-compiled VisualPQL program can connect a database when it runs, but, if you need to compile a VisualPQL program that references a database, the database must be connected first.

`JOURNAL`

Turns journaling on or off. Journaling is a database characteristic and remains as it was last set and need not be re-specified.

`PASSWORD`

Specifies the password that is required to access to the database. This is not required if the database has no password. This option shows the password in the output listing.

The database password can be supplied with the `PASSWORD` command that does not list the password in the output listing.

`PREFIX`

Specifies the database directory. This is not required if the database is in the current directory.

`SECURITY`

Specifies the read and write security passwords.

It is only necessary to specify the write security password when updating the database. To specify a write password when the read password is null, specify the read password as an asterisk (*).

If the read password matches the highest (30) level of security, then the user has Data Base Administrator (DBA) level access to the database and can run all utilities. If the database does not have any read passwords assigned, then any connected user has DBA access. See `READ SECURITY`

`CREATE .`

Creates a new database when used with the `CONNECT DATABASE` command. `CREATE` makes `CONNECT DATABASE` act identically to the `CREATE DATABASE` command.

DISCONNECT DATABASE

```
DISCONNECT DATABASE database_name
```

Disconnects a database. If this is the default database, the procedure file is set to `SYSPROC`.

SET DATABASE

```
SET DATABASE database_name
```

Sets a previously connected database as the default.

SHOW DATABASE

```
SHOW DATABASE
```

Writes a viewable list of connected databases.

LIST DATABASE

```
LIST DATABASE
```

Sends a list of connected databases to the OutputHandler callback routine in SirAPI when running in that mode.

JOURNAL ON|OFF

```
JOURNAL ON|OFF
```

Turns journaling on or off. Journaling is a database characteristic and it is recommended that journaling is left on under normal circumstances.

PASSWORD

```
PASSWORD database_password
```

Supplies the database password.

If this command **immediately** follows either the `CREATE DATABASE` (`NEW FILE...`) or `CONNECT DATABASE` (`OLD FILE...`) command. This means that the if the PASSWORD is to be specified then it MUST be on the next physical command line.

When used with `CREATE DATABASE`, this command defines the new database password.

When used with `CONNECT DATABASE`, this command supplies the password needed to connect the database. If the database has no password, this command need not be specified. If an incorrect password is specified, access to the database is denied.

If the PASSWORD command is used at any other time when a database is connected then it will change the database password.

This command does not print the password in the output listing.

SECURITY

```
SECURITY read_pw , write_pw
```

`SECURITY` supplies the read and write passwords. If an incorrect password is specified, level 0 (zero) security is assigned.

This command does not print the passwords in the output listing.

PURGE SIR FILE

Deletes the database. Use this utility to delete the current database before restoring it for recovery or restructuring. The database files are completely deleted from the disk.

```
PURGE SIR FILE
    [JOURNAL = KEEP | PURGE]
    [PROC    = KEEP | PURGE]
```

`JOURNAL`

`KEEP` specifies that the journal file is not deleted and is the default. `PURGE` specifies that the journal file is deleted when the database is deleted.

```
PURGE SIR FILE JOURNAL = PURGE
```

`PROC`

`KEEP` specifies that the procedures are not deleted. When a new database is created or a database is reloaded, the old procedure file can be used as part of that database. This keeps the procedures from a corrupt database when a restore cannot be accomplished. `PURGE` specifies that the Procedure File is deleted and is the default.

```
PURGE SIR FILE PROC = KEEP
```

DELETE SCHEMA

Deletes the schema of a record that has no data. Select the appropriate record.

```
DELETE SCHEMA recname | recnum
```

Deletes the record schema name or number from the database definition. `DELETE SCHEMA` only operates if there are no records for this record type. This is a DBA security level command.

When defining and redefining a record type, it is sometimes simpler to `DELETE SCHEMA` and redefine it through a complete new `RECORD SCHEMA` than to modify it over and over again.

When defining and testing a new database and wish to delete all of the test data for record type n prior to delete schema the following simple VisualPQL program does this (omit the case commands if it is a caseless database):

```
RETRIEVAL UPDATE
PROCESS CASE
PROCESS REC n
DELETE REC
END PROCESS REC
END PROCESS CASE
END RETRIEVAL
```

DELETE STANDARD SCHEMA

Deletes the entire standard schema. This modifies any records referencing the standard schema so that they no longer reference the standard. It is strongly advised that the database is rebuilt using export/import or unload/reload as soon as possible if this is done and multiple record definitions are affected.

CASE ID

```
 CASE ID varname [(A)|(D)] |
[NO] CASE ID
```

`CASE ID varname` establishes the database with a case structure and specifies the name of the variable used on every record as the case identifier.

`NO CASE ID` establishes the database without a case structure.

Either `CASE ID` or `NO CASE ID` is required to set up the database and this command must be used prior to any other definition. Once the case specification and case id have been defined, these cannot be modified.

The case variable may be any data type. Avoid `REAL` for keys due to the difficulty of specifying exact numbers in floating point.

`D` specifies descending sort order for cases. If order is not specified, ascending is assumed. When all cases are processed sequentially, they are retrieved in this sequence.

COMMON SECURITY

```
COMMON SECURITY rlevel, wlevel
```

Specifies the default minimum security levels for all common variables.

Rlevel (read level) and wlevel (write level) are integers between 0 (zero) the lowest, and 30, the highest. If no security levels are defined, level 0, the lowest, is assigned. Further security restrictions for individual common variables can be specified at the record level using the `VAR SECURITY` command. See `READ SECURITY` for an explanation of security levels.

Common security levels can be changed. Note, this affects only the security levels for new common variables that are defined or redefined. It does not affect the security levels of currently defined common variables.

COMMON VARS

```
RECORD SCHEMA 0 CIR
```

Specifies variables in the Common Information Record or CIR. A CIR exists for every case in a case structured database and holds counts plus the case identifier. It can hold *Common* variables that are typically those that are used repeatedly in retrieving data from the database. These variables can be referenced at any time regardless of the record type being processed. CIR entries can be updated directly when processing a case, or can take the value of the last entry in a given record type.

Note that common vars, except for the case id on a case structured database, *cannot* be used as key variables in a secondary index in a record as they are not stored as part of a record.

Specify the format here and then, when this variable is referenced on a subsequent record definition, there is no need to respecify formats except input-output columns and any batch data specifications (e.g. `RECODE`) for that record.

The specification of the CIR is identical to any other record except that the batch data input specification clauses (`ACCEPT REC`,`REJECT REC`,`COMPUTE`,`IF` and `RECODE`) are meaningless. Follow the `RECORD SCHEMA 0 CIR` with a `DATA LIST` to completely respecify the common vars or use `ADD VARS` or `DELETE VARS` to update the common vars.

Use any of the normal record variable definition commands such as `VALUE LABELS` as required.

The `RECORD SCHEMA 0 CIR` set of commands follows any `STANDARD SCHEMA` set of commands and precedes normal record definitions.

(Note: The older format of `COMMON VARS` is still supported for compatibility with earlier versions of SIR.)

DATA FILES

```
DATA FILES  'filename'
            [FROM (key,...) 'filename']
            [FROM (key,...) 'filename']
            .....
```

Specifies that the data file for this database is not a standard data file. It may have a different name, be in a different directory or may be split across multiple data files.

Specify the command at the end of the schema definition - it cannot be processed before the type of the Case variable has been specified for a case structured database. If a record type is specified as a `FROM` key, then that record type must have been defined. When SIR/XS writes a schema, this command follows any secondary indexes.

If the command does not have any `FROM` clause, it specifies the name and location of the data file. This can be in a different directory from the other database files and named something other than the database name with a `.sr3` extension.

The first specification names the original data file that holds all records up to the value specified on the first `FROM` key. The last specification names the final data file that holds all records from to the value specified on the last `FROM` key.

On a case structured database, the first key specified is the case id. If any further specification is required, the next key specified is a record number. On a caseless database, the first key specified is a record number. Subsequent keys can be specified up to the maximum number of keys on the record type.

The filenames must either be fully qualified filenames or simple filenames without any directory specification. If the files are not fully qualified then the data file is placed in the same directory as the other database files.

For example:

```
DATA FILES  'company.s31'
            FROM (500)  'company.s32'
            FROM (1000) 'company.s33'
```

A `DATA FILES` command with no other specifications removes any previous data file definition and sets the database to have a standard data file.

DATABASE LABEL

```
DATABASE LABEL 'text'
```

Specifies a label for the database. This text can be up to 78 characters and is enclosed in quotes. The label can be retrieved in VisualPQL using the `RECDOC(0,0)` function.

DOCUMENT

```
DOCUMENT text
```

Specifies that the text following the command is commentary. This text is stored in the dictionary describing the overall database.

The text cannot be partially modified. To alter the text, run the `DOCUMENT` command with new text. The new document text completely replaces the old.

ENCRYPT

```
ENCRYPT [ON |OFF]
```

`ENCRYPT` turns on data encryption for this database. This means that all data records in the database are encrypted on disk and are thus protected against scrutiny from software other than SIR/XS. The encryption method used is a version of the publicly available Blowfish algorithm using a 256 bit key.

All data records are encrypted, however keys in index blocks are held in unencrypted format. Do not use names or other recognisable strings as keys if this data is sensitive and requires protection. Unloads and journals for encrypted databases are themselves encrypted. Text files are all unencrypted. Schemas and procedures are unencrypted.

`ENCRYPT OFF` turns encryption off for a database. Encryption can be turned on and off without ill effect. Records are written according to the current setting; records are read and recognized as to whether they require decryption.

Passwords and security levels are encrypted on all databases. There are encryption/decryption functions in VisualPQL if users need to encrypt data for themselves but these use a user specified key - the SIR/XS system key is used for database encryption.

MAX INPUT COLS

```
MAX INPUT COLS n
```

Specifies the length of the largest input line for any record type in the database. N is rounded up to a number evenly divisible by eight. This command is necessary when there is any record type with a batch input format longer than 80. The `MAX INPUT COLS` can be increased at any time, but cannot be decreased once any record types have been defined.

MAX KEY SIZE

```
MAX KEY SIZE n
```

Specifies the maximum key size required for any record type in the database. The default is the size of the largest key currently defined for any record type in a database and is calculated automatically.

Only specify an explicit key size if you expect to define a new record type with a larger key than the current largest key after loading data into the database. If `MAX KEY SIZE` is set to the largest expected key, it avoids having to `UNLOAD` and `RELOAD` the database.

The keys for a record are: the Case Id, the record number and the key fields. The current `MAX KEY SIZE` can be obtained from the database statistics.

The absolute maximum key size possible, whether defined through this command or calculated from the keys specified, is 320 characters.

MAX REC COUNT

```
MAX REC COUNT n
```

For case structured databases, this specifies the default `MAX REC COUNT` for individual record definitions. The overall `MAX REC COUNT` sets the default maximum of a record type for any one case in the database. The number specified for an individual record type may be larger than specified here. The default value is 100.

For caseless databases, use the command `N OF RECORDS` to specify the maximum number of records that the database can hold.

MAX REC TYPES

```
MAX REC TYPES n
```

Specifies the maximum number of different record types that can be defined in the database. The default value is 30.

No record type number can exceed the value specified on the `MAX REC TYPES` command. For example, if `MAX REC TYPES` is 10, a record type 11 is not allowed, even if there are fewer than ten record types defined.

This number affects the size of the CIR. Space is reserved in the CIR for counts for as many record types as defined in `MAX REC TYPES`. The case level `MAX REC COUNT` determines how much space is held for the count of any record type as yet unspecified. For example, if there is a `MAX REC TYPES` of 30 and a `MAX REC COUNT` of 100, 30 bytes are reserved for record counts in each CIR. With a `MAX REC TYPES` of 100 and a `MAX REC COUNT` of 1,000,000, 400 bytes are reserved.

Changing this number requires a database `UNLOAD` / `RELOAD` once data has been loaded.

N OF CASES

```
N OF CASES n
```

Specifies the maximum number of cases, n, that can be entered in the database. The maximum number of cases is an integrity constraint that limits the number of cases that can be held in the database

The `N OF CASES` is an absolute value; it cannot be increased without doing an `UNLOAD` / `RELOAD` and so specify the value carefully to allow for the maximum number of cases ever wanted in the database.

`N OF CASES` is multiplied by the `RECS PER CASE` to establish the total number of records the database can handle. This limit cannot be exceeded and can only be changed with an `UNLOAD` / `RELOAD`.

There is no overhead with specifying a large value, the only constraint is that total number of records is a number that can be stored in one integer. This number is 2,147,483,648.

The default `N OF CASES` is 1000.

Not valid for caseless databases.

N OF RECORDS

```
N OF RECORDS n
```

Specifies the maximum number of records, n, that can be entered in a caseless database.

The `N OF RECORDS` is an absolute value; it cannot be exceeded and so specify the value carefully to allow for the maximum number of records ever wanted in the database. This limit can only be changed with an `UNLOAD` / `RELOAD`.

There is no overhead with specifying a large value, the only constraint is that total number of records is a number that can be stored in one integer. This number is 2,147,483,648.

The default `N OF RECORDS` is 1,023,000.

Not valid for case structured databases.

READ SECURITY

```
  READ SECURITY (leveln) password ....
```

Establishes the read security levels and associated passwords. There are 31 levels of security, from 0 (zero), the lowest, to 30, the highest. Repeat the complete specification for each password. Database passwords are a SIR/XS name and must conform to the name format. That is, names are no longer than 8 characters, begin with an alphabetic character and can contain alphanumeric characters and the four characters # $ @ _. For example

```
READ SECURITY (1) CLERK (2) SUPER (3) MANAGER
```

When a user logs in to the database, they specify a Read Security Password. If this matches a password in this list, then they are assigned that security level. If they do not login with a valid read security password, they are assigned level zero.

If security passwords are not defined, anyone who logs on to the database is assigned level 30 (database administrator) read permission.

One level is associated with one password. That is, there may be a password for level 1, a password for level 2, and so on. There cannot be two passwords for the same level. Read access at a particular level grants read access for all lesser levels.

A security level and associated password must be defined before that security level can be specified on a record type or variable.

RECS PER CASE

```
RECS PER CASE n
```

Specifies an average number of records per case. The default is 1023.

This is used to calculate the total number of records in the database. The product of `N OF CASES` multiplied by `RECS PER CASE` forms an upper bound on the total number of records (not including CIRs) that can be stored in the database.

The default `N OF CASES` is 1000 that means that 1,023,000 is the default total number of records for a database.

This can be updated once records have been entered without an `UNLOAD` / `RELOAD`.

This command has no meaning for caseless databases.

RECTYPE COLS

```
RECTYPE COLS n[,m]
```

When using Batch Data Input utilities, the input file can contain records of different types and an input record type is identified by its record type number. The record type number is an integer and must appear in the same position on all input records regardless of record type.

`RECTYPE COLS` specifies the columns that contain the record type number. 'n' specifies the start column 'm' specifies the last column. If the record type is in one column, i.e. `MAX REC TYPES` is less than 10, just specify the start column.

The columns specified must be within the range specified on the `MAX INPUT COLS` command. The record type number must be on the first line of any multi-line input records. (If records on input files are all of one type, the `RECTYPE=` clause can be used for the batch data input run and the record type number omitted.)

Make the number of columns large enough to hold the value of the maximum record number. For example, if up to 99 record types are allowed, specify two columns.

The default is columns 79 and 80.

SYSTEM SECURITY

```
  SYSTEM SECURITY readpw, writepw
```

Specifies the passwords associated with the `SYSTEM SECURITY LEVEL`, if specified. Log on to the database with these passwords to access the system utilities. Currently, only `UNLOAD FILE` is restricted by the `SYSTEM SECURITY LEVEL` and therefore this command is not usually specified.

Logging on with a password that is associated with the system security level gives access to all functions.

SYSTEM SECURITY LEVEL

```
SYSTEM SECURITY LEVEL n
```

N specifies the security level at which a user can perform a set of DBA-only commands. Log on to the database with the write password associated with this level to access the system utilities. The default system security level is 30.

Currently, only `UNLOAD FILE` is restricted by the `SYSTEM SECURITY LEVEL` and therefore this command is not usually specified.

TEMP VARS

```
TEMP VARS varlist
```

Names temporary variables for use in computations during entry of data with the Batch Data Input utilities. Computations include `COMPUTE`, `RECODE`, `IF`, `ACCEPT REC` and `REJECT REC` commands.

Temporary variables are not stored in the database.

WRITE SECURITY

```
 WRITE SECURITY (leveln) password ....
```

Establishes the write security levels and associated passwords. There are 31 levels of security, from 0 (zero), the lowest, to 30, the highest. Repeat the complete specification for each password. Database passwords are a SIR/XS name and must conform to the name format. That is: names are no longer than 8 characters, begin with an alphabetic character and can contain alphanumeric characters and the four characters # $ @ _. For example:

```
WRITE SECURITY (1) CLERK (2) SUPER (3) MANAGER
```

When a user logs in to the database, they specify a Write Security Password. If this matches a password in this list, then they are assigned that security level. If they do not login with a valid write security password, they are assigned level zero.

If security passwords are not defined, anyone who logs on to the database is assigned level 30 (database administrator) write permission.

One level is associated with one password. That is, there may be a password for level 1, a password for level 2, and so on. There cannot be two passwords for the same level. Write access at a particular level grants write access for all lesser levels.

ACCEPT REC IF

```
 ACCEPT REC IF (logical expression)
```

Only applies to batch data input

Specifies the criteria for accepting records using the Batch Data Input utilities. When the logical expression is `TRUE`, the record is entered into the database. Multiple `ACCEPT REC IF` commands can be defined to specify multiple acceptance criteria. If a record passes any one test, it is accepted. If `ACCEPT REC IF` is specified, all records that do not pass a test are rejected.

```
ACCEPT REC IF (AGE GE 16 AND LE 65)
```

Note: `ACCEPT RECORD IF` cannot be specified in a CIR definition. If an `ACCEPT RECORD IF` refers to a common variable then it must appear in the record schema where that common variable is referenced rather than at the CIR level.

CAT VARS

```
 CAT VARS varname ('value' .... ) varname ('value' .... ) ....
```

Specifies string variables that are held as categorical integers and defines the set of string values that can be input for the variable.

The values in the value list are each enclosed in single quote marks (') and the list for a variable is enclosed in parentheses. Specifications for multiple variables may be separated with a slash (/) for readability.

Within the database, categorical variables are held as integers that are the position of the string in the value list. For example:

```
CAT VARS SEX   ('MALE' 'FEMALE' )
         STATE ('AL' 'AK' ....'WY')
```

Specifies that the variable `SEX` is categorical. On input 'MALE' is converted to a 1, 'FEMALE' to a 2.

The variable `STATE` definition illustrates a list of abbreviations of American states. On input 'AL' is converted to a 1, 'AK' to a 2, etc.

When entering data into a `CAT VARS`, the string value is input, not the code. Note the difference to value labels, where a code is input and a string is associated with the code.

CHARACTER

```
 CHARACTER*n varname ....
```

Specifies variables as character. n may be from 1 to 4094.

Example:

```
CHARACTER*40 NAME
```

COMPUTE

```
COMPUTE variable = expression
```

Only applies to batch data input

`COMPUTE` performs arithmetic or string transformations on common, record or temporary variables in Batch Data Input as each record is read. If the computed variable has not been defined, a new database variable is created at the end of the record. It is recommended that computed variables are defined with the appropriate command.

Note: `COMPUTE` cannot be specified in a CIR definition. If a `COMPUTE` refers to a common variable then it must appear in the record schema where that common variable is referenced rather than at the CIR level.

Case and Key variables cannot be computed.

CONTROL VARS

```
CONTROL VARS variable ....
```

Declares a list of variables that are *Control* variables for the `TABULATE` procedure. These variables must be numeric and must have either `Valid Values` or `Variable Ranges` defined. By default, variables that have Valid Values or Value Labels are Control Variables. All other numeric variables are Observation Variables, that is variables with continuous values.

