Using dBASE IV SQL

Using dBASE IV SQL

9/88

Contents

Overview of SQL

Queries and Updates

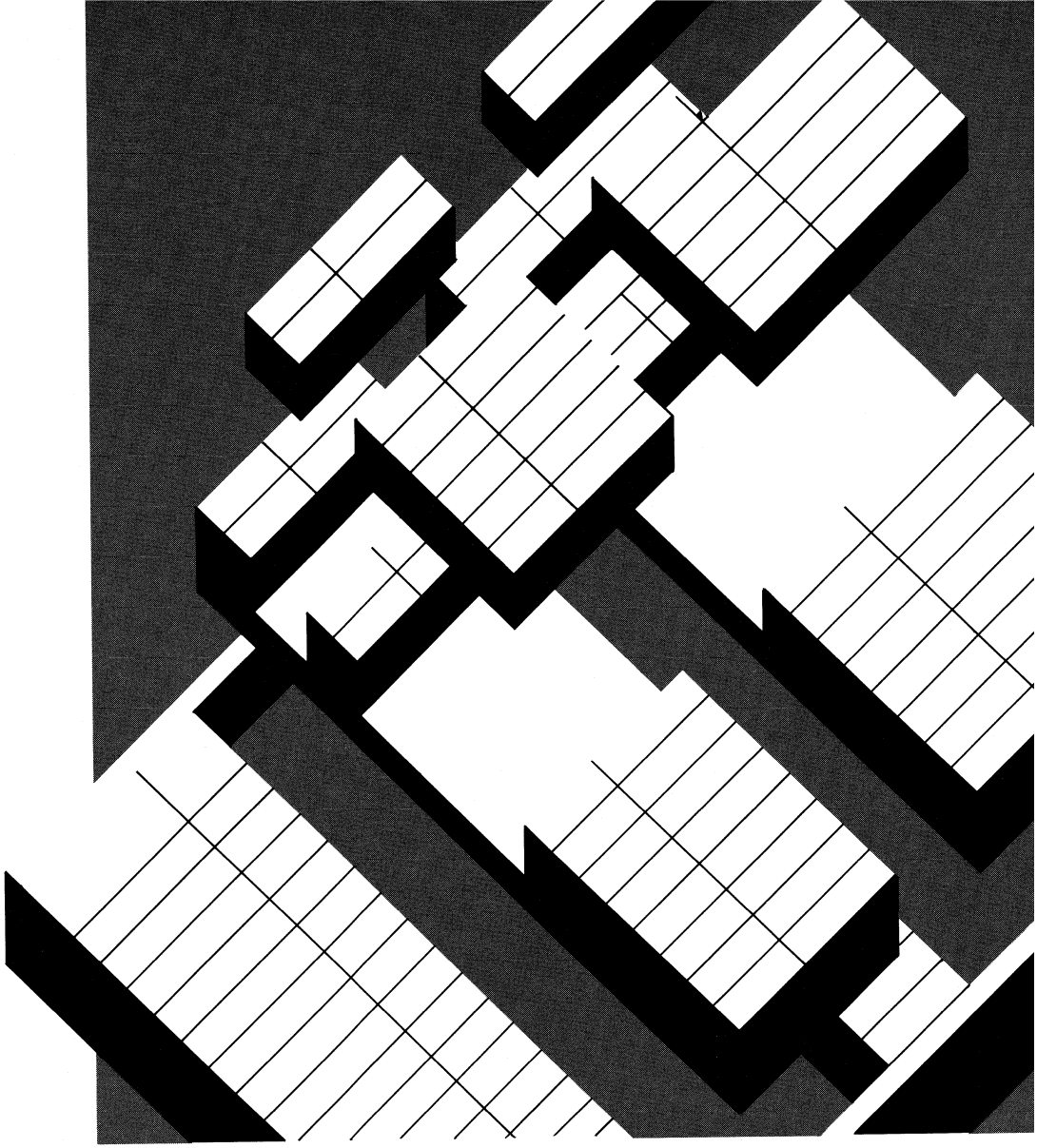
SQL and dBASE

SQL Reference

Appendices

Index

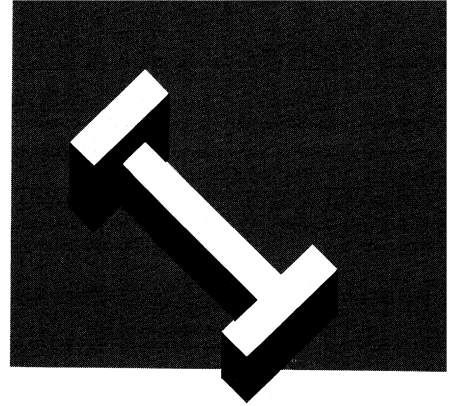
Overview of SQL



i 1 2 3 4 5 6 7 8 A

B C D In

Introduction



About This Book

This book shows you how to use the SQL (Structured Query Language) commands provided by dBASE IV™. You'll learn basic database concepts used by SQL and run examples that demonstrate SQL commands.

SQL provides a compact set of commands you can use to define, query, and update data in a database. You can execute SQL commands interactively at the SQL dot prompt, or combine them with other dBASE® commands in database application programs. The SQL commands implemented in dBASE IV are the same as those in IBM's DATABASE 2 (DB2) and SQL/DS mainframe computer database products.

How to Use This Book

How you use this book will depend on your previous experience with either dBASE or SQL. The description of each chapter given below should help you decide where to begin and how to proceed.

- Chapter 1 — Introduces SQL database concepts and describes how you can use SQL in dBASE IV.
- Chapter 2 — Shows how to start executing SQL commands, create SQL tables and views, and perform basic SQL database operations.
- Chapter 3 — Continues description of how to use SQL commands, particularly the SELECT command, to perform simple queries of data from single tables.
- Chapter 4 — Shows how to perform more complex queries, including joins and subqueries of data from more than one table at a time.
- Chapter 5 — Describes how to effectively use dBASE commands with SQL statements in the interactive SQL environment.
- Chapter 6 — Shows how to embed SQL statements in program files; demonstrates more advanced features of SQL; shows how to define and use SQL commands in program files to INSERT, UPDATE, or DELETE data.
- Chapter 7 — SQL command reference: provides syntax, description, and examples for each command.
- Chapter 8 — Describes SQL catalog tables.

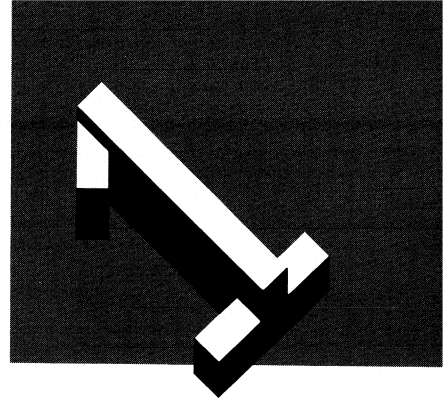
Four appendices are provided:

- Appendix A — Describes and lists SQL error messages.
- Appendix B — Provides a glossary of SQL terms.
- Appendix C — Describes dBASE commands and functions that can be used with SQL.
- Appendix D — Lists data in the SQL sample application tables: Customer, Staff, Inventry, Assembly, Sales, and Items.

Before You Begin

Before you begin, make sure you've installed dBASE IV and the associated SQL sample files. If you have not already done so, follow the instructions in *Getting Started.* If you're installing dBASE IV on a local area network, refer to instructions in *Network Installation.*

SQL Basics



This chapter provides an introduction to SQL. It describes SQL database concepts and how you can use SQL commands with dBASE IV.

A Brief History of SQL

SQL (Structured Query Language) was developed from research conducted by IBM during the mid-1970s. Since its first commercial introduction in 1979, SQL has been adopted by many companies as a database language standard for both mainframe and minicomputer environments.

The introduction of SQL to microcomputers has meant that companies can continue to standardize their development of database applications. By using SQL commands in dBASE IV, you can combine a database language standard for mainframes and minicomputers with dBASE, the standard in microcomputer database system programs.

