## Table of Contents

**Introduction** ........................................................................... vii
  - The KB_SQL Product ........................................ vii
  - About KB Systems .......................................... viii
  - How to Use this Manual .................................. x

**Part I: A Tutorial**

### Lesson 1: The Basics ................................................. 1
  - Starting the SQL Editor ................................... 2
  - How to Select, Add, or Delete a Query ................ 3
  - The SQL Text Window .................................. 11
  - Running Your Query in Background Mode .......... 14
  - Directing Your Query Results .......................... 15
  - Exiting the Current Query .............................. 17
  - Exiting the SQL Editor ................................ 17

### Lesson 2: Writing Queries .................................... 19
  - Introduction ............................................. 20
  - The SELECT Statement ................................ 21
  - Tests .................................................. 24
  - Joining Tables ....................................... 25
  - Using Set Functions to Summarize Results ....... 28
  - Subqueries .......................................... 30
  - Controlling the Report Format ....................... 33

### Lesson 3: Using DDL, DML, and DCL Statements ....................... 39

---

**KB_SQL SQL Reference Guide** iii
### Chapter 1: Windows and Keys

- Text Windows ........................................... 94
- Value Windows .......................................... 94
- Selection Windows ..................................... 95
- The Function Keys ..................................... 96
- The Cursor Movement Keys .......................... 98

### Chapter 2: Menu Bar Options

- Clear Option .......................................... 100
- Edit Option ............................................ 100
- Halt Option .......................................... 102
- Info Option .......................................... 103
- Print Option .......................................... 105
- Quit Option .......................................... 107
- Run Option .......................................... 108
- Save Option .......................................... 110
- User Option .......................................... 111

### Part III: Appendixes

- Appendix A: Background Information .............. A-1
Appendix B: Suggested Readings ........................................ B-1
Appendix C: The Demonstration Tables ............................... C-1
Appendix D: Glossary ................................................... D-1
Appendix E: KB_SQL Online Reports ................................. E-1

Index  ..................................................................................... I-1
The KB_SQL Product

The KB_SQL product is a powerful information retrieval and analysis tool that allows you to query your existing M (formerly known as MUMPS) databases. It provides you with the ability to access, manipulate, define, and control access to information without the need for a procedural programming language.

KB_SQL is a marriage of two standard languages: SQL and M. SQL provides a standard for database access and communication between systems. M provides an efficient language for processing large amounts of data.

The SQL Editor, which is KB_SQL’s text editor interface, offers a full implementation of interactive ANSI SQL commands, plus a set of reporting extensions to the SQL language. Although these extensions are not part of the ANSI SQL standard, they provide formatting and query management features to complement the standard.

Using the SQL Editor, programmers can satisfy routine information requests in minutes rather than days. System administrators can provide custom views of the application database to groups of users. End users can access their database, increase their own productivity, and reduce their dependence on programmers for routine requests.
Other features of the SQL Editor allow you to:

‰ Import and export data to other non-M applications
‰ Batch and run groups of reports together
‰ Schedule queries to be run at a later date and time
‰ Prompt the user for information each time the query is run
‰ Prototype reports by limiting the number of rows searched

In short, KB_SQL makes developers and end users more productive. By using the SQL Editor, you can produce virtually any type of report needed by your organization.

About KB Systems

KB_SQL is a product of KB Systems, Inc. Founded in 1988, KB Systems develops software tools that improve the productivity of the M application development and user community. The company is committed to the use of industry standards to provide efficient access to information for the first-time user as well as the expert. This strategy will help you keep pace with advancements in technology without wasting valuable resources on non-standard approaches.

KB Systems offers training seminars and consulting services to help you successfully implement our products. We want to ensure that you obtain maximum benefit from their use. For additional information on all our services and products, call us at 703-318-0405 or browse our Web site at http://www.kbsystems.com.
Training Seminars

- The EZQ Editor
- The SQL Editor and the SQL Language
- Database Administration with KB_SQL

Consulting Seminars

- Data Dictionary Mapping and Data Modeling
- Integration with Existing Applications
- Application Planning and Development
How to Use This Manual

Purpose

The KB_SQL SQL Reference Guide explains how to use the SQL Editor to select and control information in your database. A tutorial is provided to give you hands-on experience. The KB_SQL language is explained in the KB_SQL Syntax Guide complete with syntax, rules, and examples.

Audience

This manual is intended for anyone who wants to use the SQL Editor interface with the KB_SQL product.
The Organization of This Manual

This manual is divided into three parts: a tutorial, a section explaining the components of the KB_SQL Editor interface, and appendixes. An index of terms, prompts, and windows follows the appendixes.

Part I  A Tutorial

The tutorial takes you through the creation of several reports, introducing you to the SQL Editor interface and many of the KB_SQL features. We recommend that you complete this tutorial before writing your own queries. Depending on your understanding of SQL and relational databases, you may want to spend some time reading the background information provided in the appendixes before you begin the tutorial.

Lesson 1 explains how to enter and exit the interface. It describes the steps for adding, editing, and deleting queries as well as printing the query results.

Lesson 2 illustrates the SELECT statement through numerous sample queries. You will learn to restrict your query results and customize the report format.

Lesson 3 explains how to update your tables and data dictionary. It also shows you how to grant and revoke access privileges. Several sample statements illustrate these features.

Lesson 4 explains the components and benefits of using a data access plan. It also discusses common performance issues.
Part II  Components of the Interface

This section of the manual discusses each of the menu bar options of
the KB_SQL Editor in alphabetical order. It also discusses the various
window types of the interface and the keys that you use to interact with
the interface.

Part III  Appendixes

A discussion on the structure and value of relational databases and an
overview of SQL and M is provided in Appendix A. Appendix B lists
additional reading materials that may help you enhance your knowledge
of SQL and relational databases. Appendix C lists the tables that you
will use for the demonstration queries in this tutorial. Appendix D is a
glossary of terms used throughout this manual, and Appendix E
describes some useful reports.

**********

Error Messages

Earlier versions of this manual listed error messages in a separate
appendix. Error messages are now available by running the
SQL_ERROR_LIST query.
Style Conventions

To help you locate and identify material easily, this manual uses the following style conventions:

[key]

Key names appear enclosed in square brackets.
Example: Select an output device and press [enter].

<option>

References to the menu bar options are enclosed in angle brackets.
Example: Select the <Info> option to display the Query Information window.

CAPITAL LETTERS

For the purpose of identification, this manual prints key words and names of tables, queries, columns, modifiers, statements, and clauses in all capital letters. However, it is not necessary for you to capitalize them when typing them in the SQL text window.
Example: The CREATE statement is used to add new indices, schemas, tables, and views to the data dictionary.

italics

Italics are used to reference prompt names (entry fields) and terms that may be new to you. All notes are placed in italics.
Example: Selection windows let you pick an item from a list of possible choices.
**Windows**

The manual includes illustrations of windows from the SQL Editor interface. Window names are highlighted by a double underline.

**Prompt:**

```
data type (length) [key]
```

Each prompt includes the data type, length, and any special keys allowed.

If the prompt is followed by a colon (Prompt:), you may enter a value for the prompt.

If a prompt is followed by an equal sign (Prompt= ), it is for display purposes only.

If the prompt is followed by a question mark (Prompt?), you can enter a value of YES or NO.

◆ **Do this**

Any steps that you are expected to perform are designated by a diamond.

**Example:**

```
Press [list] and select the query whose information you want to copy.
```
Part I

A Tutorial
The SQL Editor interface is a text-editing window in which you can create, edit, and run SQL statements. The SQL Editor is composed of several interactive windows. Although each window performs a different function, the interface and conventions are consistent for all windows. Detailed information on the various window types and keys that you use to interact with the interface are explained in Part II of this manual.

This lesson explains:

- How to enter and exit the SQL Editor
- How to add, edit, and delete a query
- How to run a query and display the output.

Because of the wide range of terminals and devices in use by the M language, your interface may not appear exactly as the figures shown in this manual. Also, because KB_SQL lets you customize the interface, your database administrator may have modified the interface to make it more consistent with your own conventions.
Starting the SQL Editor

Depending on how your site administrator has set up the menus for your user group, you may either invoke the SQL Editor directly from the M command prompt or select it from your existing menu system. If you are presented with a menu, follow the directions below.

Select USER OPTIONS

To select the SQL Editor interface:

- Press ↓ to highlight the SQL EDITOR option and press [enter]. (The selection window from which you choose the SQL EDITOR option may vary depending on your user status.)

- Or-

  Try a shortcut.

  | Press S (for SQL) and press [enter].

The Query Name window will appear.
How to Select, Add, or Delete a Query

The diagram below outlines the process for selecting, adding, or deleting a query. Regardless of the process, you start the procedure at the Query Name window.

1. Enter query name and/or press [list]
2. Query found?
   - Yes: Select, Insert, Delete Query window
   - No: Add query?
     - Yes: Copy from Query window
       - No: Delete query
3. Delete query
   - Yes: Query Information window
     - No: SQL text window
4. Select query
   - Yes: Query Information window
     - No: SQL text window
5. Insert query
   - Yes: Copy from Query window
     - No: Query Information window

4 KB_SQL SQL Reference Guide
Query Name

You use the Query Name window to enter the name of your query. You may enter upper or lowercase letters, numbers, spaces, or underscores. KB_SQL automatically converts lowercase letters to uppercase letters and spaces to underscores.

Instead of entering a query name in this window, you can press [list] to display the names of all available queries. You can also enter the first few letters of the query name, and KB_SQL will perform a partial match, listing the queries whose names begin with the letters you entered. That is what we will do now.

Enter DEMO in the Query Name window and press [enter].
If one or more existing queries match the name you entered, a selection window will list all existing queries for which you have privileges and allow you to select, add, or delete a query. (If there weren’t any matching queries, the Add Query window would appear.)

Because there are several queries beginning with the word DEMO, the Select, Insert, Delete Query window appears and displays all the queries that you may access. The list contains the queries whose names match the name you entered.

Select, Insert, Delete Query

```
Select, Insert, Delete Query

| DEMO_ALTER_VIEW |
| DEMO_COLUMN_MODIFIERS |
| DEMO_COUNT |
| DEMO_CREATE_TABLE |
| DEMO_CREATE_VIEW |
| DEMO_DELETE_ROWS |
| DEMO_DROP_TABLE |
| DEMO_DECLARE |
| DEMO_GRANT |
| DEMO_GROUP |
| DEMO_GROUP_POLICY |
```

KB_SQL SQL Reference Guide
Adding a Query

To add a new query from the selection window:

1. Press [insert].

The insert function lets you create a new query. It displays the Copy from Query window. This window lets you use an existing query as the basis for your new query.

Copy from Query

To copy the description and SQL text from an existing query:

1. At the Query name prompt, type the name of the query whose information you want to copy, and press [enter]. For the purpose of this lesson, type DEMO SELECT ALL. Select the query’s name from the Select Query window by pressing [enter].

   - Or -

1. Press [list] and select the query whose information you want to copy. For the purpose of this lesson, select DEMO_SELECT_ALL.

Note: If you do not want to copy SQL text from an existing query, you can bypass the Copy from Query window by pressing [enter].
Query Information

The Query Information window allows you to edit the query name and description before proceeding to the SQL text window. For a complete description of the prompts in the Query Information window, refer to the “Info Option” entry in Part II of this manual.

Type DEMO1 at the Query name prompt and press [enter].

Press [enter] to move from prompt to prompt

-Or-

Press [skip] to transmit the information and proceed to the SQL text window.

The SQL text window is explained later in this chapter. Let’s first learn an alternate way to adding a query.

Return to the Query Name window by pressing [skip].
To add a new query from the Query Name window:

Type the name of a new query, **DEMO2**, at the Query name prompt.

Add Query

The Add Query window is displayed when no matching queries are found.

Press [→] to highlight YES, and press [enter]

-Or-

Type Y and press [enter].

At this point you would continue as you did before with the Copy from Query window.
Deleting a Query

The delete function removes old or obsolete queries. To keep disk space available, you should remove queries when they are no longer useful.

**WARNING:** These instructions are provided here for your future needs. We advise you against deleting any queries other than your DEMO2 query at this time.

To delete a query:

1. Type `DEMO2` in the Query Name window or press [list].
2. Highlight the `DEMO2` query in the selection window, and press [delete].

KB_SQL displays the Query Information window and the commit prompt. If you answer Yes to the Commit? prompt, the query and the corresponding M routine are deleted.

1. Type Y at the Commit? prompt and press [enter].
Selecting a Query for Editing

To select a query from the selection window for editing:

You may use the arrow keys to move the cursor to a desired query name or you may type the first few letters of the query and KB_SQL will move the cursor to the first query in the current window that begins with those letters. You can also use [Page Up] and [Page Down] to scroll through the entire list.

| Highlight DEMO1 and press [enter].

After you select a query, KB_SQL loads the specified query and proceeds to the SQL text window if you have update privileges. If you have select privileges, KB_SQL gives you the option to run the query.

Note: If the query you selected was originally written using the EZQ Editor, a system message will inform you of this. Any changes you make to EZQ queries using the SQL Editor are not reflected in the EZQ Editor. If you want to preserve the EZQ query, copy it into the SQL Editor and rename it.
The SQL Text Window

The SQL text window has two components: a menu bar of options and a text area to enter SQL statements. When the SQL text window is first displayed, the <Edit> option on the menu bar is highlighted. At this point, pressing [enter] will position the cursor within the text area, allowing you to edit the SQL statements. Return to the menu bar by pressing [skip], indicating that you are finished editing the query.

To select an option from the menu bar:

| Press either [→] or [←] to highlight the option, and press [enter].

-Or-

Try a shortcut.

| Press the first letter of the option and press [enter].

Following is a brief description of each option in the menu bar.
### SQL Editor Options

<table>
<thead>
<tr>
<th>Select:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Edit&gt;</td>
<td>Edit the SQL statements.</td>
</tr>
<tr>
<td>&lt;Run&gt;</td>
<td>Save and execute the query.</td>
</tr>
<tr>
<td>&lt;Print&gt;</td>
<td>Print the contents of the SQL text window and other query-related statistics.</td>
</tr>
<tr>
<td>&lt;Clear&gt;</td>
<td>Remove the text in the SQL text window.</td>
</tr>
<tr>
<td>&lt;Save&gt;</td>
<td>Save any changes made to the SQL text window.</td>
</tr>
<tr>
<td>&lt;Quit&gt;</td>
<td>Exit the SQL text window. If you make changes to the query and do not save them, this option will prompt you to save the changes.</td>
</tr>
<tr>
<td>&lt;Info&gt;</td>
<td>Display the Query Info window.</td>
</tr>
<tr>
<td>&lt;Halt&gt;</td>
<td>Stop queries that are running in the background.</td>
</tr>
<tr>
<td>&lt;User&gt;</td>
<td>Print a list of all tables that you have permission to access, and a log of all queries that you have run, within a specified range.</td>
</tr>
</tbody>
</table>

Additional information on each of the options can be found in Part II of this manual.
Editing in the SQL Text Window

The SQL text window works like a simple word processor. It is always in *insert mode*, meaning the characters you type will be inserted where the cursor is, and characters to the right of the cursor will be moved further to the right. If a line of text becomes too long it will automatically wrap to the next line. You can force a line to break by pressing [enter].

You can enter SQL commands in upper or lowercase characters. Except for literal values and M variables, KB_SQL will internally convert lowercase characters to uppercase for validation during the parse step. You can add extra spaces or blank lines to make it easier to read.

Comment lines may be added by prefixing the line with two or more hyphens (e.g., ---This is a comment). KB_SQL will ignore any text after the hyphens. This feature can be used to debug queries: instead of deleting code, you can prefix it with hyphens, and KB_SQL will ignore it when you run the query.

While in Edit mode you can access lists of SQL commands, functions, tables, and columns by pressing [list]. You can also invoke the on-line reference system by pressing [help]. More information about the List and Help Features are provided in Part II of this manual.
Running Your Query in Background Mode

If you have not selected an output device, and your query is going to modify the database, and your DBA has configured your system to allow queries to run in background mode, the Run in the Background window will appear after you select the <Run> menu option. You may respond YES or NO.