DATA LIST

```
DATA LIST [(num-lines)]
  [line-no]   varname  from-column [- to-column] [(type)]
  [line-no]   varlist  from-column [- to-column] [(type)]....
```

Defines the variables and input format for a record. You can either use the `DATA LIST` or the `VARIABLE LIST` / `INPUT FORMAT` to define the record.

The definition consists of the name, batch data input column locations and data type for each variable. The sequence of the variables determines the order in `TO` lists and the sequence of the variables wherever they are referenced, regardless of the physical order on the batch data input record.

If the batch data input record requires more than one physical record or line, then the num-lines defines the number of lines which make up the complete record and the line-no defines which line each variable is on. When the input record is only one line, omit the number of lines and line number. The line-no can be omitted for any subsequent variables on the same physical input record.

The from-column determines the start position of the variable.

The to-column specifies the ending position for variables that are longer than one column.

Define the data type of each variable as follows:

`A`

String.

`I`

Integer.

`Fn`

Single precision floating point. On batch data input, n columns at the right of the input field comprise the decimal component of the number. An explicit decimal point on input overrides the format specification.

`Dn`

Double precision floating point. On batch data input, n columns at the right of the input field comprise the decimal component of the number. An explicit decimal point on input overrides the format specification.

`DATE`

Date variable in the given date format. The specification consists of the word `DATE` and the date format, all enclosed in parentheses. See date formats for a complete description.

`TIME`

Time variable in the given format. The specification consists of the word `TIME` and the time format, all enclosed in parentheses. See time formats for a complete description.

If the type is omitted, the default is floating point with zero decimal portion.

Example:

```
DATA LIST  (2)
        1  ID        1 - 4   (I)
           POSITION  6 - 7   (I)
           STARTDAT  8 - 15  (DATE,'MMIDDIYY')
           STARTSAL  17 - 20 (F2)
           DIVISION  21      (I)
        2  NAME      6 - 30  (A)
           GENDER    31      (I)
           MARSTAT   32      (I)
           SSN       33 - 43 (A)
```

If a *varlist* is specified, (that is either a list of variable names or a list in the form varname to varname), multiple variables, all of the same size and type, can be defined. The columns specified to contain these variables must be evenly divisible by the number of variables in the list.

DATE VARS

```
 DATE VARS varname .... ('date_format') varname .... ('date_format')
....
```

Specifies that previously defined character variables are date integers with a given date format.

Date formats may be specified directly on the `DATA LIST` or `INPUT FORMAT` commands. If using `DATE VARS`, the variable is specified as a character string on the `DATA LIST` or `INPUT FORMAT` commands.

Multiple variables in the same format can be defined with one format specification; additional specifications may be separated by slashes for readability.

See date formats for a complete description of date formats.

DOCUMENT

```
DOCUMENT text
```

Specifies that the text following the command is commentary. This text is stored within the database and can be printed using the utility `LIST SCHEMA`.

When `DOCUMENT` is placed within record definition commands, it becomes part of the definition for that record type.

`DOCUMENT` text cannot be partially modified. To update the text, run the command with new text. The new document text completely replaces the old.

END SCHEMA

```
END SCHEMA
```

Specifies the end of the commands for a record set. If it is not specified, the end of the commands, `START TASK` or `END TASK` or the start of a new `RECORD SCHEMA` terminates definition of the record type. Any other commands are treated as record definition commands.

IF

```
IF (logical-condition) varname = expression; ....
```

Only applies to batch data input

Assigns the result of an expression to a variable if the logical condition is true. Multiple variables can be assigned values on a single condition. Variables referenced must be within this record or must be common vars. If the computed variable has not been previously defined, it is added to the defined database variables. For example:

```
IF (JOBCODE = 1) REVDATE = TODAY(0) + 365
```

Note: `IF` cannot be specified in a CIR definition. If an `IF` refers to a common variable then it must appear in the record schema where that common variable is referenced rather than at the CIR level.

INPUT FORMAT

```
INPUT FORMAT (format specifications)
```

`INPUT FORMAT` is associated with, and immediately follows, the `VARIABLE LIST` command.

Specify a data type, size and format for each variable on the `VARIABLE LIST`.

An input file may have multiple lines of data for each database record. Lines in an input record may be any length up to the length specified on `MAX INPUT COLS`. Specify a slash (/) to indicate the start of the second and subsequent lines of data. The slash can be used to skip one or more lines of an input record.

Separate each format specification by a comma or a space. If the format specifications require more than one line, continue the specification on the next line leaving column one blank.

Specify a single format and a repetition factor for multiple variables with the same format or groups of variables with the same format. To repeat a format, specify the number of times to repeat it, followed by either a single format or a group of formats enclosed in parentheses. For example:

```
VARIABLE LIST VAR01 TO VAR50
INPUT FORMAT (10I2, 20(I1,I4))
```

The `VARIABLE LIST` with the TO format defines 50 integer variables named `VAR01, VAR02, ... VAR50`.

The first repeating format (10I2) defines the first 10 variables that results in `VAR01` to `VAR10` as two digit integers.
The repeating *group* of formats, 20(I1,I4), defines 20 sets of two alternate variables. This results in `VAR11, VAR13` and subsequent odd numbered variables as one digit integers (I1) and `VAR12, VAR14` and subsequent even numbered variables as four digit integers (I4).

Enclose the whole format specification in parentheses. The individual format specifications are as follows:

`Fw.d or Dw.d`

A floating point variable. "F" is single precision, "D" is double precision. On batch data input the variable occupies "w" positions on the input file with the rightmost "d" positions as the decimal component. A decimal component must be specified; zero is valid. If Batch Data Input is not used the decimal component has no effect. Specifying a physical decimal point on input overrides any specification. For example:

```
VARIABLE LIST TEMP SALARY
INPUT FORMAT (F5.3, D8.2 )
```

`Iw`

An integer variable occupying "w" positions on batch data input. For example:

```
VARIABLE LIST STATUS, AGE
INPUT FORMAT ( I1 ,  I2 )
```

`Aw`

A character (alphanumeric) variable occupying "w" positions. For example:

```
VARIABLE LIST NAME ADDRESS
INPUT FORMAT ( A25 , A40 )
```

`nX`

A positioning operator for Batch Data Input utilities. It skips "n" columns of an input data record. For example:

```
VARIABLE LIST NAME ADDRESS
INPUT FORMAT ( A25 , 4X , A40 )
```

This defines two alphanumeric variables. `NAME` occupies positions 1 through 25 of the input record. 4X skips the next 4 columns (after `NAME`). `ADDRESS` is 40 characters long beginning in position 30.

`Tn`

A positioning operator for Batch Data Input utilities. It tabs to a specific column "n". The next variable begins in column "n". The "T" specification can be used to move forward or backward over the current input line and can be used to reread a particular field. For example

```
VARIABLE LIST NAME ADDRESS PHONE AREACODE
INPUT FORMAT (T3,A20,T25,A40,T72,A12,T72,A3)
```

This defines four string variables using the T operator to locate the beginning of each variable. Note that the variables `PHONE` and `AREACODE` both begin in column 72.

`DATE 'date format'`

Describes an input character variable as a date according to the specified date format. See date formats for a complete description of date formats.

`TIME'time format'`

Describes an input character variable as a time according to the specified time format. See time formats for a complete description of time formats. For example:

```
VARIABLE LIST NAME BIRTHDAY BIRTHTIM
FORMAT (A25, DATE 'MM/DD/YYYY', TIME 'HH:MM')
```

This example defines three variables for a record type. `NAME` is in the first 25 positions; `BIRTHDAY` is in the next ten positions and is a date variable; `BIRTTIM` is a time variable. The first two characters are Hours (24 hour clock), the third character is a separator and the last two are minutes.

INTEGER

```
 INTEGER*n varname ....
```

Specifies variables as integer that can hold positive or negative numbers. n may be 1, 2 or 4 and refers to the internal storage size in bytes. 1 byte holds numbers up to 123; 2 bytes holds numbers up to 32,763; 4 bytes holds numbers up to 2,147,438,643.

If this is subsequently defined as a `SCALED VAR`, the internal integer must be able to store the significant digits needed for the number. For example if Scale (-2) is specified, the largest number that can be held in I*4 is 21,474,386.43

Example:

```
INTEGER*4 SALES VAR1 to VAR5
```

KEY FIELDS

```
KEY FIELD[S] varname [(A|D|I)] ....
```

Defines the keys for the record. Any record type that has more than one single physical record per case on a case structured database and every record type with more than one single physical record in a caseless database must have a key.

The `KEY FIELD` command must be before the `DATA LIST` or `VARIABLE LIST`.

The key fields must appear in the `DATA LIST` or `VARIABLE LIST`

Key field variables cannot be created or modified by `COMPUTE`, `IF` or `RECODE` commands.

The sort specification applies to each variable individually.

- `(A)` specifies ascending sort order - this is the default.
- `(D)` specifies descending sort order.
- `(I)` specifies this is an *Auto Increment* key. When records with an auto increment key are created, the value of the specified key is tested. If the creation process sets this key value to zero, then the system automatically finds the last occurrence of the record type in the case and takes the value from that record, increments this by 1 and uses this value as the key. If a record of this type does not exist, the value 1 is used as the key. Auto increment keys must be integer variables. If a key is an auto increment key, it must be the final key in a record type.

Example:

```
CASE ID         ID
RECORD SCHEMA   3 REVIEW
KEY  FIELDS     POSITION  REVDATE (D)
VARIABLE LIST   ID POSITION REVDATE          RATING  NEWSAL IDSUPER
INPUT FORMAT   (I4 I2       DATE('MMIDDIYY') I2      F8.2   I4)
```

MAX REC COUNT

```
MAX REC COUNT n
```

For a case structured database this command specifies the maximum occurrences of a record type for any one case in the database. If a value is not specified for a record type, the `MAX REC COUNT` from the database specification is used. The number specified for a record type may be larger or smaller than specified at the database level. The default value is 100.

Counts are kept for each record type in a case in the CIR for that case. They are stored as 1, 2, or 4 byte integers depending on the count specified. A count of less than 124 takes 1 byte, less than 32,763 takes 2, and larger numbers take 4. If `MAX REC COUNT` is modified after data for that record type has been loaded, and the new number takes the same size integer, restructuring is unnecessary. If a larger size integer is needed, the database must be restructured.

For caseless databases, `MAX REC COUNT` specifies the maximum number of records of this type that the database can hold. The default value is 1,023,000. This limit cannot be exceeded but can be changed without a database restructure. There is no overhead with specifying a large value, the only constraint is that total number of records is a number that can be stored in one integer. The total number of records allowed is 2,147,483,648.

MISSING VALUES

```
MISSING VALUES varname (value1 [value2 [,value3]] ) ....
               varlist (value1 [,value2 [,value3]] ) ....
```

Specifies up to three values that are missing values for the variable(s). Missing values are excluded from statistical procedures and functions. When the variable is input or modified, and one of the specified values is input, the appropriate missing value is set.

The value can be a numeric constant, a string constant, or the keyword `BLANK.` If `BLANK` is specified as a missing value for a numeric variable, then a blank field on input results in a missing value, otherwise a blank numeric field is translated to zero.

Any variable can be missing and has a system missing value. When a record is written to the database, a variable is assigned the system missing value when it cannot be assigned a legal value or a specified missing value. This happens when:

- no value has been assigned to a given variable;
- it is set equal to another variable containing missing values;
- an assigned value does not meet the schema specification (not a valid value, outside var ranges, too large to store in an integer).

Specify any missing values for string, categorical, date and time variables as strings. If the variable is longer than the missing value, then the variable is set to missing values if the leftmost characters match the specified missing value.

If a string is read into a date or time, that string is first checked to see if it is a missing value string for the target variable. If it is then a missing value is stored. Any string can be defined as missing - it need not be a valid date. If the missing value is a valid date string then assigning that string to the variable naturally results in a missing being stored. However assigning the numeric date value corresponding to the missing string stores a valid date.

Specify any missing values for scaled variables as the unscaled value with the decimal point specified where necessary.

Example:

```
MISSING VALUES ID POSITION DIVISION (BLANK)
```

```
STARTDAT ('01/01/01')
```

OBSERVATION VARS

```
OBSERVATION VARS varname ....
```

Specifies variables that the `TABULATE` procedure uses as observation variables. An observation variable is one that is aggregated rather than treated as a control. By default, variables that have `VALID VALUES` or `VALUE LABELS` are `Control Variables`. `OBSERVATION VARS` makes these observation variables.

REAL

```
REAL*4 varname ....
REAL*8 varname ....
```

Specifies variables as real. n may be 4 or 8.

Example:

```
REAL*8 SALARY
```

REC SECURITY

```
REC SECURITY rlevel , wlevel
```

Defines the default minimum security levels required for reading (rlevel), or writing (wlevel), any variable in the record. The read and write levels are integers between 0 (zero), the lowest and 30, the highest.

This command sets the minimum `VAR SECURITY` for each variable in the record type. Individual variables within a record type can be assigned higher security levels using the `VAR SECURITY` command.

The default is 0 level security.

Example:

```
REC SECURITY 10,30
```

The variables in this record type can be read by anyone logged in with a read security password that has level 10 security or higher. Write access is restricted to personnel logged in with a write security password that has level 30 security.

RECODE

```
RECODE variable specification (recode specification)
```

Only applies to batch data input

`RECODE` changes the values of a string or numeric variable into new values. A variable can be recoded into itself or the content of the original variable can be left unchanged and a value assigned to another variable.

The `RECODE` has two parts the *Variable* specification and the *Recode* specification.

Variable Specification

The variable specification takes four forms:

- variable (recode specification) - recodes a variable into itself.
- varlist (recode specification) - recodes a list of variables into themselves.

  Old values not specified in the recode specification are not affected when recoding variables into themselves.

- newvariable = oldvariable (recode specification) - recodes a variable into a new variable.
- newvarlist = oldvarlist (recode specification) - recodes a list of variables into new variables.

  Old numeric variables can be recoded into new string variables, or vice versa. When recoding into new variables, if the data types of an old variable and new variable are the same, unrecoded old values are stored unchanged in the new variable after data editing checks are performed. If the data types of the old and new variable are different, unrecoded old values are stored as missing values in the new variable.

Recode Specification

A recode specification follows the variable specification and consists of a number of clauses, one per new value to assign. Enclose each clause in parentheses. These can take a number of forms:

- (oldvalue = newvalue) - recodes a single value to a new value.
- (oldvaluelist = newvalue) - recodes several values to a single new value.
- (oldvalue1 `THRU` oldvalue2 = newvalue) - recodes a range of values to a single new value. The keyword `LO` can be used as oldvalue1. This selects the lowest

possible value as the start of the range. The keyword `HI` can be used as oldvalue2. This selects the highest possible value as the end of the range.

- (`MISSING =` newvalue) - Specifies that all missing values are recoded.
- (`MISSING(0 | 1 | 2 | 3) =` newvalue) - Specifies that the first, second or third missing value is recoded. Zero (0) recodes undefined values.
- (`UNDEFINED =` newvalue) - Specifies that undefined is recoded. This is the same as `MISSING (0)`.
- (`BLANK =` newvalue) - Specifies that `BLANK` values are recoded. This can only be specified for numeric variables that have missing values specified as blank.
- (`ELSE =` newvalue) - Specifies a value used to recode if the variable is not recoded in any other category.

Note the sequence of the variables in the variable specification and the sequence of values in the recode specification:

- The *variable* specification to recode into a new variable, has the new variable on the left of the equal (=) sign and the old variable on the right.
- The *recode* specification has the old value on the left of the equal sign (=) and the new value on the right. viz:
-

```
Newvariable = Oldvariable (Oldvalue = Newvalue)
```

Recode Examples:

```
RECODE A (1,3,5,7,9=1)(2,4,6,8=2)
```

This sets A to 1 if it is odd, 2 if it is even, and leaves zero and missing values as is.

```
RECODE B = A (1,3,5,7,9=1)(2,4,6,8=2)
```

This sets B to 1 if A is odd, 2 if A is even and zero if A is zero. If A is missing, B is set to the value of A (whether this is a missing value for B or not). A is unchanged.

```
RECODE B = A (1,3,5,7,9='O')(2,4,6,8='E')
```

This sets B (a string variable) to 'O' for odd values of A, 'E' for even values of A and to the value of A if A is missing or has any other values.

```
RECODE STATUS = AGE (LO THRU 18 = 1) (19 THRU 21 = 2)
                    (22 THRU 65 = 3) (66 THRU HI =4)
```

This sets `STATUS` depending on the `AGE` of the subject. `STATUS` is 1 for ages 18 or under, 2 for ages 19 thru 21, 3 for 22 to 65 and 4 for over 65.

```
RECODE STATUS1 to STATUS10 =  TEST1 to TEST10(1 THRU 49=1)
                                             (50 thru 99=2)
```

This sets up ten status fields depending on the result of 10 tests.

Note: `RECODE` cannot be specified in a CIR definition. If a `RECODE` refers to a common variable then it must appear in the record schema where that common variable is referenced rather than at the CIR level.

RECORD SCHEMA

```
RECORD SCHEMA rectype [,name] 'label'
       [LOCK]
       [NOOLD]
       [NONEW]
```

This is a DBA only command.

Begins the set of commands to define a database record. This defines the name and number of the record and is a required command to define a record type. Name is optional for existing record types; if specified and different to the existing name, the record name is changed. The label is optional. Specify up to 78 characters enclosed in quotes. e.g.

```
RECORD SCHEMA 3 OCCUP 'Position Details'
```

`LOCK`

Specifies that the record type can be locked if the record redefinition requires it. Schema modifications are not done if the `LOCK` keyword is omitted and a lock condition occurs. If the record type is locked, an `UNLOAD` / `RELOAD` is required. Lock conditions occur when:

- the list of key fields is changed
- any of the key variables are modified
- a record is defined that is larger than the current data block
- a set of keys is defined that is larger than the current key size

If the `LOCK` parameter is omitted and changes are specified that would cause a locked record type, a warning is issued and the schema changes do not take place. This means that a restructure is only done when expected.

It is good practice **not** to specify lock on the record schema statements except when the change is expected to lock the record and you are prepared to do a restructure.

`NOOLD`

Specifies that existing variables cannot be modified.

`NONEW`

Specifies that no new variables can be created.

`LOCK`

Specifies that the record type may be locked if the changes to the record definition would need a database restructure. Most changes to a record definition do not require a restructure.

A restructure may be required when modifying the key structure definition of a record type that already has data loaded into it. I

REJECT REC IF

```
REJECT REC IF (logical condition)
```

Only applies to batch data input

Defines criteria for rejection of records during Batch Data Input. When the condition is true, the record is rejected and not entered into the database. Multiple `REJECT REC IF` commands can be specified.

The alternative method of specifying consistency criteria is with `ACCEPT REC IF`.

The record must pass all specified tests before being added to the database.

Note: `REJECT RECORD IF` cannot be specified in a CIR definition. If a `REJECT RECORD IF` refers to a common variable then it must appear in the record schema where that common variable is referenced rather than at the CIR level.

SCALED VARS

```
SCALED VARS varname (power) ....
```

Specifies that the previously defined integer variables are scaled. `SCALED VARS` are stored in the database as integers. This saves space and can be more accurate for fixed format numbers since it avoids the inherent inaccuracies of floating point representation.

`POWER` is a positive or negative number representing the power of ten used to scale the values.

The full, unscaled number, including the decimal point where necessary, is specified whenever the number is input by the user.

The scaled number is expected on batch data input.

Examples:

```
DATA LIST    VAR1       1-3 (I)
             VAR2       4-11 (I)
SCALED VARS  VAR1 (6)
             VAR2 (-2)
```

This declares two variables as scaled variables. When used in a program:

```
COMPUTE VAR2 = 345.67   |(34567 is stored)
COMPUTE VAR1 = 1000000  |(1 is stored)
COMPUTE VAR3 = VAR1 * 3
WRITE          VAR3     |(3000000 is printed)
```

If a batch data input record has 123 in positions 1 to 3 and 00001234 in positions 4 to 11, then `VAR1` equates to 123,000,000 and `VAR2` to 12.34.