What is SQL?

SQL (Structured Query Language) is an advanced relational database language that operates on data entirely as logical sets called *relations* (tables). You can use SQL commands within the dBASE language (which also includes more traditional record-oriented commands) to define and access the data in a dBASE IV database.

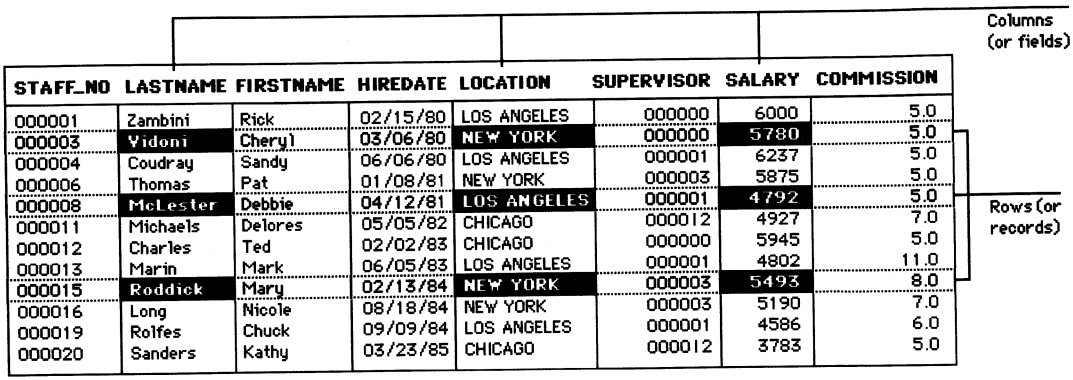
In a relational database and language, all data is defined in a single table or set of tables. Representing data in tables provides a familiar and easy-to-visualize way to access information.

SQL provides a small and concise set of commands that allows you to define, display, and update information in tables. By reducing the number of commands you need to access data, SQL saves you time and reduces the amount of programming needed to perform complex queries. SQL can also reduce the effort required to modify a database application.

SQL Tables

In SQL, each database table can be viewed as a collection of *rows* and *columns*. The intersection of each row and column position contains a data value. Figure 1-1 is an example of an SQL table.

If you're already familiar with dBASE, you'll notice that the terminology SQL uses is slightly different. In dBASE terminology, rows in a database table are called *records* and columns are called *fields*.



STAFF_NO	LASTNAME	FIRSTNAME	HIREDATE	LOCATION	SUPERVISOR	SALARY	COMMISSION
000001	Zambini	Rick	02/15/80	LOS ANGELES	000000	6000	5.0
000003	Vidoni	Cheryl	03/06/80	NEW YORK	000000	5780	5.0
000004	Coudray	Sandy	06/06/80	LOS ANGELES	000001	6237	5.0
000006	Thomas	Pat	01/08/81	NEW YORK	000003	5875	5.0
000008	McLester	Debbie	04/12/81	LOS ANGELES	000001	4792	5.0
000011	Michaels	Delores	05/05/82	CHICAGO	000012	4927	7.0
000012	Charles	Ted	02/02/83	CHICAGO	000000	5945	5.0
000013	Marin	Mark	06/05/83	LOS ANGELES	000001	4802	11.0
000015	Roddick	Mary	02/13/84	NEW YORK	000003	5493	8.0
000016	Long	Nicole	08/18/84	NEW YORK	000003	5190	7.0
000019	Rolfes	Chuck	09/09/84	LOS ANGELES	000001	4586	6.0
000020	Sanders	Kathy	03/23/85	CHICAGO	000012	3783	5.0

Figure 1-1 An SQL table

SQL Views

The SQL language allows you to create another type of table called a *view*. An SQL view is a subset of the rows and columns of one or more existing tables. A view is often referred to as a *virtual table*, since it does not actually contain data. Rather, the view reflects the data contained in one or more underlying or *base* tables on which it is built. Data displayed in a view is dynamic. That is, if data in one of the underlying base tables changes, data in corresponding views is also updated. Similarly, if you change the data in updatable views, data in underlying tables is also changed.

Set-Oriented Data Access

Using SQL commands, you specify the *data* you want from a table, rather than a *procedure* to retrieve it. The system determines the best way to obtain the data. This is different from many other database languages that require you to know how the data is stored on your computer and then also provide the steps to retrieve it.

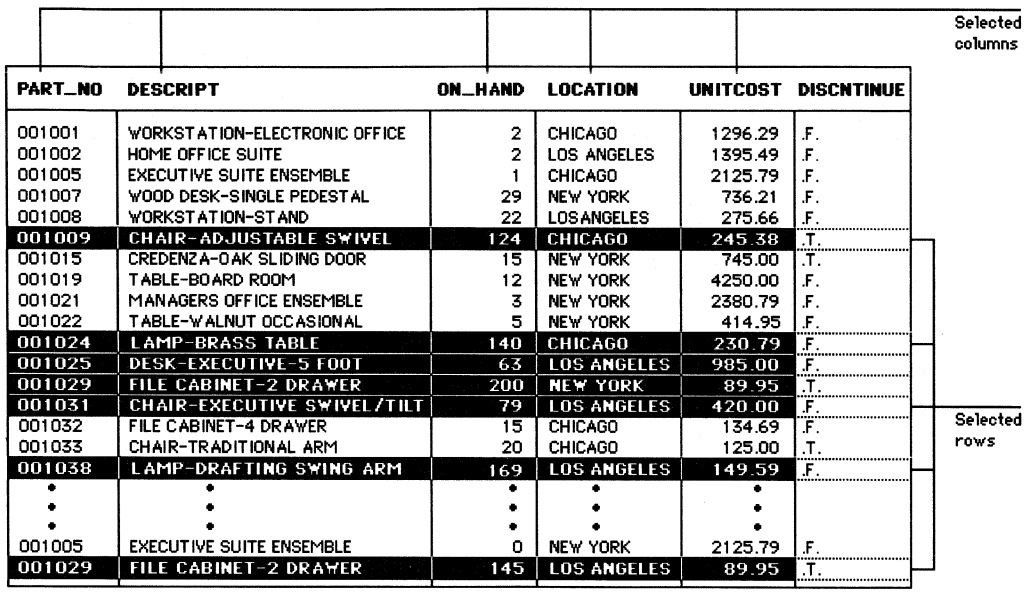
The SQL Language

The entire SQL language consists of less than 30 commands. However, it provides all the operations you need to define tables and views, and to query, update, delete, or insert data in them. SQL commands are easy to use since every SQL command refers to the same elements — the rows and columns of a table or view.

For example, you can construct the following SQL query:

```
SELECT Part_no, Descript, On_hand, Location, Unitcost
FROM Inventry
WHERE On_hand > 50;
```

Figure 1-2 shows the result of the query, also referred to as a *result table*. In this example, the SELECT command retrieves from the Inventry table all parts (and their descriptions) where the number on hand is greater than 50.



PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001008	WORKSTATION-STAND	22	LOSANGELES	275.66	.F.
001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.38	.T.
001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
001024	LAMP-BRASS TABLE	140	CHICAGO	230.79	.F.
001025	DESK-EXECUTIVE-5 FOOT	63	LOS ANGELES	985.00	.F.
001029	FILE CABINET-2 DRAWER	200	NEW YORK	89.95	.T.
001031	CHAIR-EXECUTIVE SWIVEL/TILT	79	LOS ANGELES	420.00	.F.
001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
•	•	•	•	•	
•	•	•	•	•	
•	•	•	•	•	
001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Figure 1-2 The result table from an SQL query

In this example, the SQL command contains a SELECT *clause* that identifies the columns you want to display. A FROM clause identifies the table or tables from which rows and columns are selected. Finally, the WHERE clause limits the rows of the result table to those that satisfy the condition *On_hand > 50*.

You use the same SQL commands whether you want to retrieve or update the data in a single table or in more than one table at a time. The same SQL commands can be used interactively or included in a program file. When used in a program, a single SQL statement can often replace a dozen or more instructions needed in another database language. Figure 1-3 illustrates a comparison of the SQL commands used to access data from a database versus those required using dBASE commands.

