# *Advanced CAE Applications for Professionals*

***Software that works — for you.***[SM]

# *eShell User's Manual*
# *Version 20.1*

**UNIVERSAL ANALYTICS, INC.**

**Publication EB-001**



*First Printing, November 1995*
*Second Printing, December 1997*
*Third Printing, February 1999*



**UNIVERSAL ANALYTICS, INC.**

**3625 Del Amo Blvd., Suite 370**
**Torrance, CA 90503**
**Tel: (310) 214-2922**
**FAX: (310) 214-3420**

# TABLE OF CONTENTS

## 1. INTRODUCTION

## 4. RETRIEVING DATA FROM RELATIONS



## 5. GRAPHING RETRIEVED DATA



## 6. INDEXING RELATIONAL ENTITIES

## 7. RETRIEVING DATA FROM NON-RELATIONAL ENTITIES



## 8. INSERTING DATA INTO ENTITIES



## 9. UPDATING ENTITY DATA



## 10. REMOVING DATA FROM eBase



## 11. FILE ENVIRONMENT COMMANDS



## 12. REPORT GENERATION

## 13. UTILITY FUNCTIONS

*This page is intentionally blank.*

# 1. INTRODUCTION

The Engineering Database Management System (***eBase***) developed by Universal Analytics, Inc. (UAI) provides an advanced data management facility for scientific software applications development. The design of the database allows its effective use in structuring and managing any large collection of engineering data. The Interactive ***eBase*** Shell (***eShell***) is a separate computer program that allows the design engineer to access, modify, and manage the information in an ***eBase*** database interactively. ***eShell*** accomplishes this by providing the user with a powerful query language called ***eQL*** (***eBase*** Query Language). The ***eQL*** language adheres to the ANSI Standard for Database Language - SQL X3.135-1986 as much as practical. This standard, which addresses relational databases, has been extended to accommodate the more powerful features of ***eBase***.

## THE eBase SOFTWARE SUITE

The suite of ***eBase*** products includes:

- ❒ The ***eBase*** Shell (***eShell***) and Interactive Query Language (***eQL***)
- ❒ ***eBase*:*applib***, the Application Development Library
- ❒ ***eBase*:*matlib***, the Matrix Utility Library

There are three manuals which document these products:

- ❒ This document, the ***eShell*** User's Manual (Publication EB-001)
- ❒ ***eBase*:*applib*** User's Manual (Publication EB-003)
- ❒ ***eBase*:*matlib*** User's Manual (Publication EB-004)

A fourth manual, called ***The Installation Guide and System Support Manual*** (various part numbers) provides information describing the

host-computer dependent characteristics of all of the UAI software products. ***eBase*:*applib*** is a run-time library of Fortran subroutines which is an advanced database tool for use in cost-effective scientific software development. ***eBase*:*matlib*** is a similar library of high performance matrix utility subroutines which have been optimized for many different computers. This library includes routines for matrix algebra, linear equation solving, eigenvalue extraction and others.

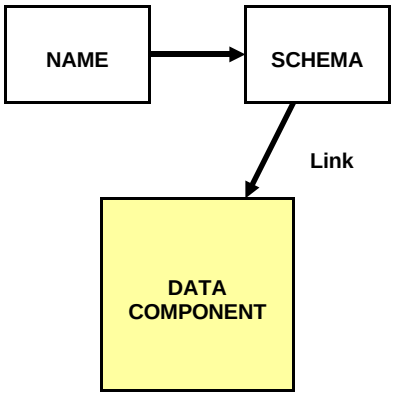
## DATA MODELING

The high-level management of engineering data is closely related to the management and performance of the actual product development process. Data is created and used within limited, well-defined domains. These include such areas as CAD Modeling, Structural Analysis, Shock and Vibration, Master Dimensions, Loads Analysis, Propulsion Systems, Performance Requirements, and many more. Typically, data is named and structured in ways which relate to the specific domain. When an engineering task has been successfully completed, the data is released by the responsible engineering management. This overall process represents a natural hierarchy that results from the decomposition of a complex process into its simpler components. Determining the most useful and efficient method for organizing this information can be called ***Data Modeling***. ***eBase*** provides powerful tools for data modeling. These tools are summarized in this Chapter.

## eBase ORGANIZATION

An ***eBase*** database is a collection of ***Entities***. An Entity can be viewed as an object with three components as shown in Figure 1-1. The Entity has a ***Name Component*** which is used to identify it, a ***Schema*** which is a set of rules that describes the data contents of the Entity, and it has ***Data Component*** which is the actual data itself. There is also a ***Link Component*** from the schema to the Data Component. ***eBase*** is a ***Multischematic Database***. This means there are different kinds of schemas. Entities which share the same type of schema are grouped into ***Entity Classes***. The ***eBase*** Entity classes are described in more detail later in this Chapter. Although the Entities contain the actual data that is being managed, ***eBase*** supports a hierarchical organization which allows you to specify the levels of data which are compatible with your way of doing business. Beyond this, ***eBase*** provides an Entity ***Subscript*** which allows storage of different revisions of the same data classes.

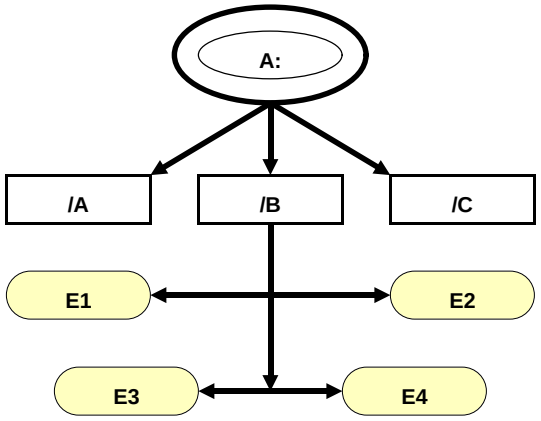
**Figure 1-1. Entity Components**



## DIRECTORY HIERARCHY

Figure 1-2 illustrates a typical way in which an ***eBase*** hierarchy may be defined. The ***eBase***, A: in the figure, contains three ***Directories***. A directory might contain the engineering data for a specific product line, or it might contain a subset of such data for a large and complex product. Each directory may, in turn, be composed of one or more ***Subdirectories*** containing related data. Each directory or subdirectory may contain different data Entities. These Entities may be organized in as many additional levels of directories as needed to fully categorize the data. In Figure 1-2, the white boxes represent directory hierarchy levels, while the shaded boxes represent actual data stored in the database.

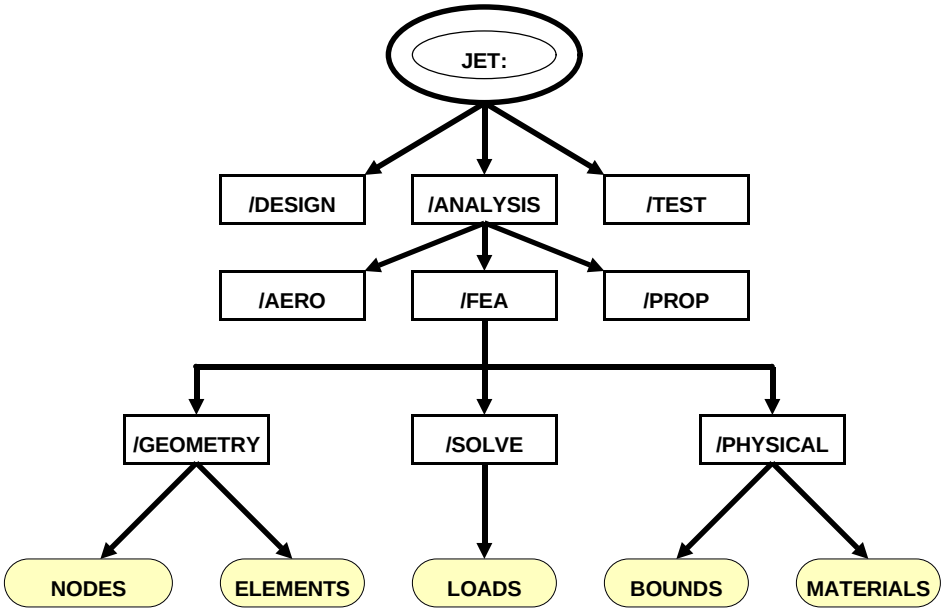
**Figure 1-2. eBASE Directory Hierarchy**




This organization is best illustrated by an example illustrated in Figure 1-3. Suppose there is a corporate product called JET. An ***eBase*** database will be used as a repository for all of the engineering activities that will be performed for this product. The data for JET has been partitioned into three subdirectories, DESIGN, ANALYSIS, and TEST. The ANALYSIS subdirectory has also been divided into three directories: one for aerodynamic analyses, AERO; one for finite element analyses, FEA; and one for propulsion analyses, PROP.

The FEA directory has also been partitioned into three subdirectories. The first, GEOMETRY, contains two Entities which define finite element model-

**Figure 1-3. Typical Data Organization**



ing data. These are called NODES, which contains grid point coordinate data, and ELEMENTS, which contains data defining the finite elements in the model.

In addition to the geometric data, there are other physical data, stored in subdirectory PHYSICAL, for the finite element model such as material properties, MATERIALS, and the boundary conditions imposed on the model, BOUNDS. Finally, there is data required for the solution of the analysis, such as environmental loads, LOADS which are stored in SOLVE.

Note that there may be many other uses and interpretations of these levels which are limited only by your imagination.

## SUBSCRIPTED ENTITIES

Each directory or subdirectory may contain one or more actual database Entities. These are, naturally, defined by their name. However, ***eBase*** allows another flexibility in that you may have several Entities which have the same basic name, but a different subscript number. This is useful if you have, for example, several sets of analysis results that you are going to correlate. There may be several Entities named RESULTS, with different indices, that contain answers for different problem conditions.

The concept of subscripts is very powerful. You can, for example, implement the version concept found on some computers by simply incrementing your subscript each time you change the data in an Entity. Or, you can simply create whole *arrays* of database Entities directly with their subscripts. This provides you with the capability to do anything that you like to better organize, access, and manipulate your data.

## THE MULTISCHEMATIC MODEL: ENTITY CLASSES

As introduced earlier, ***eBase*** is a collection of data which is organized into Entities. An Entity is simply a group of related data that is stored together. Unlike purely relational databases which store tables of data, ***eBase*** has four different data structures which are treated in a unified manner. This type of database is called ***Multischematic***. The four ***eBase*** Entity classes are:

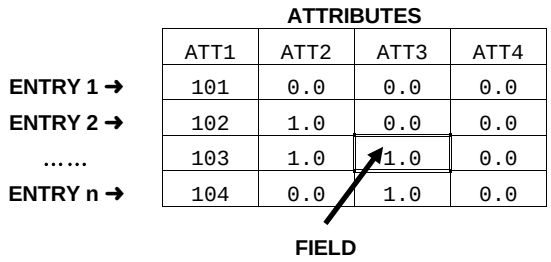
- ❒ Relational Entities
- ❒ Matrix Entities
- ❒ Freeform Entities
- ❒ Stream Entities

Each of these Entity classes is briefly described in the following sections.

## Relational Entities

You may view a Relation as simply a table of data. ***eBase*** tables have rows, which are called ***Entries***, and columns, which are called ***Attributes***. A particular data value at a given entry and attribute location is called a ***Field***. Figure 1-4 illustrates a typical Relation. The attributes of the Relation are called ATT1, ATT2, ATT3, and ATT4. Note that there are four entries for which this data has been defined. These attributes, taken together with the characteristics of the data which they contain, form the ***Schema*** of the Relation. The attribute characteristics, or ***Schema Components***, of a Relation are defined when it is created. These characteristics include the attribute name, data type and, in some cases, length, are described in detail in Chapter 3.

**Figure 1-4. RELATIONal Entity**



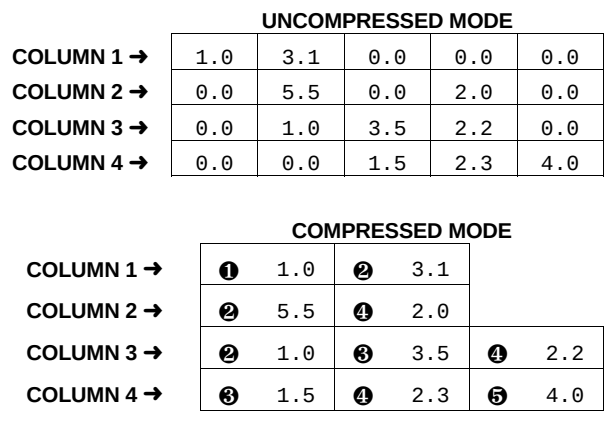

## Matrix Entities

Matrices form the second Entity class in an ***eBase*** database. Matrices are arrays of numbers used in mathematical formulae typically encountered in engineering and physical science software applications. A Matrix Entity is defined in the standard mathematical manner as an array of n ***Rows*** and m ***Columns***:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{12} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}$$

Each value $a_{ij}$ in the Matrix is called a ***Term***. The subscripts i and j indicate the row and column location of the term. Matrices also have schema components. These include the ***Orientation***, the ***Storage Mode***, the ***Numeric*** types of their terms, and their general topological ***Shape***. As with ***Relation***s, the schema is defined when the Matrix is created. A Matrix which is entered by columns has a ***Column-major*** orientation and one which is entered by rows has a ***Row-major*** orientation. The Shape of a ***Matrix*** represents its general topology such as square or rectangular. The Numeric type of a Matrix specifies the type of data defining its terms. For example, the terms can be real or complex. Finally, you may select from two possible Storage Modes for the Matrix. The first mode is ***Uncompressed*** in which case all of the Matrix terms are stored on the database. To improve efficiency, ***eBase*** uses an optional technique to minimize the disk storage requirements of matrices by storing them in the ***Compressed*** mode. Unlike the ***Uncompressed*** Matrix, only the

**Figure 1-5. MATRIX Entity**



non-zero terms of Compressed matrices are stored along with a small amount of control information. Consider the Matrix:

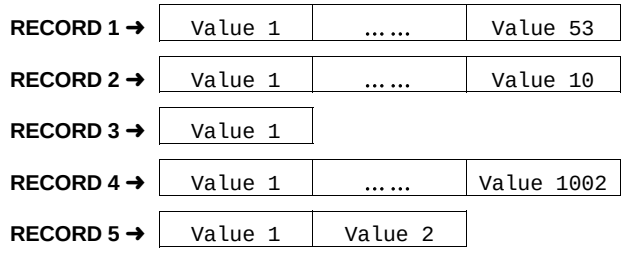
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 3.1 & 5.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 3.5 & 1.5 \\ 0.0 & 2.0 & 2.2 & 2.3 \\ 0.0 & 0.0 & 0.0 & 4.0 \end{bmatrix}$$

Figure 1-5 illustrates how this data would appear in a Column-major Orientation for both the Compressed and Uncompressed Modes.

## Freeform Entities

Freeform Entities are a form of internal data representation that can sometimes be used to improve the performance of software applications. They are collections of data with only a local and transient purpose. Most often, they are used in a software application that has been developed with the ***eBase*:*applib***. You may think of Freeform Entities as Fortran random files which have variable length ***Records***, as shown in Figure 1-6. Freeform Entities are defined with a single Schema Component that defines the records to be comprised either of a homogeneous data type or of mixed heterogeneous data types. The former, called ***Schematic Freeform*** Entities, may be viewed in ***eShell*,** as described in Chapter 7, and they may be exported and imported as described in Chapter 11. Neither is true for the mixed type. Because of these limitations, the use of non-schematic Freeform entities is discouraged for other than temporary data.

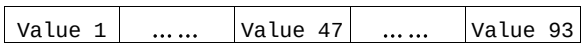
**Figure 1-6. FREEFORM Entity**




## Stream Entities

A Stream Entity, shown in Figure 1-7, is a continuous Stream of ***Data Values***, each of which has a ***Position*** within the Entity. Each Data Value may be directly and randomly addressed by reference to its Position. You may best think of this type of Entity as a low-level Unix file structure. A Stream Entity also has a very simple schema with a single Schema Component — it may contain mixed data types, in which case the limitations described for non-schematic Freeform Entities also apply, or it may contain a single numeric data type, or ***Schematic Stream***, in which case it has good portability characteristics. The length of a Stream Entity is determined by the last Position into which you insert data.

**Figure 1-7. STREAM Entity**



## SYNTAX OF COMMANDS

Later Chapters of this manual present the precise syntax of the various ***eQL*** commands. This section introduces the nomenclature used in the command descriptions.

### What is a Command?

A command is a sequence of ***Command Parts*** which ends with a terminator symbol. There are different kinds of Command Parts including Keywords, Metasymbols, and other special symbols which will help you understand ***eQL.*** The ***eQL*** termination symbol is the semicolon (**;**). Other Command Parts are discussed in the following sections.

### Keywords

A ***Keyword*** is a Command Part which you must enter exactly as shown in this manual. Keywords appear in **BOLDFACE** typewritten style.

### Abbreviating Keywords

For convenience, you may abbreviate many of the ***eQL*** Keywords. This is indicated in the manual by underlining the required part of the Keyword, as:

**SELECT**

### Simple Metasymbols

A ***Metasymbol*** is a special Command Part that indicates you must enter a requested data value. These symbols are shown in *italics*. A precise definition of each Metasymbol appears at the time it is introduced in the manual. For example, a command that you will learn later shows both Keywords and a Metasymbol:

**SET SCRIPT TO** *file_name*;

### Optional Command Parts

An optional Command Part, whether a Keyword or a Metasymbol, is indicated by enclosure in square brackets (**[]**). For example:

**COMMAND_NAME** *user_list* [ **KEYWORD** ]

indicates that the Keyword **KEYWORD** in the command is optional.

## Command Part Selectors

A ***Selector*** defines a group of Command Parts, one or more of which must be selected. Selectors are enclosed in curley braces (**{}**):

*justification* ⇒ { **LEFT** | **CENTER** | **RIGHT** }

Naturally, Selectors may also be optional in which case they are enclosed in brackets.

## Metasymbol Suffixes

Many commands allow lists of user input. In these cases, the command description uses the suffix:

*_list*

to indicate this. Each term in the list is suffixed with:

*_term*

Unless otherwise specified, the terms in a list are separated by commas:

*user_list* ⇒ *user_term* [ , *user_term* [ , ... ] ]

Throughout the manual you will find commands which include list Metasymbols. If the list has the form shown above, then only the terms will be defined.

## Complex Metasymbols

For a complex command, there may be one or more Command Parts which are themselves composed of many Command Parts. In such cases, each complex Command Part is discussed separately and the relationship of the Command Part to the whole command is shown in the left margin while the exact definition appears in the body of the text adjacent to it.

**SELECT** *attr_name_list*
FROM *rel_name_list*
[ ***WHERE_clause*** ]

*WHERE_clause* ⇒ ……

Note that the Metasymbol is highlighed in boldface italics.

### Important Information

Two other conventions are used to assist your learning of ***eShell***. These are callouts which provide important information or give you reminders useful when performing the tutorial examples.

This is important information!!

This is a reminder!!

## NAMING eBase OBJECTS

Many objects within the ***eBase*** database are given names. These include directories, Entities, views, Relational attributes, and symbols. Additionally, there are special rules for Entity names and host-computer files. These are described in the following sections.

### Basic Naming Rules

A ***Basic Name*** must be no longer than 32 characters. It must begin with a letter (A-Z) and the remaining characters may be letters (A-Z), digits (0-9), or the special symbols underscore ( _ ), and dollar ($).

### Path Naming Rules

A ***Path*** defines a database and a directory hierarchy. The general form of a path is:

*path* ⇒ [ *database_name* : ] [ / ] [ *dir_name_list* ]

where the *database_name* is the logical name of an open ***eBase*** database and the *dir_name_list* is a series of one or more *dir_name*s separated by slashes:

*dir_name_list* ⇒ *dir_name* [ / *dir_name* [ / ... ] ]

where each *dir_name* is a directory name.

## Entity Naming Rules

An ***eBase*** Entity is described by an optional Path, a Basic Name, and an optional subscript list:

*ent_name* ⇒ [ *path* ] *basic_name* [ *subscript* ]

The optional *subscript*, which is enclosed in square brackets (**[]**), has the form:

*subscript* ⇒ **[** *sub_list* **]**

Each *sub_term* is an integer value. The number of *sub_term*s must match the number specified when the first subscripted version of *ent_name* was created.

| VALID ENTITY NAMES | ILLEGAL ENTITY NAMES |
|---|---|
| **ELEMDATA** | **COORD@X** (Illegal character) |
| **Grid_Coordinates** | **1LOAD** (Starts with number) |
| **KAA[1,101]** | **KAA[2]** (Wrong number of subscripts) |
| **Displacement[6]** | **MAT[A]** (Subscript not integer) |

## File Naming Conventions

There are a number of host computer files that may be used during your ***eShell*** session. These include the actual database name, ***eQL*** Command Files, Archive Files, Report Files, and Interface Files. File names which do not satisfy the Basic Naming Rules described above must be enclosed in tics, the actual format of the name used depends on your ***eShell*** host computer and is described in the System Interface Manual.

## USING THE eShell PROGRAM

This section provides you with instructions for using ***eShell***, selecting ***eBase*** databases, and entering commands. Generally, ***eShell*** is ***Case-insensitive***; upper and lower case letters may be used interchangeably. When names or command parameters contain embedded blanks or other special characters, they must be enclosed in single quotation marks, sometimes called *tics*, as in `'HI THERE'`. Also note that when enclosed in tics, letters are case sensitive.

### Getting Started

Usually, ***eShell*** is installed on the host computer as a system procedure. You must check with your computer system manager to determine if your system has been installed differently. To execute the program from the command line, you enter:

```
eshell [-ps prefname] [-pu prefname]
                      [-pl prefname] [database]
```

where:

| | |
|---|---|
| *prefname* | Specifies the substitution string used to generate Preference File names. You may specify a different string for the System (**-ps**), the User (**-pu**) and the Local (**-pl**) preference files. If you have the unusual case where all of these files have the same name, you may use the option **-p** followed by the *prefname*. |
| *database* | Is the name of a database to be opened with read access. |

This places you in the command mode. Unless directed otherwise by ***eShell*** commands, all subsequent output will be sent to the terminal device. The optional *prefname* information is an advanced feature used for customizing ***eShell*** which is described in the Installation Guide and System Support Manual.

### Online HELP

***eShell*** provides an online **HELP** feature to provide documentation of the features available in the program. The command used is:

```
HELP [ command_part_list ];
```

If **HELP** is specified without any additional parameters, a listing of available *command_parts* is given. You may then obtain additional information by picking from a menu. If the *command_part_list* is provided, then information relating to the named *command_parts* is presented without menu interaction.

Additionally, you may use the **uaidoc** utility to view this manual online using the Adobe® Acrobat® Reader 3.0 which is delivered with your software. To do this, you launch the Reader in a new window.

## Accessing Databases

The first command used selects an ***eBase*** database that you wish to open for access. This is done with the command:

```
OPEN database_name [ = 'phys_name'] { NEW                    } ['params'] ;
                                    { TEMP                   }
                                    { WITH {READ }           }
                                    {      {WRITE}           }
                                    {      {ADMIN}           }
```

The *database_name* specifies the name of an existing ***eBase*** database or a database that you are creating during your current session. The optional *phys_name* overrides the name of the database as viewed by the operating system. On operating systems which use the file name to identify the location of the files, this also gives you the ability to override the configuration parameters which would otherwise control the location of the database files. For nonexisting databases, you may specify a status of **NEW** or **TEMP**orary when you open it. Temporary databases do not have any password protection since they are deleted at the end of your ***eShell*** session. For a new database, you will be prompted for an ***ADMINISTRATION*** password. This password will also be used for ***READ*** and ***WRITE*** access. You may change any of the passwords with the **SET PASSWORDS** command described later in this Chapter.

For existing databases, you are prompted for the appropriate access password for the database. Some host-computers require additional inforation which is specified as optional *params*; see the Installation Guide and System Support Manual for a discussion of these.

The initial part of the ***eShell*** session is shown in the following example.

Example 1-1. Execute the ***eShell*** program for an ***eBase*** named MY_EB.

```
eshell
.... eShell Version 20.0 dd-mmm-yy 14:22
.... Copyright (c) 1992-1997, Universal Analytics, Inc.
.... All rights reserved
eSh> OPEN MY_EB WITH ADMIN;
eSh> Enter ADMINISTRATION Password: MAGIC
.... eBase "MY_EB" Open with "ADMINISTRATION" Privilege
      {   Your eQL Commands
eSh> END;
.... eShell Ended at 09:30 dd-mmm-yy
.... Elapsed Time 00:16:21   CPU Time 00:02:21
```

Note that the password is not a security measure for entering the ***eShell*** program, but the password under which ***eBase*** databases will be accessed. ***eShell*** then verifies that the password allows the access you have requested.

You may use two or more ***eBase*** databases simultaneously. Each database may be opened with any privilege level. The sequences of events shown in Example 1-1 occurs each time you open a database as shown in the following example.

Example 1-2. Execute the ***eShell*** program for two ***eBase***s.

```
eshell
.... eShell Version 20.0 dd-mm-yy 14:22
.... Copyright (c) 1992-1997, Universal Analytics, Inc.
.... All rights reserved
eSh> OPEN FOO WITH ADMIN;
eSh> Enter "ADMINISTRATION" Password: MAGIC
.... eBase "FOO" Open with "ADMINISTRATION" Privilege
eSh> OPEN BAR WITH READ;
eSh> ENTER "READ" PASSWORD: SCRT
.... eBase "BAR" Open with "READ" Privilege
     .....
```

All of the operations described in the remainder of this manual may now be performed. You may terminate work on a given database by simply entering the command:

**CLOSE** *database_name* [ **DELETE** ] **;**

The **DELETE** option will delete all of the database physical files from your host-computer. To use this option, you must have opened the database with the ***ADMINISTRATION*** privilege. When you end your ***eShell*** session, all open databases are automatically closed. It is not necessary to explicitly close each one.

## eQL Command Entry

Because the ***eQL*** language allows you to perform complex operations on the database, it is possible for commands to get rather long. To support long commands and to allow their entry in a readable and structured manner, ***eQL*** allows commands to be entered on multiple lines. When this is done, ***eQL*** provides line numbers for the command as it is being entered. This is illustrated in the following example.