STANDARD SCHEMA

The schema command `STANDARD SCHEMA` is similar to a `RECORD SCHEMA` command in that it signifies the start of a set of variable definitions. The set is ended with an `END SCHEMA` command. Variables are defined using a `DATA LIST` command together with any of the normal variable definition commands such as `MISSING VALUES`, `VALUE LABELS` or `VAR RANGES`. e.g.

```
STANDARD SCHEMA
DATA LIST
                POSITION                       *             (I1)
                SALARY                         *             (I2)
                SALDATE                        *
(DATE'MMIDDIYY')
VAR RANGES      POSITION                       (1 18)
                SALARY                         (600 9000)
VAR SECURITY    SALARY                         (30,30)
MISSING VALUES  POSITION                       TO
                SALDATE                        (BLANK)
VALUE LABELS    POSITION                       (1)'Clerk'
                                               (2)'Secretary'
                                               .............
VAR LABEL       POSITION                       'Position'
                SALARY                         'Salary'
                SALDATE                        'Date Salary Set'
END SCHEMA
```

Once a variable has been defined in the standard schema it can be referenced in any normal record definition with the `STANDARD VARS` command. The benefit of this is that coding does not have to be repeated for the variable when it occurs in multiple records. Further, if the standard definition details are updated (such as value labels), the change is reflected in all records referencing the standard.
Note that the extended batch data input processing definitions of `ACCEPT REC`,`REJECT REC`,`COMPUTE`,`IF` and `RECODE` are not specific to a variable and thus cannot be specified as standard and copied in.

STANDARD VARS

`STANDARD VARS varname [AS stdvarname]` The `STANDARD VARS` command names a variable or list of variables that have previously been defined as part of this record (on the `DATA LIST`). This includes all the standard definitions for the variable as part of this record without the need to respecify these definitions. If these definitions are changed for the standard variable, all derived record definitions are updated.

Certain definitions can be supplied locally. The `VAR SECURITY`, `VAR LABEL` and `VAR DOC` can be specific to the variable in this record type and override any specified as standard. All other definitions such as `MISSING VALUES`, `VALUE LABELS` etc, are taken from the standard definition.

Optionally the record variable can have one name and can refer to a standard variable with the `AS` keyword. e.g.

```
RECORD SCHEMA 1 EMPLOYEE
DATA LIST
              ID                                      1 -     4 (I2)
              NAME                                    6 -    30 (A25)
              GENDER                                 31         (I1)
              MARSTAT                                32         (I1)
              SSN                                    33 -    43 (A11)
              BIRTHDAY                               44 -    51
(DATE'MMIDDIYY')
              EDUC                                   52         (I1)
              NDEPENDS                               53 -    54 (I1)
              CURRPOS                                55 -    56 (I1)

STANDARD VARS CURRPOS AS POSITION
```

If an existing variable is modified to be a standard variable, any local definitions are overwritten. Submitting local definitions for components of the schema that are derived from the standard is treated as an error. To change a variable from a standard definition to be a normally defined variable is not possible; delete the standard variable and add a new variable (with the same name if necessary).

TIME VARS

```
TIME VARS varname .... (time format) varname .... (time format) ....
```

Declares string variables as times. See time formats for a complete description of time formats.

Example:

```
TIME VARS   ELAPSED       ('HH:MM')
            MIN1 to MIN10 ('MM:SS')
```

VALID VALUES

```
VALID VALUES varname  (value, .... )
             varlist  (value, .... ) ....
```

Defines the set of valid values for a variable. Valid values can only be specified for numeric variables. (Use the `CAT VARS` command to create a list of valid values for string variables.)

The specified valid values are checked whenever a variable is input or modified.

If a value is assigned to the variable that is not in the `VALID VALUES` list or in the `MISSING VALUES` list, the value is not stored and the variable is set to undefined.

The varlist may be a specific list of variables or may be in the format `VARA to VARX` that specifies all the numeric variables between the named variables (listed in sequential order) in the record schema.

Example:

```
VALID VALUES JOBCODE          (1,3,4,5)
             TEST1 to TEST9   (1,2,3,4,5,9)
             TYPE KIND REASON (1,2,3,9)
```

VALUE LABELS

```
VALUE LABELS  varname1  (value1) 'label1'
                        (value2) 'label2' ....
              varlist   (value1) 'label1' ....  ....
```

Specifies labels for particular values of a variable.

Enclose each value for a variable in parentheses. The value may be numeric or character depending on the variable type. Enclose character strings in quotes. The keywords `UNDEFINED` and `BLANK` can be used as a value to assign labels to the system missing value (undefined) or blank missing values.

The label is up to 78 characters long. It is good practice to enclose the label in quotes as this is a character string. If the label contains slashes or brackets then the label must be enclosed in quotes.

The varlist may be a specific list of variables or may be in the format `VARA to VARX` that specifies all the numeric variables between the named variables (listed in sequential order) in the record schema.

Example:

```
VALUE LABELS  GENDER         (1) 'Male'
                             (2) 'Female'
              MARRIED        ('Y') 'Married'
                             ('N') 'Not Married'
              TEST1 to TEST9 (1) 'Passed'
                             (2) 'Failed'
```

VARIABLE LIST

```
 VARIABLE LIST varname1 varname2 ....
```

Names the variables on the input record for a given record type. Together with the `INPUT FORMAT`, this describes the input record. There is a one-to-one correspondence between the variables on the `VARIABLE LIST` and the format specifications on the `INPUT FORMAT`.

The sequence of the variables on the `VARIABLE LIST` determines the order in `TO` lists and the sequence of the variables wherever they are referenced, regardless of the physical order on the batch data input record.

Separate variable names by spaces or commas. To continue the list on multiple lines, leave column one blank on each subsequent line.

Example:

```
VARIABLE LIST ID JOBCODE REVDATE SALARY
```

VAR DOC

```
VAR DOC varname text
 text line 2
 text line .....
```

Specifies documentation for a single variable.

The documentation can extend over multiple lines, each up to 254 characters. There are no restrictions on characters and the format of lines is maintained. Each continuation line must leave the first column blank. Note that because documentation can contain any characters, the command must be specified separately for each variable - it cannot be continued for multiple variables.

Example:

```
VAR DOC  ID    This is the main identification for participants in the
study.
 The code is in two parts separated by slash '/' - the first part
identifies the institution, the second the individual.
VAR DOC  DOB   ...........................
 ........................................
VAR DOC  VAR1  ...........................
```

VAR LABEL

```
VAR LABEL
       varname1 'label'
       varname2 'label' ....
```

Specifies a label for variables. The label is a string up to 78 characters. It is good practice to enclose the string in quotes. This is used by SIR/XS instead of the variable name wherever it is appropriate, for example, on report headings, screen layouts, etc.

Example:

```
VAR LABEL   ID        'Identification Number'
            POSITION  'Position Level'
            STARTDAT  'Starting date'
            STARTSAL  'Starting salary'
            DIVISION  'Division'
```

In this example, "Starting date" is printed or displayed instead of "`STARTDAT`" and "Starting salary" instead of "`STARTSAL`".

VAR RANGES

```
 VAR RANGES varname (min,max) ....
```

Specifies the range of values for a variable. The ranges defined for a variable are checked whenever that variable is input or modified. Input values outside the specified range are set to undefined.

Example:

```
VAR RANGES  POSITION (1 18)
            STARTSAL (10000 90000)
            DIVISION (1 3)
            STARTDAT ('01/01/00', '01/01/06'
```

If both `VAR RANGES` and `VALID VALUES` are specified, both specifications apply. Only values consistent with both specifications are allowed into the database. Normally, specify only one of these.

VAR SECURITY

```
 VAR SECURITY varname (rlevel, wlevel) ....
```

Defines security level values for reading and writing individual variables.

Rlevel and wlevel are integers between 0 (zero) the lowest, to 30 the highest. The `VAR SECURITY` assigned to an individual variable must be higher than the security level assigned to the record through `REC SECURITY`. (Assigning a lower level is possible but has no effect.)

Example:

```
RECORD SCHEMA ....
REC SECURITY 0 5
   .
VAR SECURITY SALARY (6 10)
```

In this example, anyone can read the data in this record type, but only users logged in with passwords at level 5 or above for write security can write data. The variable `SALARY` is made more secure, since it requires that read access is at level 6 or above and write security at least at level 10.

CREATE DBINDEX

```
CREATE [UNIQUE] DBINDEX index_name
        ON [database.]recname
        (var_name [ASC|DESC] [UPPER] [, ...] )
```

`CREATE DBINDEX` creates the index structure. If the command is part of a schema definition, it must follow the complete definition for the record type referenced. There may be multiple indexes for a given record type.

If the record type already contains data, the index is automatically built from the values of any existing records. As records are added, deleted or modified through any of the SIR/XS modules including VisualPQL, batch data input, Forms and SQL, the index is automatically maintained. VisualPQL and PQLForms are the only modules that process database records using secondary indexes.

Indexes are maintained across cases in a case-structured database and, if a record is retrieved using a secondary index, the appropriate case is automatically found. Index variables can be any record variable plus the case id. Common vars cannot be used in an index.

Indexes are rebuilt whenever the database is recovered i.e. from an Import or a Reload. The backups contain only the index definition not the index data.

`UNIQUE`

Specifies the index can only contain unique values. If data already exists and a non-unique value is found, the index cannot be built. If an attempt is made to add or modify a data record such that a non-unique value would result, the update is rejected.

`index_name`

The index name must follow the rules for SIR/XS names and must be unique within the database.

`ON recname`

Specify the record name or number, qualified by a database name if this is not the default database. The database must be connected.

`var_name`

Specify the name(s) of the variable(s) used for this index. These must be variables in the named record (not common vars except for the case id).

Specify the keyword `DESC` to maintain the index in descending sequence of the variable.

Specify the keyword `UPPER` to use an uppercase representation of a character variable in the index.

DROP DBINDEX

```
DROP DBINDEX index_name ON [database.]recname | ALL
```

`DROP DBINDEX` deletes either a specific index or all indexes for a database.

SIR SCHEMA LIST

```
SIR SCHEMA LIST
    [FILENAME= 'fileid']
    [DETAILED|LABELS|REGULAR]
    [LONG|SHORT]
    [STRUCTURED]
    [CIR]
    [RECTYPES={recname|recnum} [(variable list)]... ]
```

Produces a listing of the current data definitions. The `FILENAME` specifies the output file. If this is omitted, the output is displayed in the scrolled output window.

By default, all record types in the database are listed. By default, all variables in the record type are listed. Each variable has a label that describes the variable and its position within the record. Positions are shown as Cnnn, Rnnn, or Tnnn where C stands for common variable, R for record variable and T for temporary variable. The nnn denotes the position of the variable within the record.

The exact format of the report depends on the options specified:

- `DETAILED` lists all the information in the `REGULAR` listing, plus the value labels for each variable.

  `LABELS` lists just the variables and the variable label. This is the shortest form of the report.

  `REGULAR` lists the following information for each variable:

  - variable name and label
  - variable type and size
  - read and write security levels
  - position and format on the input record
  - range
  - valid values
  - missing value designators

  This is the default.

- `SHORT` specifies that just the variable label (up to 78 characters) is printed and no headings or document text.

`LONG` specifies labels and all document text. This is the default.

- `STRUCTURED` lists just case id, key fields and document text for each record type.
- `CIR` lists all variable in the CIR. If CIR is specified, only the CIR is listed. Specify individual `RECTYPES` if these are required.
- `RECTYPES` specifies the name or number of individual record types to list and individual variables to list for each record type.

Examples: To list the name and labels of all the variables in record type 1 and the specified variables in record 3.

```
SIR SCHEMA LIST LABELS
                RECTYPES = 1 , 3 (POSITION REVDATE RATING)
```

To list all the variables in all record types without value labels.

```
SIR SCHEMA LIST
```

Example Output:

```
1.1       LIST SCHEMA LONG     07/09/05  10:05:57
        *** RECORD 1 (EMPLOYEE) DEFINITION ***
        Record Type 1 contains demographic information.
        There is one record per employee.  The record contains
        current position level and salary and the date
        on which the salary was last changed.

MAXIMUM NUMBER OF RECORDS/CASE         1
NUMBER OF THIS TYPE IN FILE            20
LINES IN EACH INPUT RECORD             1
ENTRY USE COUNT                        1
CASE IDENTIFIER:                       ID       (A)

        *** INPUT VARIABLE DEFINITIONS ***
C1.     ID,       Identification Number
        INPUT  ON LINE:  1
               COLUMNS:  1 - 4
                FORMAT:  I4
        DATA TYPE:       I*2
        MISSING VALUE:   (BLANK)

R1.     NAME,     Name
        INPUT  ON LINE:  1
               COLUMNS:  6 - 30
                FORMAT:  A25
        DATA TYPE:       A*27
        MISSING VALUE:   (BLANK)

R2.     GENDER,   Gender
        INPUT  ON LINE:  1
                COLUMN:  31
                FORMAT:  I1
        DATA TYPE:       I*1
        RANGE:           1/2
        MISSING VALUE:   (BLANK)
        VALUE LABELS:    (1) Male
                         (2) Female
```

WRITE SCHEMA

```
WRITE SCHEMA
    [FILENAME= fileid]
    [RECTYPES= rectype list]
    [CIR]
    [COMMON]
    [FORMS]
         [BOOLEANS]
         [COMPUTES]
    [MASKPW]
    [NOMAXKEY]
    [NOTO]
    [NOTXS]
    [PASSWORD]
    [PQLFORM]
         [NOUPDATE]
         [SUBROUTINE]
    [SECURITY]
    [VARLIST]
    [VARSEQ]
```

Writes a text file containing the schema definition. This might be used to recreate the database without data or procedures or, if it is the schema for a single record type, as the starting point to update that record definition. If mass changes are required to a database definition, it is sometimes easier to create a schema file and use a text editor to do global changes rather than modifying individual records through menus.

With the `PQLFORMS` parameter, this utility creates a default PQLForm.

With the `FORMS` parameter, this utility creates a default old Form.

If no record types are specified, the database definition and all record definitions are written excluding passwords.

`FILENAME`

Specifies the output file. If this is omitted, the output is displayed in the scrolled output window.

`RECTYPES`

Specifies the record types selected to generate a record or form definition.

`CIR`

Specifies that a record definition is written for the CIR. This is done anyway when all record types are selected, so this is only needed when selecting record types. If just `CIR` is specified, then just the CIR is written.

`COMMON`

The record definition for the CIR includes definitions for all the common variables and so these do not need to be repeated on subsequent record types that include a common variable. Specify `COMMON` if commands are to be produced for each common variable on every record where it occurs.

`FORMS`

Creates a default old style form. Unless otherwise specified, the CIR (Common Information Record) is not included in the form definition. Specify the keyword `BOOLEANS` to use any `ACCEPT REC IF` or `REJECT REC IF` specifications to generate appropriate commands in the output form. Specify the keyword `COMPUTES` to use any `COMPUTE` and `IF` commands to generate appropriate commands in the output form.

`MASKPW`

Specifies that an asterisk (*) is written instead of passwords.

`NOMAXKEY`

Suppresses the writing of a specific MAX KEY command to the output file. Specify this option whenever schema modifications are being made to allow the system to recalculate the maximum key size unless a larger key is required for future expansion.

`NOTO`

Where contiguous record variables share the same specification, a shorthand specification is written using the keyword `TO`. e.g.

```
MISSING VALUES  NAME
            TO  NDEPENDS                         (BLANK)
```

Specify the `NOTO` keyword to suppress this shorthand and to list all variables individually.

`NOTXS`

Specifies that the schema is as compatible as possible for use by earlier versions of SIR. New features such as `STANDARD VARS` and `RECORD SCHEMA 0` are not written and slashes are generated as separators.

`PASSWORD`

Specifies that the database password is written to the file.

`PQLForms`

Creates a default PQLForm. Unless otherwise specified, the CIR (Common Information Record) is not included in the PQLForm definition. Specify the keyword `NOUPDATE` for a read-only PQLForm. Specify the keyword `SUBROUTINE` for a PQLForm that compiles and saves as a sub-routine.

`SECURITY`

Specifies that the database security level passwords are written to the file.

`VARLIST`

The standard style of schema output is to write variable names and input definitions as a `DATA LIST`. `VARLIST` specifies that variable names are written as a `VARIABLE LIST` command followed by input definitions as an `INPUT FORMAT` command. e.g.

```
VARIABLE LIST ID NAME GENDER MARSTAT ...
INPUT FORMAT (I4,T6,A25,I1,I1,...
```

`VARSEQ`

The standard style of schema output is to write each definition command once and to list variables within the command. The `VARSEQ` keyword specifies that the output is in variable sequence and that all commands that apply to a single variable are grouped together. e.g.

```
VAR LABEL       NAME                             'Name'
MISSING VALUES  NAME                             (BLANK)

VAR LABEL       GENDER                           'Gender'
VAR RANGES      GENDER                           (1 2)
MISSING VALUES  GENDER                           (BLANK)
VALUE LABELS    GENDER                           (1)'Male'
                                                 (2)'Female'

VAR LABEL       MARSTAT                          'Marital status'
VAR RANGES      MARSTAT                          (1 2)
MISSING VALUES  MARSTAT                          (BLANK)
VALUE LABELS    MARSTAT                          (1)'Married'
                                                 (2)'Not married'
```

Resubmitting Whole Schema

If you specify a `DATA LIST` for a record schema that is already defined, then all existing definitions are ignored and the record is defined according to the new schema. Otherwise, `RECORD SCHEMA` modifies the definition. Modifications include labels and codes. You can add new variables and delete existing variables. You can modify a variable's data type, external format or position for batch data input.

When modifying data types for variables that already have data stored in the database, the data must be reformatted. Be careful if modifying a variable's data type. The following table indicates the result of changing data types.

T - Transfer value C - Convert to new type X - Convert to numeric if string is a valid number U - Convert to undefined		To					
		String	Cat Var	Date	Time	Integer	Real
From	String	T	C	C	C	X	X
	Catvar	C	T	U	U	T	C
	Date	C	U	T	U	T	C
	Time	C	U	U	T	T	C

	Integer	C	T	T	T	T	C
	Real	C	C	C	C	C	T

When transferring values, ranges, missing values, valid values and all schema definitions are checked and appropriately handled. Strings are truncated if they cannot fit in the new definition. Floating point numbers are truncated if they are converted to unscaled integers.

Dynamic Restructuring

Some changes to the schema affect the way existing data is treated. These changes include changes to the size or type of variables, additions or deletions of variables. This necessitates a restructure of existing data records that is done *dynamically* wherever possible. All changes except changes to the keyfields are dynamically restructured. Other changes such as changes in variable labels, value labels and security do not affect data storage at all.

Only changes that affect the index require the restructuring of the entire database. Dynamic restructuring means that the restructuring takes place as the data is used. Once the schema modification is made, the data is always seen as it is currently defined. Data that was stored under the old format is transformed into the new format every time that it is read. The record is physically restructured only when a write operation is performed. There is a small overhead involved in restructuring records dynamically each time they are read. If a series of changes had been made a simple VisualPQL program that reads and writes every record of that type, forces a physical restructure. For example:

```
RETRIEVAL UPDATE
. PROCESS RECORD employee
.  COMPUTE name = name
. END PROCESS REC
END RETRIEVAL
```

When a record schema is changed, internal tables are saved, indicating what changes have occurred and at what update levels. When a data record is accessed for a record type that has been changed, transformations are applied to bring it up to the current schema level. If the record is rewritten to the database, the restructured version is made permanent.

Database Restructure

Changes that affect the index require the restructuring of the entire database. This occurs if keys are redefined, a new record is defined with a key length greater than `MAX KEY SIZE` or the overall database constraints are respecified. Such a change does not take effect unless the `LOCK` parameter is specified on the record schema. This is because once a change has been made, the record is locked and the database must be restructured. The absence of the `LOCK` parameter prevents this happening unexpectedly.