If you chose to run the query in the background or if you selected an output device (you can use the SET DEVICE statement at the beginning of the query to assign an output device), and if your DBA has configured your system to allow you to run your query at a later date or even on a scheduled basis, the Query Start Parameters window will appear.

The start date and time values will be filled with TODAY and NOW, respectively. You may choose to change them in order to run the job at another date/time. You may also press [list] to select from a list of start date values in the Select SQL Custom Start Dates window.
Directing Your Query Results...

To the Screen or to a Printer

After KB_SQL prepares your query, the Print on Device window appears. The Print on Device window allows you to control where the query results print.

Note: Your DBA can overwrite the default device selection with an alternative device selection. See your DBA for more information.

To print your query results on the screen:

| Press [enter] from the Device name prompt.

By including the SET SCROLL = YES statement in your query, you can page up and back through the query results.

To send the query results to a printer:

| Enter the output device name (up to 30 characters). Press [enter]. If you want KB_SQL to perform a partial match, enter only the first few letters of the device.

-Or-

| Press [list] to view all valid output devices.

KB_SQL will display a selection window of devices. If you entered a device name, the list will begin with the devices that match the name.
you entered. If you entered only the first few letters of the device, KB_SQL will list those devices that match the character(s) that you entered.

Select an output device and press [enter].

If the selected device has more than one output mode, a second selection window of device sub-types will be displayed. This window allows you to select the best output mode for this particular query. For example, sub-types can be used to change the print quality (draft or letter quality), size (narrow or wide paper), and orientation (landscape or portrait) of the printer.

**To a File**

If you want to send your output to a file, you can use the SET statement to assign a file name that will contain the query results. The SET statement must precede all other SQL statements in your query, except for the READ or DECLARE statements. (For more information on using the SET statement and the valid output formats, refer to the Format and SET Statement entries and to Parameters within a SET statement in the *KB_SQL Syntax Guide*.)

The two SET statements below produce the same result: the query results are printed to a file in the same format that would be used if the results were printed on paper.

```
SET FORMAT=REPORT
SET FILE='C:\DATA.TXT', FORMAT='REPORT'
```

If you use the SET FORMAT=REPORT statement, you will be prompted to enter a filename. If the file already exists, and you want the contents to be overwritten, you must answer YES to the Replace if exists prompt in the Output to file window.
Exiting the Current Query

To exit your current query and return to the Query Name window:

- Select the <Quit> option. If you made any changes to the query since the last time you saved it, you will be given the choice to save the changes. After you select YES or NO and press [enter], the Query Name window will appear.

-Or-

- Press [skip] from the menu bar.

**IMPORTANT:** KB_SQL automatically saves your query when you press [skip].

Exiting the SQL Editor

To exit the Editor:

- From the Query Name window, press [skip].
This lesson presents numerous examples of the SQL SELECT statement and the results that the statement produces. If your DBA loaded the demonstration queries, you can try them on-line as you read about them in this lesson. You can either select the existing queries or create a new query and enter the statements yourself.

This lesson explains:

‰ How to use the SELECT statement to retrieve information from your database
‰ How to use tests to restrict the results of your query
‰ How to join tables and summarize results
‰ How to customize the report’s format
Introduction

This lesson presents several examples of the SQL SELECT statement. SQL statements consist of a series of clauses. Each clause starts with a key word (e.g., SELECT) followed by one or more values. The values are usually the names of tables and columns in the data dictionary. Usually, a given statement includes some clauses that must be specified and optional clauses that are not required but can be included to modify the statement’s result.

Note: For the purpose of identification, this manual prints key words in all capital letters. However, it is not necessary for you to capitalize the key word when typing it in the SQL text window.

In order to execute the demonstration queries presented in this lesson, you must follow the steps outlined below. Each step is described in more detail in Lesson 1: The Basics.

1. Select the SQL EDITOR option from the selection window.

2. Select the demonstration query name from the list of available queries (displayed when you press [list] in the Query Name window).

   If you do not have the demonstration queries available to you, you will need to insert a new query. In the SQL text window, select the <Edit> option from the menu bar and type the statements as they appear in this manual.

3. Select the <Run> option to execute the query.
The SELECT Statement

The SELECT statement is the basis for all queries and reports. The statement must have both a SELECT clause and a FROM clause. The SELECT clause starts with the key word SELECT, followed by one or more values. Each value is separated by a comma. If you use the default report formatting features, these values appear in the report as the column names.

The FROM clause starts with the key word FROM, followed by the names of the tables you want to access. The values are retrieved from these tables.

The SELECT statement can also include any or all of the following four optional clauses: WHERE conditions, GROUP BY values, HAVING conditions, and ORDER BY values. The purpose of each clause is described below.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Identifies values to display</td>
</tr>
<tr>
<td>FROM</td>
<td>Identifies the tables which contain the data</td>
</tr>
<tr>
<td>WHERE</td>
<td>Restricts which rows are searched</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>Summarizes the results by specific values</td>
</tr>
<tr>
<td>HAVING</td>
<td>Restricts which results are returned</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Sorts the results</td>
</tr>
</tbody>
</table>
Selecting all Rows and Columns

This example uses the DEMO_SELECT_ALL query. The SELECT statement in DEMO_SELECT_ALL is used to retrieve all rows and columns from the EMPLOYEES table identified in the FROM clause.

DEMO_SELECT_ALL

```
SELECT * 
FROM employees
```

Result

```
DEMO_SELECT_ALL
Printed on 03/22/93 at 3:44 PM

<table>
<thead>
<tr>
<th>EMP_SSN</th>
<th>NAME</th>
<th>SALARY</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>102-44-3545</td>
<td>JOHNSON</td>
<td>9.00</td>
<td>616-02-6316</td>
</tr>
<tr>
<td>142-62-0335</td>
<td>BIXTER</td>
<td>7.50</td>
<td>616-02-6316</td>
</tr>
<tr>
<td>144-36-7056</td>
<td>SMITH</td>
<td>8.00</td>
<td>621-04-0911</td>
</tr>
<tr>
<td>203-12-5909</td>
<td>KING</td>
<td>25.00</td>
<td></td>
</tr>
<tr>
<td>216-56-7993</td>
<td>JONES</td>
<td>10.50</td>
<td>621-04-0911</td>
</tr>
<tr>
<td>323-44-2104</td>
<td>CANNON</td>
<td>11.00</td>
<td>621-04-0911</td>
</tr>
<tr>
<td>116-02-6316</td>
<td>GREEN</td>
<td>16.00</td>
<td>283-12-9509</td>
</tr>
<tr>
<td>521-04-0911</td>
<td>ROBERTS</td>
<td>10.50</td>
<td>283-12-9509</td>
</tr>
</tbody>
</table>
```

Before writing a “real” query, you can use the SELECT * syntax to review the contents of a table, including column names and formats. The asterisk means to select all columns. Because the query does not contain a WHERE clause, the query will also select all rows.

Notice that the result is formatted as a table and includes headers for each column. These headers are the column names from the EMPLOYEES table. In addition, there is a two-line header at the top of the page. The first line is the query name, in this example DEMO_SELECT_ALL. The second line indicates the date and time the query was executed. This is valuable information for reports that contain time-sensitive data.
All of these formatting features are provided for you automatically by KB_SQL. You will see in subsequent examples that you can control these features, but for many reports the automatic format is all you will need.

Selecting Specific Rows and Columns

Most reports will display only certain rows and columns. You can specify particular columns to display by listing them in the SELECT clause. You can also control which rows are included by adding a WHERE clause with a restriction limiting which rows will be included in the result. This example uses the DEMO_SELECT_SPECIFIC_ROW_COLMN query.

DEMO_SELECT_SPECIFIC_ROW_COLMN

```
SELECT proj_no, task
FROM tasks
WHERE proj_no BETWEEN 100 AND 200
```

Result

```
PROJ_NO  TASK
----------------
 100  FORMULARY
 100  REPORTS
 100  IU'S
 200  HEMATOLOGY
 200  CHEMISTRY
 200  MICRO
End>
```
The SELECT statement in DEMO_SELECT_SPECIFIC_ROW_COLMN retrieves just two columns, PROJ_NO and TASK from the TASKS table. In addition, the WHERE clause restricts the result to only those rows that have a value between 100 and 200 for the PROJ_NO column. This restriction is formally referred to as a predicate; however, this manual uses the term test.

Notice that the sequence in which columns appear in the SELECT clause is also the sequence in which they appear in the report.

Also, make sure the column names are separated by commas. Do not place a comma after the last column in the SELECT clause.

Tests

KB_SQL supports a wide variety of comparison tests. For a complete list and description of these tests, refer to the “Tests” entry in the KB_SQL Syntax Guide.

If you want to combine two or more tests, you can use the AND key word to indicate that a row of data must pass all the tests to be included in the result, or you can use the OR key word to specify that a row of data needs to pass only one of the tests to be included.

When your query contains one or more tests and you use both the AND and OR operators, it is wise to group your tests within parentheses to avoid any ambiguity. KB_SQL evaluates conditions according to the SQL standards. For example, it applies the AND operator before it applies the OR operator. To ensure that your SQL statements are evaluated in the order that you want, use parentheses to group your test conditions.
Joining Tables

A join combines two or more tables into a single result table. KB_SQL performs a join when a test references columns from two or more tables. This test is referred to as a join condition.

The query DEMO_JOIN performs a join to combine the correct tasks with each project. The query returns four columns of data. The PROJECT and BUDGET values come from the PROJECTS table, and the TASK and STATUS values come from the TASKS table. Both tables contain a column called PROJ_NO. To distinguish between the two values, we use a prefix for each column name. The prefix consists of the table name followed by a period (e.g., PROJECTS.PROJ_NO and TASKS.PROJ_NO). This “table.column” syntax should be used in any SQL statement when the column name is included in more than one table.

DEMO_JOIN

```
SELECT project, budget, task, status
FROM projects, tasks
WHERE projects.proj_no = tasks.proj_no
  AND budget > 30000
ORDER BY project, task
```
In this query, we included a test (value = value) in the WHERE clause to restrict the result to rows with the PROJ_NO from the PROJECTS table equal to the PROJ_NO in the TASKS table. This test is a join condition because it references columns from more than one table.

A second test (value > value) restricts the result to rows with a BUDGET value greater than 30000. Because this test references a column from only one table, it is not a join condition.

The two tests are combined using the key word AND, indicating that both tests must be true for the row to be included in the result. You can use the key word OR to include rows that meet either test. In addition, you can add the word NOT to any test to reverse its meaning, providing a rich syntax for selecting result rows.

To avoid problems when joining tables, we suggest you follow these rules:

1. Make sure each table in the FROM clause is joined to at least one other table in the FROM clause.

2. For best performance, use primary keys when possible, and join as many primary keys as possible.
The next query joins three tables: PROJECTS, TASKS, and EMPLOYEES. The primary key from the PROJECTS table is used to join to one of the primary keys of the TASKS table. The LEADER column from the PROJECTS table is used to join to the primary key of the EMPLOYEES table.

```sql
DEMO_JOIN_3_TABLES
SELECT name CHANGED, project CHANGED, task
FROM projects, tasks, employees
WHERE projects.proj_no = tasks.proj_no AND
      projects.leader = employees.emp_ssn
ORDER BY project, task
```

Result

```
<table>
<thead>
<tr>
<th>NAME</th>
<th>PROJECT</th>
<th>TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROBERTS</td>
<td>LABORATORY</td>
<td>CHEMISTRY</td>
</tr>
<tr>
<td></td>
<td>HEMATOLOGY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MICRO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEDICAL</td>
<td>RECORDS</td>
</tr>
<tr>
<td></td>
<td>MEDICAL</td>
<td>RECORDS</td>
</tr>
<tr>
<td></td>
<td>PATIENT</td>
<td>E/E</td>
</tr>
<tr>
<td></td>
<td>REPORTS</td>
<td></td>
</tr>
<tr>
<td>GREEN</td>
<td>PHARMACY</td>
<td>FORMULARY</td>
</tr>
<tr>
<td></td>
<td>IV'S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPORTS</td>
<td></td>
</tr>
</tbody>
</table>

DEMO_JOIN_3_TABLES
Printed on 05/05/93 at 4:27 PM
```

Notice the effect of the CHANGED modifier on the NAME and PROJECT columns. The CHANGED modifier displays a value only when the value is different from the previous row. This eliminates duplicates. **Note:** If you use the CHANGED modifier, you must apply it to the column you want to affect and all columns to the left of that column in order to obtain the expected results.
Using Set Functions to Summarize Results

The previous queries have selected particular columns and rows from tables. However, many reporting requests need to summarize many rows of data into a more manageable form. These reports provide answers to questions like: How many? How much? Or how long? SQL satisfies this type of request by using set functions. A set function returns one value for a set of rows.

DEMO_COUNT uses the COUNT(*) function to return the total number of rows in the TASKS table that have a status of INCOMPLETE.

Functions do not supply column headers. To include a header for the column, this query uses an alias for the COUNT function. Another option would be to use the HEADING column modifier, as does the DEMO_GROUP query that follows. Both the ALIAS function and the HEADING column modifier are explained in more detail in the KB_SQL Syntax Guide.

DEMO_COUNT

```
SELECT COUNT(*) AS total
FROM tasks
WHERE status = 'INCOMPLETE'
```

Result

```
DEMO_COUNT
Printed on 08-05-96 at 10:58:56 AM

<table>
<thead>
<tr>
<th>TOTAL</th>
<th>4</th>
</tr>
</thead>
</table>

End>
```
### Set Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AVG(value)</code></td>
<td>Averages all non-null values</td>
</tr>
<tr>
<td><code>COUNT(*)</code></td>
<td>Counts all rows</td>
</tr>
<tr>
<td><code>COUNT(value)</code></td>
<td>Counts all non-null values</td>
</tr>
<tr>
<td><code>MAX(value)</code></td>
<td>Returns the maximum (highest) value</td>
</tr>
<tr>
<td><code>MIN(value)</code></td>
<td>Returns the minimum (lowest) value</td>
</tr>
<tr>
<td><code>SUM(value)</code></td>
<td>Returns the sum (total) all values</td>
</tr>
</tbody>
</table>

Most summary reports need more than a single answer. The `GROUP BY` clause allows you to produce one result row for each distinct set of `GROUP BY` values. This allows you to create reports like “Revenue by Department” or “Expense by Project.”

**DEMO_GROUP** provides a break down of incomplete tasks by project name. You can include additional values in the `GROUP BY` clause to create a more detailed breakdown.

```sql
SELECT project, count(*) AS 'Number of Incomplete Tasks' FROM projects, tasks WHERE projects.proj_no = tasks.proj_no AND status = 'INCOMPLETE' GROUP BY project
```

**Result**

```
<table>
<thead>
<tr>
<th>PROJECT</th>
<th>Number of Incomplete Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABORATORY</td>
<td>3</td>
</tr>
<tr>
<td>PHARMACY</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Printed on 08/05/96 at 11:00.51 AM
Subqueries

Sometimes you need the result of more than one query to obtain the desired product. By using tests with set functions, you can compare a value to the result of another query, usually referred to as a subquery.

The subquery feature allows a query to be embedded within another query. The embedded query is referred to as a subquery, and the query which contains the subquery is referred to as the primary query.

Assume we want to list all employees who make more than the average salary. The average salary can be determined by using the AVG function. However, a second query would be needed to retrieve employees with a salary GREATER THAN the average. This can be accomplished in one query using the subquery feature.

In DEMO_ONE_SUBQUERY, a subquery (SELECT AVG(salary) FROM employees) is embedded within the WHERE clause of the primary query (SELECT name, salary FROM employees WHERE...).

**DEMO_ONE_SUBQUERY**

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary)
               FROM employees)
```
In most respects subqueries are similar to primary queries except that subqueries return only one column and subqueries cannot have an ORDER BY clause.

Subqueries may also be embedded within other subqueries; i.e., you could determine which employees make more than the average salary of employees who work for Green.