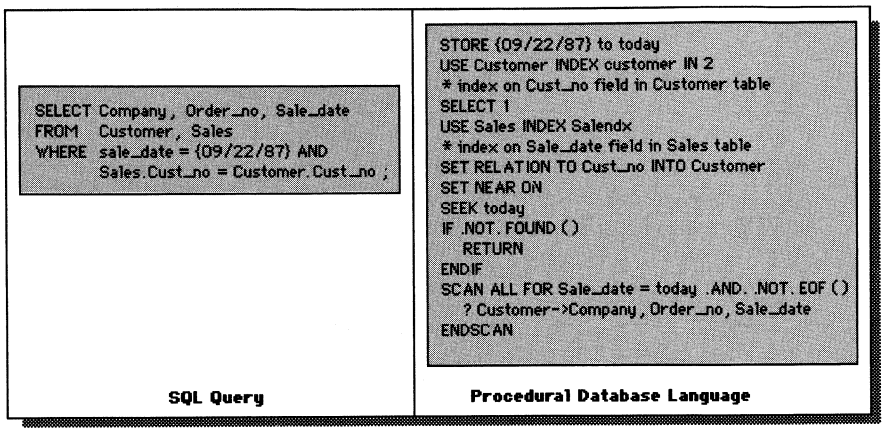


Figure 1-3 Query comparison of SQL and dBASE commands

Using SQL Commands in dBASE IV

SQL commands provide an alternative to using dBASE commands to display or access data. The set of commands you decide to use to access data depends on your familiarity and experience with either SQL or dBASE language commands.

Interactive and Embedded SQL

In dBASE IV, SQL commands can be used interactively or in program files. Using *interactive* SQL, you execute commands one at a time, and the results are displayed immediately after each command. This mode is similar to entering dBASE commands at the dot prompt.

Using *embedded* SQL, you can create dBASE program files that include SQL commands. In mainframes and minicomputer environments, SQL is typically used with languages such as COBOL, FORTRAN, or C to build a database application program. In dBASE IV, you can combine SQL with other dBASE commands to create database application programs to run on microcomputer systems.

Combining dBASE Commands with SQL

dBASE IV provides a full spectrum of commands that can make using SQL commands, either interactive or embedded, easier. For example, you can use dBASE commands to create and print reports, set up a printer, and print the results of SQL commands.

If you want to embed SQL commands in dBASE programs, dBASE IV provides commands to construct menus, design forms for report and data entry, and control the operation of a program.

Switching Between dBASE and SQL Modes

dBASE IV provides two basic modes in which to enter commands. The default mode allows you to use only traditional dBASE commands; the second mode allows you to use SQL commands but restricts some of the dBASE commands you can use. You need to specify the SQL mode to use SQL commands because of differences in the way SQL and dBASE operate, and because certain SQL and dBASE commands and functions have the same name (for example, the SELECT command). However, you can easily switch back and forth between the two modes.

To switch interactively to SQL from the dBASE mode at the dot prompt, use the SET SQL ON command. To switch back again, use SET SQL OFF. When creating dBASE program files, you specify the mode you want (dBASE or SQL) by the extension you give to each program file. Use .prg for dBASE program files, and .prs for SQL program files. When program files are executed, dBASE IV automatically switches modes depending on the extension of the current file.

In both interactive and embedded SQL, you cannot use a certain category of dBASE commands and functions: those that open a database file or require a file in use. In SQL mode, SQL commands perform those operations instead. (Appendix C indicates the dBASE commands and functions that can also be used in SQL mode.)

SQL Catalogs

SQL is able to work efficiently by maintaining and referencing information on each SQL table or view that it uses. This information is stored in files called *catalog tables*. When you create an SQL database using the SQL CREATE DATABASE command, a set of these catalog tables is automatically created for that database. (See Chapter 8 for a description of the SQL catalog tables.)

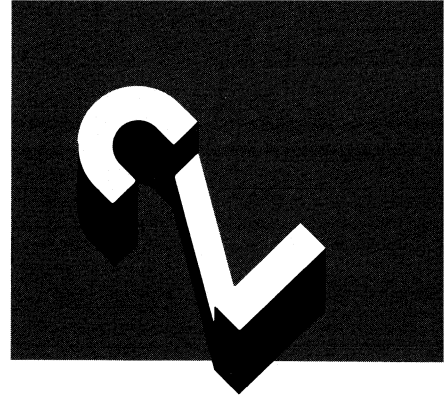
Single-User and Local Area Network (LAN) Operation

dBASE IV provides both single-user and network SQL support for both interactive and embedded SQL operations. SQL program files that you create in dBASE IV will automatically run on a local area network.

This means that when more than one user is running dBASE IV, data in tables is automatically *locked* during SQL operations that insert, update, or delete data. If an operation cannot be performed because the data is currently in use, you can instruct dBASE IV to retry the operation until it is successful.

dBASE IV also provides transaction processing with BEGIN TRANSACTION, END TRANSACTION, and ROLLBACK commands. If an operation cannot be completed, you can specify ROLLBACK to restore files to their original state.

Starting SQL



In this chapter, you'll learn how to use SQL commands interactively to start building and using SQL databases. The examples in this chapter show you how to create and use SQL databases, tables, views, synonyms, and indexes.

What This Chapter Covers

The topics covered in this chapter are:

- The Samples database
- Entering and editing commands at the SQL command line
- Creating and using SQL databases
- Creating and using SQL tables, views, synonyms, and indexes
- SQL catalog tables

Preparing for This Chapter

This chapter assumes that you've already started dBASE IV and have the dBASE IV dot prompt displayed on your screen. Some examples in this chapter use sample SQL tables. Follow the instructions in *Getting Started* to install them. If you're installing dBASE IV on a local area network, refer to instructions in *Network Installation.*

The Sample Database

When you install dBASE IV and the associated SQL sample files, a sample SQL database is created. The database has a default name of Samples and is in the directory named \DBASE\SAMPLES. It contains the sample tables used in this chapter and remaining chapters of this book. The tables resemble those you might create for an application of your own.

Table 2-1 Sample tables description

SQL Table	Description
Customer	Names and addresses of customers
Staff	Names and information on employees
Inventry	Description, quantities on hand, and pricing for inventoried items
Assembly	Simple subassembly parts listing for assemblies listed in the Inventry table
Sales	Office equipment orders
Items	Part number and quantity of items corresponding to order numbers in Sales table

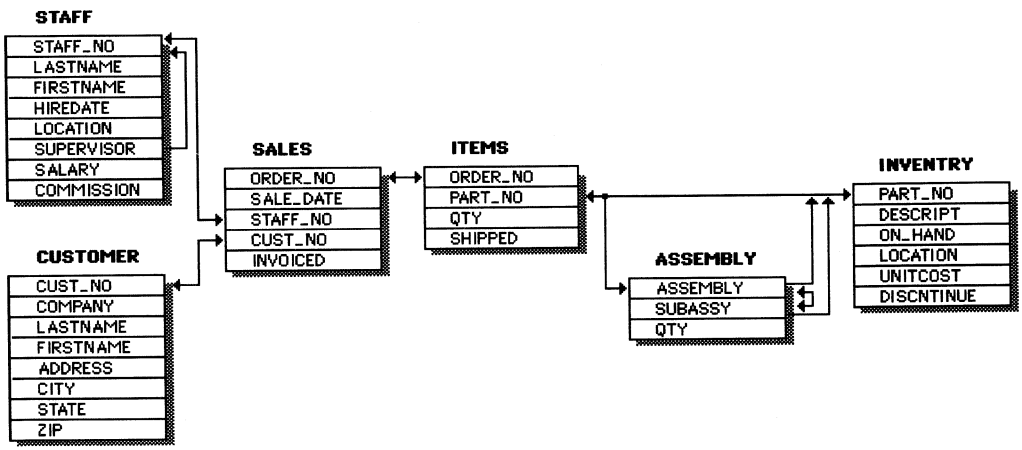


Figure 2-1 The SQL sample tables

The application for which the tables were designed is an order entry and invoicing system, as shown in Figure 2-1. Take a few minutes to become familiar with the names of the tables and the types of data they contain. (You may also want to make a copy of the table structures and the data in the tables to reference as you run samples in this book. Appendix D provides a listing of the contents of each table.)