A database restructure is done in steps;
make the schema modification;
run the `UNLOAD` utility to copy the database to an unload file;
delete the database using the `PURGE SIR FILE` utility;
reload the database with the `RELOAD` utility.

The database restructuring that takes place in an unload/reload restructures the data dictionary, the index and all of the data.

DELETE SCHEMA

```
DELETE SCHEMA n
```

Deletes the Record Schema 'n' from the database definition. `DELETE SCHEMA` only operates if there are no records for this record type. This is a DBA security level command.

When defining and redefining a record type, it is sometimes simpler to `DELETE SCHEMA` and redefine it through a complete new `RECORD SCHEMA` than to modify it over and over again.

When defining and testing a new database, delete all of the test data for a record type prior to deleting the schema. The following simple VisualPQL program does this.

```
RETRIEVAL UPDATE
PROCESS REC n
DELETE REC
END PROCESS REC
END RETRIEVAL
```

Batch Data Input Utilities

There are five batch data input utilities that provide a quick and efficient way to add or update data in the database from external text files.

The normal way to run these interactively is from the Data - File Input menu. These can also be run as commands. The batch data input utilities are :

- `ADD REC` that adds new records if they do not exist;
- `EVICT REC` that deletes existing records;
- `REPLACE REC` that replaces existing records;
- `UPDATE REC` that replaces selected variables on existing records;
- `INPUT DATA` that adds or replaces records.

The batch data input utilities use any `COMPUTE`,`IF`,`RECODE`,`ACCEPT` or `REJECT RECORD IF` clauses defined in the schema for a given record type. While you can use the utilities to input data directly into the CIR (record type zero), these clauses can only be specified at the normal record level even if they refer to common variables.

The `FILE DUMP` utility writes data from a database to a text file in a format that can be used by the data input utilities.

The `FILE LIST` utility writes a report showing the data from a database. Naturally this can be fairly voluminous.

You can also display and edit data through the SIR/XS `SPREADSHEET` utility that selects a set of data, displays it in a familiar spreadsheet style manner and allows you to update the data if necessary.

Batch Data Input Specifications

The utilities all have a very similar specification.
`ADD REC, READ INPUT DATA` and `REPLACE REC` have identical specifications.
`EVICT REC` has fewer options plus one particular keyword.
`UPDATE REC` has all the standard options plus four additional parameters.

They all have the same possible five files:

Input data
Contains the input data. The format of the data file is specified in the description of the record schema. A file in this format can be produced from an existing database by the `SIR FILE DUMP` utility or by the VisualPQL procedure `WRITE RECORDS`.
If variables on the record schema do not have input/output columns specified, these utilities automatically assign default columns at the end of any manually specified columns. If the input file is produced by `SIR FILE DUMP`, this allows database maintenance without having to assign columns manually. If you are processing a specific input file and want only to process variables with assigned input/output columns, the `NOAUTO` keyword suppresses this automatic assignment.
If the `CSV` keyword is specified, then the input file is in Comma Separated Variable (CSV) format. The input file is a text file with values for each record in a valid CSV format. The fields must be in the correct sequence that matches the sequence of fields on the database record. A file may either contain records for a single record type, in which case the record type is specified on the utility command or may contain multiple record types, in which case the first field on each input record is the record type.
A file in this format can be produced from an existing database by the `SIR FILE DUMP` utility with the `CSV` keyword or by the VisualPQL procedure `CSV SAVE FILE`.
Error Listing
A report of any errors.
Error Records
A copy of any data records with errors in the same format as the input file. (This could be reinput with an `ACCEPT` or other option.)
Summary
Update summary report.
Log
A fixed format that describes any errors. Each record contains the following:

```
COL    DESCRIPTION
-------------------------------
1-4    record number
5-6    message number
       1 record number error
       2 variable format error
       3 variable/compute error
       4 variable/recode error
       5 ACCEPT REC failed
       6 REJECT REC failed
       7 record accepted with errors
       8 record rejected
7-18   date of run
19-26  time of run
27-30  ordinal of record on this file
31-34  DBMS error number
35-42  variable name, if variable error
43-46  ACCEPT/REJECT REC num, line num
47-50  field starting column
51-54  field ending column
```

ADD REC

`ADD REC` adds new records to the database. The keys of incoming records are matched against those already in the database. If an input record matches an existing record, the incoming record is rejected with an error message.

```
ADD REC INPUT   =  filename
  [LISTFILE     =  filename]
  [ERRFILE      =  filename]
  [LOGFILE      =  filename]
  [SUMFILE      =  filename]
  [ACCEPT]
  [ALL]
  [BLANKUND]
  [CSV]
  [LOGALL]
  [NOAUTO]
  [NONEW]
  [NOSEQ]
  [ALIMIT       =  n]
  [BLIP         =  n]
  [LOADING      =  n]
  [RECTYPE      =  rectype]
  [RLIMIT       =  n]
  [SKIP         =  n]
  [STOP         =  n]
```

There are three groups of parameters. The first group specifies files, the next group specifies keywords and the last group specifies limits or other conditions. Optionally separate multiple parameters on a command with a slash "/".

FILES

`INPUT`
Specifies the input data file and must be specified.

`LISTFILE`
Specifies the file where error messages are written. If not specified, the current output file is used for error messages.

`ERRFILE`
Specifies the file where data records with errors are written. If not specified, no error file is produced, however errors messages are produced on the `LISTFILE`.

`LOGFILE`
Specifies the log file. A record is written to the log file in fixed format that describes each error.

`SUMFILE`
Specifies the file where the update summary report is written. If this is not specified the current output file is used. `SUMFILE` and `LISTFILE` can be the same file in which case the summary report is written after any error listing.

Keywords - Use these to specify the particular processing option(s) required:

`ACCEPT`
Specifies that records with erroneous values are accepted and variables with erroneous values are set to undefined. If not specified, records with erroneous values are rejected.
`ALL`
Specifies that input records with errors are written to the `ERRFILE` regardless of whether or not they are accepted into the database.
`BLANKUND`
Specifies that blank numeric fields on the input file result in `UNDEFINED` on the record. If this option is not specified, then blanks on input for a numeric field either result in a missing value, if a `BLANK` missing value is defined in the schema, or in a zero value.
`CSV`
Specifies that the input file is in CSV format.
`LOGALL`
Specifies that input records with errors are written with the log record to the `LOGFILE`.
`NOAUTO`
Specifies that only variables with specified input/output columns are processed.
`NONEW`
Specifies that no new cases are created.

Limits and Settings - Specify the keyword followed by an equals sign, "=" , followed by the value for these limits and settings.

`ALIMIT = n`
Sets a limit on the number of input records with errors. Terminates the procedure when "n" number of records have been processed with undefined values replacing errors.
`BLIP = n`
Specifies that an indication of progress is required and the approximate number of records expected on the input file. Progress is shown as a percentage of this number. Statistics of records added are displayed.
`LOADING = .n`
Specifies how data blocks are split as they become full. "N" is a number between 0.01 and 0.99. When data is reloaded or imported, blocks are filled. The normal value is 0.5, that means that a full data block is split in half. A value of .99 splits a data block with n records into one data block containing n-1 records and one data block containing 1 record. This is useful if records are added in keyfield order to keep the database file as compact as possible.
`RECTYPE = n | name`
Specifies that all the records in this run are of the given type. Rectype may be a record name or number. This is used when the data record does not contain a record type number or to override the number on the input record. Only one `RECTYPE=` keyword may be specified. If omitted, data records are identified by the

record number in the columns specified in the schema (or by the first field on a CSV file).

`RLIMIT = n`

Specifies that the run stops if "n" number of records are rejected due to errors.

`SKIP = n`

Specifies that the first "n" lines on the data input file are skipped before starting to process the data. Processing begins at line "n" + 1.

`STOP = n`

Stops the run after processing "n" lines from the data input file. If the data is in multi-line records, the entire record is always processed.

Example:

```
ADD REC INPUT    = 'INPUT.DAT'
        ERRFILE  = 'ERR.DAT'
        LOGFILE  = 'LOG.LST'
        LISTFILE = 'OUT.LST'
        SUMFILE  = 'SUM.LST'
        ACCEPT
        RECTYPE  = 1
```

EVICT REC

```
EVICT REC INPUT =  filename
  [LISTFILE     =  filename]
  [ERRFILE      =  filename]
  [LOGFILE      =  filename]
  [SUMFILE      =  filename]
  [CSV]
  [EVICTCIR]
  [LOGALL]
  [NOAUTO]
  [BLIP         =  n]
  [RECTYPE      =  rectype]
  [RLIMIT       =  n]
  [SKIP         =  n]
  [STOP         =  n]
```

Deletes records. The keys of input records are matched against those already in the database. If an input record matches an existing record, the existing record is deleted. If an input record does not match an existing record, an error message is written.

There are three groups of parameters. The first group specifies files, the next group specifies keywords and the last group set limits or other conditions. Optionally separate multiple parameters on a command with a slash "/".

FILES

`INPUT`
Specifies the input data file. Must be specified.
`LISTFILE`
Specifies the file where error messages are written. If not specified, the current output file is used for error messages.
`ERRFILE`
Specifies the file where data records with errors are written. If not specified, no error file is produced, however errors messages are produced on the LISTFILE.
`LOGFILE`
Specifies the log file. A record is written to the log file in fixed format that describes each error.
`SUMFILE`
Specifies the file where the update summary report is written. If this is not specified the current output file is used. `SUMFILE` and `LISTFILE` can be the same file in which case the summary report is written after any error listing.

Keywords - Use these to specify the particular processing option(s) required.

`CSV`

Specifies that the input file is in CSV format.
`EVICTCIR`
Specifies that cases are deleted if all records in the case are deleted. This only applies to case structured databases.
`LOGALL`
Specifies that input records with errors are written with the log record to the `LOGFILE`.
`NOAUTO`
Specifies that only variables with specified input/output columns are processed.

Limits and Settings - Specify the keyword followed by an equals sign, "=" , followed by the value for these limits and settings.

`BLIP`
Specifies that an indication of progress is required and the approximate number of records expected on the input file. Progress is shown as a percentage of this number. Statistics of records added are displayed.
`RECTYPE`
Specifies that all the records in this run are of the given type. Rectype may be a record name or number. This is used when the data record does not contain a record type number or to override the number on the input record. Only one `RECTYPE=` can be specified. If omitted, data records are identified by the record number in the columns specified in the schema (or by the first field on a CSV file).
`RLIMIT`
Specifies that the run stops if "n" number of records are rejected due to errors.
`SKIP`
Specifies that the first "n" lines on the data input file are skipped before starting to process the data. Processing begins at line "n" + 1.
`STOP`
Stops the run after processing "n" lines from the data input file. If the data is in multi-line records, the entire record is always processed.

Example:

```
EVICT REC  INPUT     = 'INPUT.DAT'
           ERRFILE   = 'ERR.DAT'
           LOGFILE   = 'LOG.LST'
           LISTFILE  = 'OUT.LST'
           SUMFILE   = 'SUM.LST'
           RECTYPE   = 1
```

READ INPUT DATA

```
READ INPUT DATA INPUT  =  filename
    [LISTFILE          =  filename]
    [ERRFILE           =  filename]
    [LOGFILE           =  filename]
    [SUMFILE           =  filename]
    [ACCEPT]
    [ALL]
    [CSV]
    [LOGALL]
    [NOAUTO]
    [NONEW]
    [NOSEQ]
    [ALIMIT           =  n]
    [BLIP             =  n]
    [LOADING          =  n]
    [RECTYPE          =  rectype]
    [RLIMIT           =  n]
    [SKIP             =  n]
    [STOP             =  n]
```

Adds new records and replaces existing records. The keys of incoming records are matched against those already in the database. If an input record matches an existing record, the existing record is replaced; if the keys do not match, a new record is added.

There are three groups of parameters. The first group specifies files, the second group specifies keywords and the last group set limits or other conditions. Optionally separate multiple parameters on a command with a slash "/".

FILES

`INPUT`
Specifies the input data file. Must be specified.
`LISTFILE`
Specifies the file where error messages are written. If not specified, the current output file is used for error messages.
`ERRFILE`
Specifies the file where data records with errors are written. If not specified, no error file is produced, however error messages are produced on the `LISTFILE`.
`LOGFILE`
Specifies the log file. A record is written to the log file in fixed format that describes each error.
`SUMFILE`

Specifies the file where the update summary report is written. If this is not specified the current output file is used. `SUMFILE` and `LISTFILE` can be the same file in which case the summary report is written after any error listing.

Keywords - Use these to specify the particular processing option(s) required.

`ACCEPT`
Specifies that records with erroneous values are accepted and variables with erroneous values are set to undefined. If not specified, records with erroneous values are rejected.
`ALL`
Specifies that input records with errors are written to the `ERRFILE` regardless of whether or not they are accepted into the database.
`CSV`
Specifies that the input file is in CSV format.
`BLANKUND`
Specifies that blank numeric fields on the input file result in `UNDEFINED` on the record. If this option is not specified, then blanks on input for a numeric field either result in a missing value if a `BLANK` missing value is defined in the schema or in a zero value.
`LOGALL`
Specifies that input records with errors are written with the log record to the `LOGFILE`.
`NOAUTO`
Specifies that only variables with specified input/output columns are processed.
`NONEW`
Specifies that no new cases are created.

Limits and Settings - Specify the keyword followed by an equals sign, "=" , followed by the value for these limits and settings.

`ALIMIT`
Sets a limit on the number of input records with errors. Terminates the procedure when "n" number of records have been processed with undefined values replacing errors.
`BLIP`
Specifies that an indication of progress is required and the approximate number of records expected on the input file. Progress is shown as a percentage of this number. Statistics of records added are displayed.
`LOADING`
Specifies when data blocks are split as they become full.

"N" is a number between 0.01 and 0.99. When data is reloaded or imported, blocks are filled. On subsequent update runs, the normal value is 0.5, that means that a full data block is split in half. A value of .99 splits a data block with n records into one data block containing n-1 records and one data block containing 1 record. This is useful if records are added in keyfield order to keep the database file as compact as possible.

`RECTYPE`
Specifies that all the records in this run are of the given type. Rectype may be a record name or number. This is used when the data record does not contain a record type number or to override the number on the input record. Only one `RECTYPE=` keyword may be specified. If omitted, data records are identified by the record number in the columns specified in the schema (or by the first field on a CSV file).

`RLIMIT`
Specifies that the run stops if "n" number of records are rejected due to errors.

`SKIP`
Specifies that the first "n" lines on the data input file are skipped before starting to process the data. Processing begins at line "n" + 1.

`STOP`
Stops the run after processing "n" lines from the data input file. If the data is in multi-line records, the entire record is always processed.

Example:

```
READ INPUT DATA INPUT = 'INPUT.DAT'
      ERRFILE = 'ERR.DAT'
      LOGFILE = 'LOG.LST'
      LISTFILE = 'OUT.LST'
      SUMFILE  = 'SUM.LST'
      ACCEPT
      RECTYPE = 1
```

REPLACE REC

```
REPLACE REC INPUT   =  filename
    [LISTFILE       =  filename]
    [ERRFILE        =  filename]
    [LOGFILE        =  filename]
    [SUMFILE        =  filename]
    [ACCEPT]
    [ALL]
    [CSV]
    [LOGALL]
    [NOAUTO]
    [NONEW]
    [NOSEQ]
    [ALIMIT         =  n]
    [BLIP           =  n]
    [LOADING        =  n]
    [RECTYPE        =  rectype]
    [RLIMIT         =  n]
    [SKIP           =  n]
    [STOP           =  n]
```

Replaces existing records. The keys of input records are matched against those already in the database. If an input record does not match an existing record, it is rejected with an error message. If a match is found, the existing record is replaced by the input record.

There are three groups of parameters. The first group specifies files, the next group specifies keywords and the last group sets limits or other conditions. Optionally separate multiple parameters on a command with a slash "/".

FILES

`INPUT`

Specifies the input data file. Must be specified.

`LISTFILE`

Specifies the file where error messages are written. If not specified, the current output file is used for error messages.

`ERRFILE`

Specifies the file where data records with errors are written. If not specified, no error file is produced, however errors messages are produced on the `LISTFILE`.

`LOGFILE`

Specifies the log file. A record is written to the log file in fixed format that describes each error.

`SUMFILE`

Specifies the file where the update summary report is written. If this is not specified the current output file is used. `SUMFILE` and `LISTFILE` can be the same file in which case the summary report is written after any error listing.

Keywords - Use these to specify the particular processing option(s) required.

`ACCEPT`
Specifies that records with erroneous values are accepted and variables with erroneous values are set to undefined. If not specified, records with erroneous values are rejected.
`ALL`
Specifies that input records with errors are written to the `ERRFILE` regardless of whether or not they are accepted into the database.
`CSV`
Specifies that the input file is in CSV format.
`BLANKUND`
Specifies that blank numeric fields on the input file result in `UNDEFINED` on the record. If this option is not specified, then blanks on input for a numeric field either result in a missing value if a `BLANK` missing value is defined in the schema or in a zero value.
`LOGALL`
Specifies that input records with errors are written with the log record to the `LOGFILE`.
`NOAUTO`
Specifies that only variables with specified input/output columns are processed.
`NONEW`
Specifies that no new cases are created.

Limits and Settings - Specify the keyword followed by an equals sign, "=" , followed by the value for these limits and settings.

`ALIMIT`
Sets a limit on the number of input records with errors. Terminates the procedure when "n" number of records have been processed with undefined values replacing errors.
`BLIP`
Specifies that an indication of progress is required and the approximate number of records expected on the input file. Progress is shown as a percentage of this number. Statistics of records added are displayed.
`LOADING`
Specifies how data blocks are split as they become full. "N" is a number between 0.01 and 0.99. When data is reloaded or imported, blocks are filled. The normal value is 0.5, that means that a full data block is split in half. A value of .99 splits a data block with n records into one data block containing n-1 records and one data block containing 1 record. This is useful if records are added in keyfield order to keep the data as compact as possible.
`RECTYPE`

Specifies that all the records in this run are of the given type. Rectype may be a record name or number. This is used when the data record does not contain a record type number or to override the number on the input record. Only one `RECTYPE=` keyword may be specified. If omitted, data records are identified by the record number in the columns specified in the schema (or by the first field on a CSV file).

`RLIMIT`

Specifies that the run stops if "n" number of records are rejected due to errors.

`SKIP`

Specifies that the first "n" lines on the data input file are skipped before starting to process the data. Processing begins at line "n" + 1.

`STOP`

Stops the run after processing "n" lines from the data input file. If the data is in multi-line records, the entire record is always processed.

Example:

```
REPLACE REC INPUT    = 'INPUT.DAT'
            ERRFILE  = 'ERR.DAT'
            LOGFILE  = 'LOG.LST'
            LISTFILE = 'OUT.LST'
            SUMFILE  = 'SUM.LST'
            ACCEPT
            RECTYPE = 1
```

UPDATE REC

```
UPDATE REC INPUT    =  filename
   [LISTFILE        =  filename]
   [ERRFILE         =  filename]
   [LOGFILE         =  filename]
   [SUMFILE         =  filename]
   [ACCEPT]
   [ADD]
   [ALL]
   [COMPUTE]
   [CSV]
   [LOGALL]
   [NOAUTO]
   [NOBOOL]
   [NONEW]
   [NOSEQ]
   [ALIMIT    =  n]
   [BLIP      =  n]
   [LOADING   =  n]
   [MISSCHAR  =  a]
   [RECTYPE   =  rectype]
   [RLIMIT    =  n]
   [SKIP      =  n]
   [STOP      =  n]
```

Replaces individual variables in existing records. The keys of input records are matched against those already in the database. If a match is found, the variables in the existing record are replaced by non-blank fields in the input. If a match is not found, the input record is rejected with an error message, or, if the `ADD` keyword is specified, a new record is created.