Try this one on your own! You can find the answer on the next page.
For this discussion, refer to the EMPLOYEES table in Appendix C. The innermost subquery (SELECT emp_ssn FROM employees WHERE . . .) returns the emp_ssn for employee Green: 416-82-6316.

That emp_ssn is used for the other subquery to determine an average salary for employees who work for Green (Johnson @ 9 and Baxter @ 7.5).

That subquery returns the average salary for those employees: 8.25. The primary query lists all employees with a salary greater than 8.25.
Controlling the Report Format

KB_SQL provides default formatting to make your reports easy to read. It also provides you with features that let you format reports to your specifications. And, if you need to transfer your data to be used in a spreadsheet or included in a management presentation, you can select from the following file formats: COMMA, TAB, DIF, MAPPER, DELIMITED, REPORT, and FIXED.

**Note:** Your DBA establishes import and export methods for transferring data. These methods are customized for every site. For more information on transfer methods, refer to the Method entry in the KB_SQL Syntax Guide.

Column Modifiers

DEMO_COLUMN_MODIFIERS and DEMO_WRAP use a KB_SQL feature called column modifiers to alter the report format. Column modifiers are short clauses that follow each value in the SELECT clause. Each modifier specifies how the value should be displayed. In the first query the MANAGER column has two modifiers, CHANGED and DEFAULT 'NONE'. The CHANGED modifier suppresses the printing of duplicate values on subsequent lines. The DEFAULT modifier causes the text NONE to be displayed for any row with a null value for the MANAGER column. We suggest that if you do use the CHANGED modifier for any column that may have null values, you also use the DEFAULT modifier so you can distinguish between a null value and a duplicate value.

**Note:** If you use the CHANGED modifier, you need to apply it to the column you want to affect and all columns to the left of that column in order to obtain the expected results.
Refer to the “Column Modifiers” entry in the *KB_SQL Syntax Guide* for a complete list of column modifiers.

**DEMO_COLUMN_MODIFIERS**

```sql
SELECT manager CHANGED DEFAULT 'NONE',
       name HEADING 'EMPLOYEE' RIGHT 10,
       salary COLUMN 30,
       emp_ssn HEADING 'SOCIAL SECURITY NUMBER'
FROM employees
ORDER BY manager;
```

The vertical bar in the HEADING clause splits the column header into three lines.

**Result**

```
+-------------------+-----------------+---------------+---------------+
| MANAGER           | EMPLOYEE        | SOCIAL SECURITY|
| Manager Chngd     | Employee Name   | Number        |
| 203-12-9509       | GREEN           | 16.00 416-02-6316 |
|                   | ROBERTS         | 10.50 621-04-8911  |
| 416-02-6316       | JOHNSON         | 9.00 102-44-3545  |
|                   | JOHN           | 7.00 142-04-8335  |
| 621-04-0911       | SMITH           | 8.00 144-36-7956  |
|                   | JONES           | 10.50 216-56-7593  |
| 323-44-2104       | CANNON          | 11.00 283-04-8335  |
|                  | KING            | 25.00 283-12-9509  |
| End>              |                 |                |
```

**Note:** If you do not specify a justification column modifier, the justification is obtained from the data dictionary.

Lesson 2: Writing Queries 35
This query uses the WRAP and WORDWRAP column modifiers to demonstrate how they display text in a column. WORDWRAP breaks the lines at a space character, while WRAP breaks according to the column width.

**DEMO_WRAP**

```sql
SELECT proj_no,'Notify your project leader of your vacation plans before the end of the month' WORDWRAP 12 HEADER 'WORDWRAP'
    ,''Notify your project leader of your vacation plans before the end of the month' WRAP 12 HEADER 'WRAP'
FROM projects
WHERE proj_no = 100
HEADER 'WRAP and WORDWRAP Example'
```

**Result**

**WRAP and WORDWRAP Example**

<table>
<thead>
<tr>
<th>proj_no</th>
<th>WORDWRAP</th>
<th>WRAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Notify your project leader of your vacation plans before the end of the month</td>
<td>Notify your project leader of your vacation plans before the end of the month</td>
</tr>
</tbody>
</table>

End>
KB_SQL lets you control your report’s format based on the occurrence of a specific event. This feature, which is especially helpful for reports that require extensive formatting, is referred to as event blocks. An event block consists of the event name and, in most cases, one or more statements that are executed when the event occurs. KB_SQL supports several types of events. A description of each is shown below; their syntax is provided in the “Event Blocks” entry in the KB_SQL Syntax Guide.

<table>
<thead>
<tr>
<th>Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL</td>
<td>An event that occurs before the first result row is processed.</td>
</tr>
<tr>
<td>HEADER</td>
<td>An event that occurs at the top of each page. This event is used to override the default report header.</td>
</tr>
<tr>
<td>FOOTER</td>
<td>An event that occurs at the bottom of each page.</td>
</tr>
<tr>
<td>BREAK AT</td>
<td>An event that occurs before each different ORDER BY value has been processed.</td>
</tr>
<tr>
<td>BREAK AFTER</td>
<td>An event that occurs after each different ORDER BY value has been processed.</td>
</tr>
<tr>
<td>DETAIL</td>
<td>An event that occurs once for each result row. This event is used to provide more complex logic than is allowed in the SELECT clause for each result row.</td>
</tr>
<tr>
<td>FINAL</td>
<td>An event that occurs after all rows have been processed.</td>
</tr>
</tbody>
</table>

You may have one of each of the following events per query: DETAIL, FINAL, FOOTER, HEADER, and INITIAL.
You may have one BREAK AFTER or BREAK AT event for each value in the ORDER BY clause.

In DEMO_EVENT_BLOCKS, the HEADER event overrides the default report header, the BREAK AT event prints the PROJECT each time the value changes, and the FINAL event lists the count of incomplete tasks at the end of the report.

**DEMO_EVENT_BLOCKS**

```sql
SELECT task COLUMN 17 HEADING 'Task Number',
       task_no HEADING 'Task Number' FROM projects, tasks
WHERE projects.proj_no = tasks.proj_no
AND status = 'INCOMPLETE'
ORDER BY project, task

HEADER WRITE 'Incomplete Task List' CENTER 60
BREAK AT PROJECT WRITE project SKIP 2, 'Leader ' || leader
FINAL WRITE COUNT(=) 11 'Incomplete Tasks' SKIP 2
```

**Result**

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Task Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABORATORY</td>
<td></td>
</tr>
<tr>
<td>CHEMISTRY</td>
<td>5</td>
</tr>
<tr>
<td>BIOLOGY</td>
<td>3</td>
</tr>
<tr>
<td>MICRO</td>
<td>9</td>
</tr>
<tr>
<td>PHARMACY</td>
<td></td>
</tr>
<tr>
<td>REPORTS</td>
<td>2</td>
</tr>
</tbody>
</table>

4 Incomplete Tasks
End
The SELECT statement, discussed in the previous lesson, represents only a fraction of the capabilities provided by KB_SQL. In this lesson we will present additional SQL language statements that KB_SQL supports. They are divided into the following categories: the Data Definition Language (DDL), the Data Manipulation Language (DML), and the Data Control Language (DCL).

If your database administrator has loaded the demonstration queries, you can try them on-line as you read about them in this lesson. Either select the existing queries or create a new query and enter the statements yourself. Refer to Lesson 1: The Basics for detailed instructions on creating and selecting queries.

This lesson explains:

% How to create views and tables
% How to insert rows into tables
% How to delete rows and tables
% How to grant and revoke privileges
IMPORTANT: In order to use the demonstration queries in this lesson, you must have your database administrator use the SECURITY/USER EDIT option to assign the command qualifications shown in the window below.
Overview

The Data Manipulation Language (DML) lets you add, edit, and delete rows from the database. KB_SQL supports four DML statements: SELECT, INSERT, UPDATE, and DELETE. The SELECT statement was discussed in the previous lesson.

The Data Definition Language (DDL) lets you add, modify, and delete information in the data dictionary.

Using the Data Control Language (DCL), you can control access to tables, indexes, views, and queries.

<table>
<thead>
<tr>
<th>Language</th>
<th>Statement</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Manipulation Language (DML)</td>
<td>SELECT</td>
<td>Identifies the values to display</td>
</tr>
<tr>
<td></td>
<td>INSERT*</td>
<td>Adds new rows to a table</td>
</tr>
<tr>
<td></td>
<td>UPDATE*</td>
<td>Changes column values in a table</td>
</tr>
<tr>
<td></td>
<td>DELETE*</td>
<td>Removes rows from a table</td>
</tr>
<tr>
<td>Data Definition Language (DDL)</td>
<td>ALTER</td>
<td>Changes a data dictionary object</td>
</tr>
<tr>
<td></td>
<td>CREATE</td>
<td>Adds a new data dictionary object</td>
</tr>
<tr>
<td></td>
<td>DROP</td>
<td>Removes a data dictionary object</td>
</tr>
<tr>
<td>Data Control Language (DCL)</td>
<td>GRANT</td>
<td>Gives privileges to user groups</td>
</tr>
<tr>
<td></td>
<td>REVOKE</td>
<td>Takes privileges from user groups</td>
</tr>
</tbody>
</table>

*If your DBA sets the `CONFIGURATION/SITE EDIT/DEFAULT INFO/ROW COMMIT` prompt to YES, the INSERT, UPDATE, and DELETE statements will change successful rows and skip failed rows. Otherwise, if this prompt is set to NO, and an error occurs on any row that you are processing, the statement will terminate and no rows will be changed.
When making updates to the database or changes to privileges, the SQL Editor prompts you to confirm your actions by displaying the Modify Database window after you select the <Run> option.

You may answer YES to proceed with the process or NO to cancel the process. If you have not selected an output device, and your query is going to modify the database, and your site allows background processing, the SQL Editor will also display the Run in the Background window to let you choose whether or not you wish to run this process in the background.

If you chose to run the query in the background or if you selected an output device, and if your site has been configured to allow you to schedule the run of your query, the Query Start Parameters window will display.
After a query, which was to execute in the foreground, has completed or encounters an error, a system message displays the outcome of the query. The Message Options window lets you choose to View, Save, or Print the message.

The outcome of all queries, including those run in the background, may be viewed by selecting QUERY RUN LOG from the User Print Options window. Refer to Chapter 2 for more on this feature.
Creating and Using Views

Up to this point most of the queries have used the SELECT statement to retrieve information from one table. The real world is often much more complex. Many apparently simple queries require data from several tables. Correctly joining these tables can be tricky. Fortunately, the designers of SQL recognized this and provided an elegant solution, the view table.

A view table, or simply view, is the named result of a query. Views exist only in the data dictionary, and are not associated with any M globals. Because views lack physical existence they are sometimes referred to as virtual tables.

An experienced SQL programmer can write a complex query involving many tables, joins, and other clauses with the end result being a simple virtual table. Users can then access this view and build powerful queries from it, without needing to understand more difficult concepts. Once a view has been created, it looks like any other table. However, it is important to remember that views get data from other tables. Therefore when you change the tables that were used to create the view, the results of the query which uses the view also change.

Views can also provide a powerful security feature by limiting a user to particular columns and rows, and by helping to insulate queries from changes to the M global structure.
Creating a View

DEMO_CREATE_VIEW looks like a SELECT statement, but the CREATE VIEW <view name> AS clause produces a different result. Rather than translating the SQL into an M routine, KB_SQL executes the statement and responds with the message “View INCOMPLETE created.” This indicates that the view is now available for use. This view named INCOMPLETE is a virtual table containing the four columns PROJECT, BUDGET, TASK, and TASK_NO.

**DEMO_CREATE_VIEW**

```sql
CREATE VIEW incomplete AS
SELECT project, budget, task, task_no
FROM projects, tasks
WHERE projects.proj_no = tasks.proj_no
AND status = 'INCOMPLETE'
```

DEMO_USE_VIEW produces the same results as the SELECT statement in DEMO_CREATE_VIEW which joined two tables. Any SELECT statement that you would use on tables PROJECTS and TASKS may now be performed on the view INCOMPLETE. It is important to remember that all of the data listed in the result is from the PROJECTS and TASKS tables, the INCOMPLETE table does not really exist. When values in the PROJECTS and TASKS tables change, they will be reflected in queries that use this view.

**DEMO_USE_VIEW**

```sql
SELECT * FROM incomplete
```
Altering a View

Generally, when you compose a view you create a statement, run the statement, and evaluate the results. This often leads to refining the original statement and running it again. To refine a view definition, perhaps to add a new column or a new test, you may use the ALTER view statement. This statement lets you change the view definition without having to drop it and then recreate it. The syntax for the ALTER view statement is identical to the CREATE view statement, simply replace CREATE with ALTER.

DEMO.Alter_VIEW

```
ALTER VIEW incomplete AS
SELECT project, budget, task
FROM projects, tasks
WHERE projects.proj_no = tasks.proj_no
AND status = 'INCOMPLETE'
```
Run DEMO_USE_VIEW again to see the results of the ALTER statement.

Result

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>BUDGET</th>
<th>TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHARMACY</td>
<td>50000.00</td>
<td>REPORTS</td>
</tr>
<tr>
<td>LABORATORY</td>
<td>00000.00</td>
<td>HEMATOLOGY</td>
</tr>
<tr>
<td>LABORATORY</td>
<td>00000.00</td>
<td>CHEMISTRY</td>
</tr>
<tr>
<td>LABORATORY</td>
<td>00000.00</td>
<td>MICRO</td>
</tr>
</tbody>
</table>

**Dropping a View**

If a view is no longer needed it may be removed from the data dictionary using the DROP VIEW statement. For this example when the process completes, the message “View INCOMPLETE dropped” displays.

**DEMO_DROP_VIEW**

DROP VIEW incomplete
Creating Tables

Although views provide an extremely powerful capability, there are times when you will want to actually create a new base table. Unlike a view, a base table physically exists in the database. It actually stores the data in an M global. This is particularly useful when you want to preserve time-dependent information, for example year-to-date statistics, or extract a subset of data from a table for further study or refinement.

The CREATE TABLE statement adds a new table definition to the data dictionary. In the example below, we have created a table called INCOMPLETE_TASKS. This table contains five columns: STATUS_DATE, PROJECT, BUDGET, TASK, and TASK_NO. Each column is followed by a data type clause that specifies the type of value the column will contain. For example, the STATUS_DATE is a date value. The CHARACTER, NUMERIC, and INTEGER data type clauses include the length of the field. The NUMERIC data type clause also specifies the scale (or number of digits to the right of the decimal point).

<table>
<thead>
<tr>
<th>DEMO_CREATE_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE incomplete_tasks</td>
</tr>
<tr>
<td>(status_date DATE,</td>
</tr>
<tr>
<td>project CHARACTER(15),</td>
</tr>
<tr>
<td>budget NUMERIC(9,2),</td>
</tr>
<tr>
<td>task CHARACTER(15),</td>
</tr>
<tr>
<td>task_no INTEGER(3),</td>
</tr>
<tr>
<td>PRIMARY KEY (status_date, project, task_no))</td>
</tr>
</tbody>
</table>

Lesson 3: Using DDL, DML, and DCL Statements 49
The CREATE TABLE statement in the DEMO_CREATE_TABLE query also contains the optional PRIMARY KEY clause. The PRIMARY KEY clause ensures that there will be only one row for each unique combination of STATUS_DATE, PROJECT, and TASK_NO values. For this example when the process completes, the message "Table INCOMPLETE_TASKS created" displays.

**Note:** If you do not specify a primary key, KB_SQL will automatically create an "extra" column using the table name with a suffix " _ID." This column will be used as the primary key. For example, if the DEMO_CREATE_TABLE query didn’t have a primary key specified, KB_SQL would have built one called incomplete_tasks_id.

**Inserting Rows**

The INSERT statement adds rows to a table. A set of new rows can be added by using the result of a SELECT statement, or by importing a non-M data file. A single row may be added by specifying a list of values.

**Note:** Tables that were created using the CREATE TABLE statement can be modified. However, other tables can be modified only if your DBA has enabled this feature.
In this example we use the result of a SELECT statement to add new rows to the INCOMPLETE_TASKS table. After you execute this query, KB_SQL displays the message “4 rows inserted” indicating that it added four new rows to our table.