Interactive SQL

The easiest and quickest way to start using SQL is to execute commands interactively. This will help you quickly learn how to use each SQL command. Executing SQL commands interactively provides immediate feedback: you enter an SQL command and it displays a result. You can also ask for Help, which displays the syntax and other useful information for the command you're entering.

Interactive SQL is also well suited for experienced SQL users and application developers. It is a good tool to use when you're designing, building, and testing database applications. You can try out SQL commands interactively and then include them when you build an application program. (Chapter 6 describes how you can use SQL when creating applications.)

The Interactive SQL Dot Prompt

To start executing SQL commands, enter the SET SQL ON command from the dBASE dot prompt. If you are at the Control Center, first exit from the menus and display the dBASE dot prompt.

The prompt displayed on your screen changes to **SQL.** to show that you've entered SQL mode. To exit SQL mode, enter the command SET SQL OFF.

NOTE
To start in SQL mode automatically, enter SQL = ON or COMMAND = SET SQL ON in your Config.db file.

Entering SQL Commands

You can enter SQL statements directly on the SQL dot prompt command line, or press **Ctrl-Home** to activate an editing window in which to enter SQL statements. You enter commands at the SQL dot prompt in the same way you enter commands at the dBASE dot prompt. However, at the end of an SQL statement, you type a semicolon, and then press ↵ to execute the statement or press **Ctrl-End** if you're in the editing window.

NOTE
When you enter interactive SQL commands, semicolons are only used to indicate the end of a statement that contains an SQL command. In SQL program files, however, you may also use a semicolon as a continuation character when entering a dBASE command allowed in SQL mode (see Appendix C).

Both dBASE and SQL commands may contain up to 1,024 characters. You can enter a command at the dot prompt on a single line containing up to 254 characters. In the editing window, a command may be entered on one or more lines with a combined length of up to 1,024 characters.

Command Line Editing

While entering a command, you can use the navigation and editing keys listed in Table 2-2. These keys provide the same functions as those provided when entering dBASE commands at the dot prompt.

Table 2-2 SQL navigation and editing keys

Key	Function
→	Moves cursor one character to right
←	Moves cursor one character to left
↑	Moves cursor to beginning of previous line
↓	Moves cursor to beginning of next line
Ctrl-←	Moves cursor to previous word in command
Ctrl-→	Moves cursor to next word in command
Home	Moves cursor to beginning of command line
End	Moves cursor to end of command line
Backspace	Deletes one character to left of cursor
Del	Deletes one character at cursor position
Ins	Switches modes between inserting and overwriting characters
Num Lock	Turns Numeric mode on and off (must be off for function keys on numeric keypad to operate)
Caps Lock	Switches automatic upper case mode on and off
F1 Help	Displays Help for command if pressed after entry of SQL command keyword
Ctrl-Home	Opens a full-screen edit window
Ctrl-End	Closes a full-screen edit window and returns to the SQL prompt command line
↵	Executes the command displayed on the command line.

Window Editing

When you press **Ctrl-Home**, a full-screen editing window appears. Any portion of commands you began typing at the command line will appear in the window.

In the editing window you can enter SQL commands on more than one line. You may separate portions of the same command on different lines by pressing ↵. At the end of an SQL statement, type a semicolon. After you finish entering a command, press **Ctrl-End**. The command displays again on a single line at the SQL prompt and is then automatically executed.

TIP
Use the **Tab** *or* **Spacebar** *keys to indent portions of an SQL command entered on multiple lines in the editing window. When you return to the SQL prompt, each ↵, tab, or space character will display on the command line. Later, if you return to the edit window, the command will appear as you formatted it originally in the edit window.*

SQL Command Syntax

Each command or SQL statement begins with a *keyword*, such as INSERT or SELECT, that names the basic operation performed. Many SQL commands also have one or more keyword phrases, or *clauses*, that tailor the command to meet a particular need. Each SQL command must do two things:

1. Specify the data you're interested in (a set of rows in one or more tables).
2. Indicate the action to perform with the specified data.

For example, to retrieve data, the SQL command is:

```
SELECT <columns>
    FROM <tables>
    [WHERE <condition> ]...;
```

In this example, the first clause is the SELECT clause, which identifies columns to appear in the result. SELECT is also the name of the command. The second clause is the FROM clause, which specifies the tables or views from which rows and columns are retrieved. The third clause shown is the optional WHERE clause, which limits the rows that appear in the result to those matching a specified condition.

Certain conventions used in this manual to describe command syntax are listed below:

1. Commands and keywords are shown in upper-case letters. You can enter commands using upper- and lower-case letters if you wish.
2. An ellipsis (...) indicates that there are additional optional clauses in the SELECT command (for example, the ORDER BY and GROUP BY clauses). The options of the SELECT command are described in further detail in Chapters 3 and 4.
3. The [] (square brackets) indicate that you may optionally enter the enclosed information, but without typing the brackets. The < > (angle brackets) indicate that you must enter the enclosed item.

Before using the SELECT command, you must have already started an SQL database that contains the SQL tables you want to view. For example, you can choose the Samples SQL database by typing:

```
SQL. START DATABASE Samples;
```

NOTE
You may specify a default database that is automatically started when you enter SQL mode by setting SQLDATABASE in your Config.db file, for example, SQLDATABASE = SAMPLES. The database must already exist prior to being specified with the SQLDATABASE setting or the START DATABASE command. SQL databases are created using the CREATE DATABASE command.

Then, to display data in the Inventry table, you can construct the following SQL statement:

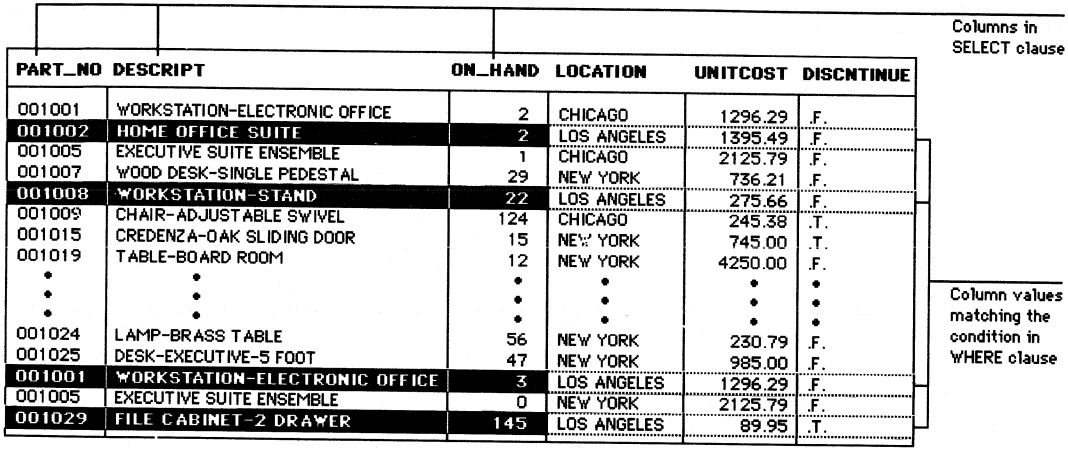
```
SQL. SELECT Part_no, Descript, On_hand
     FROM Inventry;

PART_NO   DESCRIPT                    ON_HAND
001001    WORKSTATION-ELECTRONIC OFFICE    2
001002    HOME OFFICE SUITE                2
001005    EXECUTIVE SUITE ENSEMBLE         1
    .              .                       .
    .              .                       .
    .              .                       .

001029    FILE CABINET-2 DRAWER          145
```

In this example, part numbers, descriptions, and quantities for the entire inventory are displayed. The data that appears on your screen is called a *result table.*

To limit the rows that appear, you can include a WHERE clause:

```
SQL. SELECT Part_no, Descript, On_hand
     FROM Inventry
     WHERE Location = "LOS ANGELES";

PART_NO   DESCRIPT                    ON_HAND
001002    HOME OFFICE SUITE                2
001008    WORKSTATION-STAND               22
001025    DESK-EXECUTIVE-5 FOOT           63
001031    CHAIR-EXECUTIVE SWIVEL/TILT     79
001038    LAMP-DRAFTING SWING ARM        169
001007    WOOD DESK-SINGLE PEDESTAL       62
001013    CHAIR-MODERN PNEUMATIC          35
001032    FILE CABINET-4 DRAWER           71
001001    WORKSTATION-ELECTRONIC OFFICE    3
001029    FILE CABINET-2 DRAWER          145
```

In this example, the SELECT command displays all inventoried parts in Los Angeles from the SQL Inventry table. Figure 2-2 shows the rows and columns selected using this command.



PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.38	.T.
001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
001024	LAMP-BRASS TABLE	56	NEW YORK	230.79	.F.
001025	DESK-EXECUTIVE-5 FOOT	47	NEW YORK	985.00	.F.
001001	WORKSTATION-ELECTRONIC OFFICE	3	LOS ANGELES	1296.29	.F.
001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Figure 2-2 The SELECT query

Command Keyword and Parameter Description

The entry of certain SQL command keywords and parameters must follow specific guidelines. These are described in the following table.

Table 2-3 Keyword and parameter entries in commands

Item	Description
table name	Follows DOS file naming rules: up to eight characters long, the first character must be a letter.
view name	Same as table names.
synonym	Same as table names.
alias name	Same as table names.
index name	Same as table names.
column name	Name of a column in a table or view. Up to ten characters (letters, numbers, and underscore characters). First character must be a letter.
data type	Data type of the column.
expression	A character string (up to 254 characters), a number, or calculated column.
condition	Expression that evaluates to a true or false logical value.

Re-entering SQL Commands

A memory buffer called *history* allows you to recall previously entered commands and edit or re-execute them. Use the SET HISTORY TO command to change the number of commands stored in the history buffer. (The default is 20.) Use the up and down arrow keys to display commands stored in the history buffer at the SQL dot prompt. The history buffer is also controlled by the set of dBASE commands listed in Table 2-4.

When a command is displayed at the SQL dot prompt you can press ↵ to execute it, or you can edit it first. Pressing **Ctrl-Home** displays the command in the full-screen editing window. If a command was formatted on more than one line, it will appear in the window as it was originally entered.

Table 2-4 History buffer commands

Command	Function
DISPLAY HISTORY	Displays the commands stored in the history buffer and pauses at each full screen of commands
LIST HISTORY	Lists the commands stored in the history buffer without pausing
SET HISTORY ON/OFF	Enables or disables capture of commands in the history buffer
SET HISTORY TO	Specifies the number of commands that the history buffer can capture

NOTE
Refer to Language Reference *for more information on these commands.*

Getting HELP

While in SQL mode, you can obtain help on using SQL commands and the dBASE commands allowed in SQL mode. To display a list of the SQL Help topics, you can type:

```
SQL. HELP ↵
```

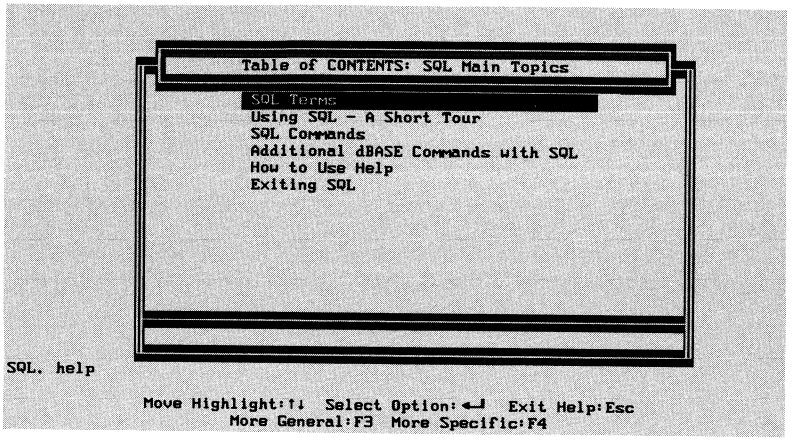


Figure 2-3 SQL Help screen

A Help box appears that allows you to select an SQL topic. To select an item, highlight it and press ↵. You can also select an item by simply typing its first letter and then pressing ↵. Pressing **Esc** returns the SQL dot prompt display.

When a Help display appears, you can press **F4** to display additional screens, or you can select from the choices appearing along the bottom of the Help box, labelled **CONTENTS**, **RELATED TOPICS**, and **PRINT**. To back up, you can press the **F3** key or choose **BACKUP** when it appears along the bottom of the help screen.

If you want help on a specific SQL or dBASE command or keyword, you can type HELP followed by the name of the command or keyword. For example, if you type **help select**, a Help box describing the syntax of the SELECT command appears. If you press **Esc**, you again return to the dot prompt. The command syntax display remains on the screen to assist you in entering the command. To clear the display, you can press ↵.

If you've begun entering a command on the command line but haven't pressed ↵ yet, you can also get help in completing the command by pressing **F1 Help**. For example, if you type **select** and then press **F1 Help**, a Help box appears that describes the specific syntax of the SELECT command.

dBASE IV provides help in one other situation. If you enter a command incorrectly and try to execute it, dBASE IV will first display an error box, as shown in Figure 2-4.

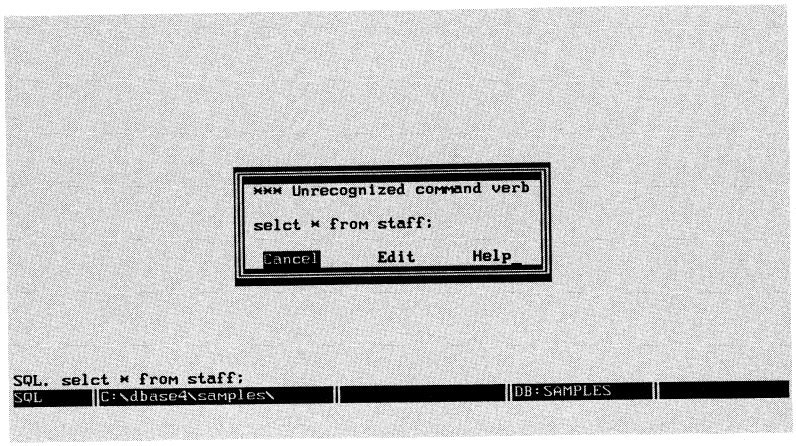


Figure 2-4 SQL error box

The error box provides three options. The first two choices, **Cancel** and **Edit**, allow you to either clear the command line or to edit the command that caused the error. The third option, **Help**, displays information about the specific error or the command that caused it, or displays a list of topics from which you can select.

SQL Databases

dBASE IV SQL uses *databases* to store all information related to the same application. An SQL database contains the following SQL objects:

- Tables — The basic structure for data in a database (sometimes called *base tables*).
- Views — A type of *virtual* table providing a *view* of data that consists of selected rows and columns from one or more base tables. The view is updated as data in the underlying tables changes.
- Synonyms — Alternate names for tables and views.
- Indexes — Adjuncts to tables that speed up data access and queries and help maintain database integrity.
- Catalogs — A set of tables in each database that describes the database and its contents. (These are different from dBASE .cat files.)

Before you begin creating or using SQL tables, you must create a database or activate (start) an existing database. Each database is assigned a specific directory in which all its tables and related files are stored. A set of catalog tables is also stored in each database directory. These tables keep track of the definitions of tables, views, indexes, and synonyms. They also maintain authorization and statistics for operations performed on tables and other files in the same database.