There are four additional parameters for `UPDATE RECORD`:

`ADD`
Specify to add new records. By default, input records must match existing records in the database.
`COMPUTE`
Specify to re-execute schema `COMPUTE` statements. By default, `COMPUTE` statements from the Schema are not re-executed.
`NOBOOL`
Specify to stop the re-execution of consistency checks from the Schema. By default, consistency checks (`ACCEPT REC IF` and `REJECT REC IF`) are performed. Any temporary variables referenced in the consistency check must be respecified on the input record to assure that the intent of the check is satisfied.
`MISSCHAR`

Specify a single character to indicate that an existing variable is set to undefined. In order to set an existing value to `UNDEFINED`, include this character on the input record in the leftmost column of the variable. A blank does not indicate a missing value and may not be used as the character. There is no default.

There are three groups of parameters. The first group specifies files, the next group comprises keywords and the last group sets limits or other conditions. Optionally separate multiple parameters with a slash "/".

FILES

`INPUT`
Specifies the input data file. Must be specified.
`LISTFILE`
Specifies the file where error messages are written. If not specified, the current output file is used for error messages.
`ERRFILE`
Specifies the file where data records with errors are written. If not specified, no error file is produced, however errors messages are produced on the `LISTFILE`.
`LOGFILE`
Specifies the log file. A record is written to the log file in fixed format that describes each error.
`SUMFILE`
Specifies the file where the update summary report is written. If this is not specified the current output file is used. `SUMFILE` and `LISTFILE` can be the same file in which case the summary report is written after any error listing.

Keywords - Use these to specify the particular processing option(s) required.

`ACCEPT`
Specifies that records with erroneous values are accepted and variables with erroneous values are set to undefined. If not specified, records with erroneous values are rejected.
`ADD`
Specifies that input records that do not match existing records are added to the database. Schema defined consistency checks and compute specifications are applied to the added records.
`ALL`
Specifies that input records with errors are written to the `ERRFILE` regardless of whether or not they are accepted into the database.
`COMPUTE`
Specifies that any `COMPUTE` specifications in the schema are re-executed.
`CSV`
Specifies that the input file is in CSV format.
`LOGALL`
Specifies that input records with errors are written with the log record to the `LOGFILE`.
`NOAUTO`

Specifies that only variables with specified input/output columns are processed.
`NOBOOL`
Specifies that any `ACCEPT REC IF` or `REJECT REC IF` specifications in the schema are bypassed.
`NONEW`
Specifies that no new cases are created.

Limits and Settings - Specify the keyword followed by an equals sign, "=" , followed by the value for these limits and settings.

`ALIMIT`
Sets a limit on the number of input records with errors. Terminates the procedure when "n" number of records have been processed with undefined values replacing errors.
`BLIP`
Specifies that an indication of progress is required and the approximate number of records expected on the input file. Progress is shown as a percentage of this number. Statistics of records added are displayed.
`LOADING`
Specifies how data blocks are split as they become full. "N" is a number between 0.01 and 0.99. When data is reloaded or imported, blocks are filled. The normal value is 0.5, that means that a full data block is split in half. A value of .99 splits a data block with n records into one data block containing n-1 records and one data block containing 1 record. This is useful if records are added in keyfield order to keep the database file as compact as possible.
`MISSCHAR`
Specifies a character to indicate that the field is set to `UNDEFINED`. When this character is in the leftmost position of a variable on input, the variable on the database is set to undefined. Specify a single character, do not enclose it in quotes.
`RECTYPE`
Specifies that all the records in this run are of the given type. Rectype may be a record name or number. This is used when the data record does not contain a record type number or to override the number on the input record. Only one `RECTYPE=` keyword may be specified. If omitted, data records are identified by the record number in the columns specified in the schema (or by the first field on a CSV file).
`RLIMIT`
Specifies that the run stops if "n" number of records are rejected due to errors.
`SKIP`
Specifies that the first "n" lines on the data input file are skipped before starting to process the data. Processing begins at line "n" + 1.
`STOP`
Stops the run after processing "n" lines from the data input file. If the data is in multi-line records, the entire record is always processed.

Example:

```
UPDATE REC  INPUT     = 'INPUT.DAT'
            ERRFILE   = 'ERR.DAT'
            LOGFILE   = 'LOG.LST'
            LISTFILE  = 'OUT.LST'
            SUMFILE   = 'SUM.LST'
            ACCEPT
            RECTYPE  = 1
            MISSCHAR  = *
```

SIR FILE DUMP

```
SIR FILE DUMP  [FILENAME  = fileid]
    RECTYPES  = {ALL | rectype (log_expr),...}
    [BOOLEAN   = (log_expr)]
    [CIR]
    [COUNT     = total [,increment[,start]]]
    [CSV]
    [DPOINT]
    [LIST      = case id list]
    [NOAUTO]
    [SAMPLE    = fraction [,seed]]
    [UNDEFINED = char]
```

Creates a text file in a form suitable for processing by the batch data input utilities. DBA read security clearance is needed to use this utility.

Optionally separate multiple parameters on the command with slashes.

`FILENAME`
Specifies the name of the output file. If this clause is not specified, the output is written to the default output file (normally the scrolled output buffer in interactive sessions).

`RECTYPES`
Specifies the record types to dump. This clause is required. The keyword `ALL` specifies all record types are dumped. A logical expression can be specified to restrict the data records selected. The expression can reference common variables or variables from the listed record type and can include PQL functions.

`BOOLEAN`
Defines a logical condition applied to common variables. This clause only applies to case structured databases. If the logical condition is true, records for that case are dumped. `BOOLEAN` is applied after any `SAMPLE`, `COUNT` or `LIST`.

`CIR`
Specify `CIR` to output common variables as a separate record (record type 0).

`COUNT`
Outputs data from a specified number of cases from the database. This clause only applies to case structured databases and cannot be used with `SAMPLE` or `LIST`. Total specifies the number of cases to retrieve. Increment specifies the "skipping factor" for retrieving cases. Start specifies the first case to select. For example, a start value of 3 begins the processing at the third case.

```
SIR FILE DUMP FILENAME =  'OUTPUT.DAT'
              RECTYPES =  ALL COUNT = 10
```

`CSV`
Specifies that the file is written in CSV format.

`DPOINT`

Specifies that, when writing a fixed format file, any numeric fields that have a decimal component, have the decimal point included. This is automatically done when in CSV format.

`LIST`

Retrieves the specified cases for case structured databases. Enclose case identifiers that are character strings in quotes. `LIST` cannot be used with `SAMPLE` or `COUNT`. For example:

```
SIR FILE DUMP FILENAME =  'OUTPUT.DAT'
              RECTYPES =  ALL  LIST = 1,3,5 thru 10
```

`NOAUTO`

Specifies that only variables with specified input/output columns are processed.

`SAMPLE`

Outputs data from a random sample of cases from the database. This clause only applies to case structured databases. Fraction specifies the sample size for selection. Seed specifies the starting seed for the random number generator. If seed is not specified, the default is used.

`UNDEFINED`

Specifies the character used to fill fields on output that are undefined on the database. Blanks are the default. For example:

```
SIR FILE DUMP FILENAME  =  'OUTPUT.DAT'
    RECTYPES = 1 (SALARY GT 2000)
    UNDEFINED = *
```

SIR FILE LIST

```
SIR FILE LIST FILENAME=filename
    [BOOLEAN=   (log-expr)]
    [LIST=      caseid list]
    [RECTYPES=  rectype [(log-expr)] ...| ALL]
    [ORDER=     ALPHA | VARNUM]
    [SAMPLE=    fraction [,seed]]
    [COUNT=     total [,incr[,start]]]
    [CIR=       varlist | NOCIR]
    [VARIABLES = rectype (var-list)]
```

Writes all or part of the data to a file for subsequent printing. Use the filename `CONSOL` to write to the screen or `STDOUT` to write to the default output file.

DBA read security is required to use this utility.

`BOOLEAN`

Specifies cases to include in the list. If the test fails, no records for that case are listed. The test may only use common variables. If `LIST`, `COUNT` or `SAMPLE` is used, the `BOOLEAN` clause is applied after that selection process. For example:

```
BOOLEAN = (ID GT 5)
```

`LIST`

Specifies cases to select. Separate entries with blanks or commas. Use the "TO" format to specify a range. Enclose case ids that are strings in quotes. `LIST` cannot be used with `SAMPLE` or `COUNT`. For example:

```
LIST= 1,3,5,7 to 10
```

`RECTYPES`

Specifies the record types to select. `RECTYPES= ALL` specifies all record types. If `RECTYPES` is omitted from the command, only the common variables are listed. A logical expression may be defined to select particular data records within a record type. If the test is `TRUE`, the record is listed. The expression may use common or record variables from the record type. For example:

```
RECTYPES= 1 (GENDER=2)
```

`ORDER`

Specifies the sequence of the listing of variables. This can be alphabetic order (`ALPHA`) of the variable name or the order the variables are defined in the record (`VARNUM`). `VARNUM` is the default. For example:

```
SIR FILE LIST  FILENAME='DATA.LIS'  ORDER= ALPHA
```

`SAMPLE`

Retrieves a random sample of cases from case-structured databases. Fraction specifies the sample size for selection. Seed specifies the starting seed for the random number generator. If seed is not specified, one is assigned by default. `SAMPLE` cannot be used with `COUNT` or `LIST`. For example:

```
SIR FILE LIST FILENAME='DATA.LIS'  SAMPLE= .5
```

`COUNT`

Retrieves a subset of cases from the database. Total specifies the number of cases to retrieve. Incr is the increment to apply to locate the next case to process. The default is 1 and processes every case. An increment of 2 processes every other case, 3 every third case, etc. Start specifies the ordinal of the first case to process. For example, a start value of 3 and an Incr of 3 starts the processing with the third case, skips cases 4 and 5 and processes 6. `COUNT` cannot be used with `SAMPLE` or `LIST`. For example:

```
SIR FILE LIST FILENAME='DATA.LIS'  COUNT= 50,3,3
CIR
```

Specifies the common variables to list. If the `CIR` clause is omitted, all common variables are listed. For example:

```
SIR FILE LIST FILENAME='DATA.LIS'  CIR= ID
NOCIR
```

Suppresses output of CIR variables. For example:

```
SIR FILE LIST FILENAME='DATA.LIS'   NOCIR
VARIABLES
```

Specifies the variables to list for a record type. All variables are listed for selected record types as the default. For example:

```
SIR FILE LIST FILENAME='DATA.LIS' RECTYPE = 3
    VARIABLES = 3 (position,revdate)
```

Sample Output:

```
SIR/XS FILE LIST                         Jan 05, 2006
10:53:36    Page  1

***CASE  ID                      1

REVIEW                                                   POSITION
4
REVDATE   04 05 03
POSITION                 4

REVIEW                                                   POSITION
4
REVDATE   06 05 03
POSITION                 4

REVIEW                                                   POSITION
5
REVDATE   12 09 04
POSITION                 5

REVIEW                                                   POSITION
5
REVDATE   02 04 05
POSITION                 5

***CASE  ID                      2

REVIEW                                                   POSITION
6
REVDATE   03 16 03
POSITION                 6
```

```
REVIEW                                                 POSITION
6
REVDATE   04 27 03
POSITION                 6
```

SIR SPREADSHEET

```
SIR SPREADSHEET
    {RECTYPE = recname [BOOLEAN = (log-expr)] |
TABLE=tabfile.table}
    [INDEXED BY indexname]
    [VARIABLES = (var1,var2,... | ALL)]
    [LABELS|UPDATE]
```

Selects data from a single database record type or from a tabfile table and displays it in a graphical form similar to a spreadsheet display. The user can insert, delete or update if allowed, and can print or export the data in a CSV format for input to other packages.

`RECTYPE =`

Specifies the record name or number to display. Specify `BOOLEAN` to select records. The specified test can use common variables and record variables.

`TABLE =`

Specifies the tabfile name and table name to display.

`INDEXED BY` (`USING` is a synonym)

Specifies the record or table is accessed via the named index. When using an index with a record, all the variables used in the index must be included in the displayed variables.

`VARIABLES`

Specifies the variables to list. `ALL` is the default.

`LABELS` (`VALLAB` is a synonym)

Specifies that value labels are displayed where these exist as opposed to actual values. This precludes `UPDATE`

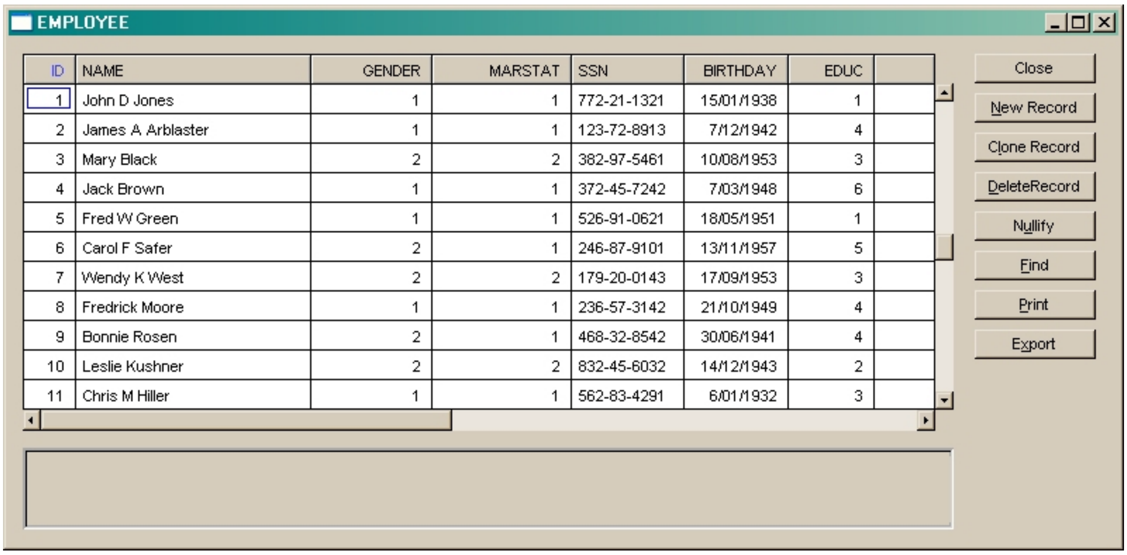
`UPDATE`

Specifies that updates to the database or table are allowed. This allows the user to add, delete or modify the data in the record or table. For update, the selected variables must included all key fields.

For example:

```
SIR SPREADSHEET RECTYPE=EMPLOYEE UPDATE
```

The record data is displayed as a spreadsheet that looks something like:



ID	NAME	GENDER	MARSTAT	SSN	BIRTHDAY	EDUC
1	John D Jones	1	1	772-21-1321	15/01/1938	1
2	James A Arblaster	1	1	123-72-8913	7/12/1942	4
3	Mary Black	2	2	382-97-5461	10/08/1953	3
4	Jack Brown	1	1	372-45-7242	7/03/1948	6
5	Fred W Green	1	1	526-91-0621	18/05/1951	1
6	Carol F Safer	2	1	246-87-9101	13/11/1957	5
7	Wendy K West	2	2	179-20-0143	17/09/1953	3
8	Fredrick Moore	1	1	236-57-3142	21/10/1949	4
9	Bonnie Rosen	2	1	468-32-8542	30/06/1941	4
10	Leslie Kushner	2	2	832-45-6032	14/12/1943	2
11	Chris M Hiller	1	1	562-83-4291	6/01/1932	3

Backup and Recovery

SIR/XS provides utilities to write all or part of the database to external files in either machine dependent or machine independent formats and to re-input these files back to a SIR/XS database. These utilities can be used for restructuring the database, backing up the database, porting all or part of the database to another database on the same operating system or creating a machine independent version of the database to move to a different operating system.

***Note:** If transferring export or other text (machine independent) files between machines using ftp, you must use ftp in **ACSII** mode (not BINARY)*

There are various procedures and the utilities to assist in protecting a database and recovering it in the case of problems.

The key to a successful recovery operation is being prepared. Do not assume that the computer, disk drives or power supply are always 100% trouble free. Be prepared for unexpected problems by taking regular backups, saving journals and, in general, take reasonable precautions against losing much time or work.

The procedure for restructuring a database is the same as for backing up and recovery.

There are utilities that are designed to work in pairs with one providing input in a suitable format for the other.

Some utilities create or use *binary files* that are specific to an operating system. These files are created in "append" mode; that is they are added to the end of any existing unload or subset file with the same name. However, if you run these through the menu system, you are given the choice of deleting the old file first.

Other utilities create *text files* that are machine independent and can be viewed and updated by any text editor. These are produced independently and overwrite existing files.

- `EXPORT` creates a file containing a text copy of the database, including the data dictionary, procedures and data. This can be used by `IMPORT` to create a new database on a different machine.

- `WRITE SCHEMA` writes just the database definition in a similar format to export.

- `SUBSET` writes a subset of the database to an unload binary file. This includes the schema and data for selected record types. This can be used by `MERGE` to combine with an existing database or `RELOAD` to create a new database that is a subset of the original.

- `UNLOAD` writes a copy of the database to an unload binary file including the schema and the procedures.

- `UPLOAD` creates a text copy from the journal of all updates that have been done. This can be used by `DOWNLOAD` to apply those changes to a second copy of the database. An example of this might be a central database with subsets on various PCs in remote locations. Updates might be done on the PCs and `UPLOAD`ed to the center, or updates might be done in the center and `UPLOAD` to the PCs to avoid re-transmitting the whole of the database.

There are two utilities that check on the contents of the system.

- `VERIFY` checks on the contents of the structure of the database and gives details of any problems discovered. This utility has a `PATCH` option that recovers from many types of corruptions. If some type of problem has occurred, a SIR/XS process may warn that a corruption has occurred. If this happens, use the `VERIFY FILE` utility to find out more about the problem, and attempt to correct it.
- `ITEMIZE` reports on the contents of either the journal file or files produced by `UNLOAD` or `SUBSET`.

`LIST STATS` reports on the current status of the database giving the number of records, data size and update level.

Journaling

Journaling can be turned "ON" for a database. This means that, each time that the database is updated (i.e. the update level increases), an entry is written to the journal file. The entry contains details of all of the updates done to take the database from one update level to the next. Each entry on the journal file consists of images of all of the data records that were updated during the update run. The images can be both before and after images of the records depending on the update. For a new record, there is an after image; for an updated record there is a before and after image; for a deleted record, there is a before image.

The journal can be used to recover in the case of an unexpected interruption in an update run and allows updates to be re-applied quickly and easily if a backup has to be restored. The journal can be used in VisualPQL to produce reports on updates or other audit trails.

If updates were incomplete or unsuccessful in some way, they can be 'undone' with a `JOURNAL ROLLBACK` that takes a database back to a previous update level. When a database is connected, its status is checked to see if it was not closed properly when being updated e.g. the system 'crashed' while the database was open for update. If this is found to be the case, you are asked if you wish to automatically recover. If you choose to try to recover, a journal rollback is done. If a database has to be recovered from backups, restore the database from the backup and then use `JOURNAL RESTORE` to bring that version of the database up to the level of the journal file.

Binary Files

Binary files are machine dependant and are NOT suitable for transferring between operating systems or different hardware. These are NOT suitable for long term archival storage. Subsequent versions of SIR may have updated file formats that are incompatible. The following utilities create binary files

.

- A full database is created by `UNLOAD FILE` and read by `RELOAD FILE`
- A partial database is created by `SIR SUBSET` and read by `MERGE`
- Database changes are created by `JOURNAL` and read by `JOURNAL RESTORE`
- A binary file can contain multiple sets of data. These are listed by `ITEMIZE`.

Text Files

Text files are suitable for transferring between machines and can be viewed with a normal editor. The following utilities create text files.

- A full or partial database is created by `EXPORT` and read by `IMPORT`. This is the recommended format for storing all long term SIR archives. It is machine and operating system independent and subsequent versions of SIR are always compatible with previous EXPORT formats.
- Database definitions are written by `WRITE SCHEMA`
- Database changes are created by `UPLOAD` and read by `DOWNLOAD`

IMPORT

To import and recreate a complete database, simply tell SIR/XS to read the export file generated by a previous `EXPORT` utility. This file is a text copy of a database consisting of commands to recreate the database and data to load into it.