**DEMO_INSERT_ROW**

```sql
INSERT INTO incomplete_tasks (status_date, project, budget, task, task_no)
SELECT today, project, budget, task, task_no
FROM projects p, tasks t
WHERE p.proj_no = t.proj_no
AND status = 'INCOMPLETE'
```

If you try to run this query more than once on the same day, an error will occur because the table definition allows only one row to be added for each distinct set of primary key values (status_date, project, task_no). The first time the query is executed each day, the results will be distinct. Each subsequent time the query is run that day, the results will be duplicates of the first run, and therefore rejected as not distinct.
Pseudo Columns and Table Alias Names

The SELECT statement in DEMO_INSERT_ROW contains two new features: pseudo columns and table alias names. The SELECT clause contains the value TODAY that is not a column in either the PROJECTS or TASKS tables. The value TODAY will always be equal to the current date, and allows us to time-stamp each new row. This value is referred to as a pseudo column. Pseudo columns may be referenced like columns, but are not stored in a table. Instead, pseudo columns allow the DBA to provide a name for the result of an M expression.

Your DBA may create additional pseudo columns to make your system more flexible. For example, you can use pseudo columns to store values that may change over time (such as tax rates) or values that are different at various clients (client name and address).

The FROM clause references two tables, PROJECTS and TASKS. After each table name is a one-character table alias which is used to rename the table (e.g., the PROJECTS table is renamed P). This allows the WHERE clause to reference the shorter column name P.PROJ_NO instead of PROJECTS.PROJ_NO.
You can also use the table alias feature to distinguish between two or more references to the same table. In DEMO_TABLE_ALIAS we produced a list of the names of managers (M.NAME) and the employees that report to them (E.NAME) by joining two copies of the EMPLOYEES table WHERE M.EMP_SSN = E.MANAGER.

**DEMO_TABLE_ALIAS**

```sql
SELECT m.name CHANGED HEADER 'Manager Name',
      c.name CHANGED HEADER 'Employees Name'
FROM employees m, employees e
WHERE m.emp_ssn = e.manager
ORDER BY m.name, c.name
BREAK AT m.name
```

Result

<table>
<thead>
<tr>
<th>Manager Name</th>
<th>Employees Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEN</td>
<td>BAXTER</td>
</tr>
<tr>
<td></td>
<td>JOHNSON</td>
</tr>
<tr>
<td>KING</td>
<td>GREEN</td>
</tr>
<tr>
<td></td>
<td>ROBERTS</td>
</tr>
<tr>
<td>ROBERTS</td>
<td>CANNON</td>
</tr>
<tr>
<td></td>
<td>JONES</td>
</tr>
<tr>
<td></td>
<td>SMITH</td>
</tr>
</tbody>
</table>

Lesson 3: Using DDL, DML, and DCL Statements
We are now able to retrieve rows from the new table (see DEMO_USE_NEW_TABLE). However, unlike the view, the rows in the INCOMPLETE_TASKS table are separate from the rows in the PROJECTS and TASKS tables. Any subsequent changes to values in the TASKS table will not affect the INCOMPLETE_TASKS table.

**DEMO_USE_NEW_TABLE**

```
SELECT *
FROM incomplete_tasks
```

**Result**

<table>
<thead>
<tr>
<th>STATUS_DATE</th>
<th>PROJECT</th>
<th>TASK_NO</th>
<th>BUDGET</th>
<th>TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/09/93</td>
<td>LABORATORY</td>
<td>3</td>
<td>00000.00</td>
<td>HEMATOLOGY</td>
</tr>
<tr>
<td>03/09/93</td>
<td>LABORATORY</td>
<td>5</td>
<td>00000.00</td>
<td>CHEMISTRY</td>
</tr>
<tr>
<td>03/09/93</td>
<td>LABORATORY</td>
<td>9</td>
<td>00000.00</td>
<td>MICRO</td>
</tr>
<tr>
<td>03/09/93</td>
<td>PHARMACY</td>
<td>2</td>
<td>50000.00</td>
<td>REPORTS</td>
</tr>
</tbody>
</table>

End>
Granting and Revoking Privileges

Security access within SQL is controlled by a user’s privileges. Privileges allow a user to perform the SELECT, INSERT, UPDATE, and DELETE commands on tables, and to run and edit queries. By default, a table or query is available to the DBA and all users in the group of the user who created the table or query.

To print or display a list of all tables to which you have access:

- Select the <User> option from the menu bar. The Print Table List window will appear. Type YES or NO to each of the prompts, depending on how much information you want to view and how you want it presented.

  ![Print Table List]

  - Print all tables from all schemas? YES
  - Print table and column descriptions? NO.
  - Start a new page for each table? YES

- Either select a print device or press [enter] to view the list on your screen.

You can use only those tables for which you have privileges. Other table names will not be displayed by the <User> option or recognized by the system.
The GRANT Statement

The GRANT statement is used to give specific privileges to a particular group of users. In this example we will give use of the SELECT statement on the INCOMPLETE_TASKS table to the group of users identified as USERS. When the process is complete, the message “GRANT command processed” displays.

DEMO_GRANT

```
GRANT SELECT ON incomplete_tasks TO users
```

The REVOKE Statement

The REVOKE statement is used to take privileges away from a group of users. This example removes all privileges, including SELECT, from the specified group of users. When the process is complete, the message “REVOKE command processed” displays.

DEMO_REVOKE

```
REVOKE ALL ON incomplete_tasks FROM users
```
Public Privileges

To make the task of managing security easier, the SQL Editor allows you to grant and revoke privileges to all users by using the PUBLIC user group. Tables with public privileges are available to any user.

DEMO_PUBLIC

```
GRANT SELECT ON incomplete_tasks TO public
```
Deleting Rows

The DELETE statement allows you to remove some or all of the rows from a table. This example will remove all rows for the laboratory project. After you run the query, KB_SQL displays a message indicating the exact number of rows deleted. By using a DELETE statement without the WHERE clause, we can remove all rows from the designated table.

DEMO_DELETE_ROWS

```
DELETE incomplete_tasks
WHERE project = 'LABORATORY'
```

Dropping a Table

If the table is no longer needed, use the DROP statement to remove the definition from the data dictionary. For this example, when the process completes, the message “Table INCOMPLETE_TASKS dropped” displays. Any data rows in the table will be deleted; when you drop the table definition you delete the data also.

DEMO_DROP_TABLE

```
DROP TABLE incomplete_tasks
```
Combining Several Features of KB_SQL

The previous examples have demonstrated many of the fundamental features of the SQL Editor. By incorporating several of the features in a single query, you can produce reports that would be difficult to write with systems not based on SQL.

DEMO_MANY_FEATURES uses several new features as well as some already covered to produce a cross-tab type report, listing projects ranked by the percentage of incomplete tasks.

Note: In the query below what appears to be a colon following PROJECT, PERCENT, and TOTAL is actually a vertical bar. The vertical bar splits the header into two lines.

```sql
SELECT project_link#project HEADING 'PROJECT NAME',
       COUNT(when status = 'INCOMPLETE' THEN 1) AS INCOMPLETE,
       ROUND(INCOMPLETE/COUNT(ALL) * 100, 2) AS PERCENT_INCOMPLETE,
       COUNT(when status = 'COMPLETE' THEN 1) AS COMPLETE,
       ROUND(COMPLETE/COUNT(ALL) * 100, 2) AS PERCENT_COMPLETE,
       COUNT(*) AS TOTAL_TASKS
FROM tasks
GROUP BY 1
ORDER BY 3 DESC
HEADER 'Projects by Percentage of Incomplete Tasks' CENTER 80
```

Result:

<table>
<thead>
<tr>
<th>PROJECT NAME</th>
<th>INCOMPLETE</th>
<th>PERCENT</th>
<th>COMPLETE</th>
<th>PERCENT</th>
<th>TOTAL TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABORATORY</td>
<td>3</td>
<td>100.00</td>
<td></td>
<td>0.00</td>
<td>3</td>
</tr>
<tr>
<td>PHARMACY</td>
<td>1</td>
<td>50.00</td>
<td>2</td>
<td>66.67</td>
<td>3</td>
</tr>
<tr>
<td>MEDICAL RECORDS</td>
<td>0.00</td>
<td></td>
<td>3</td>
<td>100.00</td>
<td>3</td>
</tr>
</tbody>
</table>

Lesson 3: Using DDL, DML, and DCL Statements
The first value referenced by the SELECT clause is a foreign key link to the PROJECTS table (i.e., the PROJECT_LINK@PROJECT value). Foreign keys are similar to pointers in other database systems, and they allow you to access other tables without a join. Refer to the “Foreign Key” entry in the KB_SQL Syntax Guide for more information on foreign keys.

Note: The use of @ is a KB_SQL extension, and is not typically found in other SQL systems. Refer to the “Column” entry in the KB_SQL Syntax Guide for more information on the @ character.

The second value in the SELECT clause is a conditional expression. It will count only those rows with an incomplete status. The key word AS is used to rename the result INCOMPLETE. This rename feature is called a column alias. It allows you to provide a simple name for complex expressions or long column names. Refer to the “Alias” and “Expressions” entries in the KB_SQL Syntax Guide for more information on conditional expressions and column aliases.

The third value in the SELECT clause performs a mathematical calculation using the alias INCOMPLETE and the count of rows for each project to determine the percentage of incomplete tasks. This result is then rounded to a number with two digits on the right of the decimal point. Refer to the ROUND function in the KB_SQL Syntax Guide for more information on rounding off numbers.

The GROUP BY clause uses a shorthand notation (1) to refer to the first value in the SELECT clause.

The ORDER BY clause uses a similar notation to sort the results by the third value in the SELECT clause, the percentage of incomplete tasks. The key word DESC indicates that the sort order will be descending, from the highest percentage to the lowest. (Sorts are usually performed from lowest to highest.) Refer to the “SELECT Statement” entry in the KB_SQL Syntax Guide for more information on the ORDER BY clause.
When you select the <Run> option to execute your query, KB_SQL first prepares your query. This consists of three steps: the parse step, the plan step, and the build step. (These steps are explained in more detail in Chapter 2.) The plan step determines the most efficient way to access the data. It is this data access plan that we are going to discuss in this lesson.

When executing a query with a SELECT, INSERT, UPDATE or DELETE statement, it is essential that the query executes efficiently. Although the plan step automatically optimizes the query, it is important for you to verify that the data access plan is reasonable. This is particularly important for queries using views. To make verification easier, the SQL Editor offers a Show Plan feature that lets you view the data access plan.

This lesson:

- Explains the contents of a data access plan
- Discusses local optimization techniques
- Presents examples of a variety of queries and their plans
- Discusses performance issues
How to Obtain the Data Access Plan

To view the data access plan:

1. Select the query whose plan you want to view.
2. Type \texttt{SET SHOW\_PLAN=YES} in the SQL text window.
3. Select the \texttt{<Run>} menu option.

In general, SET statements should be placed before other statements in your query. The SET \texttt{SHOW\_PLAN} statement works only for DML statements (\texttt{INSERT}, \texttt{UPDATE}, \texttt{DELETE}, \texttt{SELECT}). When the Show Plan feature is enabled, the translation process will pause after the plan step and display a text window with the data access plan.

\textbf{Note:} The Show Plan feature may also be used in the \texttt{EZQ Editor}. Select the \texttt{<Setup>} option and type \texttt{YES} at the Show Plan prompt in the Setup Parameters window.

The Components of the Plan

The data access plan is structured as an outline with four levels:

1. The Query #.
2. The Step #.
4. The ‘Optimize primary key’ or ‘Constrain primary key’ entries.
**PLAN COMPONENTS**

<table>
<thead>
<tr>
<th>Query #n [or part #n] Estimated cost=xx result=xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #n</td>
</tr>
<tr>
<td>Get table xxxx [using xxxx]</td>
</tr>
<tr>
<td>Optimize primary key xxxx</td>
</tr>
<tr>
<td>Use index for ORDER BY clause</td>
</tr>
<tr>
<td>Constrain primary key xxxx</td>
</tr>
<tr>
<td>Constrain results</td>
</tr>
<tr>
<td>Get view xxxx</td>
</tr>
<tr>
<td>Save result in temporary index</td>
</tr>
<tr>
<td>Get result of step n</td>
</tr>
<tr>
<td>Constrain OJ results</td>
</tr>
</tbody>
</table>

---

**Note:** Depending on how your DBA has configured the SITE EDIT option, your plan costs and results may differ from the examples in this lesson.
Terminology

Query #

Each Query # entry represents a separate search subroutine. Every command that builds an M routine contains at least a Query #1 entry. Each Query # clause contains one or more Step # entries. If the command contains views, subqueries, or the Boolean OR operator, the plan may contain additional Query # entries.

Views and subqueries may create additional query entries such as Query #2, Query #3. The OR operator may create additional entries which contain the phrase ‘or part’, for example ‘Query #1 or part 1’ and ‘Query #1 or part 2’.

The Plan # entry indicates the number of plans built before the most optimal one was determined. KB_SQL will not completely build another plan, if it determines that the earlier plan is better.

The ‘Estimated cost’ entry indicates the relative cost of searching the database.

The ‘result’ entry indicates the expected number of rows in the result.

Step #

Each Step # contains one or more ‘Get table’, ‘Get view’, ‘Get result’, or ‘Constrain results’ entries.

Get table

The ‘Get table’ entry specifies that a table will be searched. If the entry contains a ‘using’ clause, then the table will be accessed using the specified index.
Get view

The 'Get view' entry specifies that a previously computed view will be combined at this point. If this entry is present, another Query # clause will exist which computes the view. In many cases, the plan step compresses the plan that builds the view with the plan that searches the view. When the two are combined into one plan, the data access plan does not show a 'Get view' clause.

Save result in temporary index

The 'Save result' entry indicates that the result of this step will be saved in a temporary index (or cross reference) for later use by a another step.

Get result of step

The 'Get result of step' entry means the result of a previous step will be combined into this step.

Constrain results

The 'Constrain results' entry specifies that one or more tests will be applied to limit the results of the step.

Optimize primary key

The 'Optimize primary key' clause indicates the search of this key will be optimized. Tests that can be optimized include IN, LIKE, BETWEEN, =, >, <, >=, and <=.
Constrain primary key

The 'Constrain primary key' clause indicates the results will be constrained, by applying a test, after the key has been searched. If the last primary key is constrained this is equivalent to the 'Constrain results' clause.

Constrain OJ results

The 'Constrain OJ results' clause indicates that the query is retrieving information from a view that contains an outer join table, and the query contains a test against a table in the view that is not the outer join table. (The outer join table is marked by a "+"). KB_SQL tries to apply the query’s test during the build of the view so that fewer rows of data are built and therefore fewer rows of data need to be searched.
Examples

Simple Query - Without a WHERE Clause

This query will retrieve all rows from a single table. Because there is no WHERE clause or ORDER BY clause, there is no optimization. The plan step will simply search every row in the base table.

DEMO_SP_SIMPLE_QUERY

SET SHOW_PLAN = YES
SELECT NAME FROM EMPLOYEES

PLAN

Query #1  Plan #1  Estimated cost=8  result=8
Step #1
Get table EMPLOYEES
**Query with ORDER BY Clause**

In this example, the DBA has defined an index for the column NAME during the mapping process. Because the EMP_BY_NAME index exists, this query can optimize on the ORDER BY clause. If the index had not been defined, KB_SQL would have to build a temporary index which would increase the cost of the query.

<table>
<thead>
<tr>
<th>DEMO_SP_ORDER_BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET SHOW_PLAN = YES</td>
</tr>
<tr>
<td>SELECT NAME, SALARY FROM EMPLOYEES</td>
</tr>
<tr>
<td>ORDER BY NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query #1  Plan #3  Estimated cost=2  result=2</td>
</tr>
<tr>
<td>Step #1</td>
</tr>
<tr>
<td>Get table EMPLOYEES using EMP_BY_NAME</td>
</tr>
<tr>
<td>Use index for ORDER BY clause</td>
</tr>
</tbody>
</table>
Constrain Results - Single Table Search

This query contains a WHERE clause that does not constrain any primary key values. The plan step will search all rows in the base table EMPLOYEES and then limit the results based on the SALARY data value. Because all of the rows must be searched, the cost of this query is the same as a query without the WHERE clause. But the size of the result is less!