A master catalog table, Sysdbs, keeps track of all databases that you or other users create. Sysdbs records the name of each database, its DOS path location, the name (user ID or log-in name) of the person who created the database, and the date the database was created. The Sysdbs table is located in the SQL *home* directory (the directory in which SQL system files are installed).

dBASE IV SQL provides five commands related to SQL databases. These are CREATE DATABASE, SHOW DATABASE, START DATABASE, STOP DATABASE, and DROP DATABASE.

Creating a Database

The CREATE DATABASE command creates a database. It can only be run in interactive SQL mode. The syntax of this command is:

CREATE DATABASE [< path >] < database name > ;

To create a database named Myapp, you could enter the following command:

```
SQL. CREATE DATABASE Myapp;
```

The system displays the message **Database MYAPP created.**

A subdirectory of your current directory, called Myapp, is created. A set of catalog tables is created in the Myapp database directory to keep track of any SQL tables created while this database is open. Also, an entry is made in the Sysdbs master catalog table to include the name of the database, your user name (if PROTECT is installed), the current date, and the DOS directory path to the new database.

When you begin to create SQL tables of your own, you can use this database or create other databases. When you create a database, it is activated automatically. Any previously active database is closed.

NOTE

1. *Each database name must be unique. The number of databases you can create is limited only by the number of directories allowed by DOS.*

2. *You can specify an explicit DOS path (up to 64 characters) preceding the database name. Do not insert a space between the path and the name of the database.*

3. *If a specified directory already exists, no new directory is created. Instead, that directory will be defined as an SQL database and a complete set of catalog tables copied into it.*

Listing Current Databases

The SHOW DATABASE command lists the names of all current SQL databases.

```
SQL. SHOW DATABASE;

Existing databases are:
NAME      CREATOR    CREATED    PATH
SAMPLES              09/30/88   C:\DBASE\SAMPLES
MYAPP     RICK       09/30/88   C:\DBASE\MYAPP
```

The *Creator* column only contains entries for SQL databases created after dBASE IV password protection is installed using PROTECT.

Activating a Database

The START DATABASE command activates an existing database. The syntax for this command is:

START DATABASE [< database name >];

After you activate a database, all SQL commands you issue refer to tables and files in that database. You must have activated a database before issuing any SQL statement that defines or accesses data, such as CREATE TABLE, INSERT, or SELECT.

You may only have one active database at a time. If you activate a database with START DATABASE, any previously active database is closed. You also may explicitly close a database using the STOP DATABASE command. You must STOP a database before DROPping it.

NOTE
Developers may omit the database name when creating a single-database application. See "Developing SQL Applications" in Chapter 6.

Dropping (Deleting) Databases

The DROP DATABASE command deletes a database. The syntax of this command is:

DROP DATABASE < database name > ;

A warning prompt appears when you issue the command, allowing you to cancel the operation by pressing **Esc**.

This command deletes all SQL objects (and their associated .dbf and .mdx files), deletes the catalog table files from the database directory, and deletes the entry for the database in the Sysdbs master catalog table. The database directory is not deleted.

The DROP DATABASE command cannot drop an active database. Before issuing this command, activate another database or deactivate the database with the STOP DATABASE command.

WARNING
Be careful when you use the DROP DATABASE command, as this command will delete the database and its contents. You will not be able to recover the SQL tables or the other files deleted in the database.

SQL Tables

Tables are the basic component of an SQL database. Before you can manipulate data in a database, you need to define how the data is placed there.

dBASE IV SQL provides different ways to define tables. You use the CREATE TABLE command to define the structure of the table, and the ALTER TABLE command to add columns to an existing table. The INSERT, DELETE, and UPDATE commands allow you to add, change, or remove data in tables.

dBASE IV also provides utility commands to add data from or send data to non-SQL files. The DBDEFINE command allows you to use database files created with dBASE commands in SQL. The LOAD DATA command allows you to import dBASE II, dBASE III, and dBASE III PLUS files, data from Lotus 1-2-3, MultiPlan and Visicalc spreadsheets, ASCII and delimited file formats, Framework II™, and RapidFile™. The UNLOAD DATA command exports SQL tables to those same file types.

Creating Tables

To define a new table in SQL, you use the CREATE TABLE command. The syntax of this command is:

```
CREATE TABLE <table name>
    (<column name>  <datatype>
    [,<column name>  <datatype> ...]);
```

Using this command, you specify the name of the table you want to create and the individual columns within the table. Table names must be unique within the same database.

You assign to each column a *data type* defining the kind or type of data a column may hold. For some data types (for example, the character data type), you must also specify a *column width*. The column width is the maximum number of characters or digits allowed within a column.

The data types you can specify when you create an SQL table are listed in Table 2-5.

Table 2-5 SQL table data types

Data Type	Description
SMALLINT	Holds an integer with up to six digits (including sign). Values entered may range from −99,999 to 999,999.
INTEGER	Holds an integer containing up to 11 digits (including sign). Values entered may range from −9,999,999,999 to 99,999,999,999.
DECIMAL(x,y)	Holds a signed fixed decimal point number with *x* total digits (including sign) and *y* decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 19 and *y* may range from 0 to 18. For example, DECIMAL(6,2) allows entry of an unsigned value up to 9999.99 or signed values ranging between −999.99 and 9999.99.
NUMERIC(x,y)	Holds a signed fixed decimal point number with *x* total digits (including sign and decimal point) and *y* decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 20 and *y* may range from 0 to 18. For example, NUMERIC(6,2) allows entry of an unsigned value up to 999.99 or signed values ranging between −99.99 and 999.99. Precision for fixed decimal point numbers is specified with the SET PRECISION command.
FLOAT(x,y)	Holds a signed floating point number with *x* total digits (including sign and decimal point) and *y* decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 20 and *y* may range from 0 to 18. The range of numbers you may store is 0.1×10^{-307} to $0.9 \times 10^{+308}$. A number may be specified using scientific (exponential) notation, for example, −9.99E+235.
CHAR(n)	Holds a character string of up to *n* characters. *n* may range from 1 to 254. Values may be entered from character columns, character type memory variables, or a character string.
DATE	Holds a date in the format specified by the SET DATE and SET CENTURY commands. The default format is *mm/dd/yy*. Values are entered from date columns, date type memory variables, or date strings specified as { / / }, or converted with the dBASE CTOD() function, for example, *CTOD("02/15/86")*.
LOGICAL	Holds logical true or false values. *.T.* represents a true value and *.F.* represents a false value. Values are entered from dBASE logical memory variables or columns, or by the constants .T., .t., .Y., .y., .F., .f., .N., and .n..

Figure 2-5 provides an example of a table containing a list of employees, perhaps similar to one you might already have on paper or in another database file. This table is the sample table Staff in the Samples database.

STAFF_NO	LASTNAME	FIRSTNAME	HIREDATE	LOCATION	SUPERVISOR	SALARY	COMMISSION
000001	Zambini	Rick	02/15/80	LOS ANGELES	000000	6000	5.0
000003	Vidoni	Cheryl	03/06/80	NEW YORK	000000	5780	5.0
000004	Coudray	Sandy	06/06/80	LOS ANGELES	000001	6237	5.0
000006	Thomas	Pat	01/08/81	NEW YORK	000003	5875	5.0
000008	McLester	Debbie	04/12/81	LOS ANGELES	000001	4792	5.0
000011	Michaels	Delores	05/05/82	CHICAGO	000012	4927	7.0
000012	Charles	Ted	02/02/83	CHICAGO	000000	5945	5.0
000013	Marin	Mark	06/05/83	LOS ANGELES	000001	4802	11.0
000015	Roddick	Mary	02/13/84	NEW YORK	000003	5493	8.0
000016	Long	Nicole	08/18/84	NEW YORK	000003	5190	7.0
000019	Rolfes	Chuck	09/09/84	LOS ANGELES	000001	4586	6.0
000020	Sanders	Kathy	03/23/85	CHICAGO	000012	3783	5.0

Figure 2-5 A sample database table

To create the structure for this table, enter the following command. (If you have not already done so, first type **start database Samples** to activate the Samples database.)

```
SQL. CREATE TABLE Staff1
     (Staff_no      CHAR(6),
     Lastname       CHAR(15),
     Firstname      CHAR(10),
     Hiredate       DATE,
     Location       CHAR(15),
     Supervisor     CHAR(6),
     Salary         NUMERIC(6,0),
     Commission     NUMERIC(4,1));
```

Notice in the example that the command is on multiple lines. SQL statements can be entered on multiple lines (using **Ctrl-Home** to edit commands in a full-screen window). You enter a semicolon to indicate the end of the statement.