Example 1-3. Entering an ***eQL*** command on multiple lines.

```
eSh> SELECT * FROM GRID
  2>      WHERE X > 3.0
  3>      AND   Y < 2.0;
```

Note that the command must end with a semicolon (**;**). Each command line continuation is numbered, beginning with 2. The line entered at the prompt is called line 1. ***eShell*** performs no action until the complete command, including the semicolon, has been entered. This command is referred to as the ***Active Command***. After it is entered, the active command is saved until you enter the next active command. This allows you to modify a command in the event it contains an error, or, in order to create a new command that is similar to it. You may list the active command using:

**LIST** [ *line_1* [ **TO** *line_n* ] ];

where *line_1* represents the first line number to be listed and *line_n*, the last. If no line numbers are given, the entire active command is listed,

and if only *line_1* is given the single line is listed. The active command is positioned to the single line most recently **LIST**ed. This position is called the ***Current Position***. You may delete lines in an analogous manner:

**DELETE** [ *line_1* [ **TO** *line_n* ] ];

However, when the **DELETE** has no optional line number specified, only the line at the current position is deleted. A new line may be added to the active command with:

**ENTER** *'new_line'*;

where the *new_line* is inserted immediately after the current position. Note that if the quotation mark is part of *new_line*, it is represented as two consecutive quotation marks. You may also edit part of the line at the current position. To move to the portion of the active command that you will change, you use the LIST command. The change is then made with the command:

**CHANGE** *#string_1#string_2#* ;

where the first occurrence of *string_1* is replaced by *string_2*. Although the sharp (#) character is shown in the command, nearly any delimiter may be used as long as it does not appear in either target string. Exceptions are (**|**, \, **~**, **"**, **'**, **{**, and **}** ).Note that embedded blanks are significant and they may appear between the delimiters.

Example 1-4. List the previous command and change Y to Z.

```
eSh> LIST 1 TO 3;
  1    SELECT * FROM GRID
  2       WHERE X > 3.0
  3*      AND   Y < 2.0;
eSh> CHANGE /Y/Z/;
  3*      AND   Z < 2.0;
```

Notice that an asterisk (*****) marks the current position within the active command. You execute the active command by entering:

**RUN;**

The **RUN** command is typically invoked after modifying or correcting the previous command. One last feature of command entry is the blank, or null, line entry. If you discover that your command is in error prior to entering the semicolon, you may enter a blank line at the prompt. This closes command input activity and allows you to edit the command immediately.

At the end of a session, you enter the ***eQL*** command:

**END;**

This command closes all the active databases, provides a summary of computer resources that were used during the session, and returns to the operating system of the host computer.

## Symbol Substitutions

You may at any time during your ***eShell*** session define symbols which may be used in subsequent commands. Symbols are defined using the command:

**DEFINE** [ *symbol_name* [ **=** *value* ] ] **;**

The *symbol_name* must contain 32 or fewer characters and begin with a letter. The *value* is used to set the symbol. It is always treated as a simple character string which will be substituted for appearances of the symbol. To use a symbol, you use the substitution notation illustrated in the previous section:

**&***symbol_name*

If a symbol that you reference has not been defined, you will be prompted for it when you invoke a command. Both cases are illustrated by the next example.

Example 1-5. Define the symbol X and use the symbols X and Y to perform the query shown:

```
eSh> DEFINE X = 3.0;
eSh> SELECT * FROM GRID
  2       WHERE X > &X
  3*      AND   Y < &Y;
eSh> ENTER VALUE FOR SYMBOL "Y": 2.0
```

Once a symbol is defined either explicitly or through a prompt, it retains its value until you remove the symbol using the command:

**UNDEFINE** { *symbol_name* | ***** } **;**

The asterisk (*****) may be used to **UNDEFINE** all previous definitions. Several other conventions may be used for symbol definition. If you want a symbol to be concatenated with additional characters, then you follow the substitution with a period and the additional characters.

For instance,

```
eSh> DEFINE FOO = XYZ;
eSh> SEL * FROM &FOO.01;
```

results in:

```
eSh> SEL * FROM XYZ01;
```

Finally, if it is necessary to include a period in the string immediately following a symbol, then you must specify two periods:

```
eSh> DEFINE BAR = ABC;
eSh> SEL &BAR..GID FROM &BAR;
```

results in:

```
eSh> SEL ABC.GID FROM ABC;
```

If you enter the **DEFINE** command without an argument, all currently defined symbols will be listed, and if you enter **DEFINE** with just a symbol name, then the value of that symbol will be shown.

## DATABASE PROTECTION

***eBase*** databases have three levels of ***Passwords*** which can be used to secure the data against inadvertent destruction or unauthorized use. These passwords allow ***READ***, ***WRITE***, and ***ADMINISTRATION*** privileges. Because the principal intent of ***eBase*** is the development of software applications, it has not been designed to have the comprehensive security protections often found in the financially oriented products.

The ***READ*** privilege allows a user to read all of the Entities on the database and to perform any operation which does not create, modify or delete information from it. The ***WRITE*** privilege extends access to allow Entities to be modified and deleted from the database. Finally, the ***ADMINISTRATION*** privilege allows all possible database operations. The command which allows you to change your ***eBase*** passwords during an ***eShell*** session is:

**SET PASSWORDS** [ **ON** *database_name* ] *password_list>* **;**

where *password_list* is a list of one or more *password_term*s which define new values for any or all of the passwords. These terms have the form:

*password_term* ⇒ { { **READ** / **WRITE** / **ADMIN** } *password* | **CLEAR** { **READ** / **WRITE** / **ADMIN** } }

If you do not specify a *database_name*, then the passwords are changed for the current database. You may only change the passwords if you have the ***ADMINISTRATION*** privilege level for the database. The **CLEAR** option may be used to remove any level of password protection.

## DATABASE INTEGRITY

Because the primary design goals of ***eBase*** are flexibility and performance in scientific software application, it does not provide extensive data integrity features which result in the high overhead of many non-scientific database products. It does, however, provide several levels of data integrity. This is done through configuration parameters which force frequent physical data transfers.

## WORKING WITH MULTIPLE DATABASES

***eShell*** allows you to use any number of ***eBase*** databases that you have previously created or obtained from an external source. As you will see in subsequent Chapters of this manual, you may define multiple databases and address the entire directory structure of each.

## MOVING DATABASES BETWEEN COMPUTERS

***eShell*** allows you to move ***eBase*** databases from one computer to another. Options are available to export a database in either character format or in binary format using the IEEE-754(1985) Binary Floating-Point Arithmetic Standard. Such database files may then be transferred to other computers and then imported into ***eShell***. This may only be done successfully for completely schematic data which includes all Relational Entities, all Matrix Entities, Schematic Freeform and Schematic Stream Entities. Details of this procedure are given in Chapter 11 of this manual.

## eShell LIMITATIONS

Many databases have severe size limitations and their query languages long lists of Reserved Words, those which you cannot use for other purposes. ***eShell*** and ***eBase*** have reduced such problematic limits dramatically as described in the following sections.

### Reserved Words

***eQL*** has been designed to have as few restrictions as possible. While there are no categorically reserved words in the language, there are some which cannot be used in particular circumstances. Rather than enumerating these, it is best if you do not use the words shown below for names that you use in your database.

| | |
|---|---|
| **BINARY** | **INTER or INTERSECTION** |
| **DIFF or DIFFERENCE** | **MAT or MATRIX** |
| **DISTINCT** | **ORDER** |
| **FORMATTED** | **SEL or SELECT** |
| **FREE or FREEFORM** | **STR or STREAM** |
| **FROM** | **UNION** |
| **GROUP** | **WHERE** |

### Size Limitations

There are no practicable limitations on the size or composition of an ***eBase*** database. All limitations are imposed by the architecture of the host computer. The only quantifiable limitation is that any Command Part which requires an integer value, such as as Matrix row or column number, a Freeform record number, or a Stream Data Value, cannot exceed the largest representable integer on the host computer. The smallest value for ***eShell*** computers is greater than ***two billion***, i.e. $2x10^9$.

## USING THE TUTORIAL

This manual provides all of the information necessary to use ***eShell*** and the ***eQL*** language. Command usage is liberally illustrated by examples taken from a small sample database shown in Figure 1.8. This database has been delivered with ***eShell*** and is intended to be used as a tutorial to teach you the fundamentals of ***eShell***. Contact your Administrator if you wish to have your own copy of the tutorial database, called `TESTEB:`, and a copy of the Command Files which contain the ***eQL*** commands used in the manual. The following brief description of the Entities on the database will provide you with a physical interpretation of the data.

There are four directories in the database. The first is `/GEOM`, the second is `/MODEL`, the third is `/RESULT`, and the fourth is `/PLOT`. The Entities themselves are:

- `/GEOM/GRID` is a Relational Entity that contains, for each grid point in a finite element model, the identification number (**GID**), a coordinate system identification number (**CID**), and the three spatial coordinates (**X**,**Y**,**Z**) assumed to be in some common coordinate system.
- `/GEOM/QUAD4` is a Relational Entity that contains finite element definitions for quadrilateral plate elements. This data includes the element identification number (**EID**), the element property identification number (**PID**) and the four grid point identification numbers defining the element (**G1**,**G2**, **G3**,**G4**).
- `/MODEL/PSHELL` is a Relational Entity containing element property data which is referenced by the **PID** in the **QUAD4** Relation. It contains the **PID**, a material property identification number (**MID**) and the thickness (**T**) of each element referencing the property.
- `/RESULT/Q4STR[1]` and `/RESULT/Q4STR[2]` are Relational Entities containing three output stress values (**SIGX**,**SIGY**,**TAUXY**) representing Normal-X stress, Normal-Y stress and shear stress, respectively, for each element (**EID**). The two subscripted Relations represent the static solution to two distinct loading conditions.
- `/RESULT/Q4S` is a Relational Entity containing the same data as the previous Relations using a different organization in which the loading condition data is stored as a Relational attribute.

Note that these two different data forms simply illustrate different data modeling methods; there is no need to have both forms to perform any particular access. Only the retrieval performance is affected by the modeling method.

- /RESULT/BIGREL is a Relational Entity containing a large number of entries which depends on your host-computer. It is used in the examples presented in Chapter 6.
- /MODEL/KGG is a Matrix Entity that represents the global stiffness matrix of the finite element model. The Matrix has the Column-major Orientation, Compressed Mode, Numeric Type real, double precision, **RDP**, and Symmetric Shape. The contents of this Matrix are:

| | | | | | |
|---|---|---|---|---|---|
| 100.0 | 200.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 200.0 | 300.0 | 400.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 400.0 | 500.0 | 600.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 600.0 | 700.0 | 800.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 800.0 | 900.0 | 1000.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 1000.0 | 1100.0 |

- /MODEL/TESTFREE is a Freeform Entity containing five Records of varying length each of which is comprised of real, single precision, **RSP**, data. The data are:

| RECORD | DATA | LENGTH |
|---|---|---|
| **1** | 1.0,2.0,...,10.0 | 10 |
| **2** | 2.0,4.0,...,40.0 | 20 |
| **3** | 3.0,6.0,...,90.0 | 30 |
| **4** | 4.0,8.0,...,160.0 | 40 |
| **5** | 5.0,10.0,...,250.0 | 50 |

- /MODEL/STRM is a Stream Entity of type integer, **INT**, containing 1000 Data Values which are equal to the Position within the Entity.
- The /PLOT directory includes a number of Relations that contain larger amounts of data suitable for demonstrating the XYPLOT commands presented in Chapter 5. You may use the DIR and DESCRIBE commands to get information about these.

**Figure 1-8. Sample *eBase* Database TESTEB**

| RELATION /GEOM/GRID | | | | |
|---|---|---|---|---|
| GID | CID | X | Y | Z |
| 1 | 0 | 0.0 | 1.0 | 0.0 |
| 2 | 0 | 1.0 | 1.0 | 0.0 |
| 3 | 0 | 2.0 | 1.0 | 0.0 |
| 4 | 0 | 3.0 | 1.0 | 0.0 |
| 5 | 0 | 4.0 | 1.0 | 0.0 |
| 6 | 0 | 0.0 | 0.0 | 0.0 |
| 7 | 0 | 1.0 | 0.0 | 0.0 |
| 8 | 0 | 2.0 | 0.0 | 0.0 |
| 9 | 0 | 3.0 | 0.0 | 0.0 |
| 10 | 0 | 4.0 | 0.0 | 0.0 |

| RELATION /RESULT/Q4STR[1] | | | |
|---|---|---|---|
| EID | SIGX | SIGY | TAUXY |
| 1 | 1.0+6 | 2.0+6 | 4.0+4 |
| 2 | 3.0+6 | 1.0+7 | 6.0+3 |
| 3 | 2.0+6 | 3.0+6 | 4.0+4 |
| 4 | 2.0+6 | 1.0+6 | 5.0+4 |

| RELATION /RESULT/Q4S | | | | |
|---|---|---|---|---|
| EID | CASE | SIGX | SIGY | TAUXY |
| 1 | 1 | 1.0+6 | 2.0+6 | 4.0+4 |
| 2 | 1 | 3.0+6 | 1.0+7 | 6.0+3 |
| 3 | 1 | 2.0+6 | 3.0+6 | 4.0+4 |
| 4 | 1 | 2.0+6 | 1.0+6 | 5.0+4 |
| 1 | 2 | 7.0+7 | 0.0 | 3.0+3 |
| 2 | 2 | 6.0+7 | 0.0 | 2.0+3 |
| 3 | 2 | 5.0+7 | 0.0 | 1.0+4 |
| 4 | 2 | 4.0+7 | 0.0 | 4.0+3 |

| STREAM /MODEL/STRM | |
|---|---|
| Numeric Type | INT |
| Length | 1000 |

| RELATION /GEOM/QUAD4 | | | | | |
|---|---|---|---|---|---|
| EID | PID | G1 | G2 | G3 | G4 |
| 1 | 1 | 1 | 2 | 7 | 6 |
| 2 | 2 | 2 | 3 | 8 | 7 |
| 3 | 1 | 3 | 4 | 9 | 8 |
| 4 | 2 | 4 | 5 | 10 | 9 |

| RELATION /MODEL/PSHELL | | |
|---|---|---|
| PID | MID | T |
| 1 | 101 | 0.1 |
| 2 | 201 | 0.5 |
| 3 | 301 | 0.3 |

| RELATION /RESULT/Q4STR[2] | | | |
|---|---|---|---|
| EID | SIGX | SIGY | TAUXY |
| 1 | 7.0+7 | 0.0 | 3.0+3 |
| 2 | 3.0+6 | 1.0+7 | 6.0+3 |
| 3 | 5.0+7 | 0.0 | 1.0+4 |
| 4 | 2.0+6 | 1.0+6 | 5.0+4 |

| RELATION /RESULT/BIGREL | | |
|---|---|---|
| ATT1 | ATT2 | ATT3 |
| ***See Text*** | | |

| MATRIX /MODEL/KGG | |
|---|---|
| Orientation | COLUMN MAJOR |
| Mode | COMPRESSED |
| Numeric Type | RDP |
| Shape | SYMMETRIC |
| Number of Rows | 6 |
| Number of Columns | 6 |
| Density | 0.44 |

| FREEFORM /MODEL/TESTFREE | |
|---|---|
| Numeric Type | RSP |
| Number of Records | 5 |
| Longest Record | 50 |

| FOR DIRECTORY /PLOT<br>You Must Query the Database |
|---|

# 2. DIRECTORIES AND ENTITIES

As introduced in Chapter 1, an ***eBase*** database may be logically configured using a hierarchical structure of ***Directories***. This Chapter describes the commands which you may use to manipulate and manage a directory structure and those used to describe the entities within the directory structure.

## REFERENCING DIRECTORIES IN COMMANDS

When you use a hierarchical set of directories within ***eShell***, you must indicate where a database entity may be found within the directory structure. This is done by specifying the Path to the entity.

If the Path name begins with a slash, then ***eShell*** searches for the entity beginning at the root of the directory structure. Otherwise, ***eShell*** begins at the current directory and searches downward.

For example, many ***eShell*** commands require that you enter one or more entity names. Each entity name may be in one of three forms:

- ❒ It may be a fully qualified name. This means specifying the entire directory Path, including the database descriptor:

  `DB2:/A/B/C/MYDATA`

- ❒ It may be an ***Absolute Directory***. In this case, the specified ***PATH*** is used and only the database descriptor is obtained from the current path default:

  If `DB2:/A/B` is the default path,
  Then `/D/E/MYDATA` refers to `DB2:/D/E/MYDATA`

- ❒ It may be a ***Relative Directory***, in which case the specified ***PATH*** is appended to the default:

If DB2:/A/B is the default path,
Then C/MYDATA refers to DB2:/A/B/C/MYDATA

- A second form of relative directory allows you to follow the directory chain upward by using a double dot (..) to move up one level in the hierarchy:

  If DB2:/A/B/C is the default path,

  Then ../../X/MYDATA refers to DB2:/A/X/MYDATA

  If you are in the Root Directory, the double dot has no effect.

## CREATING DIRECTORIES

A newly created ***eBase*** database has a single directory, called the ***Root Directory***, which is denoted symbolically by a slash (/) character. You may create a new directory using the command:

**MKDIR** *path* **;**

where the *path* ends with the directory name which will be created. Often, directories other than the root directory are called ***Subdirectories***.

Example 2-1: Create the directory ANALYSIS and a subdirectory called FEA on the TESTEB database.

```
eSh> MKDIR /ANALYSIS;
eSh> MKDIR /ANALYSIS/FEA;
```

## THE WORKING DIRECTORY

When you open an ***eBase*** database, you are be placed in the Root Directory of the database that you OPEN. Your location is called the ***Working Directory***. Each time you open another database, your working directory becomes the Root Directory of that database. To move to another directory, you use the command:

**CD** [ *path* ] **;**

where *path* specifies the directory name to which you will move. If no *path* is specified, then ***eShell*** identifies your current Working Directory.

Example 2-2: Move to the directory ANALYSIS/FEA.

```
eSh> CD ANALYSIS/FEA;
          or
eSh> CD ANALYSIS;
eSh> CD FEA;
eSh> CD;
.... Working Directory: TESTEB:/ANALYSIS/FEA
```

## REMOVING DIRECTORIES

You may remove an existing directory using the command:

**RMDIR** *path* **;**

where the *path* ends with the directory name which will be removed.

Example 2-3: While in the Root Directory, remove the directories FEA and ANALYSIS from the TESTEB database.

```
eSh> RMDIR ANALYSIS/FEA;
eSh> RMDIR ANALYSIS;
```

Note that all entities must be **PURGE**d from the directory before it may be removed. See Chapter 10.

## LISTING DIRECTORIES

There is an ***eQL*** command available that is used to determine information about the contents of a directory. There are two forms of the command to do this. The first is used for complete directories, and the second is used for entities. These are described in the following sections.

### Using a Path Specification

To obtain a directory listing for a selected directory, you use the command:

**DIRECTORY** [ *path* ] [ **ALL** / **RELATION** / **MATRIX** / **FREEFORM** / **STREAM** / **DIRECTORY** ] [ **DATE** / **SUMMARY** ] **;**

where *path* is a valid Path name. If the *path* is omitted, a listing of the entities in the current Working Directory is printed as illustrated in the example below.

Example 2-4: While in the root directory, print the directory listing for the TESTEB database.

```
eSh> DIR;
.... Directory TESTEB:/
.... Name    Class  Size
.... ------  -----  ----------
.... GEOM     DIR   2 Entities
.... MODEL    DIR   3 Entities
.... RESULT   DIR   3 Entities
.... Directory Contains 3 Subdirectories
```

In this case, there are no entities in the root directory. However, a listing of the subdirectories within a directory are shown along with a count of the number of entities in each. The next example shows the results obtained when a *path* is specified in the **DIR** command.

Example 2-5: List each directory on the TESTEB database.

```
eSh> DIR GEOM;
.... Directory TESTEB:/GEOM
.... Name   Class  Size
.... -----  ----   ----------
.... GRID    REL   10 Entries
.... QUAD4   REL    4 Entries
.... Directory Contains 2 Entities
eSh> DIR RESULT;
.... Directory TESTEB:/RESULT
.... Name   Class  Size
.... -----  -----  -------------
.... Q4S     REL   8 Entries
.... Q4STR   REL*  1 Subscript
....               2 Versions
....                 RelVer: [2]
.... Directory Contains 2 Entities
eSh> DIR MODEL;
.... Directory TESTEB:/MODEL
.... Name      Class  Size
.... --------  -----  ------------
.... PSHELL     REL      3 Entries
.... TESTFREE   FRE      5 Records
.... STRM       STR   1000 Values
.... KGG        MAT      6 Columns
.... Directory Contains 3 Entities
```

The listing shows the name of each entity, its class, and a description. Subscripted entities are indicated by an asterisk (*****) which follows the class. The description depends on the entity class:

- For Relations, it is the number of entries.
- For Matrix entities, it is the number or columns or rows depending on the storage mode.
- For Stream entities, it is the number of Data Values, and
- For Freeform entities, it is the number of records.

For all classes of subscripted entities, it is the number of subscripts used, the number of versions which exist, and the subscript of the Released Version, if any. A discussion of Released Versions is found later in this Chapter.

You may limit the directory listing by using one of the options shown in the command. For example, to restrict the listing to a single entity class or to the subdirectories within the directory, you use the corresponding Keyword. The **SUMMARY** option simply lists the last line of the listing which gives a count of the number of entities and subdirectories within the directory. If you select the **DATE** option, which applies to the restriction Keywords as well, the listing includes two dates, the date that each entity was created and date when it was last modified.

## The Entity Directory

You may also request a directory listing by referencing a fully qualified entity name. The **DIR** command form is then:

**DIRECTORY** *ent_name* [ **ALLVER** ] [**DATE** | **SUMMARY**] ;

where *ent_name* is a fully qualified entity name. The options **ALLVER** and **SUMMARY** are only used when *ent_name* has subscripted versions. The following examples show the directory listing produced by this command.

Example 2-6: Request a directory listing for entity GRID:

```
eSh> DIR /GEOM/GRID;
.... Relation TESTEB:/GEOM/GRID
.... Name  Size
.... ----  ----------
.... GRID  10 Entries
```

If *ent_name* has subscripted versions, and you enter only the Basic Name of the entity, then you obtain the directory list for the Released Version:

Example 2-7: Request a directory listing for the Released Version of entity Q4STR:

```
eSh> DIR /RESULT/Q4STR;
.... Relation TESTEB:/RESULTS/Q4STR[2]
.... Name      Size
.... --------  ---------
.... Q4STR[2]  4 Entries
```

By including the **ALLVER** option, you will obtain the directory for each of the subscripted versions of the entity:

Example 2-8: Request a directory listing for all subscripted versions of entity Q4STR:

```
eSh> DIR /RESULT/Q4STR ALLVER;
.... Relation TESTEB:/RESULT/Q4STR
.... Name      Size
.... --------  ---------
.... Q4STR[1]  4 Entries
.... Q4STR[2]  4 Entries
.... There are 2 Subscripted Versions
```

The resulting description shows the existing subscripts and the entity class. To then obtain the specific information about one of these, its subscript is explicitly included in the command.

## DESCRIBING DATABASE ENTITIES

To determine information about the entities on the ***eBase*** database, you use the command:

**DESCRIBE** *ent_name* **;**

where *ent_name* may be the fully qualified name of a database entity of any class.

Example 2-9: From the root directory, move to the subdirectory MODEL and request descriptions for Relation PSHELL, Matrix KGG, Stream STRM, and Freeform FREE.

```
eSh> CD MODEL;
eSh> DESCRIBE PSHELL;
.... Relation TESTEB:/MODEL/PSHELL
.... Schema is:
.... Attribute  Type    Len  Null  Descriptor
.... ---------  ----  -----  ----  ------------
.... PID        INT       1    NO
.... MID        INT       1   YES
.... T          RSP       1   YES
.... Current Contents  3 Entries
```

The meaning of the attribute type, length (**Len**), **Null** specifier and the **Descriptor** is presented in Chapter 3.

```
eSh> DES KGG;
.... Matrix TESTEB:/MODEL/KGG
.... Column Major, Compressed, Real, Double Precision, Symmetric
....   6 Rows,   6 Columns,   Density = 44.4%
```

The Matrix description includes information about the Orientation, Storage Mode, the Numeric Type of the terms in the Matrix, the Shape of the Matrix, and its size and density. The density is the ratio of the number of stored terms to the total possible terms and it indicates the amount of data compression that has been performed. A more complete discussion of these descriptors appears in Chapter 3.

```
eSh> DESCRIBE TESTFREE;
.... Freeform TESTEB:/MODEL/TESTFREE
....    5 Records, Longest Record is 50, Numeric Type INT
eSh> DESCRIBE STRM;
.... Stream TESTEB:/MODEL/STRM
....    1000 Data Values, Numeric Type RSP
```

For Freeform entities, the information reported includes the number of records in the entity and the length of the longest record. Similarly, for Stream entities, the Numeric Type and number of Data Values is reported. As discussed in Chapter 1, no operations other than **DESCRIBE** or **PURGE** may be performed on Freeform or Stream entities, unless they are Schematic. To **DESCRIBE** subscripted entities, there are two options as shown in the next two examples.

Example 2-10: Request a description for subscripted entity Q4STR[2]:

```
eSh> DESCRIBE /RESULT/Q4STR[2];
.... Relation TESTEB:/RESULT/Q4STR[2]
.... Schema is:
.... Attribute  Type    Len
.... ---------  ----  -----
.... EID        INT       1
.... SIGX       RSP       1
.... SIGY       RSP       1
.... TAUXY      RSP       1
.... Current Contents  4 Entries
```

In this example, an explicit subscripted version was included as part of the *ent_name* given. If an entity has subscripted versions and you ***do not*** specify a version, then you will obtain a description of the Released version of the entity. If there is no Released Version, then you will receive a message to that effect.

## RELATIONS WITH INDEXED ATTRIBUTES

When you **DESCRIBE** a Relation for which you have created one or more indexes, the results are modified slightly:

Example 2-11: Request a description of Relation Q4S and suppose that two indexes have been created for it, one on attribute EID and one on the combined attributes EID and CASE:

```
eSh> DESCRIBE /RESULT/Q4S;
.... Relation TESTEB:/RESULT/Q4S
.... Schema is:
.... Attribute  Type    Len
.... ---------  ----  -----
.... EID        INT       1
.... CASE       INT       1
.... SIGX       RSP       1
.... SIGY       RSP       1
.... TAUXY      RSP       1
.... Current Contents  8 Entries
....  2 Indexes Exist:
....  1  on  EID
....  2  on  EID and CASE
```

A complete discussion of indexing is found in Chapter 6 of this manual.

## RELEASING A SUBSCRIPTED VERSION

As shown in earlier sections of this Chapter, you may ***Release*** one of the subscripted versions of a subscripted entity. This is done with the command:

**RELEASE** *ent_name* **;**

The significance of a Released Version is that all ***eQL*** commands can now use the Basic Name of the entity without reference to a subscript. Naturally, if you have not Released a version, such commands will fail. The Release a different version of the entity, you simply use the **RELEASE** command and specify the new subscript. If you want to remove a released version without assigning a new one, then you use the command:

**UNRELEASE** *ent_name* **;**

## MANIPULATING ENTITIES

There are several operations that you may perform on entities, or groups of entities, that may simplify data management. The first allows one entity to be copied to another thereby replicating all of the physical data associated with that entity including any Indexes it may have. This command is:

**COPY** *entity_name_1* **TO** *entity_name_2* [ **ALLVER** ] **;**

where the **ALLVER** option allows all subscripted versions of the entity to be copied at one time.