DEMO_SP_CONSTRAIN_RESULTS

<table>
<thead>
<tr>
<th>SET SHOW_PLAN = YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT NAME FROM EMPLOYEES WHERE SALARY &gt; 10</td>
</tr>
</tbody>
</table>

PLAN

| Query #1 Plan #2 Estimated cost=8 result=4 |
| Step #1 Get table EMPLOYEES Constrain results |
Optimize Primary Key - Single Table

The WHERE clause in this query references the primary key of the EMPLOYEES table. The plan step will use the test to limit the number of rows searched, reducing the cost of the query. Using >, <, or = will optimize the query search. Using BETWEEN and IN may optimize the search or at least constrain it.

**DEMO_SP_OPTIMIZE_PK**

```
SET SHOW_PLAN = YES
SELECT NAME FROM EMPLOYEES
WHERE EMP_SSN > '200-00-0000'
```

**PLAN**

<table>
<thead>
<tr>
<th>Query #1</th>
<th>Plan #2</th>
<th>Estimated cost=4</th>
<th>result=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
<td></td>
<td>Get table EMPLOYEES</td>
<td>Use &gt; on EMP_SSN to optimize search</td>
</tr>
</tbody>
</table>

Lesson 4: Using the Show Plan Feature
Constrain Primary Key - Single Table

The WHERE clause in this query also references the primary key of the EMPLOYEES table. The not equals (<> constraint cannot be used to optimize the search. Notice that the result is the same as the Optimize Primary Key query but the cost is greater.

The plan step will use the test to limit the number of rows included in the result. Using <> or the CONTAINS operator ( ) in a query will constrain the search.

**DEMO_SP_CONSTRAIN_PK**

```sql
SET SHOW_PLAN = YES
SELECT NAME FROM EMPLOYEES
WHERE EMP_SSN <> '200-00-0000'
```

**PLAN**

<table>
<thead>
<tr>
<th>Query #1</th>
<th>Plan #2</th>
<th>Estimated cost=8 result=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get table EMPLOYEES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use &lt;&gt; on EMP_SSN to limit search</td>
<td></td>
</tr>
</tbody>
</table>
Optimize Primary Key with Constraints

This query contains two tests that are combined with the AND operator. The test against the EMP_SSN will optimize the primary key and reduce the search cost. The test referencing SALARY will further constrain the results or the search.

DEMO_SP_OPT_PK_CONSTRAINTS

SET SHOW_PLAN = YES
SELECT NAME FROM EMPLOYEES
WHERE EMP_SSN > '200-00-0000' AND SALARY > 10

PLAN

Query #1  Plan #2  Estimated cost=4  result=4
Step #1
  Get table EMPLOYEES
  Use > on EMP_SSN to optimize search
  Constrain results
Two-Table Join on Primary Keys

This query references two tables by comparing the PROJ_NO columns. The PROJ_NO column is used as a primary key in both tables. The plan step will determine which table to search first, and then use the results to optimize the primary key of the second table.

DEMO_SP_TWO_TABLE_JOIN

| SET SHOW_PLAN = YES                  |
| SELECT PROJECT, TASK                |
| FROM PROJECTS, TASKS               |
| WHERE PROJECTS.PROJ_NO = TASKS.PROJ_NO |

PLAN

Query #1 Plan #2 Estimated cost=13 result=9
Step #1
  Get table PROJECTS
  Get table TASKS
  Use = on PROJ_NO to optimize search
**One-Part OR Query**

This query contains two tests combined by the OR operator. In this example the plan step will choose to search the entire table and then apply both tests to the result. This produces the lowest cost because the test against the SALARY column, which is not a primary key, would require a complete search of the table.

**DEMO_SP_ONE_PART**

```sql
SET SHOW_PLAN = YES
SELECT NAME FROM EMPLOYEES
WHERE SALARY > 10 OR EMP_SSN > '200-00-0000'
```

**PLAN**

```
Query #1  Plan #2  Estimated cost=8  result=4
Step #1
  Get table EMPLOYEES
  Constrain results
```
**Multi-Part OR Query**

This query references two tables joined together using both the AND and the OR operator. Because the OR operator takes precedence over the AND operator, this query is essentially an OR query. The plan step will solve the query using separate plans, identified by the ‘or part’ clause, and then combine the results. The alternative would require joining all of the entries in both tables. Therefore, the multi-part plan provides superior performance.

**DEMO_SP_MULTIPART_OR**

```sql
SET SHOW_PLAN = YES
SELECT PROJECT, TASK
FROM PROJECTS, TASKS
WHERE PROJECTS.PROJ_NO = TASKS.PROJ_NO
  AND PROJ_NO = 100
  OR
  PROJECTS.PROJ_NO = TASKS.PROJ_NO
  AND PROJ_NO = 300
```

**PLAN**

1. **Query #1 or part 1**
   - Plan #2
   - Estimated cost=3 result=2
   - Step #1
     - Get table PROJECTS
       - Use = on PROJ_NO to optimize search
     - Get table TASKS
       - Use = on PROJ_NO to optimize search

1. **Query #1 or part 2**
   - Plan #2
   - Estimated cost=3 result=2
   - Step #1
     - Get table PROJECTS
       - Use = on PROJ_NO to optimize search
     - Get table TASKS
       - Use = on PROJ_NO to optimize search
Two-Step Query

This query performs a two-table join on columns that are not part of the primary key of either table. This type of join is referred to as a data-driven join. While this is an uncommon form of query, it provides a powerful analytical tool. Because a primary key is not used, the system tries to optimize by building a temporary index.

Step 1 will search the PROJECTS table and create a temporary index to projects by the LEADER column. The PROJECTS table was chosen because it is smaller than the EMPLOYEES table and therefore building a temporary index for it would take less resources. Step 2 will search the EMPLOYEES table and then join to the temporary index created by Step 1.

Note: Queries of this type may require considerable computer resources to create the temporary index. If an alternative join clause is available that would use primary key columns, it should be utilized instead of, or in addition to, the data-driven join clause.
DEMO_SP_TWO_STEP
SET SHOW_PLAN = YES
SELECT PROJECT, NAME FROM PROJECTS, EMPLOYEES
WHERE LEADER = MANAGER

PLAN
Query #1  Plan #2  Estimated cost=28  result=8
Step #1
  Get table PROJECTS
  Save result in temporary index
Step #2
  Get table EMPLOYEES
  Get result of step #1
Two-Plan Query - Using Subquery

This query produces two plans, one for the primary query and one for the subquery. Because subqueries contain a FROM clause, and optionally a WHERE clause, the plan step will always create at least one plan for each subquery. The primary query is always Query #1, and therefore subqueries will be Query #2-n.

Note: The Query # does not indicate the actual execution sequence of the queries. Some subqueries (non-correlated subqueries) and all views will be performed before the primary query (Query #1). Other subqueries (correlated subqueries) may be performed during the execution of the primary query.

### DEMO_SP_TWO_PLAN

```sql
SET SHOW_PLAN = YES
SELECT NAME, SALARY FROM EMPLOYEES
WHERE SALARY > (SELECT AVG(SALARY) FROM EMPLOYEES)
```

### PLAN

<table>
<thead>
<tr>
<th>Query #2</th>
<th>Plan #1</th>
<th>Estimated cost=8</th>
<th>result=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
<td>Get table EMPLOYEES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query #1</th>
<th>Plan #2</th>
<th>Estimated cost=8</th>
<th>result=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
<td>Get table EMPLOYEES</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constrain results</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
One-Plan Query - Using a View

This query uses a view instead of a base table. The view represents the results of a query—in this case the preliminary query. Queries of this type will sometimes require two plans: one for the query that produces the view, and one for the query that references the view. In this example the plan step is able to combine both plans eliminating the need to produce the view as an intermediate result. The first 'Constrain results' reflects the WHERE clause in the view definition, and the second 'Constrain results' is the WHERE clause in the query that references the view. This type of optimization is extremely useful for most views.

DEMO_SP_ONE_PLAN_STEP1

CREATE VIEW ONEPLAN AS
SELECT PROJECT, TASK, STATUS
FROM PROJECTS  P, TASKS T
WHERE P.PROJ_NO = T.PROJ_NO
AND TASK='REPORTS'

DEMO_SP_ONE_PLAN_STEP2

SET SHOW_PLAN = YES
--CREATE VIEW ONEPLAN AS
--SELECT PROJECT, TASK, STATUS
--FROM PROJECTS  P, TASKS T
--WHERE P.PROJ_NO = T.PROJ_NO
--AND TASK='REPORTS'

SELECT * FROM ONEPLAN
WHERE TASK IS NOT NULL

PLAN
Query #1  Plan #2  Estimated cost=13  result=9  

Step #1  
- Get table PROJECTS P  
- Get table TASKS T  
  - Use = on PROJ_NO to optimize search  
- Constrain results  
- Constrain results

**Constrain OJ Results**

This query is similar to the one-plan query using a view, except that the view in this query contains an outer join—indicated by the plus sign following PROJECTS in the CREATE view statement. Because of the outer join, all projects will be preserved (included) whether or not they have a task.

The outer join changes how the view can be optimized. The view is still compressed with the query into a single plan, but the query’s WHERE clause ‘TASK is not null’ is applied after the view has been built and appears in the plan as a ‘Constrain OJ results.’

```
CREATE VIEW OJPLAN AS
SELECT PROJECT, TASK, STATUS
FROM PROJECTS+ P, TASKS T
WHERE P.PROJ_NO = T.PROJ_NO
AND TASK='REPORTS'
```
DEMO_SP_CONSTRAIN_OJ_STEP2

SET SHOW_PLAN = YES
--CREATE VIEW OJPLAN AS
--SELECT PROJECT, TASK, STATUS
--FROM PROJECTS+ P, TASKS T
--WHERE P.PROJ_NO = T.PROJ_NO
--AND TASK='REPORTS'

SELECT * FROM OJPLAN
WHERE TASK IS NOT NULL

PLAN

Query #1    Plan #2    Estimated cost=13  result=9
Step #1
Get table PROJECTS P OJ
Get table TASKS T
   Use = on PROJ_NO to optimize search
Constrain results
Constrain OJ results
Two-Plan Query - Using a View

This query uses a view that cannot be compressed because the view definition contains a GROUP BY clause. The plan step will use 'Query #2' to compute the value of the view, and then process the results using 'Query #1'. Views of this type often require more resources than views that can be compressed.

DEMO_SP_TWO_PLAN_STEP1
CREATE VIEW TWOPLAN (MANAGER, PAY_BUDGET) AS
SELECT MANAGER, SUM(SALARY)
FROM EMPLOYEES
WHERE MANAGER IS NOT NULL
GROUP BY MANAGER

DEMO_SP_TWO_PLAN_STEP2
SET SHOW_PLAN = YES
-- CREATE VIEW TWOPLAN (MANAGER, PAY_BUDGET) AS
-- SELECT MANAGER, SUM(SALARY)
-- FROM EMPLOYEES
-- WHERE MANAGER IS NOT NULL
-- GROUP BY MANAGER

SELECT * FROM TWOPLAN
WHERE PAY_BUDGET > 20
<table>
<thead>
<tr>
<th>Query #2  Plan #2  Estimated cost=12  result=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
</tr>
<tr>
<td>Get table EMPLOYEES</td>
</tr>
<tr>
<td>Constrain results</td>
</tr>
<tr>
<td>Query #1  Plan #2  Estimated cost=4  result=2</td>
</tr>
<tr>
<td>Step #1</td>
</tr>
<tr>
<td>Get view TWOPLAN</td>
</tr>
<tr>
<td>Constrain results</td>
</tr>
</tbody>
</table>
**Optimize Foreign Key**

KB_SQL optimizes fkey@column references in the WHERE clause by converting the foreign key to a multi-table join. Joins provide more optimization options than foreign keys.

**DEMO_SP_FOREIGN_KEY**

```sql
SET SHOW_PLAN = YES
SELECT TASK
FROM TASKS
WHERE PROJECT_LINK@PROJ_NO=200
```

**PLAN**

```
Query #1  Plan #2  Estimated cost=3  result=2
Step #1
  Get table PROJECTS fkey_opt_51
  Use = on PROJ_NO to optimize search
  Get table TASKS
  Use = on PROJ_NO to optimize search
Constrain results
```
Equivalent Predicate Optimization (EPO)

The equivalent predicate optimization (EPO) technique allows additional tests to be built that will join or constrain tables. The optimum plan always applies tests at the first possible opportunity, to minimize the size of intermediate results. The EPO process evaluates the tests provided by the WHERE clause and attempts to generate additional algebraically equivalent tests. These new tests may be applied before the original predicates or in concert with the original predicates to produce a plan with a lower cost.

For example, consider the three-table join shown below:

```
SELECT a, b, c FROM ta, tb, tc WHERE ta.a = tb.b AND tb.b = tc.c
```

This query contains two tests: (ta.a=tb.b) & (tb.b=tc.c). Depending on the size and structure of the underlying M globals, the most optimal data access plan might require searching ta first, followed by tc, and then finally tb. However, the query did not specify a join between tables ta and tc. In reality, users are often unaware of the size and structure of the M globals. In fact, much of the appeal of a relational database is due to its non-procedural approach which allows you to specify what will be accomplished without worrying about how it will be accomplished! And it is certainly desirable for the optimizer to attempt to understand your intent, and produce the best possible access plan.

This is where the EPO technique provides its benefits. The EPO logic produces additional tests before the other optimization techniques are applied. In this example, since ta.a=tb.b and tb.b=tc.c, then the test ta.a=tc.c is also true. It is then possible for the other optimization techniques to use the new test to create the optimal access plan, without requiring you to “tune” your query.
Performance Issues

When queries experience performance problems it’s often because of an incomplete or missing join constraint. If your query contains an incomplete or missing join constraint, the plan step may perform a full or partial Cartesian product, consuming tremendous resources. The plan step will attempt to warn the user of a Cartesian product situation if more than one table is referenced, and at least one table is not constrained in the WHERE clause. However, either the use of the Boolean OR operator, or some missing join constraints can cause a partial Cartesian product, which may not be reported, but can adversely affect the query performance. Even more important than the performance, a Cartesian product may produce well-formatted but incorrect results.

The primary warning sign for a Cartesian product is a ‘Get table’ entry that does not contain either an ‘Optimize primary key’ or ‘Constrain primary key’ clause. And, the ‘Get table’ entry follows a ‘Get’ entry in the same step. The ‘Get table’ will be searched once for each entry returned by the first ‘Get’. The first ‘Get’ in each step does not need any primary key optimization or constraints. Only the ‘Get table’ entries after the first ‘Get’ indicate a performance issue. Any ‘Get view’ or ‘Get result’ will automatically provide an optimized index structure and should not be a performance problem.

Other things that may affect performance include views that must be created by a second query, correlated subqueries, and multi-step queries. These types of queries may require building temporary data structures that could be large or require more than one pass through the database.
A view definition that contains an outer join, a set function (COUNT, AVG, MAX, MIN, or SUM), a GROUP BY clause, or a HAVING clause may require a separate query to build the view. If the amount of data (rows) in the view is large, building the view could require considerable resources.

**Cartesian Product**

This query references two tables in the FROM clause, but does not contain a WHERE clause. Note the second 'Get table' does not contain either an 'Optimize primary key' or a 'Constrain primary key' clause. Also note that the cost is much higher than the query with a WHERE clause (see Two-Table Join on Primary Keys).