This can be done in a number of ways:

- From the `Database\Recover\Import` menu
- By running the system in batch with the import file named as the `IN =` parameter
- By "running" the import file from the `Procedures - File` menu

There is a command "`IMPORT`" that indicates that data, in a suitable format for importing, follows the command. eg:

```
IMPORT
0/1/1/2/4/1/12/John D Jones1/1/11/772-21-
1321129754/1/M1/5/2150/145851/2/4/
145120/1500/1/2/5/145733/2000/1/3/4/145241/5/1650/2/3/4/145180/4/
1600/2/3/5/
145851/5/2150/2/3/5/145794/4/2100/2/0/2/1/1/3/1/25/James A
Arblaster        1/1/
...
```

EXPORT

```
EXPORT FILENAME  = fileid
     [BOOLEAN  = (logical_expr)]
     [COUNT    = total [,increment [,start]]]
     [LIST     = caseid, ...]
     [RECTYPES = ALL | rectype(logical_expr) ...]
     [SAMPLE   = fraction [,seed]]
     [DATABASE = new database name]
     [PASSWORD = new database password]
     [COMMON]
     [NODATA]
     [NOINDEX]
     [NOMAXKEY]
     [NOPASSWORDS]
     [NOPROCS]
     [NOTO]
     [NOTXS]
     [VARLIST]
     [VARSEQ]
```

Creates a file of text records containing the data dictionary, procedures and data from the database. Exports all non-compiled procedures, i.e members with a :T, :M or :P suffix (text, menu and picture (template) members). Compiled procedures (:E, :O and :V) are not machine independent and cannot be exported. Compiled procedures must be re-compiled after an `IMPORT`.

Database administrator security is required to use `EXPORT`. If the export does not run because record types in the database are locked (due to schema modifications) then restructure the database before rerunning the export.

Optionally separate multiple parameters on the command with slashes.

`FILENAME`
Specifies the name of the file to contain the exported database. This is required.
`BOOLEAN`
Specifies tests applied to cases before the case is written to the export file. A case is only written if the expression is `TRUE`. Only use common variables in the expression and only use for databases with a case structure. The test is applied

after any `SAMPLE, COUNT` or `LIST`. If the case id is a categorical, date or time variable, specify either a string or numeric test and the equivalent variable value is used. VisualPQL functions can be used. For example:

```
BOOLEAN = (ID GT 5)
```

`LIST`

Exports only the specified cases. Specify the keyword `THRU` to select a range. For example:

```
LIST = 12 THRU 29, 33, 37
```

`RECTYPES`

Selects the data records written to the export file. Schema definitions are written for all record types, regardless of which rectypes are selected on the `RECTYPES` clause. If this clause is omitted, data for all record types in the database are exported.

Specify a logical expression to select records. Records that meet the criteria of the logical expression are selected. For example:

```
RECTYPES = EMPLOYEE (SALARY GT 2000)
```

`SAMPLE`

Exports a random sample of cases from the database. Fraction specifies the sample size (a decimal number) for selection. Seed specifies the starting seed for the random number generator. If seed is not specified, a default seed is used. For example, `SAMPLE = .25` exports 25% of the cases from the database using a default seed.

`COUNT`

Exports a subset of cases from the database. Total is the number of cases to retrieve. Increment is a number that specifies the "skipping factor" for retrieving cases. For example, an increment of 3 produces every third case. Start specifies the first case processed. For example, a start of 3 starts with the third case.

`DATABASE`

Specifies a new name for the database on the export file. If this clause is omitted, the current database name is used.

`PASSWORD`

Changes the database password on the export file. If this clause is omitted, the current password is used.

`COMMON`

The record definition for the CIR includes definitions for all the common variables and so these do not need to be repeated on subsequent record types that include a common variable. Specify `COMMON` if commands are to be produced for each common variable on every record where it occurs.

`NODATA`

Specifies that no data records are written to the export file.

`NOINDEX`

Specifies that no specifications for secondary indexes are written to the export file if transferring from SIR/XS or later to earlier versions that did not support indexes.

`NOMAXKEY`

Suppresses the writing of a specific MAX KEY command to the output file. Specify this option whenever schema modifications are being made to allow the system to recalculate the maximum key size unless a larger key is required for future expansion.

`NOPASSWORDS`

Specifies that the database password, security passwords and member passwords are not written to the export file. This has the effect of removing all password protection from the new database. New passwords can be assigned once the new database has been imported to its new location and recreated.

`NOPROCS`

Specifies that the procedures are not written to the export file.

`NOTO`

Where contiguous record variables share the same specification, a shorthand specification is written using the keyword `TO`. e.g.

```
MISSING VALUES  NAME
            TO  NDEPENDS                         (BLANK)
```

The `NOTO` keyword suppresses this shorthand and all variables are individually listed within the specification.

`NOTXS`

Specifies that the export is as compatible as possible for use by earlier versions of SIR (2002). New features such as `STANDARD VARS` and `RECORD SCHEMA 0` are not written and slashes are generated as separators.

`VARLIST`

The standard style of schema output is to write variable names and input definitions as a `DATA LIST`. `VARLIST` specifies that variable names are written as a `VARIABLE LIST` command followed by input definitions as an `INPUT FORMAT` command. e.g.

```
VARIABLE LIST ID NAME GENDER MARSTAT ...
INPUT FORMAT (I4,T6,A25,I1,I1,...
```

`VARSEQ`

The standard style of schema output is to write each definition command once and to list variables within the command. The `VARSEQ` keyword specifies that the output is in variable sequence and that all commands that apply to a single variable are grouped together. e.g.

```
VAR LABEL       NAME                             'Name'
MISSING VALUES  NAME                             (BLANK)

VAR LABEL       GENDER                           'Gender'
VAR RANGES      GENDER                           (1 2)
MISSING VALUES  GENDER                           (BLANK)
VALUE LABELS    GENDER                           (1)'Male'
                                                 (2)'Female'

VAR LABEL       MARSTAT                          'Marital status'
VAR RANGES      MARSTAT                          (1 2)
MISSING VALUES  MARSTAT                          (BLANK)
VALUE LABELS    MARSTAT                          (1)'Married'
                                                 (2)'Not married'
```

Examples:
To export the entire database:

```
EXPORT FILENAME = 'COMPANY.EXP'
```

To export record types 5, 6, and 8 of the first 1000 cases:

```
EXPORT  FILENAME= 'SUBSET.EXP'
        RECTYPES= 5 6 8
        COUNT= 1000
```

The export procedure writes out a number of messages. These note the beginning and end of various stages of the export (Begin export of schema/Export of schema complete, Begin export of procedures/etc.).

Export writes a summary of the data records exported. This lists the number of cases, each record type exported and the number exported.

SIR SUBSET

```
SIR SUBSET FILENAME = filename
        [ BOOLEAN  = (logical expression)]
        [ LIST     = caseid list]
        [ RECTYPES = rectype [(logical expression)] ...]
        [ SAMPLE   = fraction [,seed]]
        [ COUNT    = total [,increment [,start]]]
        [ DATABASE = newdbname]
```

Creates a subset of a database. The subset file is a binary file in identical format to an unload. The schema and procedures are written in their entirety. Only the data that meets the criteria is subset. Database administrator security is required to use this utility.

`FILENAME`

Specifies the name of the output file. If this file already exists, the new subset is appended to the end of the file provided that the file is a valid unload/subset file for this database. If the file has multiple database copies, use the `ITEMIZE FILE` utility to determine the copies that are there.

`BOOLEAN`

Specifies conditions based on the values of common variables. `BOOLEAN` is applied after `SAMPLE` and `COUNT`. For example:

```
SIR SUBSET FILENAME = 'SUBSET.UNL'
            BOOLEAN =  (ID GT 5)
```

`LIST`

Subsets the specified cases. Enclose case ids that are character strings in quotes. Cannot be used with `SAMPLE` or `COUNT`. For example:

```
SIR SUBSET FILENAME = 'SUBSET.UNL'
    LIST= 1,3,5 thru 10
```

`RECTYPES`

Specifies the set of record types to copy. Specify a logical expression to select on data values. The expression may use common variables and variables in the rectype specified. For example:

```
SIR SUBSET FILENAME = 'SUBSET.UNL'
    RECTYPES = 1 (SALARY GT 2000),3
```

`SAMPLE`

Selects a random sample of cases from the database. Fraction specifies the sample size for selection. Seed specifies the starting seed for the random number generator. If seed is not specified, the default is used. Cannot be used with `COUNT` or `LIST`.

`COUNT`

Selects a specified number of cases from the database. Total specifies the number of cases to retrieve. Increment specifies the "skipping factor" for retrieving cases. For example, an increment of 3 produces every third case. Start specifies the ordinal of the first case processed. For example, a start value of 3 begins the processing at the third case. Cannot be used with `SAMPLE` or `LIST`

`DATABASE`

Specifies the name of the new subset database. The subset database password is the same as the password for the original database. For example:

```
SIR SUBSET FILENAME = 'SUBSET.UNL'
    DATABASE = TESTDBMS
```

UNLOAD FILE

```
UNLOAD FILE  FILENAME = filename
           [JOURNAL  = KEEP | PURGE]
           [NEWDB    = newname]
           [NEWPW    = newpassword]
```

Creates a machine dependent copy of the database. `UNLOAD` is used for backup and restructuring. Database administrator security is required to use this utility.

Use the `UNLOAD FILE` utility to back up the database. The old journal file can be deleted once an unload file is produced. A database may be recovered from an unload file plus any journals from the point the unload was done. Make sure that there is either a journal file that covers the entire history of the database, or an unload file and a journal file that covers modifications made to the database since the unload. The suggested procedure is:

- Always have journaling `ON` for the database.
- Backup the database with `UNLOAD FILE` on a regular basis.
- It is good practice to run a complete `VERIFY FILE` before doing an unload)
- After a successful backup has been made, copy the backup file to secure external media.

At this point, previous unload files and journal files can be renamed or deleted.

The options and keywords are:

`FILENAME`
Specifies the name of the output file. If the output file already exists as an unload file for this database, the utility **adds** the latest output to the end of the file. Use `ITEMIZE FILE` to see what is on the output file. If multiple copies of a database are on one physical file, specify the file number or update level to restore the correct copy of the database. For example:

```
UNLOAD FILE FILENAME = 'COMPANY.UNL'
```

`JOURNAL`
`KEEP` is the default and specifies that the current journal file is retained.
`PURGE` specifies that the current journal file is deleted when the unload run is completed. Journaling then starts on a new file.

`NEWDB`

Specifies a database name for the database copy. By default, the name of the database is used.

`NEWPW`

Specifies a new database password for the database copy. By default, the current password is used. For example

```
UNLOAD FILE FILENAME  = 'COMPANY.UNL'
            NEWDB  = TESTDBMS
            NEWPW  = TESTPASS
```

UPLOAD

```
UPLOAD FILENAME= filename
     [JOURNAL  = filename]
     [UPDATE   = update level [THRU update_level]]
     [RECTYPES = ALL | rectype (variable_list), ...]
     [TITLE    = 'upload_file_title']
```

Reads a journal file and outputs all the journaled changes to a file. This file is a text file so that it can be transferred to another machine. The `DOWNLOAD` utility reads the file produced by `UPLOAD` and applies the changes to the new database. Database administrator security is required to use this utility.

`FILENAME`
Specifies the name of the output file. This is required.
`JOURNAL`
Specifies the journal file. If the journal file has a different name, specify the name used. The current journal file, (database file 5), is the default.
`UPDATE`
Specifies update levels or date/time stamps to upload from the journal file. Specify a specific update level, update date, a range of update levels or a range of update dates. The default is the most recent, single set of updates on the journal file.

If this is specified, a report is produced showing each update level that is written to the upload file. For example:

```
UPLOAD  FILENAME = 'JOURNAL.UPL'
        UPDATE= 10 THRU 30
```

`RECTYPES`
Selects rectypes to upload. A variable list specifies individual variables. If the variable list is omitted, all variables are processed. The keyword `CIR` selects the common information record variables. `ALL` selects all record types, including CIR and is the default. For example:

```
UPLOAD  FILENAME = 'JOURNAL.UPL'
        RECTYPES= 1,3
```

`TITLE`

Specifies the title of the upload file. This is written as the first line of the file and is used to identify the file. `DOWNLOAD` prints this title in the summary report. This title may be up to 45 characters and is enclosed in quotes. For example:

```
UPLOAD FILENAME = 'JOURNAL.UPL'
       TITLE= 'Department 3 Changes'
```

ITEMIZE FILE

```
ITEMIZE FILE   [FILENAME= fileid]
```

Reports on the contents of a unload and journal files. An unload file may contain multiple unloads taken at different update levels. A journal file typically has journals from multiple update levels. This information is necessary when restoring a database or applying journals.

The options on the command are:

`FILENAME`
Specifies the name of the binary file. The default is the journal (fifth database file).

The report produced is similar to the following:

```
Itemize File 'C:\sir2004\alpha\COMPANY.sr5' is a JOURNAL file for
database COMPANY
Update level:    1 -    2  Dec 08, 2005/10:46:13 Journal data
to Dec 08, 2005/10:47:07
Update level:    2 -    3  Dec 08, 2005/10:48:07 Journal data
to Dec 08, 2005/10:49:03

Itemize File 'C:\sir2004\alpha\COMPANY.unl' is an UNLOAD file for
database COMPANY
Update level:           2  Dec 02, 2005/13:08:17 Unload schema
Record:     1
Update level:           2  Dec 02, 2005/13:08:17 Unload data
Record:     2
Update level:           3  Dec 08, 2005/10:49:03 Unload schema
Record:     3
Update level:           3  Dec 08, 2005/10:49:03 Unload data
Record:     4
```

The information reported is the name of the file, the type of file and the database that the file refers to. This is then followed by a list of the records on the file. Each entry has the following information:

UPDATE LEVEL

The update level is a sequential number incremented each time the database is updated. On a journal, it is the update level from - to where these are always one different and the 'to' is the update level that resulted after the update run. Journals are expected to be contiguous and a warning is given if any update levels are missing. On an Unload, it is the current update level at the time of the unload.

DATE & TIME

The date that the update was done followed by the time the update was done. On an unload, this is *not* when the unload was done but rather the date and time of the last update that resulted in that update level on the database.

TYPE OF RECORD

The record may by a journal of a schema or a data update or may be an unload for schema or for data.

Record Number

Each record on the journal or unload is assigned a sequential number. When specifying processing on the file, the unload/journal to be processed can be selected with either the update level or the record number.

LIST STATS

```
LIST STATS
```

Provides a status report about the database similar to the following:

```
Statistics for  COMPANY
Database name                       COMPANY

Creation Date/Time                  Dec 06, 2005      10:46:12
Last update Date/Time               Dec 08, 2005      10:49:03
Update level                        3

Average Records per Case            1023
Max/Current Number of Cases         1000/20
Max/Current Number of Records       1023000/114

Max/Current Number of Record Types  30/3
Maximum Input Columns/Lines         80/1
Rectype Column                      5
Journal For Database                ON
Case Id Variable                    ID
(A)

Number of Index Levels              2
Max Entries Per Index Block         509
Index/Data Block Size               1019/1019
Active/Inactive Data Blocks         2/0
Active/Inactive Index Blocks        2/0

Keysize In Bytes                    8
Min/Max Record Size                 1/8
Number of Temporary Variables       0
Maximum Number of Data Variables    10

Record Record                         Number of  Maximum
Total In  Size In  Entry Use
 No.   Name                           Variables  Per Case
Database  Words     Count
----   ------------------------------  ---------  --------  ---
-----  -------  ---------
   0   CIR                                    1         1
20        5          1
   1   EMPLOYEE                              10         1
20        8          1
   2   OCCUP                                  4       100
30        1          1
```

```
   3   REVIEW                                    5       100
64        2          1

Secondary Indexes
Index Name                              Record
Variables
-------------------------------         ------------------------
------- ------------------------------
NAME                                    EMPLOYEE
NAME ASC
BIRTHDAY                                EMPLOYEE
BIRTHDAY ASC
EDUC                                    EMPLOYEE
EDUC ASC

GENDER ASC
EDUCID                                  EMPLOYEE
EDUC ASC

GENDER ASC

ID ASC
```

The information includes:

A) Overall Database Information

- database name
- current update level
- date of creation, last update and last access
- the current and maximum number of cases
- the current and maximum number of records
- the maximum key size
- the minimum and maximum record size
- the total number of common variables
- size of the CIR

B) Information about each Record Type

- number of variables
- maximum records per case
- total currently in the database
- length of the record
- number of times the record schema has been defined

C) Restructure Information (if any)

- number of original variables
- number of restructured variables

- update level for restructured records

D) Secondary index information (if any)

- name of each index
- record indexed
- variables indexed and whether Ascending/Descending and if Upper case

JOURNAL RESTORE

```
JOURNAL RESTORE   [FILENAME = fileid]
[FROM = n]
[THRU = n] | [COUNT = n]
[NEXT]
```

Applies journal files to a database to update it to a more current level. Any schema changes are applied as well as updates to the data. The process expects that the database has been recovered from a backup and, by default, looks for journal records that correspond to updates starting at the current update level on the database. It then applies all journals forward from that point to arrive at the most up to date database possible from that journal.

Update level information may be obtained by `LIST STATS` and `ITEMIZE FILE`. The update level listed for journals is the level the database was at *after* the update was originally done. So, for example, if the restored database is at level 40, the first journal to be applied would be update level 41.

`JOURNAL RESTORE` can restore partial journal records from abrupt interruptions of journaled update sessions. If a premature End-of-Record condition is encountered, the database is restored to a useable (non-corrupt) state, with as much data intact as possible. However if a logical set of updates were being done and were interrupted, data may be in an inconsistent state between records. It is recommended that a `VERIFY FILE` is done after a journal has been restored.

The options on the command are:

`FILENAME`
Specifies the name of the file that contains the journal. The default journal file is database file .sr5. For example:

```
JOURNAL RESTORE  FILENAME = 'COMPANY.JNL'
```

`FROM`
Specifies that when journal entries are applied, instead of starting from the current database level, they start from the specified level. This may be higher or lower than the current database level. Specify the starting update level which is one less than the first journal to be applied.

`NEXT`
Specifies that one journal entry is applied to the database to take it to the next update level.
`THRU`
Specifies that journal entries going up to and including the one at update level "n" are applied to the database. `UPDATE` is a synonym. For example:

```
JOURNAL RESTORE  FILENAME = 'COMPANY.JNL'
        THRU = 42
```

`COUNT`
Specifies that journal entries on the file from the start including the "nth" specified on the count are applied to the database. This is an alternative to specifying update level, which is the recommended approach. Do not specify both options. For example:

```
JOURNAL RESTORE  FILENAME = 'COMPANY.JNL'
        COUNT = 10
```

JOURNAL ROLLBACK

```
JOURNAL ROLLBACK   [FILENAME = fileid]    [UPDATE = n]  [COUNT =
n]
```

Applies journal files to a database to undo updates and roll it back to a previous level. Only applies to data updates.

If a database update run is interrupted, this might be used to roll back to a known update level before re-running the update process

`JOURNAL ROLLBACK` can restore partial journal records from abrupt interruptions of journaled update sessions. It is recommended that a `VERIFY FILE` is done after a journal has been rolled back.

The options on the command are:

`FILENAME`

Specifies the name of the file that contains the journal. The default journal file is database file .sr5. For example:

```
JOURNAL ROLLBACK FILENAME = 'COMPANY.JNL'
```

`UPDATE`

Specifies that the database is rolled back to this update level. If no update level is specified, it is expected that the database update run was interrupted and that the update level was not changed. This means that only journal records that were created as part of the last, interrupted run are rolled back. The database remains at its current update level and, after the rollback, should be in the same state as when the aborted run started. This would normally be what was wanted. Update level information may be obtained by `LIST STATS` and `ITEMIZE FILE`.