You may change the name of an entity, or all of its versions, with the **RENAME** command:

**RENAME** *entity_name_1* **TO** *entity_name_2* [ **ALLVER** ] **;**

It is sometimes useful to create a new name for an entity or all of its versions. This is done with the **ALIAS** command:

**ALIAS** *entity_name* **TO** *alias_name* [ **ALLVER** ] ;

The **ALIAS** command may not be used across databases.

Finally, there are two commands which allow you to convert a Matrix entity from one Storage Mode to another. This is done using:

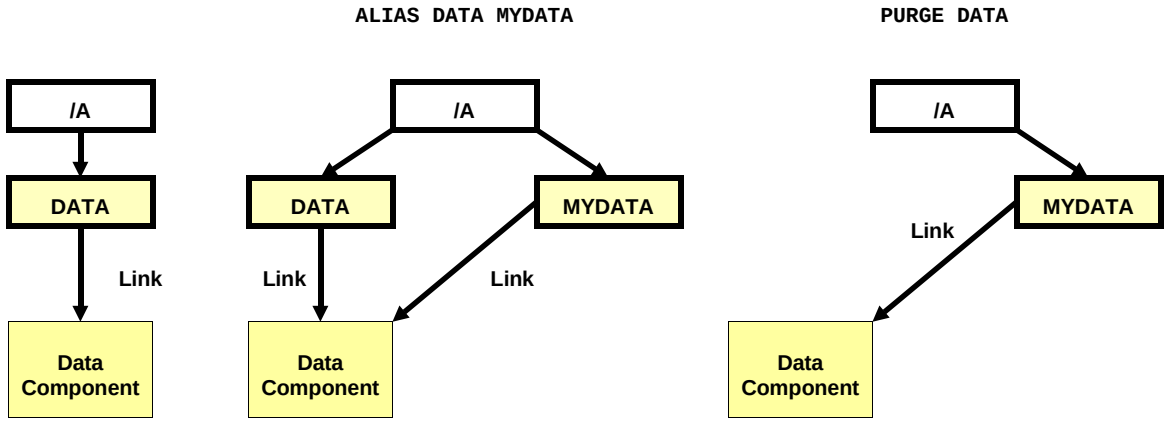
**COMPRESS** *mat_name_1* **TO** *mat_name_2* [ **ALLVER** ] ;

**UNCOMPRESS** *mat_name_1* **TO** *mat_name_2* [ **ALLVER** ] ;

For all of these commands, you may specify the Basic Name of an entity if you are operating on all Subscripted versions of it. Naturally, you may also include a Subscript to explicitly select just one version.

The **ALIAS** command does not make any physical copy of the data — it simply defines new name, *alias_name*, and Link Components which share the same data component as *entity_name*. This is illustrated in Figures 2-1a and 2-1b. This results in several side effects. If you modify the information in the data component of DATA, then these changes will appear in the alias, MYDATA, as well. If you PURGE either entity, then the name component will be removed from the database, but the data component will remain and may be referenced by the *alias_name*, as shown in Figure 2-1c.

**Figure 2-1. Effect of ALIAS Command**




a. Initial State of Database

b. Link Created by ALIAS Command

c. Original Name Removed but ALIAS Link Remains

*This page is intentionally blank.*

# 3. CREATING eBase ENTITIES

In addition to using an ***eBase*** database created by a software application that uses the ***eBase:applib*** tools, you may also create a new database or add new entities to an existing one. This Chapter describes the commands and techniques for performing these operations.

## CREATING RELATIONS

A new Relation may be added to the ***eBase*** database and, as will be seen in Chapter 8, data inserted into it. The command to do this is:

**CREATE RELATION** *rel_name*

$$\left\{ \begin{array}{l} \texttt{(}\ \textit{schema\_list}\ \texttt{)} \\ \textbf{LIKE}\ \textit{old\_rel} \end{array} \right\} \texttt{;}$$

The *rel_name* is the entity name to be created and *schema_list* is a list of one or more *schema_term*s. These terms specify the attributes which define the Relation and their data characteristics. Each *schema_term* in this list has the form:

CREATE RELATION *rel_name*

$$\left\{ \begin{array}{l} \texttt{(}\ \textit{schema\_list}\ \texttt{)} \\ \texttt{LIKE}\ \textit{old\_rel} \end{array} \right\} \texttt{;}$$

*schema_term* ⇒ *attrib_name attrib_type*
[ **(** *attrib_len* **)**] [ **NOT NULL** ] ]
[ ' descriptor ' ]

The *attrib_name* must follow the valid naming rules. The attribute types, *attrib_type*, that may be selected are shown in Table 3-1. The *attrib_len* parameter is an integer that defines the length of the attribute. For numeric arrays, this is used only when the array has two or more elements. It is also possible to specify the **NOT NULL** option for each

**Table 3.1: Relational Attribute Types**

| *attrib_type* | DESCRIPTION | *attrib_len* |
|---|---|---|
| **INT** | Integer Value | Length of Array |
| **RSP** | Real, Single Precision Value | Length of Array |
| **RDP** | Real, Double Precision Value | Length of Array |
| **CSP** | Complex, Single Precision Value | Length of Array |
| **CDP** | Complex, Double Precision Value | Length of Array |
| **CHAR** | Character String | Number of Characters |

attribute. The meaning of **NULL** attributes is discussed later in this Chapter. Finally, you may attach a *descriptor* to an attribute. A *descriptor* is simply text which allows you to provide an amplified description of the meaning of the attribute. It is necessary for you to have the ***ADMINISTRATION*** privilege in order to create a Relation. The Relation may also use the schema previously defined for an existing Relation, *old_rel* by using the **LIKE** clause.

The use of the **CREATE RELATION** command is quite simple as shown in the next example.

Example 3-1. Create a new directory TUTOR, and in this directory create a Relation Q4S2 with the same schema as the Relation Q4STR[2] contained on the TESTEB: database.

```
eSh> MKDIR TUTOR;
eSh> CD TUTOR;
eSh> CREATE REL Q4S2
  2>   ( EID   INT,
  3>     SIGX  RSP,
  4>     SIGY  RSP,
  5>     TAUXY RSP );
        or
eSh> CREATE REL Q4S2 LIKE /RESULT/Q4STR[2];
```

✔ The /TUTOR/ Directory will be used for many of the examples in the remainder of this manual.

Attributes that are defined as arrays, those having *attrib_len* greater than one, are intended to store data that is exclusively used on an all-or-nothing basis. This might be the case, for instance, when storing transformation matrices.

Example 3-2. In subdirectory TUTOR, create a new Relation called TRANS. Each entry contains an identification number, TID, and a small 3x3 Matrix called T. Assume that the Matrix T will only be accessed in its entirety for computational purposes and can thus be stored as an array of RSP values. Also, add appropriate descriptors to the attributes.

```
eSh> CREATE RELATION TRANS
  2>   ( TID INT 'Transformation ID Number',
  3>     T   RSP(9) 'Transformation Matrix' );
```

## THE "NULL" FIELD CONCEPT

Many Relational databases include the concept of a **NULL** field within an entry of a Relation. This feature is useful for entities which have many attributes which can be logically grouped into subsets. A simple example is the case in which an EMPLOYEE Relation contains many attributes describing each employee. One attribute might be COMMISSION_EARNED. Clearly, this applies only to sales people who qualify for commission earnings. Therefore, this field is typically **NULL** for other classes of employees.

An ***eBase*** Relation may contain **NULL** fields when only a subset of data has been inserted into the entity, as described in Chapter 8. This occurs when insertions are performed using a projection that does not include all of the attributes of the Relation. There are certain ***eQL*** operations that can internally create such fields. If you ever see the word **NULL** appear in a display, it is because an operation has created such a field. It is possible to update entries which contain a **NULL** field.

## CREATING MATRICES

New matrices can also be created on the ***eBase*** database. This is done by using the command:

**CREATE MATRIX** *mat_name*

$$\left\{ \begin{array}{l} \texttt{(}\ \textit{mat\_attrib}\ \texttt{)} \\ \textbf{LIKE}\ \textit{old\_mat} \end{array} \right\}\ ;$$

where the *mat_name* is the name of the Matrix entity. A Matrix is further defined by its characteristics which include its Storage Mode, the Numeric Type of the data that it contains, its general Shape, and its ***Static Dimension***. The Static Dimension specifies the maximum number of Rows or Columns. This implicitly defines the Orientation of the entity.

These characteristics are specified with:

```
CREATE MATRIX mat_name
    { ( mat_attr_list ) }
    { LIKE old_mat      } ;
```

*mat_attr_term* ⇒ { **MODE** *mode_type* | **TYPE** *num_type* | **SHAPE** *shape* | { **ROWS** | **COLUMNS** } *number_of_rorc* }

Table 3-2 presents the available options for the Storage Mode, *mode_type* and Numeric Type, *num_type*. The Matrix *num_type*s are somewhat different from those of Relational entities. The *number_of_rorc* depends on the Storage Mode. The ***eBase*** Matrix storage options allow one dimension of a Matrix to be fixed and the other dimension to be dynamic. In the case of the Column-major Orientation, the number of columns is dynamic while the number of rows is fixed. For Row-major matrices, the opposite is true — the number of rows is dynamic and the number of columns static. As a result, if your Matrix is stored in Column-major mode, then the number of **ROWS** is entered; if it is in Row-major form, then the number of **COLUMNS** is entered. In both cases, these are entered as an integer value. Note that the ***ADMINISTRATION*** privilege is also required to create a Matrix and that a **LIKE** clause may be specified to use the attributes of an existing Matrix.

**Table 3.2: Matrix Attribute Types**

| SYMBOL | KEYWORD | DESCRIPTION |
|---|---|---|
| ***mode_type*** | **COMPRESS** | Compressed Storage Mode |
| | **UNCOMPRESS** | Uncompressed Storage Mode |
| ***num_type*** | **INT** | Integer Terms |
| | **RSP** | Real, Single Precision Terms |
| | **RDP** | Real, Double Precision Terms |
| | **CSP** | Complex, Single Precision Terms |
| | **CDP** | Complex, Double Precision Terms |
| ***shape*** | **RECTANGULAR** | $A_{ij}$ , i = 1,...,m; j=1,...,n |
| | **SQUARE** | $A_{ij}$ , i = 1,...,n; j=1,...,n |
| | **SYMMETRIC** | $A_{ij} = A_{ji}$, i = 1,...,n; j=1,...,n |
| | **DIAGONAL** | $A_{ij} = 0$ when $i \neq j$ |
| | **IDENTITY** | $A_{ii} = 1.0$, $A_{ij} = 0$ when $i \neq j$ |

Example 3-3. Create Matrix NEWKGG, in directory /TUTOR, with the same characteristics as KGG on the TESTEB: database:

```
eSh> CD ../MODEL;
eSh> CREATE MATRIX NEWKGG
  2>   ( MODE   COMPRESS,
  3>     TYPE   RDP,
  4>     SHAPE  SYMMETRIC,
  5>     ROWS   6 );
         or
eSh> CREATE MATRIX /MODEL/NEWKGG LIKE KGG;
```

## CREATING FREEFORM ENTITIES

New Freeform entities may be created as long as they are Schematic. Recall from Chapter 1 that this means they must have a homogeneous data type. You create a Freeform with the command:

**CREATE FREEFORM** *free_name*

$$\left\{ \begin{array}{l} \textbf{( TYPE } \textit{num\_type} \textbf{ )} \\ \textbf{LIKE } \textit{old\_free\_name} \end{array} \right\} ;$$

where the *free_name* is the entity name. Each Freeform entity has a single characteristic, its Numeric Type, *num_type*. The allowable values are shown in Table 3-3. Again, ***ADMINISTRATION*** privilege is required to create these entities and the **LIKE** clause may be specified to use the attributes of an existing entity.

Example 3-4. Create Freeform NEW_FREE, in directory /TUTOR, with the same characteristics as FREE on the TESTEB: database:

```
eSh> CD TUTOR;
eSh> CREATE FREEFORM NEW_FREE
  2>   ( TYPE  RDP );
         or
eSh> CREATE FREEFORM NEW_FREE LIKE /MODEL/FREE;
```

**Table 3.3: Freeform and Stream Numeric Data Types**

| SYMBOL | KEYWORD | DESCRIPTION |
|---|---|---|
| *num_type* | **INT** | Integer Terms |
| | **RSP** | Real, Single Precision Terms |
| | **RDP** | Real, Double Precision Terms |
| | **CSP** | Complex, Single Precision Terms |
| | **CDP** | Complex, Double Precision Terms |
| | **MIXED** | Heterogeneous Terms |

## CREATING STREAM ENTITIES

New Stream entities may be also be created as long as they are schematic. They are created with the command:

**CREATE STREAM** *stream_name*

{ **( TYPE** *num_type* **)** } **;**
{ **LIKE** *old_stream_name* }

where the *stream_name* is the entity name. Again, the Stream has a single characteristic, its Numeric Type, *num_type* also shown in Table 3-3. ***ADMINISTRATION*** privilege is required to create these entities and the **LIKE** clause may be specified to use the attributes of an existing entity.

## CREATING SUBSCRIPTED ENTITIES

When you create a subscripted entity of any class, the first creation determines the ***Dimensionality***, or number of subscripts, that will be used for **all** subsequent subscripted versions of the entity that you create. An error will occur if you attempt to specify an entity with the wrong Dimensionality in any ***eQL*** command.

Clean-up your /TUTOR/ Directory to restore your TESTEB: database to its initial state.

# 4. RETRIEVING DATA FROM RELATIONS

The most powerful use of the ***eQL*** language is its ability to retrieve data from an ***eBase*** database. This Chapter describes the commands used to retrieve data from Relations. The results of such a retrieval are called a ***Query***, and the act of the retrieval is often referred to as ***Querying***. The flexibility and options available for retrieval of data are of a complex nature. Therefore, this Chapter is organized in such a manner that successively more advanced uses of the **SELECT** command are defined and illustrated by many examples. It is best to read all of the material thoroughly and to work through the examples using the sample database.

## THE SELECT COMMAND

Data retrieval is accomplished with the **SELECT** command. The general form of this command is:

```
[ RELATION ] SELECT select_list FROM_part
                    [ WHERE_part ]
                    [ GROUP_part ]
                    [ ORDER_part ];
```

The *select_list* and *FROM_part* are required. The *WHERE_part*, *GROUP_part* and *ORDER_part* are each optional, but when any or all of them appear in the **SELECT** command they must appear in the order indicated. The richness of this command requires numerous examples. The examples which follow are presented in increasing levels of complexity, starting with the simplest form of the command:

**SELECT** *attrib_list* **FROM** *rel_name* **;**

where the *attrib_list* is a list of one or more attribute names, separated by commas, that are contained in the Relation specified by *rel_name*.

Example 4-1. Query all of the columns from Relation QUAD4.

```
eSh> CD /GEOM
eSh> SELECT EID,PID,G1,G2,G3,G4 FROM QUAD4;

     EID       PID        G1        G2        G3        G4
--------  --------  --------  --------  --------  --------
       1         1         1         2         7         6
       2         2         2         3         8         7
       3         1         3         4         9         8
       4         2         4         5        10         9

.... 4 Entries Selected
```

Note that there is a special shorthand that can be used when selecting all the attributes:

**SELECT * FROM GRID;**

Any subset of attributes may be selected, and the resulting output presents the attributes in the order in which they are named on the **SELECT** command.

Example 4-2. Select only the G1 and G4 attributes from Relation QUAD4, first selecting G4.

```
eSh> SELECT G4,G1 FROM QUAD4;

      G4        G1
--------  --------
       6         1
       7         2
       8         3
       9         4

.... 4 Entries Selected
```

The **SELECT** command also allows you to only select entries from the Relation which have distinct values for all **SELECT**ed attributes. The command is modified slightly as:

**SELECT** [ **DISTINCT** ] *attrib_list*
**FROM** *rel_name* **;**

Example 4-3. Select the distinct property identification numbers from Relation QUAD4.

```
eSh> SELECT DISTINCT PID FROM QUAD4;

     PID
--------
       1
       2

.... 2 Entries Selected
```

## THE OUTPUT FORMAT

Examples 4-1 and 4-2 illustrate the type of output that appears when a query is performed. A description of the general format of this output and the manner in which you may modify it are given in Chapter 12.

## ATTRIBUTES WHICH ARE ARRAYS

If an attribute of a Relation is an array , i.e. its length is greater than one, then all of the elements of the array are displayed when queried. Although the sample database does not contain a Relation with such attributes, consider the following example.

Example 4-4. Suppose that the Relation QUAD4 was created in a slightly different manner to place the grid point identification numbers (G1, G2, G3 and G4) in a single attribute called GRIDS of data type **INT** and length 4. The following output would result from the indicated query.

```
eSh> SELECT EID,PID,GRIDS FROM QUAD4;

     EID       PID                                  GRIDS(4)
--------  --------  ---------------------------------------
       1         1         1         2         7         6
       2         2         2         3         8         7
       3         1         3         4         9         8
       4         2         4         5        10         9

.... 4 Entries Selected
```

Individual elements within an array attribute cannot be referenced or selected on an individual basis, nor can they be used in any computational manner.

## REFERENCING A PATH DURING THE QUERY

The *rel_name* specified in the *FROM_part* of a query may include a full path name in addition to a simple entity name. Alternately, you may set a default path prior to the query or you may define a global symbol that are used in the name. These are shown in the following example.

Example 4-5. Suppose that the Relation QUAD4 resides in the directory /BIGPLANE/WING. The following methods may be used to query the Relation.

```
eSh> SELECT EID,PID,GRIDS FROM /BIGPLANE/WING/QUAD4;
            or
eSh> cd /bigplane/wing;
eSh> SELECT EID,PID,GRIDS FROM QUAD4;
            or
eSh> DEFINE PATH='/bigplane/wing/';
eSh> SELECT EID,PID,GRIDS FROM &PATH.QUAD4;
```

## QUALIFYING THE SELECTION

In many cases, it is desired to select only those entries in a Relation that satisfy a certain condition or combination of conditions. This is accomplished by using the following variation of the **SELECT** command:

**SELECT** *attrib_names* **FROM** *rel_name*
**WHERE** *search_condition*;

where a *search_condition* is a *logical_expression* of arbitrary complexity that involves any, or all, of the attributes in the Relation. The terms in a *logical_expression* are themselves *Relational_expression*s. All of the rules for evaluating logical and Relational expressions follow the standard rules of Fortran.

Example 4-6. Select those grid points from Relation GRID that have X coordinates greater that 3.0.

```
eSh> SET TOLERANCE TO 0.0;
eSh> SELECT * FROM GRID WHERE X > 3.0;

     GID       CID             X             Y             Z
--------  --------  ------------  ------------  ------------
       5         0   4.00000E+00   1.00000E+00   0.00000E+00
      10         0   4.00000E+00   0.00000E+00   0.00000E+00

.... 2 Entries Selected
```

Note that the SET TOLERANCE command is used to override your configuration default value for this and subsequent examples.

Note that when Relational operations are performed between floating point values it is possible for the numeric representations to cause testing failure. This is corrected by the use of a floating point **TOLERANCE** on the operations, as described in Chapter 13.

Table 4-1 lists the allowable logical and Relational operators that may be used in a *search_expression*. The next example illustrates the use of a logical combination of two Relational terms.

**Table 4-1. Logical and Relational Operators**

| OPERATOR | PURPOSE |
|---|---|
| **AND** | Logical conjunction |
| **OR** | Logical disjunction |
| **NOT** | Logical Negation |
| **=** | Equality |
| **<>** | Inequality |
| **<** | Less than |
| **>** | Greater than |
| **<=** | Less than or equal |
| **>=** | Greater than or equal |

| LOGICAL TRUTH TABLES | | |
|---|---|---|
| **AND** | **T** | **F** |
| **T** | **T** | **F** |
| **F** | **F** | **F** |
| | | |
| **OR** | **T** | **F** |
| **T** | **T** | **T** |
| **F** | **T** | **F** |

Example 4-7. Select the grid point identification number, `GID`, for all grid points with X-coordinates greater than `2.0` and Y-coordinates equal to `0.0`.

```
eSh> SELECT GID FROM GRID WHERE X > 2.0 AND Y = 0.0;

     GID
--------
       9
      10

.... 2 Entries Selected
```

Parentheses may be used to group and clarify more complex expressions appearing in the *`search_condition`*.

Example 4-8. Select the grid point identification number, `GID`, and X-coordinate for all grid points with X-coordinates greater than `2.0` and Y-coordinates equal to `0.0`, or whose `GID` is `1`.

```
eSh> SELECT GID,X FROM GRID
  2>     WHERE ( X > 2.0 AND Y = 0.0 )
  3>     OR      GID = 1;

     GID             X
--------  ------------
       1   0.00000E+00
       9   3.00000E+00
      10   4.00000E+00

.... 3 Entries Selected
```

## SELECTING FROM A SET

A special operator, **IN**, is available to allow a selection of data to be based upon a set of specific values. A ***Set*** is a finite group of values with consistent data types. You may also select from values not in the set by using the optional keyword **NOT**. The general form of the **WHERE** clause to do this is:

**WHERE** *attribute_name* [ **NOT** ] **IN (** *set_definition* **)**

where the *set_definition* is one or more values, separated by commas, for the *attribute_name* which are to be selected from the Relation.

Example 4-9. Select all grid points with X coordinates of 3.0 or 4.0 using a set definition.

```
eSh> SELECT GID,X,Y,Z FROM GRID WHERE X IN (3.0,4.0);

     GID             X             Y             Z
--------  ------------  ------------  ------------
       4   3.00000E+00   1.00000E+00   0.00000E+00
       5   4.00000E+00   1.00000E+00   0.00000E+00
       9   3.00000E+00   0.00000E+00   0.00000E+00
      10   4.00000E+00   0.00000E+00   0.00000E+00

.... 4 Entries Selected
```

## COMPARING TO A SET

It is also possible to compare an attribute to a set of values using the form:

**WHERE** *attribute_name rel_operator*

$$\left\{\begin{matrix}\textbf{ALL}\\ \textbf{SOME}\\ \textbf{ANY}\end{matrix}\right\} \textbf{(}\ \textit{subquery}\ \textbf{)}$$

Any of the *rel_operators* shown in Table 4-1 may be used in this expression. This comparison is used when you wish to quantify the members of a *subquery* which is described later in this Chapter.

## USING ARITHMETIC EXPRESSIONS

The last two sections have illustrated how logical and Relational expressions can be used to qualify the data to be selected from a Relation. ***eQL*** commands can also contain arithmetic expressions that combine attribute names, constants and functions. Such expressions can be used as selections. When this is done, the selection is called a ***Virtual Attribute***, because it only exists as the result of the query. All of the rules for arithmetic expression evaluation also follow the standard rules of Fortran.

**Table 4-2. Arithmetic Functions**

| FUNCTION | PURPOSE | ALLOWED INPUT TYPES | | | | |
|---|---|---|---|---|---|---|
| | | INT | RSP | RDP | CSP | CDP |
| **ABS(x)** | Absolute value | ■ | ■ | ■ | ■ | ■ |
| **ACOS(x)** | Inverse trigonometric cosine | ■ | ■ | ■ | | |
| **ASIN(x)** | Inverse trigonometric sine | ■ | ■ | ■ | | |
| **ATAN(x)** | Inverse trigonometric tangent | ■ | ■ | ■ | | |
| **CMPLX(x,y)** | Convert to CSP | ■ | ■ | ■ | | |
| **COS(x)** | Trigonometric sine | ■ | ■ | ■ | | |
| **COSH(x)** | Hyperbolic cosine | ■ | ■ | ■ | | |
| **DCMPLX(x,y)** | Convert to CDP | ■ | ■ | ■ | ■ | |
| **DBLE(x)** | Convert to RDP | ■ | ■ | ■ | ■ | ■ |
| **EXP(x)** | Exponential function $e^x$ | ■ | ■ | ■ | ■ | ■ |
| **INT(x)** | Convert to INT | ■ | ■ | ■ | ■ | ■ |
| **LOG(x)** | Natural (base e) logarithm | ■ | ■ | ■ | ■ | ■ |
| **LOG10(x)** | Common (base 10) logarithm | ■ | ■ | ■ | | |
| **MOD(x)** | Remainder | ■ | ■ | ■ | | |
| **REAL(x)** | Convert to RSP | ■ | ■ | ■ | ■ | ■ |
| **SIN(x)** | Trigonometric sine | ■ | ■ | ■ | | |
| **SINH(x)** | Hyperbolic sine | ■ | ■ | ■ | | |
| **SQRT(x)** | Square root | ■ | ■ | ■ | ■ | ■ |
| **TAN(x)** | Trigonometric tangent | ■ | ■ | ■ | | |
| **TANH(x)** | Hyperbolic tangent | ■ | ■ | ■ | | |

Example 4-10. List the grid point identification numbers and the distance that the grid point is from the origin for grid points having Y-coordinates of 1.0.

```
eSh> SELECT GID,SQRT(X**2+Y**2+Z**2) FROM GRID
  2>      WHERE Y = 1.0;

     GID  SQRT(X**2+Y**2+Z**2)
--------  --------------------
       1           1.00000E+00
       2           1.41421E+00
       3           2.23607E+00
       4           3.16228E+00
       5           4.12311E+00

.... 5 Entries Selected
```

Note that the column label is identified by the actual equation used to define the virtual attribute, and the field width is lengthened to accommodate it. Similarly, arithmetic expressions may be used as constraints in a *WHERE_part*.

Example 4-11 Select all element identification numbers from Relation Q4STR[1] where the Normal-Y stresses, SIGY, are at least twice the Normal-X stresses, SIGX.

```
eSh> SELECT EID,SIGY FROM /RESULT/Q4STR[1] WHERE
  2>      SIGY >= 2.0*SIGX;

     EID          SIGY
--------  ------------
       1   2.00000E+06
       2   1.00000E+07

.... 2 Entries Selected
```

Built-in arithmetic functions, also found in Fortran, are available to support the most frequently used operators. These are shown in Table 4.2 along with the allowable Numeric Types of their arguments. The results of each function are coerced to the required Numeric Type.

## THE JOIN OPERATION

Previous sections have illustrated the manner in which a single Relation is queried. A special operation, called the ***Join***, is available to allow the selection of data from more than one Relation and to combine this into a single result. The Relationship of entries in two or more Relations is determined by the field values in the entry. For example, both the QUAD4 Relation and the PSHELL Relation have an attribute called PID, the property identification number. This commonality of data allows entries in QUAD4 to be related to those in PSHELL.

***eBase*** is **not** a pure Relational database. The Relational schema are not independent of the entities. This means that it is possible for two Relations to have attributes of the same name which are not in fact from the same domain. You must therefore make certain that any operations between Relations are valid.