<table>
<thead>
<tr>
<th>DEMO_SP_CARTESIAN_PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET SHOW_PLAN = YES</td>
</tr>
<tr>
<td>SELECT PROJECT, TASK</td>
</tr>
<tr>
<td>FROM PROJECTS, TASKS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query #1 Plan #1 Estimated cost=52 result=36</td>
</tr>
<tr>
<td>Step #1</td>
</tr>
<tr>
<td>Get table PROJECTS</td>
</tr>
<tr>
<td>Get table TASKS</td>
</tr>
</tbody>
</table>
Build View

This query references a view. Because the view contains a GROUP BY clause, the view must be built before being referenced. If the cost of building the view is large, the cost of performing the query will increase.

**DEMO_SP_BUILD_VIEW_STEP1**

```sql
CREATE VIEW BUILDPLAN (MANAGER, COUNT_EMP) AS
SELECT MANAGER, COUNT(*)
FROM EMPLOYEES
GROUP BY MANAGER
```

**DEMO_SP_BUILD_VIEW_STEP2**

```sql
SET SHOW_PLAN = YES
-- CREATE VIEW BUILDPLAN (MANAGER, COUNT_EMP) AS
-- SELECT MANAGER, COUNT(*)
-- FROM EMPLOYEES
-- GROUP BY MANAGER

SELECT *
FROM BUILDPLAN
```

**PLAN**

```
Query #2  Plan #1  Estimated cost=16  result=8
  Step #1
    Get table EMPLOYEES
Query #1  Plan #1  Estimated cost=8  result=8
  Step #1
    Get view BUILDPLAN
```
Multi-Step Query - No Primary Key Constraint

A multi-step query can adversely affect performance if the temporary index is large. This query references two tables in the FROM clause, but the join condition in the WHERE clause does not include any primary key columns. This query is valid, but because a primary key is not used, the cost of performing the query will increase if the tables contain many rows. Ideally, in these cases, KB_SQL creates a temporary index on the smallest table.

DEMO_SP_MULTI_STEP

SET SHOW_PLAN = YES
SELECT PROJECT, NAME
FROM PROJECTS, EMPLOYEES
WHERE LEADER = MANAGER

PLAN

Query #1  Plan #2  Estimated cost=28  result=8
Step #1
Get table PROJECTS
Save result in temporary index
Step #2
Get table EMPLOYEES
Get result of step #1
Correlated Subquery

A correlated subquery, is a subquery that requires information from a parent query. This type of subquery may be executed once for each row of the parent query. Depending on how the plan step is able to optimize the subquery, subqueries may increase the cost of performing the query.

This query contains a correlated subquery that references the E.EMP_SSN column from the parent query. The subquery must be executed for each row produced by the parent query. If the parent query processes a large number of rows, or if the subquery must process a large number of rows, the performance time of the query will increase.

DEMO_SP_CORRELATED_SUBQUERY

```sql
SET SHOW_PLAN = YES
SELECT NAME
FROM EMPLOYEES E
WHERE EXISTS (SELECT NAME
              FROM EMPLOYEES
              WHERE E.EMP_SSN = MANAGER)
```

PLAN

```sql
Query #2  Plan #2  Estimated cost=8  result=4
  Step #1
  Get table EMPLOYEES
  Constrain results
Query #1  Plan #2  Estimated cost=8  result=4
  Step #1
  Get table EMPLOYEES E
  Constrain results
```
It is not always possible to avoid all resource-intensive queries. In fact, much of the power and flexibility of the KB_SQL system is because of its ability to combine any set of tables or views regardless of the physical implementation of the underlying M globals. However, it is important to assess the potential cost on any complex query or view and determine if a less expensive solution exists.
Part II

Components of the Interface
The SQL Editor is composed of several interactive windows. Although each window performs a different function, the interface and conventions are consistent for all windows. This chapter introduces you to the various window types and to the keys that you use to interact with the SQL Editor.

Because of the wide range of terminals and devices in use by the M language, your interface may not appear exactly as the figures shown in this manual. Also, because KB_SQL lets you customize the interface, your database administrator may have modified the interface to make it more consistent with your own conventions.
Text Windows

*Text windows* provide simple word-processing functions. You can use the backspace key to eliminate the character preceding the cursor and the delete key to erase the character above the cursor.

The SQL text window, shown below, is an example of a text window.

```
SELECT *
FROM employees
```

Value Windows

*Value windows* allow the entry of single-valued fields. The overwrite mode is the default mode for typing text in a value window. To insert type into existing text in a value window, press [insert] before typing.

The Query Name window is an example of a value window.
Selection Windows

Selection windows let you pick an item from a list of possible choices. In a selection window, the choices may be displayed vertically or horizontally, depending on the organization of the window.

If there are more choices to view in a vertical-display selection window, a plus sign will appear in the lower right-hand corner of the window. Press [Page Down] to display the next page of items. If a minus sign appears in the upper right-hand corner of a vertical-display selection window, press [Page Up] to view the previous page of choices.

The Select USER OPTIONS window is an example of a vertical-display selection window.

The Save Before Exit window is an example of a horizontal-display selection window.
The Function Keys

To interact with the SQL Editor you use keyboard characters and special functions. Basic functions, such as [help], [list], [insert], and [delete] are accessed by pressing a particular key. Pressing [enter] processes the current option or value. For example, when the cursor is positioned over an option, [enter] means *perform that option*. When the cursor is positioned at a prompt field in a value window, [enter] means *edit and process that value*.

Depending on the terminal that you are using, the actual function key(s) you press to transmit a function varies. The functions can be mapped to be consistent with the key sequences used by your organization. For example, the [skip] function may be assigned to [F2] at one site and [F4] at a different site. For these reasons when we refer to a function, we refer to its name not to the key(s) to which it may be assigned.

The most commonly used functions along with their associated keytop (e.g., skip=F4) are shown at the bottom of the screen. The function keys that the SQL Editor displays vary depending upon the action that you are performing. To view all valid function keys, press the key assigned to the [keys] function.
Following are the descriptions of the most commonly used function keys.

<table>
<thead>
<tr>
<th>Press:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[enter]</td>
<td>process input and continue</td>
</tr>
<tr>
<td>[skip]</td>
<td>accept current values, and if all values are valid, exit the window</td>
</tr>
<tr>
<td>[list]</td>
<td>produce a list of values</td>
</tr>
<tr>
<td>[keys]</td>
<td>see a list of all available function key values</td>
</tr>
<tr>
<td>[insert]</td>
<td>add a new entry</td>
</tr>
<tr>
<td>[delete]</td>
<td>remove an entry</td>
</tr>
<tr>
<td>[undo]</td>
<td>undo any changes within the window and exit the window</td>
</tr>
<tr>
<td>[help]</td>
<td>display the on-line reference</td>
</tr>
</tbody>
</table>

*Note: To stop the execution of a query, press the space bar.*
The Cursor Movement Keys

To move around in the interface, use the cursor movement keys (arrow keys). The [→] and [←] keys are handy for selecting menu bar options. The [↑] and [↓] keys are useful when selecting an entry from a vertical list or when moving among prompt fields in a value window.

<table>
<thead>
<tr>
<th>Press:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[↑]</td>
<td>move up 1 line or row</td>
</tr>
<tr>
<td>[→]</td>
<td>move 1 space or item to the right</td>
</tr>
<tr>
<td>[←]</td>
<td>move 1 space or item to the left</td>
</tr>
<tr>
<td>[↓]</td>
<td>move down 1 line or row</td>
</tr>
</tbody>
</table>

Note: If you use the arrow keys to move from field to field in a value window, any value that you type will be processed as if you had pressed the [enter] key. However, if you press an arrow key and the value you typed is not accurate, KB_SQL will prompt you to correct your entry and will not transfer you to the direction you specified by the arrow key.
Menu Bar Options

Introduction

This chapter discusses the function of each of the SQL Editor’s menu bar options. To select an option, either highlight its name and press [enter] or press the first letter of the option followed by [enter].

The SQL Text Window
Clear Option

The <Clear> option in the SQL Editor interface is used to remove the text of an old query from the SQL text window. This option is similar to the <Edit> option, except that it deletes the contents of the SQL text window before positioning the cursor in the window.

If you accidentally select this option, you can retrieve the SQL text by immediately returning to the menu bar, selecting the <Quit> option, and answering NO at the Save Before Exit window. Then select your query again from the Query Name window.

Edit Option

The <Edit> option in the SQL Editor interface positions the cursor within the SQL text window and allows you to edit the current query. While in Edit mode you can access lists of SQL commands/key words, functions, tables, and columns by pressing [list]. You can also invoke the on-line reference system by pressing [help].

Using the List Feature

From the SQL text window, you can press [list] to display a selection window of available lists of the SQL commands/key words, functions, tables, and columns. When you pick a list type, the system displays all valid entries for that type. For example, selecting MODIFIERS causes a list of all valid column modifiers to appear. Hint line descriptions are provided for most list types.
The TABLES list and COLUMNS list display a value window and allow you to enter a partial match to the table or column name to restrict the length of the list. The TABLES list displays the schema name to the right of the table name.

When you pick an item from a list, KB_SQL inserts the item into the SQL window at the current cursor position. If you do not want to insert the entry, press [skip].

Using the On-Line Reference

You can invoke HyperHelp (the on-line reference system) by pressing [help] from the SQL text window. This feature allows you to view the KB_SQL Syntax Guide, using a hypertext-style interface.

Exiting the SQL Text Window

After you have finished editing the text in the SQL text window, you may press [skip] to return to the menu bar.
Halt Option

If your system supports running queries in the background, this option in the SQL Editor interface lets you stop queries that you have started. Each query running in the background checks a halt flag periodically, and if the flag is set, the query stops executing. This process may take a few seconds to complete.

If a query has been halted it cannot be restarted. The <Halt> option displays a selection window containing all the queries that you have started that are either running or waiting to run.

If the query you want to stop is not displayed in the selection window, then it is no longer under the control of KB_SQL. This can happen under operating systems which support print spooling. If this happens, you will need to contact your DBA or computer operator to stop the query.

If you select one of the displayed options, the Halt query prompt will appear. If you answer YES to this prompt, KB_SQL will set a halt flag for this query. If you answer NO, KB_SQL will reset the halt flag.
The `<Info>` option in the SQL Editor interface displays the Query Information window. This window allows you to edit the query name and description before proceeding to the SQL text window. Following is a description of the prompts in the Query Information window.

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query name</td>
<td>You must enter a query name so that KB_SQL can distinguish one query from another. The query name must be a valid SQL_IDENTIFIER. KB_SQL will automatically convert any spaces to underscores.</td>
</tr>
<tr>
<td>Routine</td>
<td>New queries and queries that have not yet been run do not usually have a value for the M routine name. This is because KB_SQL does not assign a routine name until the first time the query is run. If the DBA has designated you as a programmer in the User Edit procedure, you may edit this field. When editing the routine name, make sure you do not overwrite an existing routine.</td>
</tr>
</tbody>
</table>

Note: An SQL_IDENTIFIER is a name, starting with a letter (A-Z), followed by letters, numbers (0-9), or underscores `_`. The last character in the name cannot be an underscore. The length of the name must not exceed 30 characters.
Description: character (60)
This description helps you identify the query. You may type in any
characters including punctuation and lowercase letters.

Run message: character (60)
A run message is optional. If you enter a run message, it will appear
above the Print on Device window when the query is executed. Using a
run message allows you to provide important runtime instructions, such
as the necessary paper type or size, to the person that is executing the
query.

The Last edit date/time, Compiled date/time, By user, and Run
date/time prompts are all maintained by KB_SQL.
The <Print> option in the SQL Editor interface allows you to print the contents of the SQL text window and other query-related information to an output device. If the query has not been edited after the last compile, the data access plan will print along with the SQL text. The data access plan displays the same information that is displayed when the SHOW_PLAN parameter is used. The data access plan is a valuable tool for determining how the query is executed. This feature makes it easier to debug and document queries by providing both the SQL and the plan in the same printout.

### Print on Device

**Device name:** character (30) [list]

To print the query definition on the screen, press [enter] at the Device name prompt. To send the report to a printer, enter the printer’s name, or press [list] to view all valid output devices.

If the device you select has more than one output mode, a second selection window of device subtypes may be displayed from which you can select the best output mode for this report. For example, subtypes can be used to change the print quality (draft or letter quality), size (narrow or wide paper), and orientation (landscape or portrait) of the printer.
Sample output from <Print> option

Site Name: XB Systems, Inc. [1]          XB_SQL_U3.0
Query Name: DEMO1                          Routine: SQLT4
Printed: 02/28/95 at 11:46 AM
Description: Demonstration query
Last edit: 02/28/95 at 11:45 AM by DBA using SQL Editor
Last compile: 02/28/95 at 11:46 AM

SQL Text
=======
SELECT *
FROM employees

Access Plan
============
Query #1 Plan #1 Estimated cost=0 result=0
  Step #1
    Get table EMPLOYEES
End>
Quit Option

The <Quit> option in the SQL Editor interface exits the SQL text window and returns you to the Query Name window. If you changed the text in the SQL text window, KB_SQL will give you the option to save the changes.

Save Before Exit

If you want to save the changes, answer YES; otherwise, answer NO.
Run Option

To execute your commands in the SQL text window, select the <Run> option from the menu bar. The <Run> option first prepares your query. This consists of three steps: the parse step, the plan step, and the build step. The output from these three steps is an M routine. After the <Run> option prepares the query, it executes the M routine. The first time a query is run, or when the query or any related object has been edited, the <Run> option first saves the query and then prepares and executes it.

Parsing the Query

The parse step checks both the syntax of the SQL statement and the security privileges of the user. This ensures that the statement is valid and the user is authorized to perform the particular command on the designated database tables. During the parse step, KB_SQL reports any errors that may exist in the query.
The Plan Step

The plan step evaluates the statement and determines the best (fastest) method for performing the statement. All possible data access paths are considered, including pointers and indices. To review the plan before the build step, just add the line SET SHOW_PLAN=YES before the SELECT statement in your query. For more information on the Show Plan feature, please refer to Lesson 4 in this manual.

During the plan step, the system determines if the query requires a Cartesian product of the referenced tables. A Cartesian product occurs when two or more tables are referenced, and there is no join condition between the tables. This means each row in each table is joined to every row in every other table. If a Cartesian product is detected, the plan step displays the following window:

```
WARNING: 0659: Query contains cartesian product
Continue? NO
```

Unless you intentionally want to produce a Cartesian product, press [enter] to answer NO. This will terminate the translation and return you to the menu bar.

The Build Step

The build step actually writes an M routine. When the query is run, this routine is executed. After KB_SQL has built an M routine, the translation is not repeated unless the SQL statement is edited, or some table, column, or other related object referenced by the query changes.
Save Option

The <Save> option saves any changes you made to your query in the SQL text window. Before saving the query, KB_SQL displays the Save As window.

Save As

Query name:  
Character (30)

To save the query using the current name, press [enter]. To save the SQL commands as a new query, enter a different name. Typing a new query name will not rename the old query! To rename a query you must use the Query Information window. After KB_SQL saves the changes you made, it returns you to the menu bar.
**User Option**

The <User> option allows you to print a list of all tables to which you have access. And it lets you view a log of all the queries you have run.

**User Print Options**

When you select the PRINT TABLE LIST option, the window shown below appears.

**Print Table List**

Print all tables from all schemas? YES/NO

Print table and column descriptions? NO

Start a new page for each table? YES

---

Print all tables from all schemas? YES/NO

Answer YES to print all tables and views that you can access. To limit the report to a particular schema, answer NO, and the Select Schema window will appear.
Print table and column descriptions? YES/NO
Answer YES to print table and column descriptions in addition to the table and column names. This will make the report longer, but may provide valuable information. To skip the descriptions, answer NO.

Start a new page for each table? YES/NO
Answer YES to start each table on a new page. While this will leave unused space at the bottom of some pages, it makes it easier to add updated pages if some of the tables have changed.

Select Schema

Schema name: character (30) [list]
Enter a schema name or press [list] to display all valid schemas. The report will be limited to tables from the selected schema.