The message **Table STAFF1 created** appears after you run the command and the table has been created. The new table contains eight columns: Staff_no, Lastname, Firstname, Hiredate, Location, Supervisor, Salary, and Commission.

When you create an SQL table, dBASE IV constructs a corresponding dBASE .dbf file. Database table size limits are the same as with other .dbf files created in dBASE IV: 255 columns of up to 4,000 bytes.

Besides creating a dBASE .dbf file when you define a table, dBASE IV also updates the database's catalog tables for the new table definition. A description of the table is entered in the Systabls table and descriptions of columns are entered in the Syscols table.

Inserting Data

SQL provides the commands INSERT and LOAD DATA to add new rows to tables. Using the INSERT command, you can specify particular values to insert into the columns of a table, or insert data from specified rows qualified by a SELECT statement.

The syntax of the command to INSERT rows into a table from a list of values is:

```
INSERT INTO <table name>
    [(<column list>)]
    VALUES (<value list>);
```

For example, to add the first row of data (shown in Figure 2-5) to the Staff1 table, you would type:

```
SQL. INSERT INTO Staff1
     VALUES ("000001", "Zambini", "Rick", {02/15/80},
     "LOS ANGELES","000000", 6000, 5.0);
```

The message **1 row(s) inserted** appears.

Notice that you did not have to enter a column list in the example. You can omit the column list if the order of values you've entered is the same as the columns of the table in which the values are to be inserted.

You do not need to specify values to be inserted for every column in the table. You can specify a columns list to indicate particular columns to update. Or, you can specify a partial list of values in the same order as columns in the table for those columns you want to update. The data type of each value must also match that of the column into which the data is inserted.

To insert data from selected rows of an existing table, the syntax of the command is:

```
INSERT INTO <table name>
    [(<column list>)]
    <SELECT statement>;
```

You might use this second method of inserting data if you have an existing file to which you regularly append data from an update or transaction table. To insert rows, you could type:

```
SQL. INSERT INTO Staff1
     SELECT *
     FROM Staff
     WHERE Lastname <> "Zambini";
```

11 row(s) inserted

This INSERT statement adds all rows from the Staff table into the new Staff1 table you created, except those where the Lastname column contains the name *Zambini*. The operator < > specifies a "does not equal" condition. The asterisk (*) is a special symbol specifying that the SELECT clause include all columns in the Staff table.

If you now want to see all the rows in the table, you can use the SELECT command:

```
SQL. SELECT *
     FROM Staff1;
```

All the rows and columns in Staff1 will display. The list of rows and columns returned by the SELECT statement is known as a *result table*. It is not a permanent table like one you create with the CREATE TABLE command. You can, however, save the result of a SELECT statement in a dBASE database file using the SELECT command's SAVE TO clause, as you'll learn in Chapter 5.

Updating Data

The SQL UPDATE command allows you to change the values in one or more selected rows of a table. The syntax of the UPDATE command is:

```
UPDATE <table name>
   SET <column name> = <expression>
   [,<column name> = <expression> ...]
   [<WHERE clause>];
```

For example, perhaps you want to increase the commission of all employees hired before a certain date. To update the commission column in the Staff1 table for qualified rows (hiredate before July 5, 1982), type:

```
SQL. UPDATE Staff1
     SET Commission = (Commission * 1.25)
     WHERE Hiredate < CTOD("07/05/82");
```

The message **6 row(s) updated** appears.

In this example, the WHERE clause specifies a simple condition. You can use the WHERE clause to qualify rows using a simple condition as well as more complex operations such as SELECT. Chapters 3 and 4 provide examples for more complex SELECT statements and more complex uses of the UPDATE command.

Deleting Data

The SQL DELETE command allows you to delete selected rows. The syntax of this command is:

```
DELETE FROM <table name>
   [WHERE <clause>];
```

For example, to delete a row for an employee who has left your company, you would type:

```
SQL. DELETE FROM Staff1
     WHERE Lastname = "Long";
```

The message **1 row(s) deleted** appears.

In this example, the row in which the Lastname column in the Staff1 table contained the string "Long" was deleted.

WARNING
Be careful when you use this command. If you forget to specify the WHERE clause, all rows will be deleted from your table. A prompt appears when you issue DELETE without a WHERE clause. Pressing **Esc** will cancel execution of the command.

Modifying Tables

As your database grows, you may have to change the structure of your SQL tables. dBASE IV SQL allows you to add columns to an existing table, but does not allow you to delete or modify them. You can, however, effectively modify a table by creating a new table and inserting rows from the existing table into the new table.

Adding New Columns

The ALTER TABLE command allows you to add one or more new columns to an existing table. The syntax of this command is:

```
ALTER TABLE <table name>
    ADD (<column name> <datatype>
    [,<column name> <datatype> ...]);
```

For example, to add a new 13-character column to the Staff1 table to contain the phone numbers of employees, you would type:

```
SQL. ALTER TABLE Staff1
     ADD (Phone CHAR(13));
```

The message **Column PHONE added to table STAFF1** appears after the column has been added. The column is added to the table after the last existing column.

Restructuring Tables

You can effectively modify an existing table by creating a new table and inserting rows from the existing table into the new one. First you create a new table with the columns you want. Then you insert the rows from your existing table into the new table.

For example, to modify the Staff1 table so that the Lastname and Firstname columns are combined into a new column called Fullname, first create a new table named Staff2 with the information below. Note that the new Fullname column has the combined width of the Lastname and Firstname columns and is the same data type.

```
SQL. CREATE TABLE Staff2
     (Staff_no      CHAR(6),
     Fullname       CHAR(25),
     Hiredate       DATE,
     Salary         DECIMAL(6,0),
     Commission     DECIMAL(4,1));
```

The message **table STAFF2 created** appears.

Next, you insert data from the old Staff1 table into the new Staff2 table:

```
SQL. INSERT INTO Staff2
     SELECT Staff_no, (Firstname + Lastname),
     Hiredate, Salary, Commission
     FROM Staff1;
```

The message **11 Row(s) inserted** appears.

The INSERT INTO Staff2 SELECT statement inserts rows of the existing Staff1 table into the new Staff2 table.

In this example, you do not have to specify column names following the INSERT statement since the order of columns in the SELECT clause matches that of the new table Staff2. Data from the expression (Firstname + Lastname) is transferred to the new Fullname column because of the relative position of the expression to the Fullname column in the Staff2 table.

NOTE
Any columns in the new table for which no data is INSERTed is initialized with a null (empty) character string or zero (depending on the data type of each column). Date columns are initialized with a blank date string ("mm/dd/yy") and logical columns with the value .F.

Dropping (Deleting) Tables

To delete a table that is no longer needed, use the DROP TABLE command. If you drop a table, the data in it is lost and cannot be restored. All indexes, views, and synonyms defined with that table are also dropped.

For example, if you decided you no longer needed the Staff2 table, you would drop it by typing **drop table staff2;**

The message **Table STAFF2 dropped** appears.

SQL Views

dBASE IV SQL allows you to define a special kind of table called a *view*. Views (or *virtual tables* as they are sometimes called) combine rows and columns selected from one or more tables. You can use SQL commands to select and display information from a view just as you do with a table. You can also insert and update information in the underlying base tables of a view when the view is only based on columns from a single table.

One difference between a table and a view is that the view does not contain data. Rather, the view displays the data present in the tables when the view is used. Thus, as information in the tables changes, so does the data that appears in the view.