This process is defined more formally by the **JOIN** operation which is performed with a query as shown in the next example. Suppose that you want to know the material property identification number (MID) of QUAD4 element number 1. The QUAD4 Relation does not contain the MID, but the PSHELL Relation does. This is similar to the pointer concept familiar to those using finite element analysis systems. By inspection, it is seen that EID 1 has a PID of 1 and that, from the PSHELL Relation, PID of 1 has an MID of 101. This is the answer that is desired.

Example 4-12. Find the element identification number, EID, and property identification number, PID, for element 1 from the QUAD4 Relation and determine the material property identification number, MID, of the element from Relation PSHELL.

```
eSh> SELECT EID,QUAD4.PID,MID
  2>      FROM QUAD4,/MODEL/PSHELL
  3>           WHERE EID = 1
  4>           AND   QUAD4.PID = PSHELL.PID;

     EID       PID       MID
--------  --------  --------
       1         1       101

.... 1 Entry Selected
```

The syntax of the **SELECT** command is different from that seen earlier. The first difference occurs in the *attrib_list* being selected. The second attribute, QUAD4.PID, has a different form. It indicates that the PID attribute should be taken from the Relation QUAD4. The general form for the *attrib_list* is now modified to allow this. The list is comprised of *attrib_term*s which have the form:

*attrib_term* ⇒ [ Relation_*name* . ] *attribute_name*

The second difference appears in the *FROM_part*, which now has more than one Relation. Indeed, it lists all Relations from which the *attrib_list* will be drawn. Note that only attributes which appear in more than one Relation must be prefixed by the Relation name.

The *WHERE_part* specifies that only the entry from Relation QUAD4 with an EID of 1 will be retrieved, and then it specifies that if the PID of this entry is the same as the PID in a PSHELL entry, join the entries. The earlier portion of the **SELECT** has requested only that the EID, PID and MID fields of the resulting joined entry should be printed.

If there is only a single search condition in the *WHERE_part*, and this search condition is equality, then the operation is called an ***Equi-join***.

Example 4-13. Join the QUAD4 Relation to the PSHELL Relation and retrieve the element identification number, EID, property identification number, PID, and material identification number, MID, for each element.

```
eSh> SELECT EID,QUAD4.PID,MID
  2>      FROM QUAD4,/MODEL/PSHELL WHERE
  3>           QUAD4.PID = PSHELL.PID;
     EID       PID       MID
--------  --------  --------
       1         1       101
       2         2       201
       3         1       101
       4         2       201

.... 4 Entries Selected
```

When you perform a **JOIN** operation using Relations from more than one Directory, you use a special form of the *FROM_list* and the *SELECT_list*. The *FROM_list* specifies the names of the Relations in the form:

*rel_name_1* [ *alt_1* ] , *rel_name_2* [ *alt_2* ], ...

where *alt_1* is a Basic Name that is used to reference the specified *rel_name_1* everywhere within the query. For example, the query of Example 4-13 can be written as:

```
eSh> SELECT EID,A.PID,MID FROM
  2>        QUAD4 A,/MODEL/PSHELL WHERE
  3>           A.PID = PSHELL.PID;
```

where the Relation QUAD4 has been assigned the symbol A. The use of *alt* names in the *SELECT_list* is useful if the Relations referenced in the *FROM_list* are specified as fully qualified names. For example:

```
eSh> SELECT EID,A.PID,MID FROM
  2>        /GEOM/QUAD4 A,/MODEL/PSHELL B WHERE
  3>            A.PID = B.PID;
```

A special convention allows you to use the Basic Name of the Relation as the *alt* name:

```
eSh> SELECT EID,QUAD4.PID,MID FROM
  2>      /GEOM/QUAD4,/MODEL/PSHELL WHERE
  3>         QUAD4.PID = PSHELL.PID;
```

If several subscripted versions on an entity are involved in a query, then it is necessary to use an alt name to differentiate the selected attributes:

```
eSh> SELECT A.SIGX,B.SIGX FROM
  2>        /RESULT/Q4STR[1] A, /RESULT/Q4STR[2]
```

## GROUPING DATA DURING THE SELECTION

You may group the results of a query by using the *GROUP_part* in the **SELECT** command. The grouping operation partitions the Relation into a number of subsets based on identical values of one or more attributes. The *GROUP_part* follows the *WHERE_part* and has the form:

```
SELECT select_list
    FROM_part
    [ WHERE_part ]
    [ GROUP_part ]
    [ ORDER_part ];
```

*GROUP_part* ⇒ **GROUP BY** *attribute_list*

where the *attribute_list* is a list of one or more attribute names.

The grouping on floating point attributes (**RSP**, **RDP**, **CSP** or **RDP**) is not allowed because, even though a **TOLERANCE** may be specified, the partitioning is ambiguous.

Example 4-14. Query the Relation QUAD4 and group the results on the property identification number, PID.

```
eSh> SELECT PID FROM QUAD4
  2>     GROUP BY PID;

     PID
--------
       1
       2

.... 2 Entries Selected
```

The grouping of the results of a query is especially useful when group operations are performed on the results.

Example 4-15. Find the maximum Normal-X stress for each property identification number used for QUAD4 elements.

```
eSh> SELECT QUAD4.PID,MAX(Q4STR.SIGX)
  2>     FROM QUAD4,/RESULT/Q4STR[1]
  3>     WHERE QUAD4.EID = Q4STR.EID
  4>     GROUP BY QUAD4.PID;

     PID     MAX(SIGX)
--------  ------------
       1   2.00000E+06
       2   3.00000E+06

.... 2 Entries Selected
```

Notice that a join condition, described in more detail later in this Chapter, was required to attach the common attribute EID between the Relations QUAD4 and Q4STR[1]. The maximum stress was then extracted from each of the groups. Also note that the Q4STR[1] relation was fully qualified since in was not in the Current Working Directory.

## SORTING DATA DURING THE SELECTION

You may also order, or sort, the results of a query by using the *ORDER_part* in the **SELECT** command. The sorting operation may be performed on one or more of the attributes which appear in the *select_list*. When present, the *ORDER_part* appears last in the query. The form of this clause is:

```
SELECT select_list
    FROM_part
    [ WHERE_part ]
    [ GROUP_part ]
    [ ORDER_part ];
```

*ORDER_part>* ⇒ **ORDER BY** *sort_list*

where the *sort_list* is a list of one or more *sort_term*s, separated by commas, which have the form:

*sort_term>* ⇒ { *attribute_name* | *ordinal* } [ **ASCENDING** | **DESCENDING** ]

where each *attribute_name* must appear in the *select_list*. The alternate form *ordinal* allows the attribute to be specified as an integer value which corresponds to the order of the appearance of the attribute in the *select_list*. Either form may be used interchangeably — some attributes may be named while others are selected by their ***Ordinal***. It is required that Virtual Attributes be referenced by their Ordinal. All sorting is performed in ascending order, **ASCENDING**, unless the optional keyword **DESCENDING** is present for a particular attribute, in which case the sort is in descending order. Sorting is performed in the sequence specified in the *ORDER_part*.

Example 4-16. Query the Relation Q4STR[1] and order the results on SIGX and SIGY.

```
eSh> SELECT * FROM /RESULT/Q4STR[1]
  2>          ORDER BY SIGX,SIGY;

     EID          SIGX          SIGY         TAUXY
--------  ------------  ------------  ------------
       1   1.00000E+06   2.00000E+06   4.00000E+04
       4   2.00000E+06   1.00000E+06   5.00000E+04
       3   2.00000E+06   3.00000E+06   4.00000E+04
       2   3.00000E+06   1.00000E+07   6.00000E+03

.... 4 Entries Selected
```

If an *ORDER_part* is used with a *GROUP_part*, then the sorting operation is performed separately on each group created as illustrated in the next example.

Example 4-17. Repeat example 4-13 ordering the resulting stresses in descending order.

```
eSh> SELECT QUAD4.PID,MAX(Q4STR.SIGX)
  2>        FROM QUAD4,/RESULT/Q4STR[1]
  3>        WHERE QUAD4.EID=Q4STR.EID
  4>        GROUP BY QUAD4.PID
  5>        ORDER BY 2 DESC;

     PID     MAX(SIGX)
--------  ------------
       2   3.00000E+06
       1   2.00000E+06

.... 2 Entries Selected
```

## THE SUBQUERY

One of the most powerful aspects of the ***eQL*** language is its provision for subqueries to appear in **SELECT** commands. A ***Subquery*** is simply a complete **SELECT** that is used in the *WHERE_part* of another query command. Consider the following example.

Example 4-18. Determine the thickness of QUAD4 element EID=4 by first determining its property identification number, PID, and then finding the thickness used by elements with that property from Relation PSHELL.

```
eSh> SELECT PID FROM QUAD4 WHERE EID=4;

     PID
--------
       2

.... 1 Entry Selected
eSh> SELECT T FROM /MODEL/PSHELL WHERE PID=2;

           T
------------
 5.00000E-01

.... 1 Entry Selected
```

Note that in this case the result of the first query was used to determine the selection criteria for the second. The same result may be obtained by using a subquery.

Example 4-19. Determine the same information as in Example 4-16 using a single query with a subquery in the *WHERE_part*.

```
eSh> SELECT T FROM /MODEL/PSHELL WHERE
  2>      PID = ( SELECT PID FROM QUAD4 WHERE
  3>                EID=4 );

           T
------------
 5.00000E-01

.... 1 Entry Selected
```

The subquery in the last example returned a single value which was used as the selection criteria. Subqueries may also return a set of values that may be used to determine the selection.

Example 4-20. Find the shear stress , TAUXY, for Subcase 1 ,Q4STR[1], for all elements whose TAUXY is greater than all shear stresses in Subcase 2, Q4STR[2]. Select both the element EID and the TAUXY value.

```
eSh> SELECT EID,TAUXY FROM /RESULT/Q4STR[1] WHERE
  2>      TAUXY > ALL ( SELECT TAUXY FROM /RESULT/Q4STR[2] );

     EID         TAUXY
--------  ------------
       1   4.00000E+04
       3   4.00000E+04
       4   5.00000E+04

.... 3 Entries Selected
```

The result of the subquery is a list of all TAUXY from the Relation Q4STR[2]. Since it is desired to find the TAUXY which is greater than all of these values, the quantifier **ALL** must be used. More than one subquery may be used to perform the selection, as shown next.

Example 4-21. Find the QUAD4 elements having a thickness of 0.5 and whose Normal-X stress, SIGX, for Subcase 1, Q4STR[1], is greater than 2.0E+6.

```
eSh> SELECT EID FROM QUAD4
  2>  WHERE PID IN ( SELECT PID FROM /MODEL/PSHELL
  3>                     WHERE T = 0.5 )
  4>  AND EID IN ( SELECT EID FROM /RESULT/Q4STR[1]
  5>                     WHERE SIGX > 2.0E+6 );

     EID
--------
       2

.... 1 Entry Selected
```

Additionally, subqueries may be nested to create complex selection criteria.

Example 4-22. Find the GID and X-coordinate of each grid point that appears as the first grid point in a QUAD4 element which has a thickness of 0.1.

```
eSh> SELECT GID,X FROM GRID
  2>   WHERE GID IN ( SELECT G1 FROM QUAD4 WHERE
  3>              PID IN ( SELECT PID FROM /MODEL/PSHELL WHERE
  4>                       T = 0.1 ));

     GID             X
--------  ------------
       1   0.00000E+00
       3   2.00000E+00

.... 2 Entries Selected
```

## GROUP OPERATORS

In addition to the arithmetic and logical operators that have been described, ***eQL*** also supports certain group operators. These operators are so designated because they are performed on one or more attributes across a group of entries from a Relation. These group functions are shown in Table 4-3.

The use of the group functions is like that of normal arithmetic functions, namely the argument must be enclosed in parentheses.

Example 4-23. Find the average X-coordinate for all grid points having a Y-coordinate of 1.0.

```
eSh>  SELECT AVG(X) FROM GRID WHERE Y = 1.0;

       AVG(X)
------------
 2.00000E+00

.... 1 Entry Selected
```

Naturally, the *WHERE_part* restricts the set of data for which the function, **AVG** in this case, is applied. Multiple attributes may be computed

**Table 4-3. Group Operators**

| OPERATOR | DESCRIPTION |
|---|---|
| **AVG** | Computes the average value of the specified attribute expression for all entries satisfying the selection criteria. |
| **SUM** | Computes the arithmetic sum of the expression. |
| **MIN** | Finds the algebraically smallest value of the expression. |
| **MAX** | Find the algebraically largest value of the expression. |
| **COUNT** | Counts the number of entries satisfying the given conditions. |

with group functions,but these functions may **not** be used in other arithmetic expressions.

Example 4-24. Find the maximum Normal-X stress and minimum Normal-Y stress for Subcase 1.

```
eSh> SELECT MAX(SIGX),MIN(SIGY)
  2>      FROM Q4STR[1];

   MAX(SIGX)     MIN(SIGY)
------------  ------------
 3.00000E+06   1.00000E+06

.... 1 Entry Selected
```

When using group functions, it is not allowed to also select individual attributes such as:

```
eSh> SELECT GID,AVG(SQRT(X**2+Y**2+Z**2)) FROM GRID;
```

unless the individual attributes are later specified in a *GROUP_part* of the query. This is because attributes refer to fields within each entry whereas the group operation refers to all entries.

Example 4-25. Find all grid points which have the greatest X-coordinate.

```
eSh> SELECT GID FROM GRID
  2>      WHERE X = ( SELECT MAX(X) FROM GRID );

     GID
--------
       5
      10

.... 2 Entries Selected
```

There is one case in which single attribute names may appear in a **SELECT** command along with group operators. This is allowed when a *GROUP_part* is included as part of the selection criteria, as shown in the next example. It is this feature that differentiates the grouping and ordering operations. Also note that, in this case, the subquery returns a single value. Therefore, the strict equality is specified without resorting to one of the quantifiers **ALL**, **ANY** or **SOME**. If groups exist, then a single value, or list of values, is returned for each group.

Example 4-26. Find the maximum grid point identification number used as G3 in the QUAD4 Relation for each property identification number, PID.

```
eSh> SELECT PID,MAX(G3) FROM QUAD4
  2>      GROUP BY PID;

     PID   MAX(G3)
--------  --------
       1         9
       2        10

.... 2 Entries Selected
```

This operation is possible because the *GROUP_part* has partitioned the Relation into groups upon which the group operator **MAX** can function. Note that the *ORDER_part* does not allow this type of operation to be performed because it does not partition the data into distinct groups. The **COUNT** function is used to count the number of entries that satisfy the given selection criteria. The general form of this function is:

**COUNT (** { ***** | **DISTINCT** *attrib_name* } **)**

where the shorthand notation **(*)** indicates that all entries will be counted. If **DISTINCT** *attribute_name* is specified, then the number of entries which have distinct nonnull values for that attribute will be counted.

Example 4-27. Count the number of QUAD4 elements with a PID of 2.

```
eSh> SELECT COUNT(*) FROM QUAD4
  2      WHERE PID = 2;
COUNT(*)
--------
       2

.... 1 Entry Selected
```

Example 4-28. Count the number of unique values of PID in Relation QUAD4.

```
eSh> SELECT COUNT(DISTINCT PID) FROM QUAD4;

COUNT(DISTINCT PID)
-------------------
                  2

.... 1 Entry Selected
```

## INTERSECTION, UNION AND DIFFERENCE

There are three additional Relational operations that are often useful. Their primary use is to compare or merge two Relations having ***exactly*** the same schema. The intersection of two Relations is the set of entries that occur in both Relations that are identical in all fields. The command to perform this operation is:

**SELECT INTERSECTION OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

The **INTERSECT** command is a special form of **SELECT**. Note that only the distinct entries common to both Relations will result from this command and that no *WHERE_part*, *GROUP_part*, or *ORDER_part* may be used. If you wish to save the resulting Relation, the optional **AS** clause is used. Both of these facts are true of the next two commands, **SELECT UNION** and **SELECT DIFFERENCE** as well. This is of no significance because the new Relation, *rel_name_3*, is created from these commands. It may then be queried in any manner. You must have the ***ADMINISTRATION*** privilege on the databases containing *rel_name_1*, *rel_name_2*, and *rel_name_3*.

Example 4-29. Find the intersection of Relations Q4STR[1] and Q4STR[2].

```
eSh> CD /RESULT;
eSh> SELECT INTER OF Q4STR[1] AND Q4STR[2];
.... No Entries Selected
```

This example verifies that there are no common entries in the two versions of Q4STR!

The union of two Relations having the same schema is defined as the collection of all distinct entries that appear in either of them. This union is performed with the command:

**SELECT UNION OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

Example 4-30. Find the union of Relations Q4STR[1] and Q4STR[2].

```
eSh> SELECT UNION OF Q4STR[1] AND Q4STR[2];

     EID        SIGX        SIGY       TAUXY
--------  ----------  ----------  ----------
       1  1.00000E+6  2.00000E+6  4.00000E+4
       1  7.00000E+7  0.0         3.00000E+3
       2  1.00000E+6  1.00000E+7  6.00000E+3
       3  2.00000E+6  3.00000E+6  4.00000E+4
       3  5.00000E+7  0.0         1.00000E+4
       4  2.00000E+6  1.00000E+6  5.00000E+4

.... 6 Entries Selected
```

Finally, the difference of two Relations is the set of all entries in *rel_name_1* that are not in *rel_name_2*. The command to perform this operation is:

**SELECT DIFFERENCE OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

Example 4-31. Find the difference of Relations Q4STR[1] and Q4STR[2].

```
eSh> SELECT DIFF OF Q4STR[1] AND Q4STR[2];

     EID        SIGX        SIGY       TAUXY
--------  ----------  ----------  ----------
       1  1.00000E+6  2.00000E+6  4.00000E+4
       3  5.00000E+7  0.0         1.00000E+4

.... 2 Entries Selected
```

The important point to remember is that these operations function only for two Relations having the same schemata.

*This page is intentionally blank.*

# 5. GRAPHING RETRIEVED DATA

You should familiarize yourself with retrieving data from Relations as described in Chapter 4 prior to using the features described in this Chapter.

This Chapter describes the commands used to create graphs, or plots, from data which you retrieve from Relations. The flexibility and options available for the plotting of retrieved of data are as complex as those of the query. This Chapter is organized in the manner of Chapter 4 such that successively more advanced uses of the **XYPLOT** command are shown and illustrated by many examples.

The graphics functions are available only for Unix workstation versions of ***eShell*** under the X-Windowing System.

## THE PLOTTING WINDOWS

All of the plots that you create are drawn in one or more Graphic Windows. You can control the number of these windows and the plots that each of them contains.

For efficiency, each plot that you create is saved temporarily during your ***eShell*** session. This makes it possible for you to recover a plot that you may have deleted previously. Because the data are saved, you are not required to recreate the query that was used for creating earlier plot.

### Selecting the Plot Window

The plot windows are independent graphic display windows. You may have any number of display windows open at a given time. When you create your first plot, Plot Window 1 is created and your plot appears in this window. This is called the ***Active Window***.

You may define a new plot window with the command:

**SET ACTIVE WINDOW TO** *plot_win_id* **;**

where *plot_win_id* represents the number of the active plot window.

If *plot_win_id* already exists, then any new plot that you create will destroy the plot that previously occupied that window.

Once you have set the current plot window, it will be used for all subsequent plots that you create until you redefine it explicitly, or clear it using the command:

**CLEAR** { **PLOT WINDOW** *plot_win_id* | **ALL PLOT WINDOWS** } **;**

If you **CLEAR** all of the plot windows, the active window is reset to one. As noted above, you may replot the data is a plot window with the command:

**REPLOT** [ *plot_win_id* ];

If you do not specify a plot_win_id, then the data in the current active window are **REPLOT**ted.

## THE PLOTTING COMMANDS

Data retrieval and plot creation are accomplished with the 2-D plotting commands **XYPLOT**, **ADDCURVES**, and **MXYPLOT**. These are described in the following sections.

## THE XYPLOT COMMAND

Data retrieval and graph creation is accomplished with a single **XYPLOT** command. The general form of this command is:

**XYPLOT** *x_attrib, y_attrib_list FROM_part*
[ *WHERE_part* ]
[ *GROUP_part* ]
[ *ORDER_part* ];

The *x_attrib, y_attrib_list* and *FROM_part* are required. As with all queries, the *WHERE_part*, *GROUP_part* and *ORDER_part* are each optional, but when any or all of them appear in the **XYPLOT** command they must appear in the order indicated. The richness of this command requires numerous examples.

The simplest form of the command is:

**XYPLOT** *x_attrib , y_attrib_list* **FROM** *rel_name* **;**

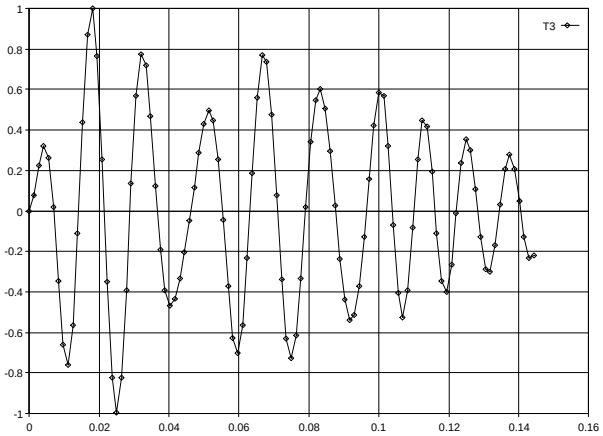
where *x_attrib* and the *y_attrib_list*, are, respectively, a single attribute and a list of one or more attribute names, separated by commas, that are contained in the Relation specified by *rel_name*. The *x_attrib* defines the x-axis of the graph and the *y_attrib_list* define the y-axis data that will be plotted. Each attribute in the list results in a separate curve on a single plot.

The maximum number of curves that may appear on a single plot is nine. Therefore, the number of elements specified in the y_attr_list may not exceed nine.

Example 5-1. Create a plot of displacement component, T3, versus time step, TIME, from Relation THISTORY.

```
eSh> CD PLOT;
eSh> XYPLOT TIME,T3 FROM THISTORY;
```

which results in the graph:

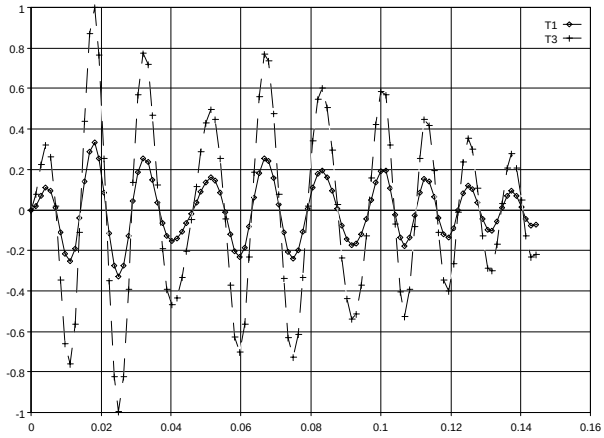



An example which shows multiple curves is illustrated next.

Example 5-2. Create a plot, using Relation THISTORY, of the two real displacement components, T1 and T3, versus the time step, TIME.

```
eSh> XYPLOT TIME,T1,T3 FROM THISTORY;
```

This results in the plot:




## THE GRAPH ELEMENTS

Examples 5-1 and 5-2 illustrate the type of output that appears when a query is performed. The XY-Plots consist of certain Graph Elements which you may control with various ***eShell*** commands. The available families of commands are described in the following sections.

### Symbols and Lines

When you create a graph, each data point is indicated by a symbol and the symbols are connected by lines. You may manipulate these options with the commands:

**SET SYMBOL TO** { **ON** | **OFF** } **;**

**SET DRAWLINE TO** { **ON** | **OFF** } **;**

### Titling

There are three different titling options for a graph. These are the graph frame title, the x-axis title, and the y-axis title. A legend is automatically placed on the plot that associates attribute names with their plot symbols. The titles are set using the commands:

**SET FTITLE TO** *'text'* **;**

**SET** { **XTITLE** | **YTITLE** } **TO** *'text'* **;**

In each command, *text* is any character text that you wish to use for titling information. Once you have set the titles, they remain until you explicitly clear them using the commands:

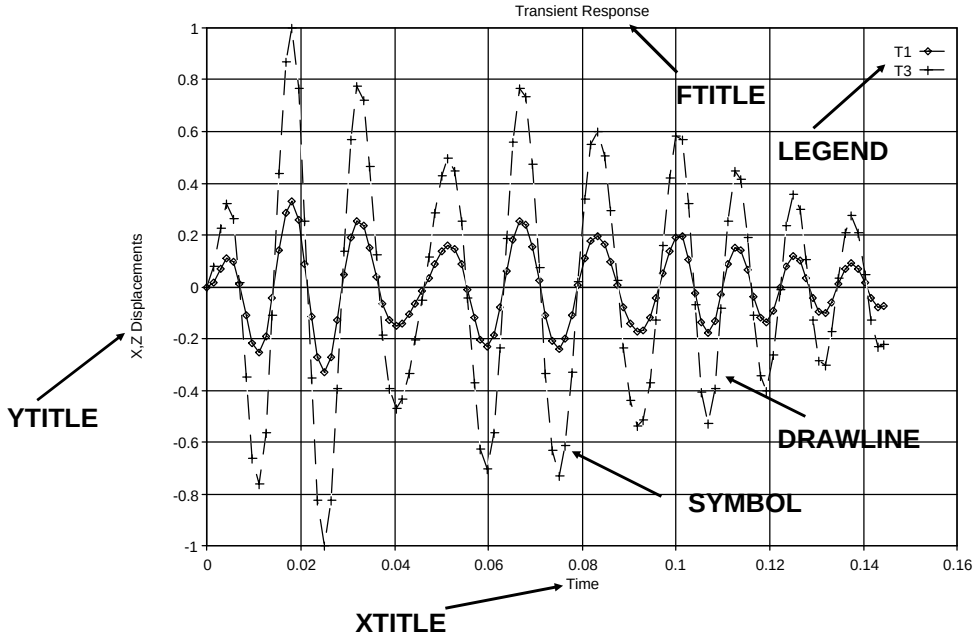
**CLEAR** { **FTITLE** | **XTITLE** | **YTITLE** } ;

Remember that the titles that you set remain in effect until you either redefine them or CLEAR them.

Example 5-3. Repeat example 5-2 adding titles to the graph.

```
eSh> SET FTITLE TO 'Transient Response';
eSh> SET XTITILE TO 'Time';
eSh> SET YTITLE TO 'X,Z Displacement';
eSh> XYPLOT TIME,T1,T3 FROM THISTORY;
```

which results in the following graph, with the graphics elements annotated:




## Customizing the Axes

There are several commands which allow you to control the x- and y-axes of your graphs which are described in this section.