Print all tables? YES/NO
Answer YES to print all tables from the selected schema. Answer NO to specify a range of table names. If you answer NO, the Print User Tables window appears.
### Print User Tables

**From table:** character (30)
Enter a starting value for the range of table names that you would like to print.

**Thru table:** character (30)
Enter an ending value for the range of table names that you would like to print.

**Sample Table List**

<table>
<thead>
<tr>
<th>CHAR_DATE</th>
<th>KEY</th>
<th>CHAR_NAME</th>
<th>KEY</th>
<th>CHARACTER(11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_NAME</td>
<td>FOREIGN KEY TO EMPLOYEES</td>
<td>EMPLOYEE_NAME</td>
<td>CHARACTER(11)</td>
<td>NUMERIC(3,1)</td>
</tr>
<tr>
<td>PROJ_LINK</td>
<td>KEY</td>
<td>PROJ_NAME</td>
<td>FOREIGN KEY TO PROJECTS</td>
<td>INTEGER(3,0)</td>
</tr>
<tr>
<td>TASK_LINK</td>
<td>KEY</td>
<td>TASK_NAME</td>
<td>FOREIGN KEY TO TASKS</td>
<td>INTEGER(2,0)</td>
</tr>
</tbody>
</table>

**Table List For User Group DBAS**

For Schema SQL_TEST

From Table CHARGES thru EMPLOYEES

Printed on 05/19/93 at 4:37 PM
When you select the QUERY RUN LOG option, the window shown below appears.

Print User Query Run Log

You may constrain the output of the report by entering a date range. All queries executed within that date range will be included in the report. A sample report is shown below.

Sample Query Run Log

<table>
<thead>
<tr>
<th>Time</th>
<th>Query</th>
<th>Total Time</th>
<th># rows searched</th>
<th># rows selected</th>
<th># pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:16 PM</td>
<td>SQL_SCHEMA_ZZZ</td>
<td>0:00:06</td>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1:14 PM</td>
<td>SQL_FUNCTION_LIST</td>
<td>0:00:02</td>
<td>80</td>
<td>80</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>err: 0116: Query execution halted by user request</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:03 AM</td>
<td>CREATE_VIEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>err: 0357: Task has been deleted from queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:37 AM</td>
<td>DEMO_EVENT_BLOCKS</td>
<td>0:00:01</td>
<td>13</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

page 3, more below, press ENTER to continue, any other key to stop>
Part III
Appendixes
The KB_SQL product employs the benefits of a relational database, the simplicity of the Structured Query Language (SQL), and the power of M to give you easy, efficient access to your M globals.

This appendix provides an overview of SQL and M, and it gives you an introduction to relational databases: how they are structured and why they are used. For those new to the concept of a relational database, we suggest you refer to Appendix B for a list of additional publications on this topic.
The Relational Database Model

The relational database model combines a simple presentation of information with a powerful language for manipulating the data (SQL).

In a relational database, data is presented as a collection of tables. The result of every operation performed on one or more tables is another table. Each table is a two-dimensional structure, containing some number of rows and columns. Each column in a table has a unique name and contains a particular type of data such as characters, numbers or dates. Each row contains one value for each of the columns in the table.

Regardless of the complexity of your M globals, the tabular structure provided by a relational database makes it easier for you to understand and access your data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel, William</td>
<td>123 Madonna Ln</td>
<td>Sterling</td>
<td>765-7901</td>
</tr>
<tr>
<td>Abrams, George</td>
<td>142 Rolfe St</td>
<td>Fairfax</td>
<td>698-3823</td>
</tr>
<tr>
<td>Adams, Alice</td>
<td>3243 Wakley Ct</td>
<td>Vienna</td>
<td>979-2904</td>
</tr>
<tr>
<td>Adams, Stephen</td>
<td>12 Woods Ave</td>
<td>Fairfax</td>
<td>790-9773</td>
</tr>
<tr>
<td>Adham, Frances</td>
<td>104 Argyle Dr</td>
<td>Olney</td>
<td>237-9499</td>
</tr>
<tr>
<td>Ahmed, Jamil</td>
<td>32 Pelican Ct</td>
<td>Ashburn</td>
<td>450-0284</td>
</tr>
</tbody>
</table>
Why Relational?

The underlying principle of relational databases is that data is presented as a set of tables, each with multiple rows and columns. Each table can be related to another table in the database at any time. This design allows for easy access to data. Consequently, relational systems have become the most widely accepted way to manage information.

1. **Relational databases are easy to use.**

Relational database management systems are easier to use than systems based on the hierarchical or network models. In hierarchical and network databases, the user must understand the structure of the underlying M globals and correctly navigate through these structures. Failure to accomplish this task can result in slow responses and incorrect results.

In a relational database, the user sees the data as a collection of rectangular tables, consisting of rows and columns. The user is not concerned how the data is actually stored, and can be insulated from changes to the M global structure. This makes the database easier to learn for new users. It also allows for a high-level data manipulation and control language with automatic handling of global structures.

2. **Relational databases provide automatic optimization.**

Access to data in a full-fledged relational system is optimized by the system rather than by programmers or users. In a large and complex database, determining an efficient access path is difficult, requiring knowledge of all available data structures and the distribution of data. Favorable access paths may be rendered unfavorable or even inoperable when storage structures change.

In contrast, a well designed optimizer is adaptable to changes to the data dictionary and is ever improving. Whenever a data dictionary change occurs, the optimizer checks all available access paths and
quickly computes the relative cost for each approach. As new techniques for optimization are developed, these can be incorporated into new KB_SQL releases in a way entirely transparent to users and existing applications.

3. Relational databases provide controlled access to information. Databases are shared resources. As such, they must be protected from unauthorized access and updates that violate data consistency. In relational systems, privileges control access to information. The privilege to select, insert, update, and delete table information can be granted or revoked from a user group. In addition, you can control access to rows, columns, and even particular values in a table.

4. Relational databases provide multiple views of data. In SQL, all data manipulation operations produce new tables by combining old ones. With a single statement, a user can retrieve an entire collection of records and name the result. This result, which SQL calls a view can be queried just like a table. This ability to create many different logical views of the database is one of the most powerful features provided by the SQL environment. Users can be presented with data in the way they want to see it regardless of the number or complexity of tables involved. Views can be used to simplify the user’s perception of the database. Views can complement the security system, limiting users’ access to predefined subsets of data. Overall, views allow the same data to be seen by different users in different ways at the same time.
5. Relational databases improve productivity.
The fundamental concern of application developers is how to improve both the quality and pace of applications development. Relational databases are a significant step forward in the improvement of both developer and end user productivity. A fully implemented relational system presents a user with a simple data description and a language that allows him to ask questions instead of writing programs. For the application programmer, it lifts programming to the level of problem solving. For the casual user, it provides a method for easy access and manipulation of the information in the database.

Relational Tables. Each table contains columns that are used to access information from other tables. For example, we can use the ORDERS table’s ACCT_NO to access information from the VISITS table by comparing the values of the ACCT_NO columns. The column names do not have to be identical in order to compare their values. Also the columns being compared do not need to be key fields.
Structured Query Language (SQL) is a relational language used to define, query, manipulate, and control access to information stored in a relational database. SQL is a nonprocedural language: the user specifies what to do, not how to do it.

Developed by the IBM research center in 1974, SQL is a concise, powerful, and comprehensive database language. SQL has gained widespread acceptance by developers as well as end users. In 1979, Oracle introduced the first commercial implementation of the relational database based on the SQL database language. In 1981, IBM followed with SQL/DS for the DOS/VSE environment.

SQL came under review of the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in the 1980s. SQL was declared the standard language for relational database management systems in 1986. The standard was revised in 1989 and 1992, and is currently undergoing another revision. The KB_SQL product conforms to the ANSI and ISO definitions of SQL.

SQL is now implemented by many vendors across many different computing environments. It is widely used by commercial applications and its many uses include Decision Support Systems (DSS) and client/server applications.
M (formerly known as MUMPS) is a programming language/operating system with an embedded database structure, a powerful instruction set, and true multi-tasking, multiuser capabilities. Originally developed by Massachusetts General Hospital in 1968 on PDP minicomputers, M has migrated to all types of hardware platforms, from mainframes to micros. It runs as a stand-alone operating system or under many other operating systems including VM, VMS, UNIX, AOS, DOS, and OS/2. Although M started in the medical world and is often used there, it has no medical bias. It is used worldwide for financial, inventory control, manufacture processing, and other general business applications. In 1984, M was the third language (after COBOL and FORTRAN) to be standardized by ANSI.

M is a powerful, procedural language. It provides fast, random access to information and can support a large number of simultaneous users. SQL is a powerful, nonprocedural language. It uses simple English-like statements to access, manipulate, define, and control access to information stored in a database. Together, M and SQL provide a strong foundation on which to build your applications.

KB Systems is committed to providing software for all M systems. KB Systems uses only standard M for our products. Along with our support for standard SQL, this ensures that your investments in M and SQL technologies will be preserved.
How Does KB_SQL Translate SQL to M?

The first time a query is run, or when the query or any related object (e.g., a table, view, or function) changes, KB_SQL automatically saves the changes and translates the statement into an M routine. The translation is composed of three distinct steps: parse, plan, and build. For more information on these steps, refer to the “Run Option” entry in Part II of this manual.
For more information on relational databases and SQL, we suggest the following books:

For end users or anyone new to SQL:
Date, C.J. *Database A Primer*. Addison Wesley, 1983.

For programmers:
Date, C.J. *An Introduction to Database Systems*. Addison Wesley, 1986.

For database experts:
Date, C.J. *Relational Database Selected Writings*. Addison Wesley, 1986.
For a copy of the American National Standard SQL document, please contact:

American National Standards Institute, Inc.
(ANSI)
1430 Broadway
New York, NY 10018


A useful companion to this document is:
In order to run the queries in the tutorial, you must have access to the demonstration tables. The three demonstration tables that follow comprise a simple project management system database. If these tables do not appear in your table list, contact your database administrator (DBA). You can obtain a list of the demonstration tables to which you have access, by selecting the <User> option from the SQL Editor’s menu bar and choosing to print only those tables in the SQL_TEST schema.
The EMPLOYEES Table

There is an entry in the EMPLOYEES table for every employee in the project management system. Each employee has an employee number (SSN), a last name, an hourly salary, and a manager. The manager for the employee is specified by the manager’s employee number.

<table>
<thead>
<tr>
<th>EMP_SSN</th>
<th>NAME</th>
<th>SALARY</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>102-44-3545</td>
<td>JOHNSON</td>
<td>9.00</td>
<td>416-82-6316</td>
</tr>
<tr>
<td>142-62-8335</td>
<td>BAXTER</td>
<td>7.50</td>
<td>416-82-6316</td>
</tr>
<tr>
<td>144-36-7056</td>
<td>SMITH</td>
<td>8.00</td>
<td>621-84-8911</td>
</tr>
<tr>
<td>203-12-9509</td>
<td>KING</td>
<td>25.00</td>
<td></td>
</tr>
<tr>
<td>216-56-7593</td>
<td>JONES</td>
<td>10.50</td>
<td>621-84-8911</td>
</tr>
<tr>
<td>323-44-2104</td>
<td>CANNON</td>
<td>11.00</td>
<td>621-84-8911</td>
</tr>
<tr>
<td>416-82-6316</td>
<td>GREEN</td>
<td>16.00</td>
<td>203-12-9509</td>
</tr>
<tr>
<td>621-84-8911</td>
<td>ROBERTS</td>
<td>18.50</td>
<td>203-12-9509</td>
</tr>
</tbody>
</table>

* - denotes a primary key
There is an entry in the PROJECTS table for each project defined in the system: Pharmacy, Medical Records, Laboratory, and Accounting. Each project has a project number, name, budget amount, and project leader. The project leader column contains the employee number of the employee who leads the project.

<table>
<thead>
<tr>
<th>PROJ_NO</th>
<th>PROJECT</th>
<th>BUDGET</th>
<th>LEADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>PHARMACY</td>
<td>50000.00</td>
<td>416-82-6316</td>
</tr>
<tr>
<td>200</td>
<td>LABORATORY</td>
<td>80000.00</td>
<td>621-84-8911</td>
</tr>
<tr>
<td>300</td>
<td>MEDICAL RECORDS</td>
<td>30000.00</td>
<td>621-84-8911</td>
</tr>
<tr>
<td>400</td>
<td>ACCOUNTING</td>
<td>55000.00</td>
<td>416-82-6316</td>
</tr>
</tbody>
</table>

* - denotes a primary key
The TASKS Table

There is an entry in the TASKS table for each task associated with a project. The task status is either COMPLETE or INCOMPLETE.

### TASKS data

<table>
<thead>
<tr>
<th>TASK_NO</th>
<th>PROJ_NO</th>
<th>TASK</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>FORMULARY</td>
<td>COMPLETE</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>REPORTS</td>
<td>INCOMPLETE</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>IV'S</td>
<td>COMPLETE</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>HEMATOLOGY</td>
<td>INCOMPLETE</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>CHEMISTRY</td>
<td>INCOMPLETE</td>
</tr>
<tr>
<td>9</td>
<td>200</td>
<td>MICRO</td>
<td>INCOMPLETE</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>MERGE PATIENT</td>
<td>COMPLETE</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>REPORTS</td>
<td>COMPLETE</td>
</tr>
<tr>
<td>12</td>
<td>300</td>
<td>PATIENT E/E</td>
<td>COMPLETE</td>
</tr>
</tbody>
</table>

- denotes a primary key
Alias: A temporary alternate name for tables, views, and values.

Base table: A table that contains values that are actually stored in an M global. Unlike a view table, a base table physically exists in the database.

Break: A change in the value of a column or an expression used in an ORDER BY clause. See Event.

Cartesian product: A join which combines every row from the first table with every row from the second table; a Cartesian product occurs when you specify more than one table name in the FROM clause and do not specify a join predicate.

Clause: A group of words starting with a key word (e.g., SELECT) that compose an SQL statement.

Column alias: A name assigned to complex expressions or long column names.

Column modifiers: A KB_SQL feature that allows you to alter your report’s format; a clause in the SELECT statement that modifies the display format of a value.
Columns: Components of tables and views. Each column has a name that must be different from all other column and foreign key names within the same table or view.

Data Control Language: Statements that are used to give and take away privileges to access the database and to see database data. DCL statements let one user allow other users to see, change, and use data in tables.

Data Definition Language: Statements that are used to create database objects, modify them, and delete (drop) them.

Data dictionary: A collection of information regarding the tables and columns in the relational database. To use KB_SQL to access any values from your M database, your database administrator (DBA) must first enter the names and structures of the M globals into the data dictionary.

Data Manipulation Language: Statements used to add, edit, and delete rows from the database.

Data type: A basic classification of data used for formatting and comparisons. See Domain.

Default value: A value that KB_SQL uses when the user does not specify an alternative value.

Domain: A user-defined implementation of a data type; a variation on the way the value is stored. For example, date is a data type. Billing date and birth date could be treated as domains of the date data type.

Event blocks: A KB_SQL feature that allows you to control report formatting based on the occurrence of a specific event. An event block consists of the event name and one or more procedural statements. See Event.
**Event**: Denotes at what point something happens during the execution of a query. Typical events include: INITIAL event (when the query starts), FINAL event (when the query finishes), HEADER event and FOOTER event (when the display page begins and ends), and BREAK AT and BREAK AFTER events (when a different value, designated by the ORDER BY clause, has occurred).

**Export method**: A way of exporting the results of queries. Export methods can be used to create specialized file formats or launch other applications.

**Expression**: The result of an arithmetic computation being performed on a column value.

**Foreign key**: Component of tables and views; a group of one or more columns. Each foreign key has a name that must be different from all other column and foreign key names within the same table or view. Foreign keys represent the primary key of a row in a table. Foreign keys can be used as a shortcut for joining tables and retrieving other column values.

**Formats**: File formats used to import and export data.

**Function**: A clause in an SQL statement that performs calculations on the table’s data. A function combines one or more values to produce a new value. See Row functions and Set functions.

**Globals**: An array data structure that holds information stored by an M database.

**Import method**: A way of importing data into SQL tables.

**Indices**: Components of tables. Indices provide alternative structures for searching and sorting query results.
**Join:** A feature of SQL that allows you to combine tables. To perform a join you use a test in the SQL statement.