`COUNT`

Specifies that all journal entries on the file, starting at the last and including the "nth" specified on the count, are rolled back and so 'undone'. This is an alternative to specifying update level, which is the recommended approach. Do not specify both options.

VERIFY FILE

```
VERIFY FILE [ALL]
       [CIRKEY]
       [CIRDATA]
       [CHECK]
       [CCF]
       [RECKEY]
       [RECDATA]
       [RCF]
       [COUNT= total,increment,start]
       [PATCH]
```

`VERIFY FILE` examines the database files for damage and corrects errors where possible. DBA-level security clearance is needed if any keywords are specified, since potentially secure data might be revealed.

The corruption flag is set when any errors are detected in the database. It is cleared when the database is verified and found to contain no errors.

The keywords control the amount of checking and the amount of output generated when verifying each data record. The error message number is followed by a character that signals the type of error message: I for Informative, N for Non-correctable, C for Correctable, F for Fatal. The loading factors are printed with 2 decimal digits. Errors are listed by type with informative messages as appropriate.

`ALL`
Selects all the options. Use this option carefully since the output generated is voluminous. (Not an option on the menu system but equivalent to selecting all options.)
`CIRKEY`
Lists the values of all fields in the CIR record key.
`CIRDATA`
Lists the values of CIR variables.
`CHECK`
Checks the value of each variable against its schema specified criteria. Diagnostic messages are generated when bad values are encountered.
`CCF`
Clears the corruption flag. Use this option carefully; clearing the flag may mean that the problem resurfaces in the future after more work has been done and recovery may be difficult.
`RECKEY`

Lists the values of all fields in a record key.
`RECDATA`
Lists the values of all record variables.
`RCF`
Lists the record count fields from the CIR. These counts are the number of data records of each type that belong to each case.
`COUNT`
Retrieves a subset of cases from the database. There are three values:
`Total`
specifies the number of cases to retrieve.
`Increment`
specifies the "skipping factor" for retrieving cases. e.g. 3 checks every third case.
`Start`
specifies the ordinal of the first case to process. eg. 3 starts on the third case.
`PATCH`
Repairs all repairable problems. Run `VERIFY FILE` again to verify the patched database to clear the corruption flag if no errors are detected.

VERIFY FILE Error Codes

The following error messages show the types of problems detected. Most of these errors are "major", and if any of them occur, the data file is probably unsaveable. (For an explanation of the structure of the database, what is a PRU, etc., please see Tuning and Efficiency.)

01 index pru out of range.
1 index level
2 index pru that is out of range
3 index pru in error
02 index entry count error. Printed if the header contains an illegal entry count.
1 index level
2 index entry found
3 index pru in error
03 index entry count mismatch. Printed if total records below count does not match upper level index count.
1 index level
2 index count calculated
3 index count in upper level
4 index pru in error
04 db pru out of range.
1 illegal pru ordinal
2 db pru in error
05 db entry count mismatch. Printed if the entry count in the header does not match the db.
1 db entry found
2 db entry in header
3 db pru in error

06 db size (words used) mismatch. Printed if the number of words used in the header does not match the db.
1 db used found
2 db used in header
3 db pru in error
07 CIR record count mismatch. Printed on completion of case, if the correct number of records for each record type is in error.
1 case data file ord:case ord (last case)
2 rectype
3 mismatch (- if too many, + if not enough)
08 CIR record count limit error. Printed if a record count field exceeds some limit.
1 case data file ord:case ord (last case)
2 rectype
3 count field in error
4 legal max record count for rectype
09 CIR record count exceeded. Printed for any record detected that exceeds the CIR record count field.
1 case data file ord:case ord (last case)
2 record ordinal
3 rectype
10 record locked. Informational only, not an error per se.
1 case data file ord:case ord (last case)
2 record ordinal
3 rectype
11 wrong length CIR.
1 case data file ord:case ord (last case)
2 incirn detected
3 incirn
4 db pru in error
12 wrong length data record.
1 case data file ord:case ord (last case)
2 record ordinal
3 recdrn detected
4 recdrn
5 db pru in error
13 illegal rectype encountered.
1 case data file ord:case ord (last case)
2 record ordinal
3 rectype detected
4 db pru in error
14 rectype record total mismatch. Printed if at end of run the number of records for a given rectype is in error
1 rectype
2 reccnt detected
3 reccnt

15 database record total mismatch. Printed if at end of run there is a record total mismatch
1 dinrec detected
2 dinrec
16 case total error. Printed if at end of run total number of cases found does not equal count.
1 dincas detected
2 dincas
17 Record or Case limit exceeded.
1 record of case limit reached
2 master index overflow
3 data file is full
18, 19 used ind block error. Printed if number of ind or db blocks read is in error
1 number detected
2 number should be
20 data error: missing error
21 data error: range error
22 data error: catint error
23 data error: valid error
24 Master Index is Full. If this occurs, the capacity of the database has been reached. Possible solutions are to increase PRU size or to decrease maximum key size.
25 index key out of order. Current key in an index block is not greater than the previous key
1 index block pru
2 index block level
3 case data file ord:case ord (last case)
4 record ordinal
26 data key out of order. Current key in the data block is not greater than the previous key
1 data block pru
2 case data file ord:case ord (last case)
3 record ordinal
27 non-matching index block keys. First key in index block does not match key in higher level block pointing to it
1 index block pru
2 index block level
3 case data file ord:case ord (last case)
4 record ordinal
28 non-matching data index block keys. First key in data block does not match key in index block pointing to it
1 data block pru
2 case data file ord:case ord (last case)
3 record ordinal

29 overflow block has been used message. An overflow block is reserved when the database is created. If the database requires more space and cannot obtain it, it uses the overflow block to attempt to maintain database integrity.
30 Missing CIR: case id changed but no CIR record.
1 case data file ord:case ord (last case)
2 rec ordinal
3 rectype

Secondary Index verification messages

Any secondary indexes on the database are verified. If there is something wrong, the following error messages may be produced. All of these are serious errors and you need to drop and rebuild the index:

***ERROR - couldn't read index PRU - unable to read an index block from disk.
***ERROR - Zero index PRU - should have a block number but have zero.
***ERROR - couldn't read data PRU - unable to read a data block from disk.
***ERROR - Zero data PRU - should have a block number but have zero.
***ERROR - index key mismatch - as the various index levels were processed, a mismatch on the key was found.
***ERROR - data key mismatch - at the bottom level the data block pointed to by the index did not match on key
***ERROR - index count mismatch - as the various index levels were processed, a mismatch on the counts was found ***ERROR - data count mismatch - at the bottom level the data block pointed to by the index did not match on counts
If one of these errors occurs, supplementary information is printed including:
LEVEL - The index level being processed
PRU - The block being referenced
ENTRIES - The number of entries
COUNT - The count of entries
CURRENT ENTRY - The entry being processed

DOWNLOAD

```
DOWNLOAD FILENAME=  filename
        [MESSAGES=  ON | OFF]
```

Reads the text file produced by `UPLOAD` from a journal and applies these changes to the database. Database administrator security is required to run this utility.

`FILENAME`

Specifies the name of the input file.

```
DOWNLOAD  FILENAME = 'JOURNAL.UPL'
```

`MESSAGES`

Specifies whether messages are issued. Messages include whether a record exists in the database that is marked as a new record on the upload file. By default, messages are off.

For example

```
DOWNLOAD FILENAME = 'JOURNAL.UPL' MESSAGES = ON
```

SIR MERGE

```
SIR MERGE FILENAME = input_file
          DATABASE = database [PASSWORD = password]
                              [SECURITY = read password]
          RECTYPES = ALL | source [:targetno,name] [(expression)]
         [BOOLEAN  = (log_expr)]
         [NODATA]
         [RENAME   = [source](source_list = target_list)]
         [UPDATE   = ADD | REPLACE]
```

Merges record types from a copy of one database (source) into an existing database (target) that is the database currently being used. The source is a binary file. The `FILENAME`, `DATABASE`, and `PASSWORD`, `SECURITY` clauses if required on this database, must appear before any other clauses. DBA write security for the target database is required to use this command. This utility is not available through the menu system.

If the record type is already defined in the target database schema, the source and target record type definitions must match exactly. If a new record type is being merged, the schema for the new record type is created containing everything from the source database schema definition except the `IF, COMPUTE, RECODE, ACCEPT REC IF` and `REJECT REC IF` statements.

If the target database is caseless, the case id and CIR's on the source database are ignored. A caseless source cannot be merged into a case structured target. (Use `SIR SAVE FILE` to create a case structured database from a caseless database.)

The options on this command are:

`FILENAME`
Specifies the name of the source binary file to merge.
`DATABASE`
Specifies the source database name.
`PASSWORD`
Specifies the source database password.
`SECURITY`

Specifies the read security password of the source database. The read password must be the DBA level password.

```
SIR MERGE  FILENAME  = 'COMPANY.UNL'
           DATABASE  = COMPANY
           PASSWORD  = COMPANY
           SECURITY  = HIGH
```

`RECTYPES`

Specifies the record types to merge. The CIR of the source database is merged if the variables in the target CIR match exactly.

`ALL`

Merges all source record types.

`source`

Merges the specified record types. The record type may be a name or number.

:targetno,name

Merges the source record types with the specified target record types. Do not leave any blanks between the colon : and the number. Specify both the number and name of the target record.

(expression)

Specifies a logical expression to select records. This can reference both common variables and record variables from the source record type(s).

If the `RENAME` clause is used, specify the new name of the variable in this clause.

`BOOLEAN`

Specifies a logical expression referencing common variables to select cases. If the expression is `TRUE` the case is merged.

`NODATA`

Specifies that no data is merged. The schema for the specified (new) record type(s) is added.

`RENAME`

Specifies new names for variables merged from the source record types. Use if the source and target records have different names for the same variable or to change a variable name from the source name when a new record type is being created. `RENAME` does not change variable names on existing target records. Specify the `RENAME=` rectype (source variable list = target variable list) form when more than one record type is being merged. The rectype is the source rectype:

```
SIR MERGE ... RENAME = 1 (EMPNAME = NAME)
```

`UPDATE`

Specifies the action to take when the record identifiers on the source record match those of a record in the target database.

`ADD`

Specifies that only new records are created. If a source record has a key that matches an existing record on the target database, the source record is rejected and a message is issued.

`REPLACE`

Specifies that records are only replaced. If a source record has a key that does not match a record on the target database, the source record is rejected and a message is issued.

By default, both new records are added to the database and existing records are replaced.

RELOAD FILE

```
RELOAD FILE  dbname
       FILENAME =  fileid
       [PASSWORD = password]
       [SECURITY = rsec,wsec]
       [UPDATE = n | FILE= n]
       [LOADING = n]
       [NOFCASES = n]
       [AVGREC = n]
       [RESTART]
```

Recreates a database. The input is a binary file that is a copy of a database.

The reload database name and password must be the name and password of the database on the unload file. To change database names and passwords, specify the new name and password on the `UNLOAD`.

Optionally separate multiple parameters on the command with slashes.

`FILENAME`
Specifies the name of the binary file that contains the input. If there is more than one copy of a database on the file (which happens if the database is `UNLOAD`ed to the same file more than once), specify `UPDATE= n` or `FILE= n` to reload a copy other than the first.

`PASSWORD`
Specifies the database password. Must match the password of the unloaded database.

`SECURITY`
Specifies the read and write security of the database. Specify an asterisk '*' for a null security password.

`UPDATE`
Specifies the update level to reload if there are multiple copies of the database on the unload file. `ITEMIZE FILE` reports the update levels of multiple database copies on a file.

`FILE`
Specifies the file number of the database to reload if there are multiple copies of the database on the unload file.

`LOADING`
Specifies the fraction of each disk block to fill with data.

`AVGREC`

Specifies a new value for `RECS PER CASE` in the Case Schema definition for a case-structured database. The specified value is the average number of records per case.

`NOFCASES`

Specifies a new value for `N OF CASES` in the Case Schema definition for a case-structured database. The specified value is an upper limit on the number of cases in the reloaded database.

`RESTART`

Resets the database update level to 1. This is done automatically when the update level on the reloaded database would be greater than 32268.

Example:

```
RELOAD FILE MYDBMS
       FILENAME = 'COMPANY.UNL'
       UPDATE = 52
```

Tabfiles and Tables

A **Table** is a Relational Table (or flat file) that is a number of occurrences (from 0 to n) of a single type of record that has a number of variables (or columns). For example, a `CUSTOMER` table might have all of the customers with customer number, name, address and credit limit as variables. The individual variables that make up a table are defined including the variables name, format, data type, missing values and value labels. Tables can be created, defined, populated, modified and retrieved from.

Tables are physically held in **Tabfiles**. A Tabfile is a physical file on disk that contains relational data tables, schema definitions for those tables, indexes to the tables and system tables. A tabfile is independent of all other tabfiles and is independent from any SIR/XS database. A tabfile is the largest unit that exists for security and access control. A tabfile can hold multiple tables. Before accessing a table, the appropriate tabfile must be connected.

Tables from multiple tabfiles can be accessed and retrieved by SQL, VisualPQL and FORMS.

A SIR/XS session may be connected to multiple tabfiles at the same time. A default tabfile can be defined and this tabfile is used whenever a tabfile name is not specified. Whenever tables are referenced, the tabfile can be specified explicitly or the default can be used. Tabfiles can only be updated by one user at one time.

An **Index** is a way of accessing a table using the values of a particular variable as the key. Indexes can be defined on any variable or combination of variables. An index can be defined as only allowing unique values (for example Customer Number) or can have multiple entries for records all with the same value (for example Last Name). Indexes can be used to process tables randomly or in index sequence. If a table is processed without an index, it is retrieved sequentially in the order in which it was created. Once an index is defined, it is built from any existing data and is automatically maintained as the table is updated.

Tabfiles, tables and indexes may be defined in a number of ways using SQL, the VisualPQL procedure `SAVE TABLE` or the menus. In addition, there are specific SIR/XS commands that deal with tabfiles and tables. These are:

- `CONNECT TABFILE`
- `CREATE TABFILE`
- `CREATE INDEX`

- `VERIFY TABFILE`

CONNECT TABFILE

```
CONNECT TABFILE tabfile  [ON filename]
```

Connects the specified tabfile. A tabfile must be connected before it can be used. A pre-compiled VisualPQL program can connect a tabfile when it runs, but, if you need to compile a VisualPQL program that references a tabfile, the tabfile must be connected first.

The `ON` clause identifies the physical file where the name of the physical file is not the internal tabfile name plus `.tbf`.

CREATE TABFILE

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

Creates a tabfile. The tabfile name is the name used in all other commands. This name is stored on the physical file and is the same name used to `CONNECT` to this file in subsequent sessions. A tabfile is automatically connected when created.

`FILENAME`

A filename for the tabfile. If this is not specified, the filename is created from the tabfile name plus a suffix of `.tbf` and this must be a valid filename on your operating system.

`JOURNAL filename`

Specifies that journaling is turned on and names the operating system file to use. If the journal file is not there when the tabfile is updated, a new journal is created. If the journal is there, new journal data is added to the end of the file.

`IDENT BY`

Creates the initial security definitions for access to the tabfile.

Group name and password

Specify a group name who has DBA permission for the tabfile. If this is not specified, the tabfile is created with no security; this cannot be changed and no security can be assigned to any individual table on that tabfile. Optionally specify a group password

User name and password

Further restricts DBA access to a second level of name and optional password.

`BLOCKS n`

Specifies the number of blocks to create a physical block. In general do not specify this as the default is adequate. The default of 1 gives an actual block size of 2k bytes. A specification of 2 gives 4k bytes and so on. The number must be a positive integer. A block must be able to hold the largest physical record.

CREATE INDEX

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

Creates an index for a table. An index provides direct access to a subset of records.

`ON`
Select the tabfile and table to create the index on.
`index name`
The name used to refer to the index. Must be unique on this table.
`UNIQUE`
Specifies that two rows cannot have the same index value. Rows with a value the same as an existing row are rejected. If an index is created for a table, and existing rows contain identical key values, then the index is not built and an error message is issued.
Columns
Specifies the column(s) comprising the index in major to minor sequence. For example: if (Sex, Name) is the index, this retrieves all Males by name, then all females by name. If (Name, Sex) is the index, everyone with the same name is retrieved together.
`ASC | DESC` specifies `A`scending or `D`escending sequence for a particular variable. Ascending is the default.
`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 very static and the index is not updated, or is updated sequentially, specify a low figure.

Examples:

```
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)
```

VERIFY TABFILE

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

Checks all of the tables on the specified tabfile. If a table or tables are corrupt, `VERIFY` issues a notice of the affected tables and prompts on whether to purge the corrupted tables.

If a tabfile is corrupt, you may have difficulty `CONNECT`ing to it to verify it. If you have DBA permissions, `CONNECT` to a corrupt table by specifying `READ` access only.

The `ON` clause is used to identify a the physical file where the name of the physical file is not the internal tabfile name plus `.tbf`.

Tuning and Efficiency

The information in this topic covers the way SIR/XS manages data internally. You do not need to know this to use SIR/XS successfully or to implement SIR/XS systems. It is here if required for database design on large or complex systems, or tuning particular applications.

Efficiency in an application is difficult to achieve by tuning after the system is developed. If efficiency is a concern, the best time to consider these issues is at the design stage. The first thing to determine is whether efficiency is a major concern and to identify possible areas where these concerns may arise. For example:

- Data Entry.
  Is data entry interactive ?
  How many people are going to require simultaneous access ?
  What sort of turnaround is required ?
  What response times are necessary ?
- Regular Batch Processes.
  Are there major processing tasks that must happen every day, week, month ?
  Are there reports that must be done very frequently ?
  Are there reports where the results must be available immediately ?
- On-line Queries.
  How is the data accessed ?
  Are inquiries always on particular records or are sets of records retrieved that match given conditions ?
  How up to date does the data need to be ?
- Disk Space.
  Is the amount of data to hold relatively trivial for this size machine or is it going to have a major impact on disk storage ?
  Can disk storage be ignored for all practical purposes or do you have to have to make the most efficient use possible ?
- Recovery.
  How is data going to be protected in case of problems ?
  What concurrent updates are going to happen ?
  What journaling strategy is appropriate ?

Disk Space

The amount of space that a database occupies on the disk can be a concern with larger databases. There are often trade-offs between processing efficiency and storage and there are several things that can be done to limit the size of databases.

Database Subsets

All of the data may not be needed on-line. There are utilities that create a subset of a database. There are utilities that merge subsets into the master database. For example, if an application normally only deals with data from the current year, archive the data for previous years and conserve disk space. If the old information is needed for year end reports, reload it, use it and then archive it again.

CIR Size

The common information record or CIR on a case structured database occurs once for each case and holds both record counts and common data.

Record Counts

The CIR holds space to count occurrences of each possible record type on the case. Allowing for large numbers of record types means a large CIR. In particular, it is very wasteful to allow a large number of record types (`MAX REC TYPES`) with the intention of using very few e.g. do not define a `MAX REC TYPES 1000` just to use a few record types in various ranges (100+, 200+, 300+) to mean something. For maximum efficiency, start record types at 1 and assigned numbers sequentially.

The `MAX REC COUNT` affects the size of each counter (1, 2 or 4 bytes). So a `MAX REC TYPES` of 1,000 and a `MAX REC TYPES` of 1,000,000 would mean 4K of record counts per CIR.

Common Vars

Defining a variable as a common var means that it is physically stored in the CIR, not in a record. Very often, there is only one occurrence of a common var in a case, so storage is identical whether held as a common or a record variable. If a common var is defined in a record that does occur multiple times, it is only physically stored once, this value being the latest value written.

Common vars are retrieved very efficiently regardless of a particular record type that is being processed. (Note that common vars cannot be used as keys in secondary indexes as they are not physically part of the record.)