**Changing the Axis Range.** You may change the ranges of the graph by giving the largest and smallest values that you want displayed on either axis. This is done with the commands:

**SET** { **XMIN** | **XMAX** | **YMIN** | **YMAX** } **TO** *value* **;**

Note that you may place multiple selections in as single command such as:

```
SET XMIN TO 0.0 XMAX TO 1000.0
    YMIN TO -2.0 YMAX TO 4.5;
```

Similar to the titling commands, these ranges remain in effect until cleared with:

**CLEAR** { **XMIN** | **XMAX** | **YMIN** | **YMAX** } **;**

Remember that axis ranges remain in effect until you either redefine them or CLEAR them.

**Logarithmic Scales.** In addition to specifying a data range, you may also have one or both axes represented by a logarithmic scale with the commands:

**SET** { **XLOG** | **YLOG** } **TO** { **ON** | **OFF** } **;**

You may toggle this option on or off for either or both axes.

**Axis Values and Ticmarks.** The number of values and ticmarks placed on the axes may be controlled using:

**SET** { **XDIV** | **YDIV** } **TO** *inc* **;**

where *inc* is the increment value for the selected axis.

**Drawing Grid Lines.** You may request that grid lines be drawn from each of the ticmarks on an axis with the command:

**SET GRID TO** { **ON** | **OFF** } **;**

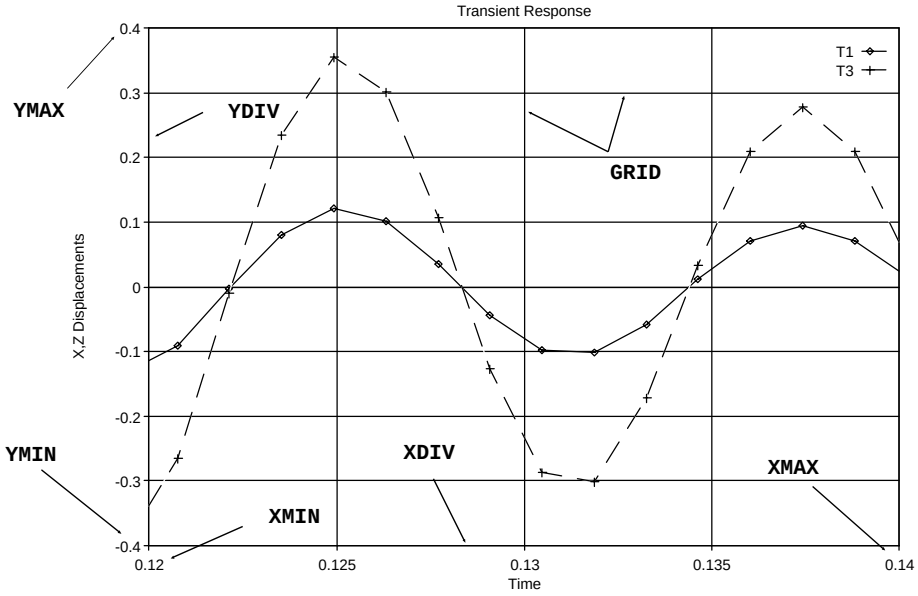
**Drawing the Axes.** You may toggle the drawing of the axes with the command:

**SET AXIS TO** { **ON** | **OFF** } b

Example 5-4. Repeat example 5-3, but this time create a blow-up between TIME values of 0.12 and 0.14. adding titles to the graph.

```
eSh> SET XMIN TO 0.12;
eSh> SET XMAX TO 0.14;
eSh> XYPLOT TIME,T1,T3 FROM THISTORY;
```

which results in the graph, with the graphics elements annotated:




## REFERENCING A PATH DURING THE QUERY

The *rel_name* specified in the *FROM_part* of a query may include a full path name in addition to a simple entity name. Alternately, you may set a default path prior to the query or you may define a global symbol that is used in the name. These are shown in the following example.

Example 5-5. Suppose that the Relation THISTORY resides in the directory /BIGPLANE/WING. The following methods may be used to plot the Relation.

```
eSh> XYPLOT TIME,T1 FROM /BIGPLANE/WING/THISTORY;
          or
eSh> CD /BIGPLANE/WING;
eSh> XYPLOT TIME,T1 FROM THISTORY;
          or
eSh> DEFINE PATH='/bigplane/wing/';
eSh> XYPLOT TIME,T1 FROM &PATH.THISTORY;
```

Refer to Chapter 2 for a detailed discussion of path names.

**Table 5-1. Logical and Relational Operators**

| OPERATOR | PURPOSE |
|---|---|
| **AND** | Logical conjunction |
| **OR** | Logical disjunction |
| **NOT** | Logical Negation |
| **=** | Equality |
| **<>** | Inequality |
| **<** | Less than |
| **>** | Greater than |
| **<=** | Less than or equal |
| **>=** | Greater than or equal |

| LOGICAL TRUTH TABLES | | |
|---|---|---|
| **AND** | **T** | **F** |
| **T** | **T** | **F** |
| **F** | **F** | **F** |
| | | |
| **OR** | **T** | **F** |
| **T** | **T** | **T** |
| **F** | **T** | **F** |

## QUALIFYING THE SELECTION

You may also create graphs for only those entries in a Relation that satisfy a certain condition or combination of conditions. This is accomplished by using the following variation of the **XYPLOT** command:

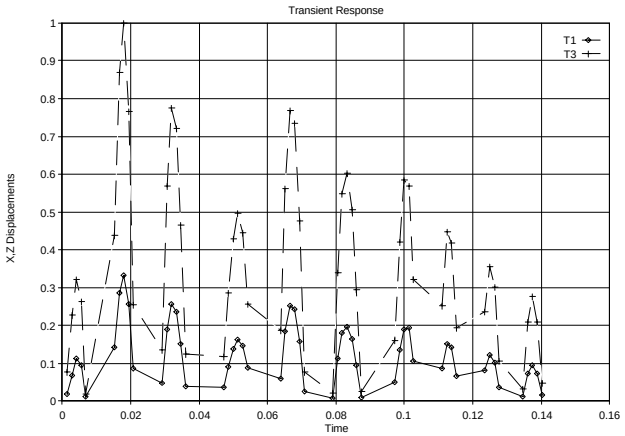
**XYPLOT** *x_and_y_list* **FROM** *rel_name*
**WHERE** *search_condition*;

where a *search_condition* is identical to that of the Relational query.

Example 5-6. Plot the T1 and T3 components in Relation GRID that have values greater that 0.0. First, clear the values of XMIN and XMAX that you set in Example 5-4.

```
eSh> CLEAR XMIN;
eSh> CLEAR XMAX;
eSh> XYPLOT TIME,T1,T3 FROM THISTORY WHERE T1 > 0.0;
```

This results in the plot:



When Relational operations are performed between floating point values it is possible for the numeric representations to cause testing failure. This is corrected by the use of a floating point **TOLERANCE** on the operations, as described in Chapter 13.

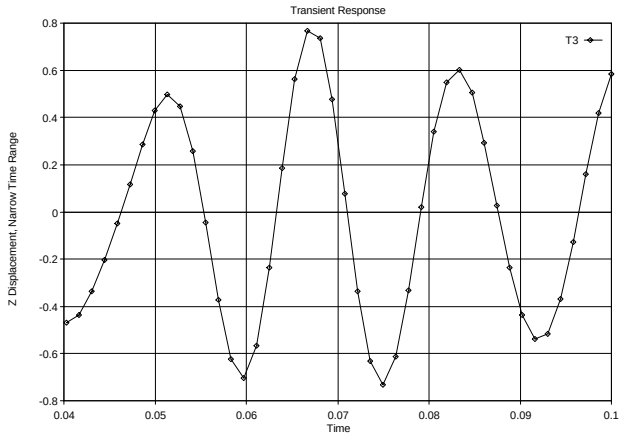
Table 5-1 reiterates the allowable logical and Relational operators that may be used in a *search_expression*.

The next example illustrates the use of a logical combination of two Relational operations.

Example 5-7. Plot the displacement component, T3, for all time steps greater than than 0.04 and less than 0.1.

```
eSh> SET YTITLE TO 'Z Displacement, Narrow Time Range';
eSh> XYPLOT TIME,T3 FROM THISTORY
  2>        WHERE TIME>0.04
  3>        AND   TIME<0.1;
```

This results in the following plot:

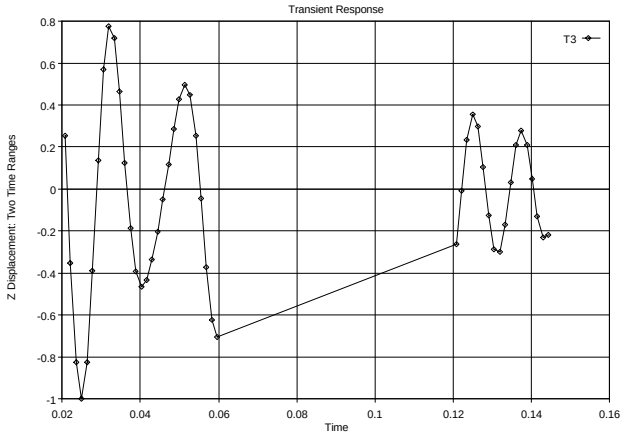



Parentheses may be used to group and clarify more complex expressions appearing in the *search_condition*.

Example 5-8. Plot the displacement component, T3, for all time steps greater than 0.02 and less than 0.06, or whose time step is greater than is 0.12.

```
eSh> SET YTITLE TO 'Z Displacement: Two Time Ranges';
eSh> XYPLOT TIME,T3 FROM THISTORY
  2>     WHERE ( TIME > 0.02 AND TIME < 0.06 )
  3>     OR      TIME > 0.12;
```

which results in:




## SELECTING FROM A SET

A special operator, **IN**, is available to allow selection of data based upon a set of specific values. A ***Set*** is a finite group of values with consistent data types. You may also select from values not in the set by using the optional keyword **NOT**. The general form of the **WHERE** clause to do this is:

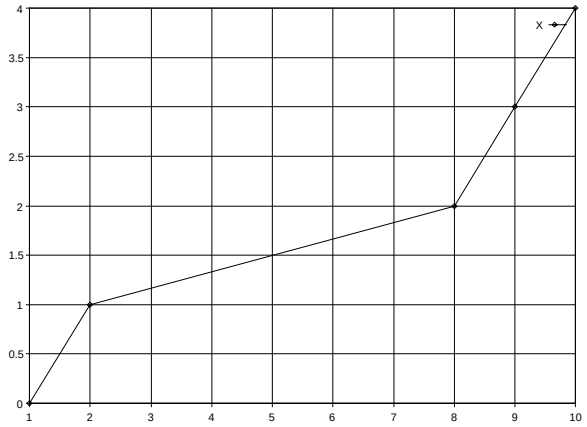
**WHERE** *attribute_name* [ **NOT** ] **IN (** *set_definition* **)**

where the *set_definition* is one or more values, separated by commas, for the *attribute_name* which are to be selected from the Relation.

Example 5-9. Plot the X coordinates versus the grid point ID from the Relation /GEOM/GRID where the GID has one of the values 1, 2, 8, 9 or 10.

```
eSh> CLEAR XTITLE;
eSh> CLEAR YTITLE;
eSh> CLEAR FTITLE;
eSh> CD /GEOM;
eSh> XYPLOT GID,X FROM GRID WHERE GID IN (1,2,8,9,10);
```

which results in:



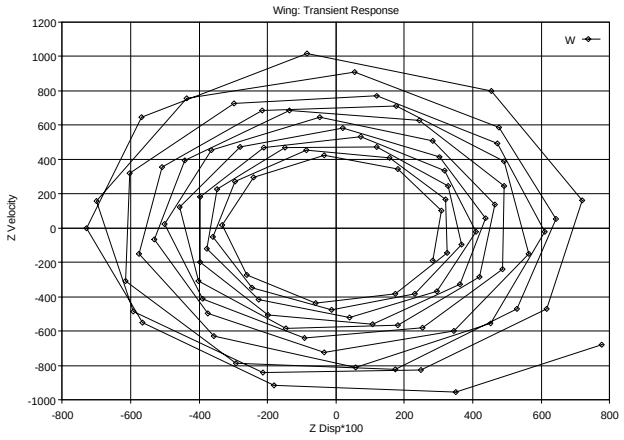
## USING ARITHMETIC EXPRESSIONS

As you have seen in Chapter 4, **eQL** commands can contain arithmetic expressions that combine attribute names, constants and functions. Such expressions can be used as selections which may be plotted. When this is done, the selection is called a ***Virtual Attribute***, because it only exists as the result of the query. All of the rules for arithmetic expression evaluation follow the standard rules of Fortran.

Example 5-10. Create a graph with two curves using data from Relation **TRANS**. The first is 100 times the z dispacement, attribute **Z**, and the second is the z velocity, attribute **W**. Label the plot accordingly.

```
eSh> CD /PLOT;
eSh> SET XTITLE TO 'Z Disp*100';
eSh> SET YTITLE TO 'Z Velocity';
eSh> SET FTITLE TO 'Wing: Transient Response';
eSh> XYPLOT 100*Z,W FROM TRANS;
```

The resulting graph is shown below:




## The ADDCURVES Command

There may be cases in which you would like to add additional curves to a plot that you have already created. You may do this with the command:

**ADDCURVES** *x_attrib*, *y_attrib_list FROM_part*
[ *WHERE_part* ]
[ *GROUP_part* ]
[ *ORDER_part* ] ;

As with the **XYPLOT** command, the *x_attrib*, *y_attrib_list* and *FROM_part* are required. And again, as with all queries, the *WHERE_part*, *GROUP_part* and *ORDER_part* are each optional, but when any or all of them appear in the **ADDCURVES** command they must

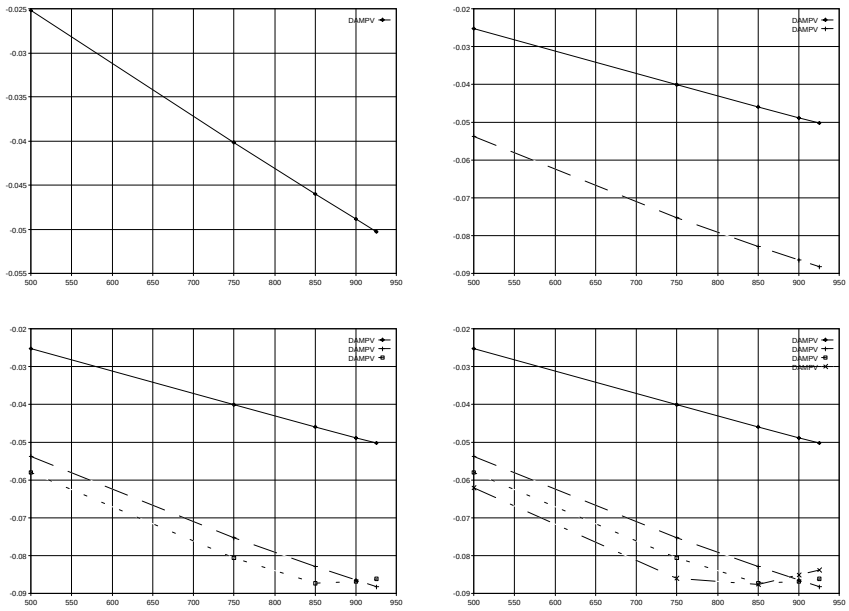
appear in the order indicated. The specified curve, or curves, will be added to the graph in the active plot window. If the domains of the *x_attrib* and *y_attrib_list* are different from those of the curves already plotted, then all data are rescaled using the union of the data domains.

Remember that the total number of curves on a plot may not exceed nine. Thus the sum of the original number of plots created by the **XYPLOT** command, plus those specified by the *y_attrib_list* of the **ADDCURVES** command cannot exceed this value.

Example 5-11: From directory `/plot` create a plot of `VELO` versus `DAMPV` from the relation `ROOTS` where `MODE=5` and `NITER=1`. After this is completed, successively add new curves for values of `NITER` equal to `3`, `5`, and `8`.

```
XYPLOT VELO,DAMPV FROM ROOTS WHERE NITER=1 AND MODE=5;
ADDCURVES VELO,DAMPV FROM ROOTS WHERE NITER=3 AND MODE=5;
ADDCURVES VELO,DAMPV FROM ROOTS WHERE NITER=5 AND MODE=5;
ADDCURVES VELO,DAMPV FROM ROOTS WHERE NITER=8 AND MODE=5;
```

Which results in the sequence of plots shown below.



## The MXYPLOT Command

Often the data stored in a relation represent repeated sets of values that vary for a given parameter. As an example, consider the following:

| PARAMETER | X | Y |
|---|---|---|
| 1 | 1.0 | 5.0 |
| 1 | 2.0 | 10.0 |
| 1 | 3.0 | 15.0 |
| 2 | 1.0 | 28.0 |
| 2 | 2.0 | 42.0 |
| 2 | 3.0 | 56.0 |
| 3 | 1.0 | 100.0 |
| 3 | 2.0 | 101.0 |
| 3 | 3.0 | 102.0 |

You would like to create three separate curves represented by the (**X**,**Y**) pairs appearing in the relation. Each curve will represent a different value of **PARAMETER**. This is done with the command:

```
MXYPLOT param, x_attrib, y_attrib FROM_part
                         [ WHERE_part ]
                         [ GROUP_part ]
                         [ ORDER_part ] ;
```

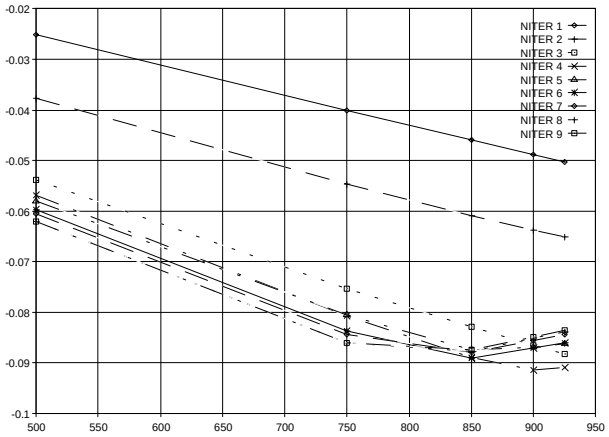
As with the other plot commands, the *x_attrib*, *y_attrib_list* and *FROM_part* are required, and the other clauses are optional.

This command results in one curve for each unique value of param retrieved from the specified relation. Only one attribute may be specified for the x-axis and y-axis of the plot. The axes are scaled based on the union of all of the plots that will be drawn.

Example 5-12: In directory `/plot`, the relation `ROOTS` contains data that represent velocity (`VELO`) and damping (`DAMPV`) curves. There are complete curves for each `MODE` and for each `ITER`ation in the solution procedure. Plot individual velocity-damping curves for each iteration for `MODE=5`.

```
MXYPLOT NITER,VELO,DAMPV FROM ROOTS
                         WHERE MODE=5;
```

This results in a single plot that includes nine curves, one for each value of `NITER`. The scaling of the plots encompasses the full range of values found in the data. The final plot is shown below.



# 6. INDEXING RELATIONAL ENTITIES

When you perform queries on very large Relational entities, they may require a significant amount of elapsed time. To improve response time, you may build one or more ***Indexes*** for a Relation. The methods for doing this are described in this Chapter.

## THE INDEX CONCEPT

An index for a Relational entity serves the same purpose as an index in a book — it allows faster access to the information that you wish to find. Similarly, **eShell** allows you to build one or more indexes for the attributes of a Relation so that the program may gain rapid access to the data. If a Relation has no index, then all queries must search through the data sequentially. As a result, indexes can dramatically increase the speed with which queries are performed.

## CREATING THE INDEX

You create an index for a Relation with the command:

```
CREATE [UNIQUE] INDEX ON   rel_name
                           ( attr_name_list ) ;
```

The *rel_name* is the name of an existing Relation, and *attr_name_list* specifies one or more attribute names for which the index will be created. These attributes form the ***Key*** for each entry in the Relation. Only attributes having data types of **INT** and **CHAR** may be used as indexes. To index other types of attributes, special artifices may be used as described later in this Chapter. A **UNIQUE** index is one in which each Key value is different from every other Key value. If the index Key is comprised of multiple attributes, although each individual

attribute may have a duplicated value, the combination of all the attributes in the Key must be unique.

Example 6-1. In Directory called TUTOR, make a copy of Relation /RESULT/Q4S called IDX_Q4S, and create an index for IDX_Q4S on attribute CASE.

```
eSh> MKDIR TUTOR;
eSh> CD TUTOR;
eSh> COPY /RESULT/Q4S TO IDX_Q4S;
eSh> CREATE INDEX ON IDX_Q4S(CASE);
.... Index Created
```

Indexes may also be used to enforce the uniqueness of one or more attributes in a Relation. Consider the following example:

Example 6-2. Now, make a copy of Relation /GEOM/GRID called IDX_GRID and create a unique index for it on attribute GID.

```
eSh> COPY /GEOM/GRID TO IDX_GRID;
eSh> CREATE UNIQUE INDEX ON IDX_GRID(GID);
.... Index Created
```

If this index has been created, then you may not insert data into the Relation which has the same index value as an existing one for the specified attributes.

Example 6-3. Add a new entry to Relation IDX_GRID which has GID=1.

```
eSh> INSERT INTO IDX_GRID
  2>      VALUES(1,1,5.,5.,5.);
ERR> Duplicate Key Encountered
```

Also note that the uniqueness check is applied when you perform an **UPDATE** operation on an indexed Relation.

## IMPROVING QUERY PERFORMANCE

Indexes for a Relation are only used when you perform a query which is qualified by a **WHERE** clause. In such cases, **eShell** uses the index data to rapidly process the entries in the entity. To understand how to model your data for optimum performance, consider the following Relation named RESULT.

| GID (INT) | TIME (RSP) | DISP_X (RSP) | DISP_Y (RSP) | DISP_Z (RSP) |
|---|---|---|---|---|
| 101 | 0.0 | 0.03 | 0.02 | 0.04 |
| 102 | 0.0 | 0.02 | 0.01 | 0.02 |
| ... | ... | ... | ... | ... |
| 101 | 0.1 | 0.05 | 0.01 | 0.03 |
| 102 | 0.1 | 0.07 | 0.03 | 0.06 |
| ... | ... | ... | ... | ... |

RESULT has five attributes, a grid point identification number, GID, a time, TIME, and three results values called DISP_X, DISP_Y and DISP_Z. Now, suppose that you want to perform the following query:

```
eSh> SELECT * FROM RESULT
  2>      WHERE GID=101;
....
....  Result of query is printed
....
```

If there are tens of thousands of entries in RESULT, it would take a large amount of time to search through the entries sequentially to extract the desired selection. By first creating an index:

```
eSh> CREATE INDEX ON RESULT(GID);
.... Index Created
```

the same query can be processed very rapidly. On the other hand, suppose you want:

```
eSh> SELECT * FROM RESULT
  2>      WHERE TIME=0.1;
....
....  Result of query is printed
....
```

This query also results in a sequential search which must check each entry. Since an attribute of type **RSP** cannot be indexed, it appears that query performance will be slow. If you intended to access the data in this manner, a simple artifice can be used to allow the creation of an attribute index. One way to do this is to model the data as:

| GID (INT) | TIME (RSP) | TIME_STEP (INT) | DISP_X (RSP) | DISP_Y (RSP) | DISP_Z (RSP) |
|---|---|---|---|---|---|
| 101 | 0.0 | 1 | 0.03 | 0.02 | 0.04 |
| 102 | 0.0 | 1 | 0.02 | 0.01 | 0.02 |
| ... | ... | ... | ... | ... | ... |
| 101 | 0.1 | 2 | 0.05 | 0.01 | 0.03 |
| 102 | 0.1 | 2 | 0.07 | 0.03 | 0.06 |
| ... | ... | ... | ... | ... | ... |

Here a new attribute called TIME_STEP of type **INT** has been introduced as a counter for the **RSP** attribute TIME. Now, to rapidly access the data, the following commands may be used:

```
eSh> CREATE INDEX ON RESULT(TIME_STEP);
.... Index Created
```

Finally, if you were directly accessing individual grid points at specific times, efficiency would be maximized by defining a two-attribute index:

```
eSh> CREATE INDEX ON RESULT(GID,TIME_STEP);
.... Index Created
```

Now, any query of the form:

```
eSh> SELECT * FROM RESULT
  2>      WHERE GID=102 AND TIME_STEP=2;
....
....  Result of query is printed
....
```

will be able to use the index for rapid access.

## INDEX PERFORMANCE

The use of indexes can dramatically improve performance when the Relational entity is accessed in the order of an attribute index or when entries are selected which have specified indexed attribute values. To see the impact of indexing, consider the following example.

Example 6-4: Study the performance of indexes using Relation /RESULT/BIGREL on the TESTEB: database.

```
eSh> CD ../RESULT;
eSh> DESCRIBE BIGREL;
.... Relation TESTEB:/RESULT/BIGREL
.... Schema is:
.... Attribute  Type    Len
.... ---------  ----  -----
.... ATT1       INT       1
.... ATT2       INT       1
.... ATT3       INT       1
.... Current Contents 10000 Entries
```

Now, perform a query of BIGREL and observe the elapsed time needed to retrieve the required entry:

```
eSh> START BIGREL;
eSh> SELECT ATT3 FROM BIGREL
  2>      WHERE ATT1 = 10000;

    ATT3
--------
   10000

.... 1 Entry Selected
```

Next, create an index for ATT1:

```
eSh> CREATE INDEX ON BIGREL (ATT1);
.... Index Created
```

Perform the query again and note the results. This has undoubtedly convinced you of the value of indexing when a Relation will be queried for specific entries.

If you have Relational entities with a large number of entries (many thousands) and you are going to perform many queries against the data, then create as many Indexes as necessary — the access speed-up will always be significant and the overhead needed to created the indexes is negligible by comparison.

## INDEX OVERHEAD

There is a certain amount of overhead when you create an index for a Relation. You will have noticed this while performing Example 6-4. However, also note that this overhead occurs only once and is well worth the investment if a large number of queries will be made on a specific Relation. At the same time, an index slows down **INSERT** and **UPDATE** operations because each of the indexes that you have created must also be updated.

## PURGING AN INDEX

Performance is improved if you purge any indexes which are not being used. This is done with the command:

**PURGE INDEX ON** *rel_name* **(** attr_name_list **) ;**

where *attr_name_list* is a list of the attributes used in a previously created index for the Relational entity *rel_name*. Recall that you may use the **DESCRIBE** command to obtain a listing of the indexes for the Relation as shown in the following example.