**Key word:** A word that KB_SQL reserves for its own use. You can not use key words for any purpose other than the one intended by KB_SQL.

**Modifier:** A clause which changes or modifies the format, position, or header of a column in a query result.

**Non-null value:** A known or existing value.

**Null value:** A data item that represents missing or unknown information.

**Operators:** Mathematical processes supported by KB_SQL (+, –, *, \, /, #, | |) that allow you to perform calculations on tables.

**Predicate:** A test or comparison used to restrict the query’s output.

**Primary key:** A group of one or more columns that is the table’s unique identifier.

**Pseudo column:** A name for the result of an M expression; a virtual column. Pseudo columns are not stored in tables.

**Query:** A named collection of SQL statements that retrieve existing data, in any combination, expression, or order. Queries always begin with the SQL reserved word SELECT, followed by the data desired, and the tables or views containing the source data. Queries do not change the data; they only retrieve data.

**Relational system:** All data is perceived by the user as tables. The result of every operation performed on the tables is another table.
**Row:** The horizontal set of values in a table; the records or entries in a table. Each row contains one value for each of the columns in the table.

**Row functions:** Functions that operate on a single row. See Function and Set functions.

**Schema:** A data dictionary object used to organize and group tables and views. SQL defines schema as being synonymous with user—a schema is the user’s environment or view of the world. KB_SQL extends this definition to mean any logical grouping of tables that is owned by a specific user group. A user can have access to any table in any schema as defined by the privileges that the DBA grants.

**Set functions:** A feature of SQL that lets you perform calculations on the data in the table. A set function returns one value for a set of rows. Set functions include: counting all rows, obtaining the average of all values, the maximum value, the minimum value, and the sum of all values.

**SQL statement:** Commands that perform an operation on the database. The three types of SQL statements are: Data Manipulation Language (DML) statements, Data Definition Language (DDL) statements, and Data Control Language (DCL) statements.

**Subquery:** An embedded query; using a subquery enables you to perform more than one query in one step.

**Table alias:** A name used in place of the table’s real name. Table aliases are usually used to reduce the amount of typing needed to enter a query.

**Tables:** The data structure in which all information is stored. Tables are two-dimensional structures composed of rows and columns. Each table within a schema must have a unique name.
User groups: A set of users. A user group may be composed of one or more actual users. Each user in a user group has access to the same tables, views, and queries. Users may be added or deleted from a user group at any time.

Value: Data items in a row; the smallest unit of data in the relational database. Each row contains only one value for each column. See Null value.

Views: Logical tables. Views are created by naming the result of a query. Views may be used for security purposes to restrict which rows and/or columns may be accessed. Views may also be used for convenience, to provide users with a different perspective than the one provided by the actual tables. A view appears to the user as a single table, but it is actually composed of one or more tables joined together.
You can produce reports that list your queries and additional related information, by running either of the following online queries: SQL_QUERIES and SQL_QUERY_DIRECTORY.

Both queries must be modified before you run them. Remove the comment hyphens preceding the two READ statements, and insert hyphens prior to the DECLARE statement. See the example below.

```sql
read :XFROM heading 'From query name' character(30) default 'N'
read :XTHRU heading 'Thru query name' character(30) default 'ZZZ'
   --declare :XFROM char(30), :XTHRU char(30)

select text column 3 heading ''
from sql_query, sql_query_line
where sql_query.sql_query_id=sql_query_line.query
   and name between :XFROM and :XTHRU
   order by name, SQL_QUERY_LINE.sequence
break at name
   write name page 3 skip 2,description skip
   header
```

After you select the <Run> menu option to run the query, you will be prompted to supply a range of query names. These prompts are case-sensitive. You may use wild cards to specify the range.

```
From query name: DOA____________________
Thru query name: ZZZ____________________
```
Following is sample output from these two queries.

**SQL QUERIES**

---

**DEMO_ALTER_VIEW**
ALTER view definition

ALTER VIEW incomplete AS

SELECT project, budget, task
FROM projects, tasks
WHERE projects.proj_no = tasks.proj_no
AND status = 'INCOMPLETE'

**DEMO_COLUMN_MODIFIERS**
Column modifiers:

SELECT manager CHANGED DEFAULT 'NONE',
name HEADER 'EMPLOYEE' RIGHT 10,
salary COLUMN 30,
page 1, more below, press ENTER to continue, any other key to stop>

---

**SQL_QUERY_DIRECTORY**

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Routine</th>
<th>Last Edit</th>
<th>By user</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_CALL_1</td>
<td>API Call listing</td>
<td>SQLX37</td>
<td>07/31/96</td>
<td>09:07:32</td>
</tr>
<tr>
<td>SQL_API_CALL_2</td>
<td>API Call listing</td>
<td>SQLX28</td>
<td>07/31/96</td>
<td>09:07:52</td>
</tr>
<tr>
<td>SQL_API_CALL_ZZZ</td>
<td>API Connection listing</td>
<td>SQLX12</td>
<td>06/23/95</td>
<td>09:29:36</td>
</tr>
<tr>
<td>SQL_API_CONNECT_OPTIONS</td>
<td>Server API Connection Options</td>
<td>SQLX2</td>
<td>02/13/96</td>
<td>10:16:44</td>
</tr>
<tr>
<td>SQL_API_DATAINFO</td>
<td>API DataInfo INFO</td>
<td>SQLX16</td>
<td>05/31/95</td>
<td>04:11:15</td>
</tr>
<tr>
<td>SQL_API_INFOOPTION</td>
<td>Information options - SQLGetInfo</td>
<td>SQLX46</td>
<td>07/08/95</td>
<td>12:23:39</td>
</tr>
<tr>
<td>SQL_API_LIST</td>
<td>SQL Server API list</td>
<td>SQLX79</td>
<td>05/30/96</td>
<td>10:50:35</td>
</tr>
<tr>
<td>SQL_API_SERVER_CONNECTIONS</td>
<td>SQL Server API list</td>
<td>SQLX96</td>
<td>04/12/96</td>
<td>0:17:58</td>
</tr>
</tbody>
</table>

---

E-2 KB_SQL SQL Reference Guide
Index

A
Add Query window, 8
Adding a query, 6, 8
ALIAS function, 28
table alias, 50, 51
ALTER statement, 41
AND operator, 70
with OR operator, 73
Arrow keys, 96

B
Background mode, 14
Base table
See table, 47
BETWEEN test
optimization of, 63
BREAK AFTER event, 36
BREAK AT event, 36, 37
Build step, 108

C
Cartesian product, 107
discussion of, 84
element of, 85
CHANGED modifier, 33
Clauses
in SQL statements, 20
Clear option, 98
Column header
supplying a value for, 28
Column modifiers
definition of, 33
element of, 33
use of, 28
Command qualifications, 40
Comment lines, 13
Committing successful operations, 41
Comparison tests, 24
Compiled: date/time prompt, 102
Conditional expression, 58
Constrain OJ results
example of, 78
Constrain OJ results entry, 64
Constrain primary key
element of, 69
Constrain primary key entry, 64
Constrain results entry, 63
CONTAINS operator, 69
Copy from Query window, 6
Correlated subquery, 88
COUNT function, 28
CREATE statement, 41
Cursor movement keys, 96

D
Data-driven join, 74
Data access plan
displaying the, 103
Data Control Language, 39, 41
Data Definition Language, 39, 41
Data Manipulation Language, 39, 41
Data transfer, 33
Database
updates to, 41
Date
for background processing, 14
DCL
See Data Control Language, 39
DDL
See Data Definition Language, 39
DEFAULT modifier, 33
DELETE statement, 41, 56
Deleting a query, 9
DEMO_ALTER_VIEW query, 45
DEMO_COLUMN_MODIFIERS query, 33, 34
DEMO_COUNT query, 28
DEMO_CREATE_TABLE query, 47
DEMO_CREATE_VIEW query, 44
DEMO_DELETE_ROWS query, 56
DEMO_DROP_TABLE query, 56
DEMO_DROP_VIEW query, 46
DEMO_EVENT_BLOCKS query, 37
DEMO_GRANT query, 54
DEMO_GROUP query, 29
DEMO_INSERT_ROW query, 49
DEMOJOIN_3_TABLES query, 27
DEMOJOIN query, 25
DEMO_MANY_FEATURES query, 57
DEMO_ONE_SUBQUERY query, 30
DEMO_PUBLIC query, 55
DEMO_REVOKE query, 54
DEMO_SELECT_ALL query, 22
DEMO_SELECT_SPECIFIC_ROW _COLMN query, 23
DEMO_SP_BUILD_VIEW_STEP1 query, 86
DEMO_SP_BUILD_VIEW_STEP2 query, 86
DEMO_SP_CARTESIAN_PRODUCT query, 85
DEMO_SP_CONSTRAIN_OJ_STEP1 query, 78
DEMO_SP_CONSTRAIN_OJ_STEP2 query, 79
DEMO_SP_CONSTRAIN_PK query, 69
DEMO_SP_CONSTRAIN_RESULT query, 67
DEMO_SP_CORRELATED_SUBQUERY query, 88
DEMO_SP_FOREIGN_KEY query, 82
DEMO_SP_MULTI_STEP query, 87
DEMO_SP_MULTIPART_OR query, 73
DEMO_SP_ONE_PART query, 72
DEMO_SP_ONE_PLAN_STEP1 query, 77
DEMO_SP_ONE_PLAN_STEP2 query, 77
DEMO_SP_OPTIMIZE_PK_CONSTRAINTS query, 70
DEMO_SP_OPTIMIZE_PK query, 68
DEMO_SP_ORDER_BY query, 66
DEMO_SP_SIMPLE_QUERY, 65
DEMO_SP_TWO_PLAN_STEP1 query, 80
DEMO_SP_TWO_PLAN_STEP2 query, 80
DEMO_SP_TWO_PLAN query, 76
DEMO_SP_TWO_STEP query, 75
DEMO_SP_TWO_TABLE_JOIN query, 71
DEMO_TABLE_ALIAS query, 51
DEMO_TWO_SUBQUERIES query, 32
DEMO_USE_NEW_TABLE query, 52
DEMO_USE_VIEW query, 44
DEMO_WRAP query, 35
Description prompt
in the Query Information window, 102
DETAIL event, 36
Device name prompt
for Print option, 103
for printing query results, 15
for User option, 112
device sub-types, 16
Distributed optimization techniques, 83
DML
See Data Manipulation Language, 39
DROP statement, 41

E
Edit option, 98
Editing a query, 10
Equivalent predicate optimization, 83
Event blocks
definition of, 36
Events
list of, 36
Exporting data, 16, 33
EZQ queries
editing in the SQL Editor, 10

F
File
sending output to a, 16
File formats
for data transfer, 16, 33
FINAL event, 36, 37
FOOTER event, 36
Foreign key
definition of, 58
optimizing a, 82
FROM clause, 21
FROM key word, 21
From table prompt
in Print User Tables window, 111
Function keys, 94
most commonly used, 95
Functions
See Set Functions, 28

G
Get result of step entry, 63
Get table entry, 62
Get view entry, 63

GRANT statement, 41, 54
GROUP BY clause, 21, 29
example of, 58, 80

H
Halt option, 100
Halt query prompt, 100
HAVING clause, 21
Header, 22
HEADER event, 36, 37
HEADING column modifier, 28

I
IN test
optimization of, 63
Info option, 101
INITIAL event, 36
INSERT statement, 41

J
Join, 71
data driven, 74
definition of, 25
ever of, 25, 71
Join condition, 25, 26

K
Key words, 20

L
Last edit: date/time prompt, 102
Last run: date/time prompt, 102
List feature, 98

M
Menu bar options
selecting them, 11
Message Options window, 42
Messages
system, 42
Modifiers
see Column Modifiers, 33
Modify Database? prompt, 48
Modify Database window, 42, 46
Modify DB Confirmation window, 48, 54, 56
Multi-part OR query, 73
Multi-step query, 87

N
Not equals operator, 69

O
On-line reference, 99
One-part OR query, 72
One-plan query, 77
Optimization techniques, 66, 74, 76, 77, 80
Optimize primary key entry, 63
Options
of the SQL Editor, 12
selecting them, 11
OR operator
example of, 72
with AND operator, 73
ORDER BY clause, 21
ever of, 58, 66
Outer join
example of, 78
Output device
selecting a, 16
Output to a file, 16
Output to file window, 16

P
Paper size
specifying in run message, 102
Paper type
specifying in run message, 102
Parentheses
using to group tests, 24
Parse step, 106
Performance issues, 84
Plan step, 59, 107
example of multi-step, 87
Predicate, 24
Primary key
example of optimization, 68, 70
Primary query
definition of, 30
Print all tables? prompt, 111
Print all tables from all schemas? prompt, 110
Print on Device window
for Print option, 103
for query results, 15
for User option, 112
Print option, 103
  sample of, 104
Print table and column
descriptions? prompt, 110
Print Table List window, 53, 110
Print User Tables window, 111
Privileges
  granting, 53
  revoking, 53
Processing messages, 108
Pseudo columns
  definition of, 50
Public privileges, 55
PUBLIC user group, 55

Q
Query
  adding a, 3, 6, 8
  clearing its text, 98
  copying a, 6
  deleting a, 3, 9
  editing a, 10, 98
  example of a two-plan, 76, 80
  example of a two-step, 74
  executing, 106
  exiting the current one, 17
  halting the execution of, 100
  messages during the running of, 108
  multi-step, 87
  parsing, 106
  printing contents of, 103
  running in the background, 14
  saving, 17, 105
  See also Query results, 98
  selecting, 3
Query #, 62
Query Information window, 7, 101
Query name prompt, 6
  in the Query Information window, 101
  in the Save As window, 109
Query Name window, 3, 4
Query results
  displaying them, 15
  printing them, 15
  scrolling through, 15
  sending to a file, 16
Query Start Parameters window, 14, 42
Query with ORDER BY clause, 66
Quit option, 17, 105
R
Rejecting failed operations, 41
Replace if exists prompt, 16
Report format, 33
REVOKE statement, 41, 54
Routine prompt, 101
ROW COMMIT prompt, 41
Rows
  deleting, 56
  inserting, 48
Run: date/time prompt, 102
Run in background? prompt, 48
Run in the Background window, 14, 42
Run message prompt, 102
Run option, 106
Runtime instructions
  providing, 102
S
Save option, 109
Save result entry, 63
Schema name prompt, 111
Scrollable queries, 15
Select, Insert, Delete Query window, 5
SELECT * syntax, 22
SELECT clause, 58
SELECT key word, 20
Select Schema window, 111
Select SQL Custom Start Dates window, 14
SELECT statement, 20, 21, 41
  to retrieve all rows and columns, 22
  to retrieve specific rows and columns, 23
Select USER OPTIONS window, 2
Selection windows
  description of, 93
SET DEVICE statement, 14
SET FILE=filename statement, 16
SET FORMAT=REPORT statement, 16
Set functions
  definition of, 28
  example of, 28
SET SCROLL = YES statement, 15
SHOW PLAN parameter, 103
Start a new page for each table? prompt, 110
Start date and time values
  entering, 14
Step #, 62
Subquery
  correlated, 88
  definition of, 30
  embedding them, 31
  example of, 30
System message
  printing, 42
  viewing, 42
T
Table
  creating, 47
  dropping, 56
  printing a list of, 53, 110
Table alias names, 51
  definition of, 50
Table list
  sample of, 113
Tests, 24
  adding NOT, 26
  equal to, 26
  greater than, 26
  optimization of, 63
Text windows
Index

description of, 92
Thru table prompt, 111
Time
  for background processing, 14
Transferring data, 33
Two-plan query, 76, 80
Two-step query, 74
Two-table join, 71

U
UPDATE statement, 41
User option, 110

V
Value windows
  description of, 92
  moving from field to field, 96
View
  altering, 45
  creating, 44
  definition of, 43
  dropping, 46
  example of building before referencing, 86
  with SHOW_PLAN feature, 77
View definition
  with GROUP BY clause, 80
Virtual table
  See View, 43

W
WHERE clause, 21
  example of, 67, 69
  use of, 24