The Loading Factor

Records are stored in "data blocks", the exact size of a data block varying from database to database. Records are added to a data block, and when that gets filled up, another block is created and so on. Records are maintained in sequence within a block and some empty space is left on each data block for the insertion of new records. The amount of space on each data block is controlled by the **loading factor** and is expressed as a decimal representing a percentage. The default loading factor on updates is .5 or 50 percent.

Using the default, when a data block fills up, it is split, with half of the records staying in the original block and half going to the new block. Fifty percent is a good figure for active databases. It means however, that as much as fifty percent of the data file may be empty. This may be unacceptable on large databases and on relatively static databases.

The amount of free space is controlled with the loading factor clause on database updating and database creating commands and utilities.

When a database is reloaded or imported, the data is in sequence and the default loading factor is set to .99 to make maximum use of disk space.

The Database Index

The index to the data records in a database is built from the key variables. The key of the first record in each data block is in the index. This means that there is some redundancy between data in the records and data in the index.

The larger the size of the keys, the larger the index. The maximum size keys in any record in a database affects the size of the index. If an application has one record type with a much longer key than all others, try to reduce this if possible. For example, do not have one record type indexed on a 60 character name, if all other record types in the database have unique numbers.

The maximum possible size of the database key or any secondary index is 320 characters.

In a series of record types that share higher level keyfields, each of these records store much of the same key information. It is therefore sensible to minimise the size of these keys.

For example, a text retrieval system might use words to index documents. However a word can be very long and storing these as keys for documents is wasteful of space. Assign each word a number, such that the text of the word is only stored once and all other key indexing is through the word number.

Variable Sizes

It takes more space to store strings than integers. Whenever there is a choice, storing a number is more space efficient. If a string has a defined set of values, either define the variable as an integer and assign value labels or define it as a categorical variable.

SIR/XS compresses string variables by stripping trailing blanks to hold only the data. Specifying a long maximum length for string variable incurs little overhead provided it is not used in any keys or secondary indexes. Note that the maximum record size is limited to 32k bytes and is tested assuming all strings are at maximum defined length.

The size of integers depends on the maximum value. One byte holds integers in the range of -127 through +123; two bytes holds integers in the range -127*256 through +127*256 (approximately 32,000), four bytes holds integers in the range -127*256*256*256 through +127*256*256*256 (approximately 2,100,000,000).

SIR/XS stores an actual value in a data field to indicate missing values. A variable can have four possible missing values. SIR/XS uses the upper four values on integers for the three missing values that can be specified and the system missing value `UNDEFINED.`

Real numbers are stored in 4 or 8 bytes.

Schema Specifications

SIR/XS assigns internal formats according to the external format of the data defined in the schema. Disk space can be saved by a careful choice of schema specifications.

For example, a variable with an input format of "I3" requires 2 bytes of storage because any value between -99 and 999 in the input field can be input. If this field contains a 2-digit variable with a leading blank or plus sign (+), specify the format as "1X,I2". This saves one byte of storage space in each data record containing this variable since 2-digit variables are stored in a single byte.

VAR RANGES

Specify `VAR RANGES` if the variable has a narrower range of values than given by the number of digits. The value is used to calculate the minimum number of bytes needed to store the data on disk. For example, specifying a `VAR RANGE` of -99 to +99 on a variable where 3 input columns are allowed saves space. Consider a potential saving of disk space by defining a proper `VAR RANGE`.

CATEGORICAL

Categorical variables offer an efficient way to store strings that are predefined. A categorical variable is a character string that has a limited number of values specified as an ordered list. When the data is input as a string, it is compared to the list and the number that corresponds to the matching position in the list is stored instead of the value. This has the advantage that only valid entries are held and considerable space is saved. In programs and reports, the full string is displayed and retrieved.

For example, a categorical variable might be a list of the names of American states. If 'Alabama' were the first entry in the list, when 'Alabama' is input, '1' is stored.

The list is held in the data dictionary and is searched sequentially. It is a very simple and easy to use facility for short lists that are not updated very often. A categorical variable takes one byte (for up to 123 values) or two bytes for longer lists.

Do not use categorical variables if there are hundreds or thousands of entries, or there is more information about each entry than just the name, or users have to modify the entries, use tables with indexes to store this type of reference data.

SCALED VARS

`SCALED VARS` stores numbers as integers when they have a predefined number of decimal places. This is more efficient than using floating point R*8 and can be more accurate than R*4.

For example, suppose a variable `XPCT` that holds a percentage and can have a range of 0 through 100 and a precision of one decimal point: Define `XPCT` as integer with an "I4" input format and specify `SCALED VARS XPCT (-1)`. On input, supply the data as a number that includes the physical decimal point, i.e. 10.3, 40.0. The `XPCT` scaled integer is only going to require two bytes to store (since the maximum physical digits stored are 1000 i.e. 100.0).

If the precision for the percentage example were 2 digits after the decimal point, specify an input format of I6 (nnn.nn) allowing for the decimal point and 5 numbers and specify `VAR RANGES (0.00,100.00)` that tells SIR/XS that 2 bytes are sufficient.

Even more storage may be saved with `SCALED VARS`, on numbers that are very small but have only a small number of significant digits. For example, the specific gravity of fluids in the human body (blood, urine, etc.), are often measured with a 3 digit precision. To maintain precision in floating point, specify an `INPUT FORMAT` of D4.3. SIR/XS would use 8 bytes of storage because of the precision. If this variable is read as "I2" integer and specify `SCALED VARS (-3)`, 6 bytes is saved per value and accuracy is preserved. (The variable can hold values up to 32.763 that is ample for an S.G. measurement.)

Processing Efficiency

Disk Input/Output (I/O) is the most time consuming operation on a computer and retrievals should be designed to minimise I/O.

Using Keys

The index is used to retrieve records whenever keys are specified in a retrieval statement. In case structured databases, the index is ordered by case, record type and by the key variables. In caseless databases, the index is ordered by record type and then key variables.

Whenever possible use the keys to retrieve records. To retrieve a single record, specify the whole key. To retrieve a set of records, specify the high level keys that define the set. Whenever possible, specify the keys as part of the retrieval statement, rather than retrieving all the records and testing values in the program.

Efficient On-Line Access

If the key values are known, then data can be retrieved efficiently. Without keys, an alternative access route is needed. Doing a serial search for particular records on-line, without knowing the high level keys is a slow process.

Define secondary indexes to provide access to subsets of records. Both databases and tables provide automatic secondary indexes.

Efficient Batch Processing

Batch processing (the running of jobs in a non-interactive way), typically means that a user is not at a terminal waiting for the job to finish. Processing speed tends not to be of the same concern as it is for on-line access. A process that takes 2 minutes as compared to 1 minute is unlikely to be of concern to anyone. However there may be some concerns when processing thousands of transactions that run for hours.

Consider sorting the input transactions to ensure that any serial processing happens only once.

Consider adding additional indexes or keys to avoid an application having to do serial searches of records to find those of interest,

One common design issue involves processing records after a certain amount of time has elapsed. For example, sending a letter to all patients who have not attended for six months. Consider a secondary index by date for planned future attendances. Update this at the time the visit data is updated and then the system can process by date rather than serially searching.

Efficiency in Batch Data Input

Batch data input is the loading of data from files into the database through the batch data input utilities. This can be done interactively or in batch mode.

For the most efficient processing, sort the data for a batch data input run into the same sequence as the data base key. Sort on :

- the CASE ID
- the record number
- the key fields

This way, the batch data input can be accomplished by an almost sequential processing of the data base.

It is efficient to process the records by record type. Each time a new record type is processed, the description of this record type must be loaded from the dictionary. If different record types are processed together and there are multiple records for each case, this saves accesses to the case block but requires multiple access to the dictionary. The most efficient processing depends on the exact mix of input.

Database Internal Structure

The records in a SIR/XS database are stored in a direct access file with an internal index sequential B-Tree index. The database contains two types of blocks: data blocks and index blocks. Data blocks contain the data records, index blocks contain the information needed to access any record in the data base. Blocks can be in any sequence on the disk. Within one block, records are held in sequence. The first record in each block is indexed.

When a record is added, it is stored on the correct block in sequence. This means that a block can get full. If this happens SIR/XS creates a second block to store the additional data, and creates another entry in the index. New blocks are created as necessary. A new block is either allocated from existing available blocks or from new blocks at the end of the file. Blocks become available if the data on them is deleted.

SIR/XS holds the case id, record number and key fields as the key. All keys are the same length, which is either the maximum length of a defined key or the `MAX KEY SIZE` specified. Pay attention to the size of the key. A key is held for each data block in the index and the key is held for each record in each data block. Defining a very large key for one record type impacts the overall database size, regardless of the number of occurrences of that record type.

At the lowest level, an index consists of a key and a pointer to the data block that has that key as the lowest value. At the highest level there is a single Master Index block. This contains a key and a pointer to the index block that has that key as the lowest value. If necessary, because of the size of a database, there may be further index levels between the Master index and the lowest level index. When a key is specified, SIR/XS uses the master index (and any other index levels) that point to the lowest level index block that corresponds to the value given and retrieves that data block.

Block Organisation

SIR/XS calculates the size of data blocks and index blocks for a particular database based on keysize, maximum record size and maximum numbers of records when it first puts any data into the database. The block size is between a minimum and maximum (from 2K bytes to 32K bytes on all current systems). The data blocks and index blocks in a database may be different sizes though in a particular database all data blocks are the same size and all index blocks are the same size.

When a block is created on disk, it is assigned a number known as the PRU or physical record unit that can then be used to retrieve the block directly. In operating system terms, a SIR block consists of multiple physical disk blocks since most operating systems write in fixed blocks.

The `LIST STATS` command gives information about the database including the '`INDEX/DATA BLOCK SIZE`'. This gives the sizes of the SIR/XS index and data blocks. Sizes are given in double words - eight bytes on current systems. Sizes do not include the control information SIR/XS holds on each block. A logical block of 2K is 256 doublewords. A typical size for logical blocks for small keys and small data records would be 253/254.

Data Blocks

Records are stored in blocks in the order of the keys:

```
case 1    CIR of case 1
records of type 1 within case 1
records of type 2 within case 1
...
...
case 2    CIR of case 2
records of type 1 within case 2
records of type 2 within case 2
...
...
case 3    CIR of case 3
...
```

SIR/XS holds all of the records in a data block in sequence and to do this it constructs an extra key area at the beginning of each record and holds keys there separately from the data. All record key areas are the same length, which is the same as the keys held for the index.

Data block size depends on the size of records defined. If there are not any very long records, SIR/XS uses one block (i.e. 2K). SIR/XS tries to allocate a block size that is big enough to hold 4 of the largest records. The largest block size is 32K. If the maximum record length is between 512 bytes and 8k bytes, then SIR/XS allocates a block size between 2K and 32K. A data record is held in one block. That is, a record is not split across blocks so the maximum size for a single record type is 32K.

Loading Factor

When SIR/XS needs to insert data that does not fit in the original data blocks, it creates a new block and splits the original data leaving some space on each block. The amount of space left on a block when it is split is determined by the "Loading Factor".

A loading factor can be specified on a retrieval update, a batch data input or on a utility update run such as RELOAD. This affects the way a full block is split. The factor is a number between 0 and 1 and the default is .5 on updates and .99 on imports and reloads. The most efficient database is one where each block is loaded to the maximum since this minimises the amount of disk space used and makes retrievals more efficient by reducing the number of disk I/Os. However, a high loading factor for existing blocks can affect the ways that updates work. To take some examples:

Example Loading Factor Effects

If a loading factor of .99 is specified on a `RELOAD`, then all the blocks are approximately full. Suppose that Batch Data Input is then used to add a large amount of data at the end of the database, say with an .99% loading factor. Again all blocks are approximately full. (Blocks have to hold whole records and each record is a different length. So when a record does not fit into a block a certain amount of space is left free. This space varies from block to block.)

Now suppose that a Retrieval Update adds records randomly using a loading factor of .8. At some point a block becomes full. The record being added at that time is inserted in the correct place and 20% of the space on that block is made available by copying those records to a new block and entering that into the index. If the original block is added to further with data that belongs in that block (i.e. with a key lower than an existing record in that block or than the lowest key in the next block) then again it becomes full and again split with the next new block again taking 20%. Thus it is possible under some sequences of updates that many new blocks are only 20% full. If the loading factor were higher, the result would be even worse. Adding data in reverse key sequence with a high loading factor would produce very poor block usage.

The best loading factor depends on the nature of the activity at the time. In general, adding in sequence at the end of the database is best served by factors nearer to 1. Randomly adding data throughout the database is best served by having enough space available for the inserts to work without splitting blocks and, without specific knowledge as to the sequence of updates, a loading factor of .5 should be used.

SIR/XS uses .5 as a default for updates and .99 as a default for reloads and imports.

The actual, exact loading is reported by the `VERIFY FILE` command. The number reported gives an average over all blocks in the database. SIR/XS does not split records across data blocks and each block contains complete records only. A block contains a mix of records. For example an 80% full block in the `EMPLOYEE` database might contain data for 3 or 4 employees and as such might have say 4 CIRs, 4 Employee records, 7 Position records and 13 review records. Thus the exact loading of the block depends on the exact mix of records.

Index Blocks

The key is comprised of:

- the CASE ID
- the record type number (0 for the CIR)
- the Key Fields

Each key has the same length - its length is either defined implicitly in the schema or by the `MAX KEY SIZE` command.

The index holds the key of the lowest record in each data block. An index block holds 'n' entries depending on the size of the key. If normal size keys are specified, say up to about 80 bytes, SIR/XS uses the minimum 2K block size; after that SIR/XS increases the index block size. The index block size is always a multiple of the minimum size. If there are very large keys or a very high number of data blocks, SIR/XS increase the size of the index block to cope with this.

Index Levels

There are always at least two levels of index, a Master Index, which is a single index block, and a low level index. There may be up to 6 levels of index. A six level index can point to the number of keys in one index block raised to the power of 6. For example, with 36 keys in a block, a six level index copes with over 2,000,000,000 data blocks.

To illustrate the way index levels work, assume there are 80 keys per index block. One index block can point to 80 other blocks. If there are less than 80 blocks of data, then there are only two index blocks. The master index and one low level index block. The master index only has one entry. With 81 to 160 data blocks, there are three index blocks, the master index with two entries, one index block for the first 80 data blocks and the second for the next eighty blocks. This continues on until there are 80*80 data blocks, 80 index blocks and one master index block with 80 entries. When the next data block is added, one of the low level index blocks is used to create two new low level blocks. The original low level index block is now a third level index that contains just two entries pointing to the two new low level indexes. As records are added, indexes split as necessary. The third level takes the index capacity up to 80*80*80 data blocks. This process continues as necessary.

At no one point in time is there any major overhead or any need to reorganise the database assuming that none of the limits specified in the schema definition are reached.

Secondary Indexes

All secondary indexes are held on a separate database file (.sr6). This is created when the first index is created and deleted if the last index is deleted.

Each secondary index is physically very similar to a standard database. It contains index blocks and data blocks. The sizes of these blocks are calculated in a similar way to the block size calculations for standard database blocks to ensure reasonably efficient processing given the size of the secondary index key and the maximum number of records of that type. Each index potentially has different block sizes.

Each record in the data block in a secondary index has the secondary key as the key and contains the standard database key as the data. Thus the size of these data blocks is affected by the size of both keys.

Size Estimating

Once records have been added to the database, each physical data block contains a number of different record types. For size estimation, calculating the number of data blocks each record type would take gives a reasonable estimate of disk space requirements. In addition, space is required for the dictionary and procedures but typically these are relatively small requirements. See Procedure File for further details on managing a procedure file.

The following discussion refers to the `LIST STATS` output listed at the bottom of this page.

- From the `LIST STATS`, find the data block size in double words. For example 254.
- From the `LIST STATS` find the keysize in bytes. Convert this to double words by dividing by eight and rounding up. For example, keysize 12 is 2 double words.
- Take the first record type in the `LIST STATS`. Take the size of the record in words and add the keysize from step 2 to find the space this record type takes. For example, "Size in Words" 8 plus key size 2 equals 10.
- Take the data block size and divide by the length from step 3. This gives the number of that record type that fit in one data block. For example, record size 10, block size 254 gives 25 records per block.
- Divide the total number of records in the database by the number per block to find the number of blocks needed. For example 1000 records at 25 per block need 40 blocks.
- Repeat steps 3 thru 5 for each record type plus the CIR. (One CIR per case). For example:

  CIR Size = 5 words plus key 2 = 7.

  254 divide by 7 = 36 per block.

  1000 cases in database means 1000 CIRs.

  1000 divided by 36 means 28 blocks.

- Add up all the blocks needed for each record type to get the total data blocks required. For example,

  CIR = 28 blocks

  Rec1 = 40 blocks

Rec2 = 32 blocks

Total = 100 blocks

This shows how many data blocks the database requires and the record types that use the space.

- Apply the 'loading factor'. If using 50%, then twice as many data blocks are needed; if using 80%, then 20% more are needed. For example, 100 blocks at 80% require 120 blocks.

Next calculate the index size:

- Take the Max Entries per Index Level from the `LIST STATS`. For example, 126.
- Take the number of data blocks and divide by the Max Index Entries. This gives the number of lowest level index blocks. For example 120 blocks divided by index entries of 126 gives 1.
- If the number of index blocks so far is less than the Max Index Entries, then do not do this step. If the number is more than the Max Index Entries then divide the answer by the max index entries to get the number of second level index blocks. If this number is greater than the max index entries, divide again to get the third level index blocks. Repeat this until the answer is less than Max Index Entries. Add up all levels.
- Add 1 for the master level.

For example, if Max Index Entries is 40 and there are 100,000 data blocks.
Bottom Level = 100,000/40 = 2,500.

Second level = 2,500/40 = 63 (Rounding up).

Third level = 63/40 = 2.

Total required 2,500 + 63 + 2 + 1 = 2,566.

This gives how many index blocks are needed. To translate these two figures into physical disk blocks or megabytes on a particular operating system, multiply by the appropriate factors:

Take the data block size and index block size from `LIST STATS` in double words and convert to physical blocks or bytes. There is a small overhead on each physical block such that the reported size is smaller than the real physical size. For example, a data block of 254 double words is 2K (which is four 512 byte blocks on some Windows file systems). 1,000 254 double word data blocks would take approximately 2 megabytes of disk space.

This is how much space the data and indexes are going to take for a database.

- Sample VERIFY FILE output

```
Index PTBYNAME                       Verified - Entries 1852
Verify database statistics
---------------------------
Data records on database             60408
Cases on database                    1852
Index blocks read                    16
Data blocks read                     1467

Average index block loading          0.91
Average data block loading           0.97
Warning messages                     0
Correctable errors                   0
Non-correctable errors               0

Verification complete  with no errors
```

- Sample LIST STATS Output

```
Statistics for  HEART
Database name                        HEART

Creation Date/Time                   Dec 13, 2005      11:55:40
Last update Date/Time                Dec 13, 2005      11:57:15
Update level                         1

Average Records per Case             25000
Max/Current Number of Cases          75000/1852
Max/Current Number of Records        1875000000/60408

Max/Current Number of Record Types   117/85
Maximum Input Columns/Lines          136/30
Rectype Columns                      1-3
Journaling                           ON
Encryption                           ON
Case Id Variable                     SSNUM (A)

Number of Index Levels               2
Max Entries Per Index Block          101
Index/Data Block Size                1011/1529
Active/Inactive Data Blocks          1467/0
Active/Inactive Index Blocks         16/0

Keysize In Bytes                     72
Min/Max Record Size                  1/346
Number of Temporary Variables        0
Maximum Number of Data Variables     958
```

```
Record Record                           Number of  Maximum
Total In  Size In  Entry Use
 No.   Name                             Variables  Per Case
Database  Words     Count
----   -------------------------------  ---------  --------  ---
-----  -------  ---------
   0   CIR                                      3         1
1852       36          1
   1   DEMO                                    89         1
1852       96          1
   2   HOSP                                   292       100
2124      140          1
   3   CLINPRES                               958       100
1266      346          1
   4   CATH                                   530       100
399      213          1
......
```