Example 6-5: Remove the indexes of Relation BIGREL that you created in Example 6-4.

```
eSh> DESCRIBE BIGREL;
.... Relation TESTEB:/RESULT/BIGREL
.... Schema is:
.... Attribute  Type    Len
.... ---------  ----  -----
.... ATT1       INT       1
.... ATT2       INT       1
.... ATT3       INT       1
.... Current Contents  x Entries
....  2 INDEXES EXIST:
....  1  on  ATT1
....  2  on  ATT1 and ATT2
eSh> PURGE INDEX ON BIGREL (ATT1);
.... Index has been Purged
eSh> PURGE INDEX ON BIGREL (ATT1,ATT2);
.... Index has been Purged
```

Clean-up your /TUTOR/ Directory to restore your TESTEB: database to its initial state.

# 7. RETRIEVING DATA FROM NON-RELATIONAL ENTITIES

In addition to the powerful querying operations for Relational entities described in the previous Chapter, you may also perform queries on the data for Matrix entities, Freeform entities, and Stream entities. The methods for doing this are described in this Chapter.

## MATRIX ENTITIES

This section describes the commands that you may use to retrieve data from Matrix entities. These commands are similar in nature to those available for querying Relations and allow the display of Matrix data in two different forms.

### The MATRIX Select Command

Data can be selected from Matrices in a manner similar to that used for Relations. The general form of the Matrix query command is:

```
MATRIX SELECT  [ format ]
               select_list
               FROM mat_name
               [ WHERE_part ] ;
```

The retrieval of the Matrix data depends on the Orientation of the Matrix. For Column-major Matrices, columns are retrieved, and for Row-major Matrices, rows are retrieved.

The *format* defines the display option that will be used. The display options are summarized in Table 7-1.

```
MATRIX SELECT [ format ]
    select_list
    FROM mat_name
    [ WHERE_part ] ;
```

*format* ⇒ ( { **FULL** | **BANDED** } )

If the display format is omitted, and the Storage Mode of the Matrix is **COMPRESSED**, then the **BANDED** option is used. For the **UNCOMPRESSED** Storage Mode, only the **FULL** option is used. The *select_list* allows you to select only some of the **COLUMN**s or **ROW**s of the entity:

```
MATRIX SELECT [ format ]
    select_list
    FROM mat_name
    [ WHERE_part ] ;
```

*select_list* ⇒ { *term_list* | * }

Where the *term_list* is a list of column or row identification numbers depending on the Orientation of the Matrix. Only the terms specified in this list will be extracted from the Matrix. You may select all values by using the asterisk (*****) shorthand notation. The Matrix entity that you are querying is specified by *mat_name*.

**Table 7-1. Matrix Print Options**

| OPTION | DESCRIPTION |
|---|---|
| **FULL** | Prints all terms in the Matrix. Default for **UNCOMPRESS**ed Matrices. |
| **BANDED** | Prints terms in each column beginning with the first nonzero term and ending with the last nonzero term. Defaultfor **COMPRESS**ed Matrices. |

The following examples illustrate the use of these options.

Example 7-1. Query the KGG Matrix obtaining a **FULL** listing.

```
eSh> MATRIX SELECT(FULL) * FROM KGG;

Column Major, Compressed, Real, Double Precision, Symmetric
   6 Rows, 6 Columns, Density = 52.0%
-------------------------------------------------------------
Column 1
100.000   200.000   0.00000   0.00000   0.00000   0.00000
-------------------------------------------------------------
Column 2
200.000   300.000   400.000   0.00000   0.00000   0.00000
-------------------------------------------------------------
Column 3
0.00000   400.000   500.000   600.000   0.00000   0.00000
-------------------------------------------------------------
Column 4
0.00000   0.00000   600.000   700.000   800.000   0.00000
-------------------------------------------------------------
Column 5
0.00000   0.00000   0.00000   800.000   900.000   1000.00
-------------------------------------------------------------
Column 6
0.00000   0.00000   0.00000   0.00000   1000.00   1100.00
```

For large, sparse Matrices, the **FULL** format can be overburdening. A more space efficient option is the **BANDED** format illustrated in the next example.

Example 7-2. Query the KGG Matrix obtaining a **BANDED** listing.

```
eSh> MATRIX SELECT * FROM KGG;

Column Major, Compressed, Real, Double Precision, Symmetric
   6 Rows, 6 Columns, Density = 52.0%
-------------------------------------------------------------
Column 1, Rows 1 through 2
100.000   200.000
-------------------------------------------------------------
Column 2, Rows 1 through 3
200.000   300.000   400.000
-------------------------------------------------------------
Column 3, Rows 2 through 4
400.000   500.000   600.000
-------------------------------------------------------------
Column 4, Rows 3 through 5
600.000   700.000   800.000
-------------------------------------------------------------
Column 5, Rows 4 through 6
800.000   900.000   1000.00
-------------------------------------------------------------
Column 6, Rows 5 through 6
1000.00   1100.00
```

## Qualifying the Columns or Rows

It is also possible to qualify, or constrain, the column or row selection with a Matrix *WHERE_part* of the form:

```
MATRIX SELECT [ format ]
    select_list
    FROM mat_name
    [ WHERE_part ] ;
```

*WHERE_part* ⇒ **WHERE** { **COLUMNS** | **ROWS** } **IN** *rorc_list*

where the *rorc_list* is a list of one or more integer row or column numbers depending on the Orientation of the Matrix entity — it is a row list for Column-major Matrices and a column list for Row-major Matrices.

Example 7-3. Query the KGG Matrix extracting Term (Row) 2 of columns 1, 2 and 3.

```
eSh> MATRIX SELECT 2 FROM KGG WHERE COLUMNS IN 1,2,3;

Column Major, Compressed, Real, Double Precision, Symmetric
   6 Rows, 6 Columns, Density = 52.0%
-------------------------------------------------------------
Column 1, Selected Row(s)
200.000
-------------------------------------------------------------
Column 2, Selected Row(s)
300.000
-------------------------------------------------------------
Column 3, Selected Row(s)
400.000
```

Note that there are no row identification numbers printed. In fact, the display options may not be specified when the *WHERE_part* is present. When the *WHERE_part* is used, the requested rows are printed using the **FULL** format.

See Chapter 12 for a description of the ways in which you can control the format of the Matrix query.

## FREEFORM ENTITIES

This section describes the commands that you may use to retrieve data from Freeform entities. These commands are similar in nature to those available for querying Matrix entities.

### The FREEFORM Select Command

Data can be selected from Freeform entities if they are Schematic. Recall from Chapter 1 that Schematic Freeform entities are those which have a single Numeric Type for all records within the entity. The general form of the Freeform query command is:

```
FREEFORM SELECT    select_list
                   FROM free_name
                   [ WHERE_part ] ;
```

The *select_list* allows you to select either all of the Data Values within a Record of the Freeform, or to select only specific values:

FREEFORM SELECT
**_select_list_**
FROM *free_name*
[ *WHERE_part* ]

$$select\_list \Rightarrow \left\{ \begin{array}{c} value\_list \\ * \end{array} \right\}$$

where *value_list* is a list of the Data Value sequence numbers that you wish to print. In the usual manner, all values can be selected by using the asterisk (*****) shorthand notation. *free_name* is the name of the Freeform entity that you are querying. The *WHERE_part*, described in the next section, allows you to specify additional selection constraints.

Example 7-4. Query the Freeform entity TESTFREE and print the first and third records.

```
eSh> FREEFORM SELECT * FROM TESTFREE WHERE RECORDS IN 1,3;

Real Single Precision, 5 Records, 50 Data Values in Longest
-----------------------------------------------------------
Record 1, 10 Data Values
1.00000  2.00000  3.00000  4.00000   ...
-----------------------------------------------------------
Record 3, 30 Data Values
3.00000  6.00000  9.00000  12.0000   ...
```

## Qualifying the Records

It is also possible to qualify, or constrain, the record selection with a Freeform *WHERE_part* of the form:

```
FREEFORM SELECT
    select_list
    FROM free_name
    [ WHERE_part ]
```

*WHERE_part* ⇒ **WHERE** { **RECORD** = *num* | **RECORDS IN** ( *record_list* ) }

where the *record_list* is a list of one or more integer record numbers. When selecting a single record, the alternate form which requires only the single *num*, may be used.

Example 7-5. Query Freeform TESTFREE selecting Data Values 1, 5, and 10 from Records 2 and 5.

```
eSh> FREE SELECT 1,5,10 FROM TESTFREE WHERE RECORDS IN 2,5;

Real Single Precision, 5 Records, 50 Data Values in Longest
------------------------------------------------------------
Record 2, 20 Data Values
2.00000  10.0000  20.000
------------------------------------------------------------
Record 5, 50 Data Values
5.00000  25.0000  50.0000
```

Note that the total length of the Record is always given. See Chapter 12 for a description of the ways in which you can control the format of the Freeform query.

## STREAM ENTITIES

This section describes the command that you may use to retrieve data from Stream entities. This command is very similar to the one for querying Freeform entities.

As with Freeform entities, data can also be selected from Stream entities if they are Schematic. The general form of the Stream query command is:

**STREAM SELECT** *select_list* **FROM** *stream_name* **;**

where the *select_list* is the same as that described for the **FREEFORM SELECT**, and *stream_name* is the name of an existing Stream entity.

Example 7-6. Query the Stream entity STRM and print the first three data values.

```
eSh> STREAM SELECT 1,2,3 FROM STRM;

Integer, 1000 Data Values
------------------------------------------------------------
        1        2        3
```

See Chapter 12 for a description of the ways in which you can control the format of the Stream query.

*This page is intentionally blank.*

# 8. INSERTING DATA INTO ENTITIES

This Chapter describes the ***eQL*** commands which are available to insert new data into ***eBase*** entities.

## ADDING NEW ENTRIES TO RELATIONS

New data entries can be added to existing Relations with the command:

```
INSERT INTO     [ RELATION ] rel_name
                [ ( proj_list ) ]
                value_part;
```

where *rel_name* is the name of an existing ***eBase*** Relation. The optional *proj_list* is a list of attributes of *rel_name*. If only a subset of the attributes is specified, then the selected attributes are called a ***Projection*** of the Relation. The *value_part* , which defines the new values to be inserted into the Relation, may take one of two forms:

```
INSERT INTO [REL]
    rel_name
    proj_list
    value_part
```

*value_part* ⇒ { **VALUES** ( *value_list* ) | *subquery* }

In the first form, the *value_list* contains the entry to be inserted into the Relation. The *value_list* must be entered in the order of the attributes as defined in the *proj_list*. The Numeric Type of each value must match the attribute type exactly. Attributes of type **CHAR** must be enclosed in apostrophes. The following example illustrates the use of this form of the **INSERT** command.

Example 8-1. Move to Directory TUTOR, make a copy of the GRID Relation called **INS_GRID** and insert a new grid point, ID=11, CID=0, and (X,Y,Z)=(5.0,0.0,0.0) into it.

```
eSh> CD /TUTOR;
eSh> COPY /GEOM/GRID TO INS_GRID;
eSh> INSERT INTO INS_GRID VALUES (11,0,5.0,0.0,0.0);
.... 1 Entry Inserted
```

If you specify a Projection, then all attributes which are not specified in the *proj_list* are given a Null value in the new entry.

If any of the attributes not in the Projection have been defined to be **NOT NULL**, then you may not **INSERT** into the Relation.

Example 8-2. Insert a new grid point, ID=12, and with X=10.0 into the **INS_GRID** Relation and verify the result.

```
eSh> INSERT INTO INS_GRID ( GID,X )
  2>    VALUES (12,10.0);
.... 1 Entry Inserted
eSh> SELECT * FROM INS_GRID WHERE GID=12;
  GID    CID           X           Y           Z
--------  -----  ----------  ----------  ----------
      12   NULL    10.00000        NULL        NULL
```

The second form of the *<value_part>* is a subquery. By using the subquery, selected portions of one Relation may be inserted into another, as shown in the next example.

Example 8-3. Create a new Relation called GRIDY1. This Relation has the same schema as GRID. Insert all grid points from INS_GRID with Y-coordinates of 1.0 into the new Relation.

```
eSh> CREATE RELATION GRIDY1 LIKE INS_GRID;
.... Relation TESTEB:/TUTOR/GRIDY1 Created
eSh> INSERT INTO GRIDY1
  2>    SELECT * FROM INS_GRID
  3>         WHERE Y=1.0;
.... 5 Entries Inserted
```

Note that the attribute names and Numeric Types are those defined in Chapter 1. The **INSERT** command requires the ***WRITE*** privilege.

## ADDING NEW COLUMNS OR ROWS TO MATRICES

An additional feature of ***eQL*** is the capability to insert new columns or rows into existing Matrix entities. The new column or row is actually appended to the current entity. That is, the column or row number inserted is one greater than the last one currently in the Matrix. You insert columns into a Column-major Matrix, and rows into a Row-major Matrix. In both cases, the command to perform this operation is:

**INSERT INTO MATRIX**
*matrix_name*
*new_value_list;*

The *new_value_list* is composed of one or more terms of the form:

INSERT INTO MATRIX
*mat_name*
***new_value_list***

*new_value_term* ⇒ **VALUES AT** { *row_id* | *col_id* } **(** *value_list* **)**

where the new column or row is entered a series of nonzero terms, *value_list*, within the new column or row. This is done by specifying the first nonzero *row_id* or *col_id* and following it with the list of data values. The values are assumed to occupy successive row positions beginning with *row_id*, or successive column positions beginning with *col_id*. The *value_list* is a list of Matrix terms whose Numeric Types are conformable to those defined in the entity schema.

Example 8-4. In Directory TUTOR, make a copy of Matrix entity KGG named INS_KGG and add a seventh column to it. Recall that KGG is stored in the Column-major Orientation and the COMPRESSED Storage Mode. Define the terms of the new column to be: KGG(2,7) = 25.0, KGG(5,7) = 30.0 and KGG(6,7) = 40.0. Verify the result.

```
eSh> COPY /MODEL/KGG TO INS_KGG;
.... 1 Entity Copied
eSh> INSERT INTO MATRIX INS_KGG
  2>    VALUES AT 2 (25.0)
  3>    VALUES AT 5 (30.0,40.0);
.... Column Added to TESTEB:/TUTOR/INS_KGG
eSh> MATRIX SELECT (FULL) * FROM INS_KGG
  2>    WHERE COLUMNS IN 7;
.... Warning, Matrix Shape may no longer be correct!
Column Major, Compressed, Real Double Precision, Symmetric
   6 Rows,   7 Columns,   Density = 52.0%
------------------------------------------------------------
Column 7
  0.0000   25.0000   0.0000   0.0000   30.0000   40.0000
```

The **WRITE** privilege is required to **INSERT** new data into Matrix entities.

When you **INSERT** into a Matrix entity, the topological Shape may be changed. You must use the **SET MATRIX SHAPE** command to make the Shape consistant with the data in the entity. This is only important if the Matrices will be used with ***eBase:matlib***.

## ADDING NEW RECORDS TO FREEFORM ENTITIES

You may also insert a new Record into a Schematic Freeform entity with the command:

**INSERT INTO FREEFORM** *free_name* **VALUES (** *value_list* **) ;**

The *value_list* is a list of Data Values whose Numeric Types are conformable to those defined in the entity schema.

Example 8-5. In Directory TUTOR, make a copy of Freeform entity TESTFREE named INS_FREE, add a sixth Record to it and verify the result:

```
eSh> COPY /MODEL/TESTFREE TO INS_FREE;
.... 1 Entity Copied
eSh> INSERT INTO FREEFORM INS_FREE
  2>    VALUES (100.,200.,300.,400.,500.);
.... Record Added to TESTEB:/TUTOR/INS_FREE
eSh> FREEFORM SELECT * FROM INS_FREE WHERE RECORDS IN 6;

Real Single Precision, 6 Records, 50 Data Values in Longest
-----------------------------------------------------------
Record 6, 5 Data Values
100.0  200.0  300.0  400.0  500.0
```

The **WRITE** privilege is also required to **INSERT** new data into Freeform entities.

When you **INSERT into a** Freeform entity with *n* Records, the new Record automatically becomes Record *n+1.*

## ADDING DATA VALUES TO STREAM ENTITIES

Additional Data Values beyond those existing in a Schematic Stream entity may be inserted using command:

**INSERT INTO STREAM** *stream_name* **VALUES (** *value_list* **) ;**

The *value_list* will be inserted into the Freeform entity *stream_name* beginning at the next available position, that is, one position beyond the last Data Value in the entity. As with Schematic Freeform entities, all of the Data Values must be conformable to the correct Numeric Type for Schematic Stream entities.

Example 8-6. Copy the Stream entity STRM to INS_STRM, add 3 new Data Values and verify the result:

```
eSh> COPY /MODEL/STRM TO INS_STRM;
.... 1 Entity Copied
eSh> INSERT INTO STREAM INS_STRM
  2>    VALUES (1001,1002,1003);
.... 3 Data Values Added to TESTEB:/TUTOR/INS_STRM
eSh> STREAM SELECT 1001,1002,1003 FROM INS_STRM;

Integer, 1003 Data Values
-------------------------------------------------------
    1001    1002    1003
```

The **WRITE** privilege is also required to **INSERT** new data into Stream entities.

✔ Clean-up your /TUTOR/ Directory to restore your TESTEB: database to its initial state.

*This page is intentionally blank.*

# 9. UPDATING ENTITY DATA

In this Chapter, commands which allow data in Relational, Matrix, Free-form and Stream entities to be updated are described. Also, the command for altering the schema of an existing Relation is given.

## UPDATING RELATIONAL ENTITIES

It is possible for you to update any fields in a selected Relational entries with the command:

**UPDATE** [ **RELATION** ] *rel_name*
**SET** *value_list*
[ *WHERE_part* ];

The *value_list* is a list of *value_term*s separated by commas, which are used to define values for one or more attributes in the entity. There are two basic forms of this list:

UPDATE [REL] *rel_name*
SET ***value_list***
*WHERE-part*

$$value\_term \Rightarrow \left\{ \begin{array}{l} att\_name = value \\ att\_name = expression \end{array} \right\}$$

where the *att_name* is an attribute name in the selected Relation, *rel_name*. The attribute can be assigned a *value* directly, or it may be computed by an arithmetic *expression*. The *expression* may contain any of the functions described in Chapter 4, and it may use any of the attributes contained in *rel_name*.

The optional *WHERE_part*, which has the standard form described in Chapter 4, can be used to update a specific range of selected data entries. If it is not used, then all of the entries in the entity are updated as specified.

Example 9-1. In Directory TUTOR, copy the Relation QUAD4 to NEW_QUAD4, change the property identification number of all element to the value 1, and verify the change.

```
eSh> COPY /GEOM/QUAD4 TO NEW_QUAD4;
.... 1 Entity Copied
eSh> UP NEW_QUAD4 SET PID=1;
.... 10 Entries Updated
eSh> SELECT * FROM NEW_QUAD4;
     EID       PID        G1        G2        G3        G4
--------  --------  --------  --------  --------  --------
       1         1         1         2         7         6
       2         1         2         3         8         7
       3         1         3         4         9         8
       4         1         4         5        10         9

.... 4 Entries Selected
```

The modification of more than one attribute is accomplished simply by listing all of the new values desired in the **UPDATE** command as illustrated in the next example.

Example 9-2. Copy the Relation PSHELL to NEW_PSHELL and change the material property identification number and the thickness to 501 and 0.25, respectively. Verify the results.

```
eSh> COPY /MODEL/PSHELL TO NEW_PSHELL;
.... 1 Entity Copied
eSh> UPDATE PSHELL SET MID=501,T=0.25;
.... 3 Entries Updated
eSh> SELECT * FROM PSHELL;

  PID    MID             T
--------  --------  ------------
       1       501   2.50000E-01
       2       501   2.50000E-01
       3       501   2.50000E-01

.... 3 Entries Selected
```

It is also possible to update a set of entries simultaneously by using a *WHERE_part* that contains the appropriate collection of conditions.

Example 9-3. Make a copy of Relation GRID called NEW_GRID, and translate the grid point having an X-coordinate of 2.0 and a Y-coordinate of 1.0 to the location having Y=2.0.

```
eSh> COPY /GEOM/GRID TO NEW_GRID;
.... 1 Entity Copied
eSh> UPDATE NEW_GRID SET Y=Y+1.0 WHERE X=2.0 AND Y=1.0;

.... 1 Entry Updated
```

Finally, subqueries may be used in the **UPDATE** command to determine the range of entries to be updated, as shown in the next example.

Example 9-4. Change the property identification number of all NEW_QUAD4 elements having normal-y stresses greater than 2.0E+6 in Subcase 1 to PID=3.

```
eSh> UPDATE NEW_QUAD4 SET PID=3
  2>      WHERE EID IN
  3>      ( SELECT EID FROM /RESULT/Q4STR[1]
  4>           WHERE SIGY > 2.0E+6 );

.... 2 Entries Updated
eSh> SELECT EID,PID FROM NEW_QUAD4 WHERE PID=3;

     EID       PID
--------  --------
       2         3
       3         3

.... 2 Entries Selected
```

In order to **UPDATE** a Relation, it is necessary to have the ***WRITE*** privilege for the database.

## UPDATING MATRIX ENTITIES

It is also possible to selectively update columns or rows of Matrix entities subject to specific restrictions. To do this, the command:

**UPDATE MATRIX** *matrix_name value_list*
**WHERE** { **ROW** | **COLUMN** } **=** *rorc_id* **;**

is used. The *matrix_name* must be the name of an existing Matrix entity. The new data values are given in the *value_list* which is composed of a list of *value_term*s:

UPDATE MATRIX *mat_name*
SET *rorc_id* TO
***value_list***

*value_term* ⇒ **SET VALUES AT TERM** *corr_id* **TO (** *values* **)**

The *corr_id* is an integer value indicating the starting column or row position of the numbers found in the list of *values*. Special rules, described in the next section, must be obeyed when specifying the *corr_id* and *values* for Matrix entities which use the **COMPRESS**ed Storage Mode. The next example shows how a Matrix column may be updated.

Finally, the *rorc_id* is the column or row identification number that will be updated. For the Column-major Orientation, it indicates a column, and for the Row-major Orientation, a row.

Example 9-5. Copy Matrix entity KGG to NEW_KGG, and change NEW_KGG(3,3) to the value 20.0. Query the original value and verify the change.

```
eSh> COPY /MODEL/KGG TO NEW_KGG;
.... 1 Entity Copied
eSh> MATRIX SELECT 3 FROM NEW_KGG WHERE COLUMNS IN 3;

Column Major, Compressed, Real Double Precision, Symmetric
   6 Rows,   6 Columns,   Density = 52.0%
-------------------------------------------------------------
Column 3, Selected Row(s)
500.000

eSh> UPDATE MATRIX NEW_KGG
  2>     SET VALUES AT TERM 3 TO (20.0) WHERE COLUMN = 3;

.... 1 Column Updated in TESTEB:/TUTOR/NEW_KGG

eSh> MATRIX SELECT 3 FROM NEW_KGG WHERE COL IN 3;
Column Major, Compressed, Real Double Precision, Symmetric
   6 Rows,   6 Columns,   Density = 52.0%
-------------------------------------------------------------
Column 3, Selected Row(s)
 20.0000
```

## RESTRICTIONS ON MATRIX UPDATING

If a Matrix entity has been created with the **COMPRESS**ed Storage Mode, then only terms that were initially placed in the Matrix can be updated. For the **UNCOMPRESS**ed Storage Mode this is not a problem because all of the terms exist by definition. An attempt to update data in the former case will lead to an error condition as shown in the next example.

Example 9-6. Set NEW_KGG(1,5) to 5.0

```
eSh> UPDATE MATRIX NEW_KGG
  2>      SET VALUES AT TERM 1 TO (5.0) WHERE COLUMN = 5;
ERR> Cannot Update Matrix NEW_KGG at Column/Row 5 Row/Column 1
```

The ***WRITE*** privilege is also required to **UPDATE** a Matrix entity.

## UPDATING FREEFORM ENTITIES

Freeform entities may be updated in a limited manner with the command:

**UPDATE FREEFORM** *free_name*
**SET** *position* **TO** *value_list*
**WHERE RECORD =** *rec_id* **;**

where the *free_name* is the name of an existing Freeform entity. The *value_list* is a list of one or more values that will replace all of the Data Items in *rec_id* beginning with *position*. Finally, *rec_id* is the Record that you wish to update.

Example 9-7. Copy Freeform entity TESTFREE to UP_FREE and change the second and third Data Values in the first Record to the values 22.2 and 33.3, respectively, and verify the change.

```
eSh> COPY /MODEL/TESTFREE TO UP_FREE;
.... 1 Entity Copied
eSh> UP FREE UP_FREE SET 2 TO 22.2,33.3 WHERE RECORD = 1;
.... Record Updated in TESTEB:/TUTOR/UP_FREE
eSh> FREE SELECT 1,2,3,4,5 FROM UP_FREE WHERE RECORD = 1;
Real Single Precision, 5 Records, 50 Data Values in Longest
------------------------------------------------------------
Record 1, 10 Data Values
1.00000  22.2000  33.3000  4.00000   5.00000
```

The entity must be a Schematic Freeform, and the *value_list* Numeric Types must be the same as that defined for the entity. The **UPDATE FREEFORM** command requires the ***WRITE*** privilege.

You may not extend the length of the Freeform Record when you perform the **UPDATE FREE** operation.

## UPDATING STREAM DATA

It is also possible for you to update any existing Data Item in a selected Stream entity with the command:

**UPDATE STREAM** *stream_name*
**SET** *position* **TO** *value_list*;

where the *stream_name* is the name of an existing Stream entity and *position* is the location of the first Data Item to be changed. The *value_list* is a list of one or more values that will replace all of the Data Items beginning with *position*.

Example 9-8. Copy Stream entity STRM to UP_STRM and change Data Values 110 through 115 to those indicated and verify the change.

```
eSh> COPY /MODEL/STRM TO UP_STRM;
.... 1 Entity Copied
eSh> UP STREAM UP_STRM SET 110 TO 1,1,1,1,1,1;
.... 5 Data Values Updated in TESTEB:/TUTOR/UP_STRM
eSh> STREAM SELECT 110,111,112,113,114,115 FROM UP_STRM;

1000 Integer Data Values
------------------------------------------------------------
       1       1       1       1       1       1
```

The **UPDATE STREAM** command requires the ***WRITE*** privilege.

You may not extend the length of the Stream entity when you perform the **UPDATE STREAM** operation. To do this, you use the **INSERT INTO STREAM** command, also described in Chapter 8.

## CHANGING THE SCHEMA OF A RELATION

It is also possible to change the schema of an existing Relation by adding a new attribute to it. This is done with the command:

**ALTER** *rel_name* **ADD (** *new_schema_list* **);**

where the *new_schema_list* is composed of *schema_term*s, already defined in Chapter 3, for the new attributes being added to *rel_name*.

Example 9-9. Using the Relation NEW_GRID created previously, add a new attribute to this entity which will be used to contain the distance of the grid point from the origin, DIST, as a real, single precision value. Verify that the attribute was added.

```
eSh> ALTER NEW_GRID ADD ( DIST RSP );
.... Attribute DIST Added to NEW_GRID
eSh> DESCRIBE NEW_GRID;

.... Attribute  Type    Len  Null  Descriptor
.... ---------  ----  -----  ----  ----------
....       GID   INT      1
....       CID   INT      1
....         X   RSP      1
....         Y   RSP      1
....         Z   RSP      1
....      DIST   RSP      1
.... Current Contents  10 Entries
```

It is necessary to have ***ADMINISTRATION*** privilege in order to **ALTER** the schema of a Relation.

Note that this operation can be quite expensive because a new Relation is created and the contents of the original are copied to the new one with the specified attributes added.

The fields for all new attributes are defined as **NULL** following the operation. In order to add data to these fields, the **UPDATE** or **INSERT** commands must be used.

Example 9-10. Query the Relation NEW_GRID for values of GID less than 4. Then, update the DIST field with the distance value and perform the same query on the result.

```
eSh> SELECT * FROM NEW_GRID
  2>    WHERE GID < 4;
    GID       CID         X         Y         Z      DIST
-------  --------  --------  --------  --------  --------
      1         0   0.00000   1.00000   0.00000      NULL
      2         0   1.00000   1.00000   0.00000      NULL
      3         0   2.00000   1.00000   0.00000      NULL

eSh> UPDATE NEW_GRID
  2>    SET DIST = SQRT(X**2+Y**2+Z**2);
.... 10 Entries Updated
eSh> SELECT * FROM TO NEW_GRID
  2>    WHERE GID < 4;
  GID    CID         X         Y         Z      DIST
-------  --------  --------  --------  --------  --------
      1         0   0.00000   1.00000   0.00000   1.00000
      2         0   1.00000   1.00000   0.00000   1.41421
      3         0   2.00000   1.00000   0.00000   2.23607
```

✔ Clean-up your /TUTOR/ Directory to restore your TESTEB: database to its initial state.

*This page is intentionally blank.*

# 10. REMOVING DATA FROM eBase

This Chapter describes the ***eQL*** commands that are used to remove entities and their data from the ***eBase*** database. Complete entities, including both their schema and contents, may be purged. Alternately, all, or selected, data entries may be deleted without removing the entity itself.

## REMOVING AN ENTITY

Any database entity may be removed from ***eBase*** with the command:

**PURGE** { **RELATION** | **MATRIX** | **FREEFORM** | **STREAM** } *entity_name* **[ ALLVER ];**

where the *entity_name* must refer to an existing entity of the selected class. This command removes the Data Component of the entity and then removes the entity Name and Schema from the database. The **ALLVER** option allows you to **PURGE** all Subscripted versions of the entity.

Because this is a potentially dangerous command, you are asked to confirm the **PURGE** activity. This is true for all of the commands described in this Chapter.

## REMOVING ENTRIES FROM RELATIONS

***eQL*** allows you to selectively remove entries from ***eBase*** Relations. This is done with the command:

**DELETE FROM** [ **RELATION** ] *rel_name* [ *WHERE_part* ];

where *rel_name* is the name of an existing Relation. The entries to be deleted can be selected with a *WHERE_part* in the usual manner. The *WHERE_part* is optional and if it is not specified, then all data entries are deleted from the Relation.

Example 10-1. In Directory TUTOR, make a copy of Relation GRID and delete entries which have Y-coordinates equal to 0.0. and verify the results.

```
eSh> CD /TUTOR;
eSh> COPY /GEOM/GRID TO DEL_GRID;
.... 1 Entity Copied
eSh> DELETE FROM DEL_GRID
  2>       WHERE Y = 0.0;
eSh> Enter YES to delete 5 Entries, leaving 5 Entries: YES
.... 5 Entries Deleted from TESTEB:/TUTOR/DEL_GRID
eSh> SEL * FROM DEL_GRID;
     GID       CID           X           Y           Z
--------  --------  ----------  ----------  ----------
       1         0         0.0         1.0         0.0
       2         0         1.0         1.0         0.0
       3         0         2.0         1.0         0.0
       4         0         3.0         1.0         0.0
       5         0         4.0         1.0         0.0

.... 5 Entries Selected
```

It is equally permissable for subqueries to be used in the *WHERE_part* as shown in the next example.

Example 10-2. Make a copy of Relation PSHELL and remove the entry which has the same property identification number, PID, as element identification number 2 in Relation QUAD4 and verify the result.

```
eSh> COPY /MODEL/PSHELL TO DEL_PSHELL;
.... 1 Entity Copied
eSh> DEL FROM DEL_PSHELL
  2>      WHERE PID = ( SELECT PID FROM /GEOM/QUAD4
  3>                      WHERE EID = 2 );
eSh> Enter YES to delete 1 Entry, leaving 2 Entries: YES
.... 1 Entry Deleted from TESTEB:/TUTOR/DEL_PSHELL
eSh> SELECT * FROM DEL_PSHELL;
     PID       MID             T
--------  --------  ------------
       1       101   1.00000E-01
       3       301   3.00000E-01

.... 2 Entries Selected
```

Note that Relation /GEOM/QUAD4 in the Subquery is fully qualified because the Working Directory is /TUTOR.

## REMOVING COLUMNS OR ROWS FROM MATRICES

It is also possible to remove selected columns or rows from Matrix entities, depending on the Orientation of the Matrix. Recalling the description of Matrix Orientation presented in Chapter 1, for matrices which are stored in the Column-major Orientation, the number of rows is fixed while its number of columns is dynamic. Similarly, for matrices which are stored in the Row-major form Orientation, the number of columns is fixed while its number of rows is dynamic. In the former case, row deletion is not allowed, and in the latter case, column deletion is not allowed. To delete one or more selected columns or rows from a Matrix, the command:

**DELETE FROM MATRIX** *mat_name* **(** *list* **);**

is used where the *mat_name* is the name of an existing Matrix entity and the *list* identifies the columns or rows to be deleted. ***eShell*** automatically determines the Orientation and deletes the data accordingly.

Example 10-3. Delete the odd numbered columns from DEL_KGG, a copy of Matrix KGG. Verify the results by querying the Matrix.

```
eSh> COPY /MODEL/KGG TO DEL_KGG;
.... 1 Entity Copied
eSh> DELETE FROM MATRIX DEL_KGG (1,3,5);
eSh> Enter YES to delete 3 Columns, leaving 3 Columns: YES
.... 3 Columns Deleted from TESTEB:/TUTOR/DEL_KGG

eSh> MATRIX SELECT (BANDED) * FROM DEL_KGG;
Column Major, Compressed, Real Double Precision, Symmetric
    6 Rows, 3 Columns, Density = 44.4%
-------------------------------------------------------------
Column 1, Rows 1 through 3
 2.00000E+02   3.00000E+02   4.00000E+02
-------------------------------------------------------------
Column 2, Rows 3 through 5
 6.00000E+02   7.00000E+02   8.00000E+02
-------------------------------------------------------------
Column 3, Rows 5 through 6
 1.00000E+03   1.10000E+03
```

If you wish to create a Matrix partition, which can be an equivalent operation to deleting the rows of a Matrix, then the **EXTRACT MATRIX** command can be used. When you **DELETE** from a Matrix entity, the topological Shape may be changed. For example, in Example 10-3, the Symmetric Shape was changed to Rectangular after the column deletion. You must use the **SET MATRIX SHAPE** command to make the Shape consistent with the data in the entity. This is only important if the Matrices will be used with ***eBase*:*matlib***.

When you **DELETE** one or more Columns or Rows from a Matrix entity, all subsequent Columns or Rows are renumbered. Therefore, you cannot reference them by their original number.

## REMOVING RECORDS FROM FREEFORM ENTITIES

Finally, you may remove an entire Record from a Freeform entity with the command:

**DELETE FROM FREEFORM** *free_name* **(** *list* **);**

where the *free_name* is the name of an existing Freeform entity and the *list* identifies the Records to be deleted.

Example 10-4. Delete the odd numbered Records from DEL_FREE, a copy of Freeform FREE and confirm the result using the **DESCRIBE** command.

```
eSh> COPY /MODEL/TESTFREE TO DEL_FREE;
.... 1 Entity Copied
eSh> DELETE FROM FREE DEL_FREE (1,3,5);
eSh> Enter YES to delete 3 Records, leaving 2 Records: YES
.... 3 Records Deleted from TESTEB:/TUTOR/DEL_FREE

eSh> DESCRIBE DEL_FREE;
.... Freeform TESTEB:/TUTOR/DEL_FREE
....    2 Records, Longest Record is 40
```

When you **DELETE** one or more Records from a Freeform entity, all Records are renumbered. Therefore, you cannot reference Records by their original number.

Clean-up your /TUTOR/ Directory to restore your TESTEB: database to its initial state.

# 11. FILE ENVIRONMENT COMMANDS

***eShell*** provides a class of commands that can modify the file environment under which the program performs certain operations. These commands allow the user to modify the source of ***eQL*** commands, to define a file upon which ***eQL*** commands may be written for archival purposes, or to create text files that may be used directly by other programs. This section describes these features and the manner in which they are used.

## THE SCRIPT FILE

A ***Script File*** is a text file that resides on the ***eShell*** host computer. It may contain a sequence of many ***eQL*** commands. When using ***eShell***, the command:

```
[ START ] file_name [ param_list ];
```

requests that ***eShell*** begin reading and executing commands from the script file, rather than from the terminal. The *file_name*, which must be enclosed in tics if it does not follow the Basic Naming Conventions described in Chapter 1, may be any valid file name on the ***eShell*** host computer.

***eQL*** Script Files may include parameters that you may change when you **START** them by specifying a *param_list*. This is done by using a substitution variable in the command. The form of such a variable is:

**&1**

You may use from one to nine parameters, i.e. **&1** through **&9**, in a command file. When you invoke the Script File, you then include the actual values of the parameters with the **START** command, as shown in the following example.

Example 11-1. Invoke the Script File named VARGRID with parameters 5.0 and 2.3:

```
eSh> START VARGRID 5.0 2.3;
```

Script Files allow you to save and recall frequently used ***eQL*** command sequences. For example, the developers of ***eShell*** used this feature to save test cases which could be rerun as often as necessary.

Only one Script File may be active at a given instant. If the **SET SCRIPT** command is issued while a Script File is active, the active file is suspended and the new one is opened. Control returns to the previous Script File when the current one is exhausted. Also note that ***eQL*** command editing cannot be performed on commands which enter ***eShell*** from the Script File because the user is not in control of the terminal.

You may include variables within the commands in your Script File by using the symbol substitution protocol discussed in Chapter 1. Namely, commands may include symbols of the form:

&*symbol_name*

You may then use the **DEFINE** command to give these variables a value prior to invoking the Script File.

## THE ARCHIVE FILE

An ***Archive File*** is a text file that contains an historical record of the commands entered during an ***eShell*** interactive session. Specifically, it is a copy of commands that have executed successfully. If a command has been edited, only the final form of the command is archived. Those which contained user errors are not put on the archive file. The principal purpose of this file is to use it during a subsequent ***eShell*** session as a script file.

To define an Archive File, the command:

**SET ARCHIVE TO** '*file_name*';

is used. The Archive File can be enabled or disabled with the command:

**ARCHIVE** { **ON** | **OFF** } ;

While enabled (**ON**), all successful ***eQL*** commands are routed to the Archive File. When disabled (**OFF**), commands are no longer written. As in the case of the Script File, only one archive file may be open. If a new **SET ARCHIVE** request is made, then the currently active file is closed and the new one is opened. Note that if the command **SET SCRIPT** is entered interactively, then it will not be written to the Archive File. On the other hand, you may enter the **SCRIPT OFF** command while the Archive File is enabled. This allows the termination of the sequence of ***eQL*** commands

previously archived and facilitates the use of the archive file as a Script File in a subsequent ***eShell*** session.

## THE REPORT FILE

The purpose of the ***Report File*** is to contain all query and report output for which you wish to obtain a permanent record. The Report File is specified with the command:

**SET REPORT TO** '*file_name*';

It is the enabled or disabled with the command:

**REPORT** { **ON** | **OFF** } ;

This file may be printed in the normal manner by using the appropriate operating system request after the ***eShell*** session is completed. As in the case of the other files, only one Report File may be opened. The request for a new file will result in the previous one being closed.

## THE INTERFACE FILE

There are often times when you may wish to extract data from the ***eBase*** database that will be further processed by some other software. An ***Interface File*** is provided to support this need. This file is defined with the command:

**SET INTERFACE TO** '*file_name*';

As in the case of Report and Archive Files, the interface file may be enabled and disabled with the command:

**INTERFACE** { **ON** | **OFF** } ;

When the Interface File is enabled, all ***eQL*** results will be written to the file in a user-defined format. A format is defined by the command:

**INTERFACE FORMAT** '*format_specifier*';

where the *format_specifier* is a Fortran FORMAT statement enclosed in apostrophes. All Fortran rules are in effect, including the need to indicate apostrophes, or tics, within the format by two consecutive tics.

Example 11-2. Create a text file containing the grid point identification numbers and coordinates in the indicated format.

```
eSh> SET INTERFACE TO 'MYINPUT.DAT';
eSh> INTERFACE FORMAT '(1X,''GID = '',I5,3E15.5)';
eSh> SELECT GID,X,Y,Z FROM GRID;
eSh> INTERFACE OFF;
```

The file is positioned to the end-of-data upon completion of this operation. If the file is enabled again, subsequent data will be appended to the end of the file. Only one Interface File may be used at a given time. If another **SET INTERFACE** command is encountered, the current open file will be closed and the new one opened.

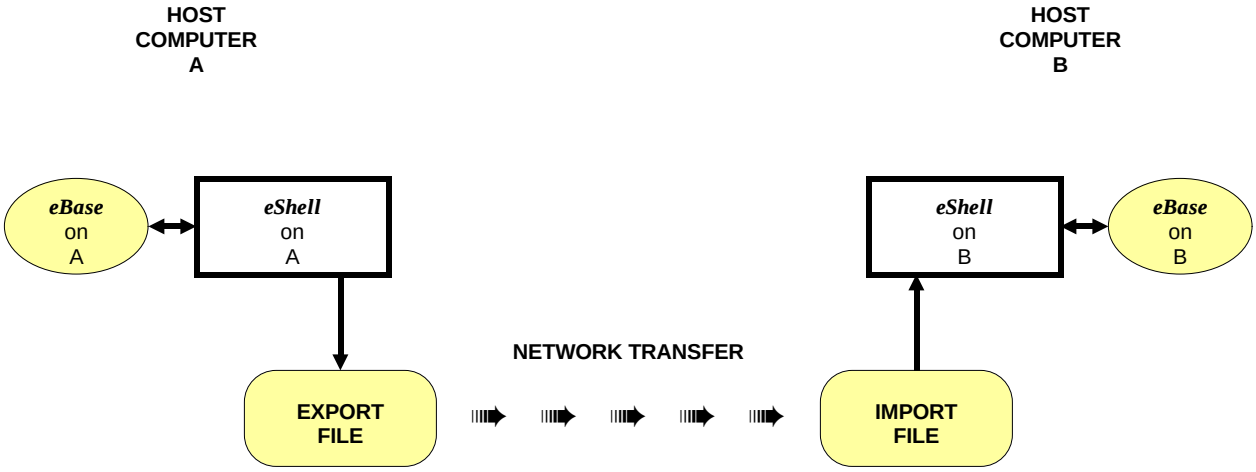
Note that no format checking is done by ***eShell***. It is your responsibility to insure that the format statement specified is compatible with the output data items.

The interface file format for Matrix data is fixed. If you need to export Matrix entities, you can read them in a program that you write which uses a special Fortran utility which is part of the ***eBase*:*applib***. Contact your ***eBase*** Administrator to obtain this routine. The actual internal file format is documented in the Installation Guide and System Support Manual.

## EXPORTING AND IMPORTING DATABASES

You may move an ***eBase*** database from one computer to another as shown in Figure 11-1. Given an ***eBase*** database residing on host computer A, you may create an ***Export File*** in one of two formats described later in this section. This file is then transferred, using the network protocols available at your facility, to host computer B. You then use ***eShell*** on this computer to load the resulting ***Import File*** into a new ***eBase*** database residing on computer B.

**Figure 11-1. Moving a Database To a New Computer**



The commands used for this purpose are:

**EXPORT** [ *path* ] { **BINARY** / **FORMATTED** } *file_name* ;

and

**IMPORT** *file_name* [ *path* ] ;

The optional *path* descriptor indicates that a recursive descent of all subdirectories, from *path*, and their contents will be **EXPORT**ed. In the case of **IMPORT**, all of the directory structure previously **EXPORT**ed will be restored treating the specified *path* as the Root Directory.

There are two basic types of file. The first is **BINARY**, which represents data using the IEEE-754(1985) Binary Floating-Point Arithmetic Standard to represent the data contents of the ***eBase***. Export Files of this type require the minimum amount of disk space and thus require the shortest transfer time across your network. The second type of file is the **FORMATTED** file. When you use this option, the resulting file shows all ***eBase*** data items as clear text. This option results in much larger Export Files. The *file_name* that you specify depends on the naming conventions of the two computers that are used for the transfer of the database. Note that **IMPORT** automatically detects the file type when restoring the database.

When you use the **BINARY** file option, it is possible that certain data may lose precision when moving from one type of computer to another. Check with ***eBase*** Customer Support at UAI when using DEC VAX(VMS) or CRAY computers because the computers do not currently use the IEEE standard.

*This page is intentionally blank.*

# 12. REPORT GENERATION

One of the major purposes of ***eShell*** is to provide capability for the post-processing of engineering analysis data and results. The previous Chapters of this manual have described many methods for manipulating ***eBase*** entities. This section presents the methods available for generating customized reports suitable for inclusion in design reports.

## FORMATTING COMMANDS

Formatting commands allow you to specify customized labels and formats for reports. Specific features include:

- ❒ Column Labels and Formats
- ❒ Page Titles
- ❒ Grouping Commands
- ❒ Page Control Commands

The sections below give detailed definitions of these commands and examples of their use.

## COLUMN LABELS AND FORMATS

As illustrated previously in this manual, selected data are printed using the names of the attributes within the schema as column titles and the data values are printed in the default formats defined within ***eShell***. However, the results of any Relational query can be presented using descriptive headings and formats selected by the user. First, consider the standard query results.

Example 12-1. Query the QUAD4 Relation.

```
eSh> SELECT * FROM QUAD4;

     EID       PID        G1        G2        G3        G4
--------  --------  --------  --------  --------  --------
       1         1         1         2         7         6
       2         2         2         3         8         7
       3         1         3         4         9         8
       4         2         4         5        10         9

.... 4 Entries Selected
```

Any or all columns of the query results may be titled by using the command:

**SET COLUMN** *attribute_name column_options* **;**

where *column_options* are a list of one or more options describing the manner in which the column will be printed. These options include:

*column_option* ⇒ [ *heading_info* ]
[ *format_info* ]
[ *justification* ]
[ **TEMP** ] [ **CLEAR** ]

Each of these options will be described in detail below. When you change the title of an attribute, this change remains in effect for your entire ***eShell*** session, or until you issue the **SET COLUMN** command with the **CLEAR** option. If you want a title to remain in effect for only the next query, you may specify the **TEMP** option when you issue the **SET COLUMN** command.

The *heading_info* is a string defining the title that you wish placed on the column. Its format is:

*column_option* ⇒
[ ***heading_info*** ]
[ *format_info* ]
[ *justification* ]
[ TEMP ] [ CLEAR ]

*heading_info* ⇒ **LABEL** { *'string'* | *'multi_line_title'* }

where a *string* may be any sequence of characters with or without embedded blanks. A *multi_line_title* is specified by separating the lines with the slash (/) character as will be shown in the next example. Should the slash character itself be wanted in the label, then two consecutive slashes (//) are entered. The optional justification parameter specifies the manner in which the title will be justified over the column.

It may be selected as:

*column_option* ⇒
[ *heading_info* ]
[ *format_info* ]
[ ***justification*** ]
[ TEMP ] [ CLEAR ]

*justification* ⇒ { **LEFT** | **RIGHT** | **CENTER** }

If not given, the default justification is used, i.e. **LEFT** for string data and **RIGHT** for numeric data.

Example 12-2. Change the headings for the QUAD4 query so that EID becomes "ELEMENT/ID NUMBER" and PID becomes "PSHELL ID".

```
eSh> SET COLUMN EID LABEL 'ELEMENT/ID NUMBER';
.... Permanent Report Column EID Created
eSh> SET COL PID LABEL 'PSHELL ID';
.... Permanent Report Column PID Created
eSh> SELECT * FROM QUAD4;

  ELEMENT
ID NUMBER  PSHELL ID        G1        G2        G3        G4
---------  ---------  --------  --------  --------  --------
        1          1         1         2         7         6
        2          2         2         3         8         7
        3          1         3         4         9         8
        4          2         4         5        10         9

.... 4 Entries Selected
```

The *format_info* specifies the exact format that you wish to use in printing the column. Its form is a subset of Fortran and may be:

*column_option* ⇒
[ *heading_info* ]
[ ***format_info*** ]
[ *justification* ]
[ TEMP ] [ CLEAR ]

*format_info* ⇒ **FORMAT** ' { **I***w* | **F***w*.*d* | **E***w*.*d* | **D***w*.*d* | **A***w* | **G***w*.*d* } ,

where *w* represents the total field width, in characters, and *d* the number of decimal places. The user may also control the character used to underline attribute names. This is done using the command:

**SET UNDERLINE TO '***underline_character***';**

where the *underline_character* is a single legal character. Note that this character must be enclosed in apostrophes. If a blank is used for the *underline_character*, then a space will be skipped between the attributes names and their values. Consider the following example.

Example 12-3. Query the Relation GRID for all grid points with X-coordinates of 4.0. Use a format of F9.5 for the X- and Y-coordinates, label the GID field "GRID ID"and use an equal sign (=) to underline attribute names.

```
eSh> SET COLUMN GID LABEL 'GRID ID';
.... Permanent Report Column GID Created
eSh> SET COLUMN X FORMAT 'F9.5';
.... Permanent Report Column X Created
eSh> SET COLUMN Y FORMAT 'F9.5';
.... Permanent Report Column Y Created
eSh> SET UNDERLINE TO '=';
eSh> SELECT * FROM GRID WHERE X=4.0;

GRID ID       CID          X          Y             Z
=======  ========  =========  =========  ============
      5         0    4.00000    1.00000   0.00000E+00
     10         0    4.00000    0.0       0.00000E+00
```

## PAGE TITLES

***eQL*** provides you with commands to place titles at either the top of each page or screen, the bottom, or both. The commands used to do this are:

**SET HEADER TO** '*header_line*'
[ *justification* ]
[ **DATE** ] [ **PAGE** ];

**SET FOOTER TO** '*footer_line*'
[ *justification* ]
[ **DATE** ] [ **PAGE** ];

Both *header_line* and *footer_line* may contain blanks. Multiple lines are denoted by the slash (/) as in column titles. The optional **DATE** and **PAGE** parameters request that the date and page number be placed on each page. The *justification* parameters and defaults are the same as for the **SET COLUMN** command. The effects of justification and titling depend upon certain page control information that are discussed later in this Chapter.

Example 12-4. Recreate the query of example 12-3 placing appropriate titles on the output report.

```
eSh> SET HEADER TO 'GRID POINTS ALONG STATION X=4.0' PAGE;
eSh> SET FOOTER TO 'TESTEB SAMPLE' CENTER DATE;
eSh> SELECT * FROM GRID WHERE X=4.0;

          GRID POINTS ALONG STATION X=4.0       PAGE   1

GRID ID       CID          X          Y             Z
=======  ========  =========  =========  ============
      5         0    4.00000    1.00000   0.00000E+00
     10         0    4.00000    0.0       0.00000E+00

 dd-mmm-yyyy           TESTEB SAMPLE

.... 2 Entries Selected
```

## GROUPING COMMANDS

***eQL*** provides a special command that allows the legibility of reports to be enhanced by spacing between related groups of data. This command is:

**SET BREAK ON** *attribute_name* [ **SKIP** *n* ] [ **PAGE** ];

where the *attribute_name* is one which appears in the next query command. The optional **SKIP** parameter requests that *n* lines be skipped when a new value of *attribute_name* is encountered. **PAGE** requests that the report begin a new page when the *attribute_name* changes value. Note that the **SET BREAK ON** command must be made prior to a query that contains any *GROUP_part* or *SORT_part* that references the same *attribute_name*.

Example 12-5. Query the QUAD4 Relation, selecting all attributes, sorting them by the property identification number, PID, and skipping a line between each property type.

```
eSh> SET BREAK ON PID SKIP 1;
eSh> SELECT * FROM QUAD4 SORT BY PID;

     EID       PID        G1        G2        G3        G4
--------  --------  --------  --------  --------  --------
       1         1         1         2         7         6
       3         1         3         4         9         8

       2         2         2         3         8         7
       4         2         4         5        10         9

.... 4 Entries Selected
```

## PAGE CONTROL COMMANDS

Certain page control information may be controlled with ***eQL*** commands. This information allows different print characteristics to be defined. The first such command is:

**SET LINEWIDTH TO** *n* **;**

which defines the width, in characters, of the display area. If not specified, **LINEWIDTH** is 80. The command:

**SET PAGELENGTH TO** *n* **;**

controls the number of lines in the display area. If not specified, this value is 24. To control the number of spaces between columns of output, the command:

**SET COLSPACE TO** *n* **;**

is used. The default spacing is two characters.

The final commands are:

**SET INTWIDTH TO** *n* **;**

**SET FLOATWIDTH TO** *n* **;**

which set the default field width for integer and floating point numeric fields, respectively. The default value depends on the data type of the attribute:

| | |
|---|---|
| I8 | for integer values (INT) |
| 1P,E12.5 | for floating point values (RSP,RDP,CSP and CDP) |

A **FLOATWIDTH** cannot be less than 8 characters.

Character attributes are printed with a field width equal to the length they were defined to have when created. For array attributes, all elements are printed following the rules above. These defaults can be modified in two ways. The first is by the specification of a **SET COLUMN** command. The second is an automatic adjustment by ***eShell*** to accommodate column title information. In the absence of a **SET COLUMN**, the minimum width printed is that which allows the full title to be displayed. This was illustrated in Examples 12-2 and 12-3.

Note that any or all page control commands may be listed in a single **SET** command by listing them separated by commas.

# 13. UTILITY FUNCTIONS

This Chapter describes several utility functions available in ***eShell*** that allow the user to control the tolerance on floating point comparisons and provide online **HELP** information and a summary of current environment variables.

## DIRECTORY TREE

The Directory structure of your ***eBase*** database can be displayed using the command:

**TREE** [ *path* ] **;**

where *path* is an optional Path name. The directory structure shown recursively descends from the *path* location.

Example 13-1. Move to the Root Directory and display the Directory tree for TESTEB:

```
eSh> CD /;
eSh> TREE;
.... TESTEB:/
....   |--- /MODEL
....   |--- /GEOM
....   |--- /RESULTS
```

## TOLERANCE FOR FLOATING POINT COMPARISONS

The ***eBase*** numeric data types (**RSP**, **RDP**, **CSP** and **CDP**) are approximate in nature so that it is possible for computed values that are in fact identical to differ in the least significant decimal places. Because ***eQL*** allows comparisons of these numbers it is usually necessary to define a comparison tolerance, called $\delta$. Let A be an attribute in a relation with a particular value, $\alpha$, and let $\rho$ be an approximate numeric value specified by the user in a query. Then the relational operators are modified as:

$\alpha = \rho$ if and only if $\rho-\delta \le \alpha \le \rho+\delta$

$\alpha \ne \rho$ if and only if $\rho-\delta > \alpha$ or $\alpha > \rho+\delta$

$\alpha > \rho$ if and only if $\alpha > \rho-\delta$

$\alpha \ge \rho$ if and only if $\alpha \ge \rho-\delta$

$\alpha < \rho$ if and only if $\alpha < \rho+\delta$

$\alpha \le \rho$ if and only if $\alpha \le \rho+\delta$

The tolerance is specified with the command:

**SET TOLERANCE TO** *value* [ **PERCENT** ];

where the *value* given corresponds to $\delta$. If the optional keyword **PERCENT** is given, then the tolerance is set to that percentage of a given $\rho$:

$\delta$ = *value* $*\rho$ / 100.0

The default value for the tolerance is 1.0E-6.

## ONLINE HELP

***eShell*** provides an online **HELP** feature to provide documentation of the features available in the program. The command used is:

**HELP** [ *command_part_list* ];

If **HELP** is specified without any additional parameters, a listing of available *command_parts* is given. The user may then obtain additional information by picking from a menu. If the *command_part_list* is provided, then information relating to the named *command_parts* is presented without menu interaction.

## ENVIRONMENT SETTINGS

Because there are many environment variables that may be **SET**, a special command is available to list their current values. This command is:

**SHOW** [ *variable_class_list* ];

If the optional *variable_class_list* is not given, then all environment parameters will be shown. Otherwise, one or more *variable_class_terms* may be selected:

*variable_class_list* ⇒ { **FILES** | **COLUMNS** | **PAGE** | **VERSION** | **CONFIG** | **OPEN DATABASES** }

where **FILES** requests a summary of the active files, **COLUMNS** a summary of current report formatting data for all attributes, **PAGE** a summary of the current report page settings, **VERSION** displays the current version number of the ***eShell*** program, **CONFIG** shows the Configuration parameters which are in effect, and **OPEN DATABASES** gives the name, status, and access level of all open databases.

*This page is intentionally blank.*

# A. eQL COMMAND SUMMARY

This Chapter provides a summary of all ***eQL*** commands and a general description of their function. All of the commands have been described in detail in previous Chapters of this manual. The ***eQL*** functions are grouped into the following categories:

- ❒ Using ***eShell*** and ***eQL*** Command Editing
- ❒ Directory Creation and Maintenance
- ❒ Entity Creation
- ❒ Data Retrieva - Relations
- ❒ Graphing Retrieved Data
- ❒ Indexing Relational Entities
- ❒ Data Retrieval - Non-Relational Entities
- ❒ Inserting Data into Entities
- ❒ Updating Data
- ❒ Removing Entities and Data
- ❒ File Environment Commands
- ❒ Report Generation Commands
- ❒ Utility Commands

Many of the ***eQL*** commands use common parts, or clauses, each of which may be very complex. These clauses, called metasymbols, are also included in this Appendix.

## Chapter 1 - Using eShell

This Chapter describes how you use the ***eShell*** program. This includes the commands to open and close databases, how to enter commands, and how to use symbols to assist in your command entry.

```
OPEN database_name [ = 'phys_name' ] [ NEW | TEMP | WITH { READ | WRITE | ADMIN } ] ['params'] ;
```

Opens an existing or new database for activity.

```
HELP [ command_part_list ] ;
```

Requests information about ***eQL*** commands.

```
CLOSE database_name [ DELETE ] ;
```

Closes an open database.

```
LIST [ line_1 [ TO line_n ] ];
```

List all of, or selected lines of, the active command.

```
DELETE [ line_1 [ TO line_n ] ];
```

Deletes current, or selected lines of, the active command.

```
ENTER 'new_line';
```

Adds a new line to, or within, the active command.

```
CHANGE /string_1 / string_2 /;
```

Changes the first occurance of a string of characters within the active command.

```
RUN;
```

Executes the active command.

```
END;
```

Terminates the ***eShell*** session.

```
DEFINE [ symbol_name [ = value ] ] ;
```

Defines a substitution symbol.

```
UNDEFINE { symbol_name | * } ;
```

Removes the definition of a substiution symbol.

**SET PASSWORDS** [ **ON** *database_name* ] *password_list* ;

Allows you to change passwords on any open database for various database privileges.

*password_term* ⇒ { { **READ** / **WRITE** / **ADMIN** } *password* } / { **CLEAR** { **READ** / **WRITE** / **ADMIN** } }

Defines the new passwords for one or more privilege levels.

## Chapter 2 - Creating and Maintaining Directories

You may structure your database to reflect the organization of your data by using a directory structure. This Chapter describes the commands used to create and maintain that structure.

**MKDIR** *path* ;

Creates a new directory.

**CD** [ *path* ] ;

Changes, or displays, current working directory.

**RMDIR** *path* ;

Removes an existing directory.

**DIRECTORY** [ *path* ] [ **ALL** / **RELATION** / **MATRIX** / **FREEFORM** / **STREAM** / **DIRECTORY** ] [ **DATE** / **SUMMARY** ] ;

Requests a directory listing.

**DIRECTORY** *ent_name* [ **ALLVER** ] [ **DATE** / **SUMMARY** ] ;

Requests a directory listing for entities.

**DESCRIBE** *entity_name* ;

Provides a description of any of the entities contained within the ***eBase*** database.

**RELEASE** *ent_name* ;

Specifies a particular subscripted version of an entity to be the Released Version.

**UNRELEASE** *ent_name* ;

Deletes the Released Version of an entity.

**COPY** *entity_name_1* **TO** *entity_name_2* [ **ALLVER** ] ;

Makes a physical copy of one or more subscripted versions of an entity.

**RENAME** *entity_name_1* **TO** *entity_name_2* [ **ALLVER** ] ;

Renames one or more subscripted versions of an entity.

**ALIAS** *entity_name* **TO** *alias_name* [ **ALLVER** ] ;

Creates a new name, or alias, by which one or more subscripted versions of an entity may be referenced.

**COMPRESS** *mat_name_1* **TO** *mat_name_2* [ **ALLVER** ] ;

To transform an UNCOMPRESSED Matrix entity to a COMPRESSED Matrix entity.

**UNCOMPRESS** *mat_name_1* **TO** *mat_name_2* [ **ALLVER** ] ;

To transform a COMPRESSED Matrix entity to an UNCOMPRESSED Matrix entity.

## Chapter 3 - Creating Database Entities

This Chapter describes the commands and techniques for adding new entities to an existing ***eBase*** database.

**CREATE RELATION** *rel_name* { ( *schema_list* ) | **LIKE** *existing_rel_name* } ;

Creates a new Relation, with a specified schema, on the database. Alternately, the schema of an existing Relation may be used to specify the schema.

**CREATE MATRIX** *mat_name* { ( *matrix_attrib* ) | **LIKE** *existing_mat_name* } ;

Creates a new Matrix entity, with specified attributes, on the database. Optionally, the attributes of an existing matrix may be used to specify the schema.

**CREATE FREEFORM** *ent_name* { ( *num_type* ) | **LIKE** *existing _ent_name* } ;

Creates a new Freeform entity with specified attributes. Optionally, the attributes of an existing entity may be used to specify the schema.

**CREATE STREAM** *ent_name* { ( *num_type* ) | **LIKE** *existing_ent_name* } ;

Creates a new Stream entity with specified attributes. Optionally, the attributes of an existing entity may be used to specify the schema.

## Chapter 4 - Retrieving Data from RELATIONs

The most powerful use of the ***eQL*** language is its ability to retrieve data from an ***eBase*** database. This Chapter describes the commands that may be used to retrieve data from Relations and also the use of indexing to improve query performance.

```
[ RELATION ] SELECT select_list FROM_part
                    [ WHERE_part ]
                    [ GROUP_part ]
                    [ ORDER_part ];
```

Selects all, or part, of a Relation and displays the result.

*WHERE_part* ⇒ **WHERE** *search_condition*

Qualifies, or places constraints on, the entries of the Relation being selected. The *WHERE_part* may also specify one or more subqueries.

*GROUP_part* ⇒ **GROUP BY** *attribute_list*

Requests that the resulting data be grouped by unique values of the selected attributes.

*ORDER_part* ⇒ **ORDER BY** *sort_list*

Requests that the results of the query be sorted on one or more selected attributes.

**SELECT INTERSECTION OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

Selects the set of entries contained in both of two Relations which have the same schema, and optionally creates a new Relation containing these entries.

**SELECT UNION OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

Selects the set of distinct entries contained in either of two Relations which have the same schema, and optionally creates a new Relation containing these entries.

**SELECT DIFFERENCE OF** *rel_name_1* **AND** *rel_name_2*
[ **AS** *rel_name_3* ];

Selects the set of entries in one Relation that are not in another. Both Relations must have the same schema, and optionally creates a new Relation containing these entries.

## Chapter 5 - Graphing Retrieved Data

This Chapter describes the commands that may be used to graph, or plot, retrieved data from Relations and control the windows where the plots are displayed.

**SET ACTIVE WINDOW TO** *plot_win_id* ;

Selects an active plot window.

**CLEAR** { **PLOT WINDOW** *plot_win_id* | **ALL PLOT WINDOWS** } ;

Removes all, or a specified, plot windows.

**REPLOT** [ *plot_win_id* ];

Replots the data is a specified plot window.

**XYPLOT** *x_attrib, y_attrib_list FROM_part*
[ *WHERE_part* ]
[ *GROUP_part* ]
[ *ORDER_part* ];

Selects all, or part, of a Relation for plotting. The *WHERE_part, GROUP-part,* and *ORDER_part* are the same as for the Relational query.

**SET DRAWLINE TO** { **ON** | **OFF** } ;

**SET SYMBOL TO** { **ON** | **OFF** } ;

Selects the drawing of lines and symbols for the graph data.

**SET FTITLE TO** *'text'* ;

**SET** { **XTITLE** | **YTITLE** } **TO** *'text'* ;

**CLEAR** { **FTITLE** | **XTITLE** | **YTITLE** } ;

Allows plot titles to be added and cleared.

```
{ SET   }  { XMIN }
{ CLEAR }  { XMAX } TO value ;
           { YMIN }
           { YMAX }
```

Allows plot data to be *windowed*

```
SET { XLOG } TO { ON  } ;
    { YLOG }    { OFF }
```

Selects one or two logarithmic scales.

```
SET { XDIV } TO inc ;
    { YDIV }
```

Controls the number of labeled divions on each axis of the plot.

```
SET GRID TO { ON  } ;
            { OFF }

SET AXIS TO { ON  } ;
            { OFF }
```

Control whether a grid is applied to the plot and whether the axes are drawn.

```
ADDCURVES x_attrib, y_attrib_list FROM_part
                  [ WHERE_part ]
                  [ GROUP_part ]
                  [ ORDER_part ] ;
```

Adds a new curve to an existing plot in the Active Window.

```
MXYPLOT param, x_attrib, y_attrib FROM_part
                         [ WHERE_part ]
                         [ GROUP_part ]
                         [ ORDER_part ] ;
```

Selects all, or part, of a Relation for plotting. The *WHERE_part, GROUP-part,* and *ORDER_part* are the same as for the Relational query. One curve is drawn for each unique value of the specified *param*.

## Chapter 6 - Indexing Relational Entities

```
CREATE [UNIQUE] INDEX ON rel_name
                              ( attribute_list );
```

Creates an index for one or more attributes of a Relation.

```
PURGE INDEX ON rel_name ( attr_list ) ;
```

Deletes a previously created index from a Relation.

## Chapter 7 - Retrieving Data from Non-Relational Entities

This Chapter describes commands that may be used to retrieve data from Matrix, Freeform, and Stream entities.

```
MATRIX SELECT  [ format ]
               select_list
               FROM mat_name
               [ WHERE_part ] ;
```

Allows selected portions of a Matrix to be displayed in a specified format.

*format* ⇒ ( { **FULL** | **BANDED** } )

Specifies the format of the Matrix selection results.

*select_list* ⇒ { *term_list* | * }

Specifies the terms within rows or columns that will be selected from the Matrix.

*WHERE_part* ⇒ **WHERE** { **COLUMNS** | **ROWS** } **IN** *rorc_list*

Restricts the Matrix selection to terms from specific rows or columns.

```
FREEFORM SELECT   select_list
                  FROM free_name
                  [ WHERE_part ] ;
```

Allows selected portions of a Freeform entity to be displayed.

*select_list* ⇒ { *value_list* | * }

Specifies the Data Values to be selected from a Freeform entity.

*WHERE_part* ⇒ **WHERE** { **RECORD** = *num* | **RECORDS IN** *record_list* }

Restricts the Freeform selection to specific Records.

```
STREAM SELECT select_list
                  FROM free_name ;
```

Allows selected portions of a Stream entity to be displayed.

## Chapter 8 - Inserting Data into Entities

This Chapter describes the ***eQL*** commands which are available to insert new data into Relational and Matrix entities.

**INSERT INTO** [ **RELATION** ] *rel_name*
[ **(** *proj_list* **)** ] *value_part*;

Inserts a new entry into the named Relation.

*value_part* ⇒ { **VALUES** ( *value_list* ) | *subquery* }

Specifies the values defining the new Relational entry.

**INSERT INTO MATRIX** *mat_name new_value_list*;

Inserts a new column or row into the specified matrix.

*new_value_term* ⇒ **VALUES AT** { *row_id* | *col_id* } **(** *value_list* **)**

Specifies the numeric values to be entered into the new matrix column or row. These are expressed in the special string format.

**INSERT INTO FREEFORM** *free_name* **VALUES (** *value_list* **) ;**

Creates a new Freeform entity Record.

**INSERT INTO STREAM** *stream_name* **VALUES (** *value_list* **) ;**

Specifies the Data Values to be appended onto a Stream entity.

## Chapter 9 - Updating Data

This Chapter describes commands which allow individual data entries in both Relations and matrices to be updated.

**UPDATE** [ **RELATION** ] *rel_name* **SET** *value_list*
[ *WHERE_part* ];

Updates, or modifies, all or part of the data in a Relational entity that meets the specified selection criteria.

**UPDATE MATRIX** *matrix_name value_list*
**WHERE** { **ROW** | **COLUMN** } **=** *rorc_id* ;

Updates, or modifies, all or part of the data in a Matrix column or row.

*value_term* ⇒ **SET VALUES AT TERM** *corr_id* **TO (** *values* **)**

Specifies the new matrix values.

**UPDATE FREEFORM** *free_name* **SET** *position* **TO** *value_list*
**WHERE RECORD =** *rec_id* **;**

Updates, or modifies, all or part of the data in a Stream entity.

**UPDATE STREAM** *stream_name* **SET** *position* **TO** *value_list* **;**

Updates, or modifies, all or part of the data in a Freeform record.

**ALTER** *rel_name* **ADD (** *new_schema_list* **);**

Adds one or more new attributes to an existing Relation.

## Chapter 10 - Removing Data from eBase

This Chapter describes the ***eQL*** commands that are used to remove entities and their data from the ***eBase*** **database.**

**PURGE** { **RELATION** | **MATRIX** | **FREEFORM** | **STREAM** } *entity_name* **[ ALLVER ];**

Removes an entity and its data from the database

**DELETE FROM** [ **RELATION** ] *rel_name* [ *WHERE_part* ];

Deletes selected entries from the named Relation that meet the selection criteria.

**DELETE FROM MATRIX** *mat_name* **(** *list* **);**

Deletes selected columns or rows from a Matrix entity.

**DELETE FROM FREEFORM** *free_name* **(** *list* **);**

Removes selected Records from a Freeform entity.

## Chapter 11 - File Environment Commands

This Chapter describes a class of commands that can modify the file environment under which ***eShell*** performs certain operations.

[ **START** ] *file_name* [ *param_list* ] **;**

Executes an ***eQL*** Script File saved as a host-computer file and, optionally, passes parameters to it.

**SET ARCHIVE TO** '*file_name*';

Defines a file on the host computer that will contain an archive created from user-input commands.

**ARCHIVE** { **ON** | **OFF** } ;

Enables and disables the archive file.

**SET REPORT TO** '*file_name*';

Defines a file on the host computer that will contain reports created from output displays.

**REPORT** { **ON** | **OFF** } ;

Enables or disables the report file.

**SET INTERFACE TO** '*file_name*';

Defines a file on the host computer that will contain output displays which are written in a user-specified format for subsequent processing by other programs.

**INTERFACE** { **ON** | **OFF** } ;

Enables or disables the interface file.

**INTERFACE FORMAT** '*format_specifier*';

Defines a Fortran format to be used in writing to the interface file.

**EXPORT** [ *path* ] { **BINARY** | **FORMATTED** } *file_name* ;

Creates an Export File used to move an ***eBase*** from one computer to another.

**IMPORT** *file_name* [ *path* ] ;

Restores a previously **EXPORT**ed ***eBase***.

## Chapter 12 - Report Generation

This Chapter presents the methods available for generating customized reports suitable for inclusion in design reports.

**SET COLUMN** *attribute_name column_option*s;

Allows the user to specify titling and format options for each column of a report.

*column_option* ⇒ [ *heading_info* ]
[ *format_info* ]

```
                        [ justification  ]
                        [ TEMP ] [ CLEAR ]
```

Defines the available column options.

*heading_info* ⇒ **LABEL** { *'string'* | *'multi_line_title'* }

Defines column heading label as one or more lines.

*justification* ⇒ { **LEFT** | **RIGHT** | **CENTER** }

Selects column titling justification option.

*format_info* ⇒ **FORMAT** ' { **I***w* | **F***w.d* | **E***w.d* | **D***w.d* | **A***w* | **G***w.d* } '

Specifies the exact format to be used in displaying a column.

```
SET UNDERLINE TO 'underline_character';
```

Sets the default character used for underlining attribute names in the output display.

```
SET HEADER TO 'header_line'
              [ justification ]
              [ DATE ] [ PAGE ];
```

Defines text information to be placed at the top of each page of a display.

```
SET FOOTER TO 'header_line'
              [ justification ]
              [ DATE ] [ PAGE ];
```

Defines text information to be placed at the bottom of each page of a display.

```
SET BREAK ON attribute_name [ SKIP n ] [ PAGE ];
```

Selects vertical spacing options as a function of changes in one or more attribute values.

```
SET LINEWIDTH TO n
SET PAGELENGTH TO n
SET COLSPACE TO n
SET INTWIDTH TO n
SET FLOATWIDTH TO n
```

Selects characteristics of the the output display, including the width of an output line, the number of lines in a display, the number of spaces be-

tween displayed attributes, the width for displaying integer values, and the width for displaying floating point values.

## Chapter 13 - Utility Functions

This Chapter describes several utility functions available in ***eShell***.

**TREE** [ *path* ] ;

Displays the Directory structure of an ***eBase*** database.

**SET TOLERANCE TO** *value* [ **PERCENT** ];

Specifies the tolerance used in comparing floating point numeric values used in query commands.

**HELP** [ *command_part_list* ] ];

Requests additional information about ***eQL*** commands.

**SHOW** [ *variable_class_list* ];

Provides a summary of current environment variables.

*variable_class_list* ⇒ { **FILES** | **COLUMNS** | **PAGE** | **VERSION** | **CONFIG** | **OPEN DATABASES** }

Selects a subset of environment variables.

*This page is intentionally blank.*

# B. GLOSSARY

***Active Command.*** ***eShell*** performs no action until you have entered a complete command, including the semicolon. This command is called the ***Active Command***.

***Active Plot Window.*** Is a graphics window, that may be defined by the user, which contains the results of the next plotting command.

***ADMINISTRATION Privilege.*** Is the ***eBase*** privilege level which allows a user called the Database Administrator, DBA, to control the use of the database and the specific access allowed by its users.

***Archive File.*** A text file that contains an historical record of the commands entered during an ***eShell*** interactive session. Specifically, it is a copy of commands that have executed successfully. If a command has been edited, only the final form of the command is archived. Those which contained user errors are not put on the archive file. The principal purpose of this file is to use it during a subsequent ***eShell*** session as a script file.

***Attributes.*** The columns of a Relational entity. Each attribute has a name and a data type associated with it.

***Column-major.*** A method of storing a Matrix entity where the number of columns in the matrix is dynamic, but the number of rows is fixed.

***Command Buffer.*** An area where all the lines of your ***eQL*** command are stored pending their execution.

***Compressed Storage Mode (Matrix).*** ***eBase*** uses an optional data compression technique to minimize the disk storage requirements of matrices. This is called ***Compressing***. Only the non-zero terms of compressed matrices are stored, along with a small amount of control information. (See also Uncompressed Storage Mode)

***Current Position.*** The line within the Command Buffer where you may perform command editing.

***Data Modeling.*** The process of designing representations of scientific data. Factors to be considered include the quantity of data and the method in which it will be accessed.

***Directory.*** A named area of the ***eBase*** database which contains entities which are related to each other.

***Entity.*** An Entity is a database object which contains a Name, a Schema, and a Data Component. (See also Entity Classes)

***Entity Classes.*** Groups of ***eBase*** entities which share the same kinds of schema. The four ***eBase*** entity classes are Relational entities, Matrix entities, Freeform entities, and Stream entities.

***Entry.*** An Entry is a row in a Relational entity.

***eQL.*** Is the ***eBase*** Query Language, based on the SQL standard for database languages.

***Export File.*** An external host-computer file which contains a binary or formatted version of one or more entities contained in a specified directory tree which can then be moved to a different host-computer. (See also Import File)

***Field.*** A Field is a single data item in a Relational entity. It is found at a single row and column position in the table.

***Freeform Entity.*** Are a form of internal data representation that can sometimes be used to improve the performance of software applications. They are collections of data with only a local and transient purpose. You may think of Freeform entities as Fortran random files which have variable length records. Because of its portability limitations, the use of the non-schematic form of this entity class is discouraged for other than temporary data.

***Import File.*** An external host-computer file which contains a binary or formatted version of one or more entities that were contained in a specified directory tree that have been moved from a different host-computer. (See also Export File)

***Index.*** An index for a Relational entity serves the same purpose as an index in a book — it allows faster access to the information that you wish to find. ***eShell*** allows you to build one or more indexes for the attributes of a relation so that the program may gain rapid access to the data.

***Integrity.*** The protection of database contents from inadvertent distruction.

***Interface File.*** An external host-computer file that contains the results of ***eBase*** queries which have been formatted in a manner which facilitates their further processing by some other software.

***Key.*** A single unique Index for a Relational entity is often called a Key.

***Keywords.*** Character string within ***eQL*** **commands that must be entered exactly as shown in this manual.**

***Matrix Entity.*** Arrays of numbers used in mathematical formulae typically encountered in engineering and physical science software applications. A matrix is defined in the standard mathematical manner as an array of n rows and m columns. (See also Compressed Storage Mode)

***Multischematic Database.*** A database which allows different types of data structures to be represented in an efficient manner. (See also Entity Classes)

***Null Field.*** Is a field within a Relational entity which is undefined. This occurs, for example, when a new entry is inserted into the entity where all of the attributes are not assigned values.

***Orientation (Matrix).*** Defines the manner in which a Matrix entity is stored. Selections Column-major storage and Row-major storage, respectively. (See also Compressed, Uncompressed, Column-major, and Row-major)

***Passwords.*** ***eBase*** databases have three levels of ***Passwords*** which can be used to secure the data against inadvertent destruction or unauthorized use. These passwords allow ***READ***, ***WRITE***, and ***ADMINISTRATION*** privileges.

***Path.*** A sequence of Directory names which indicates a location within the directory hierarchy.

***Privilege.*** The database access allowed by a user. These are defined by the ***READ***, ***WRITE***, and ***ADMINISTRATION*** passwords. (See also Passwords)

***Query.*** An ***eQL*** command which retrieves data from one or more ***eBase*** Relational or Matrix entities.

***READ Privilege.*** Allows users to read data from the ***eBase*** database.

***Relational Entity.*** A table of data. ***eBase*** tables have rows, which are called ***Entries*** and columns, which are called ***Attributes***. A particular data value at a given entry and attribute location is called a ***Field***. (See also Entries, Attributes, and Fields)

***Report File.*** A file which captures all query and report output for which you wish to obtain hardcopy.

***Root Directory.*** When you OPEN an ***eShell*** database, you will be placed in the root directory of that database.

***Row-major.*** Method of storing a Matrix entity where the number of rows in the matrix is dynamic, but the number of columns is fixed.

***Schema.*** A set of rules defining the data characteristics of a database entity.

***Script File.*** A text file that resides on the ***eShell*** host computer. Unlike a Command File, it may contain a sequence of many ***eQL*** commands.

***Shape (Matrix).*** Defines the shape of a Matrix entity. Selections are **RECTANGULAR**, **SQUARE**, **SYMMETRIC**, **DIAGONAL**, and **IDENTITY**.

***Storage Mode (Matrix).*** Defines the storage mode of a Matrix entity. Selections are **COMPRESSED** and **UNCOMPRESSED** (See also Compressed, Uncompressed, Column-major, and Row-major)

***Stream Entity.*** A continuous stream of data values that may be directly and randomly addressed. You may best think of this type of entity as a low-level Unix or DOS file.

***Subquery.*** The portion of a query in which the selection of data is based on the results of another query.

***Subscript.*** An additional modifier for an ***eBase*** entity name which allows great flexibility in data modeling.

***Substitution Symbol.*** A variable name or number which is used as an argument to an ***eQL*** command. The symbol is preceded by an ampersand (&).

***Type (Matrix).*** Defines the data type of a Matrix entity. Selections are **INT**, **RSP**, **RDP**, **CSP**, and **CDP**.

***Uncompressed Storage Mode (Matrix).*** The Matrix entity storage mode in which all matrix terms are on the database. (See also Compressed Storage Mode)

***Working Directory.*** Your current location within the database directory hierarchy is called the ***Working Directory***.

***WRITE Privilege.*** Allows users to write data onto the ***eBase*** database.

***XYPLOT.*** Is a graphic display of the results of a relational query.

# INDEX

## !



## A



## B



## C

# D



# E

# F



# G



# H



# I



# J



# K



# L

# M



# N



# O



# P



# Q



# R

# S



# T

# U



# V



# W



# X

*This page is intentionally blank.*