The simplest type of view selects rows and columns from a single table. You can build a view for a single table to reduce the number of columns that display, or use a view to restrict user access to information in certain columns. Or, you might create a view that displays only the rows that match a selection criterion.

Views based on more than one table allow you to qualify rows using conditions comparing column values in any of the underlying tables. Examples of single and multiple table views are shown in Figure 2-6.

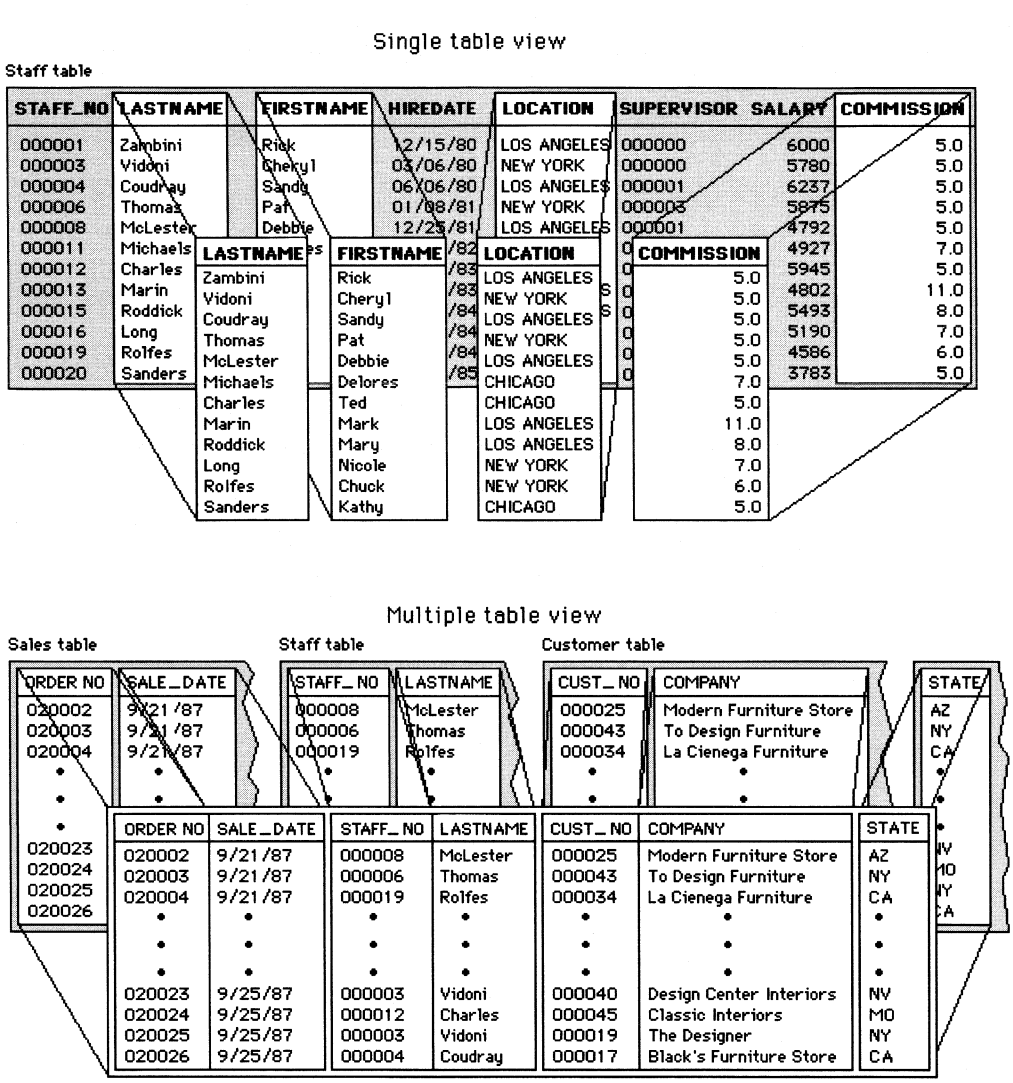


Figure 2-6 SQL views

Creating a View

You create views with the CREATE VIEW command. The syntax of this command is:

```
CREATE VIEW <view name> [(<view column list>)]
    AS <SELECT statement>
    [WITH CHECK OPTION];
```

The CREATE VIEW command has three parts. The first part names the view and optionally specifies column names in the view table. If you don't specify column names, the view will have the column names listed in the view's SELECT statement. However, you must supply a column name list if any of the view columns are derived from a dBASE function or other type of expression.

The second part of CREATE VIEW contains the SELECT statement that defines rows and columns from the underlying table(s) to be included in the view.

The third part of CREATE VIEW, the WITH CHECK OPTION, is optional. It is used with views that can be updated and specifies that inserted rows or updated rows be checked against the view definition. If you specify the WITH CHECK OPTION, you ensure that no rows are inserted in the view that do not match the condition of a WHERE clause (in the SELECT statement) that qualifies rows displayed from the view.

For example, to build a view of the Staff1 table that only includes rows for employees in Los Angeles and the three columns Staff_no, Lastname, and Hiredate, you would type:

```
SQL. CREATE VIEW La
     AS SELECT Staff_no, Lastname, Hiredate
     FROM Staff1
     WHERE Location="LOS ANGELES";
```

The message **View LA created** appears.

Once you create a view, you can use the SELECT statement to retrieve rows from the view. For example:

```
SQL. SELECT *
     FROM LA;

STAFF_NO   LASTNAME      HIREDATE
000001     Zambini       02/15/80
000004     Coudray       06/06/80
000008     McLester      04/12/81
000013     Marin         06/05/83
000019     Rolfes        09/09/84
```

The * selects all columns from the view to appear in the result. Also, because no WHERE clause is present, all rows specified in the view definition appear.

Using Views to Restructure a Database

You can use views to restructure the information in a database without creating new tables to modify column definitions, or having to create any new base tables.

For example, to create a view that combines the Lastname and Firstname columns of the Customer table in a format more acceptable for reports, you would type:

```
SQL. CREATE VIEW Address
     (Fullname, City, State)
     AS SELECT Firstname+Lastname, City, State
     FROM Customer;
```

In this example, the Firstname and Lastname columns are combined and displayed as the Fullname column in the view.

In a previous example, CREATE and INSERT were used for a similar restructuring of data. CREATE TABLE was used to create a new table and INSERT was used to insert rows into the new table. Using views to restructure data, however, has some advantages. Views do not duplicate the data in the existing table. Also, you can develop several different views on the same table. Finally, when you use a view, it will reflect the most up-to-date information in the tables on which the view is based.

Constructing a View for More than One Table

You can create a view that combines rows and columns from more than one table. For example, to create a view that combines data from columns in two different tables, the Staff table and the Sales table, you would type:

```
SQL. CREATE VIEW Orders
     (Order_no, Sale_date, Seller)
     AS SELECT Sales.Order_no, Sales.Sale_date, Staff.Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no;
```

The message **View ORDERS created** appears.

NOTE
In the example above, column names are preceded by the table name from which information is drawn. You need to specify the table name whenever it's not clear where the column information should come from (for example, when two tables both have a column with the same name).

The view you just created combines columns from both tables, Staff and Sales. The result lists the salesperson's last name (from the Staff table) with the order number and date of each sale from the Sales table.

To see the results of the view you built, you can type:

```
SQL. SELECT *
     FROM Orders;

ORDER_NO  SALE_DATE   SELLER
020002    09/21/87    McLester
020003    09/21/87    Thomas
020004    09/21/87    Rolfes
020005    09/21/87    Zambini
   .         .          .
   .         .          .
   .         .          .
020024    09/25/87    Charles
020025    09/25/87    Vidoni
020026    09/25/87    Coudray
```

This example shows how quickly a request for information can become complex. Yet a single SQL statement creates the view that represents your request for data. Then, treating the view just like a table, you can use the SELECT command to display the results. You can sometimes insert, delete, and update data using a view. For information about updating views, see the entry for the CREATE VIEW command in Chapter 7.

Dropping Views

You can continue to use a view until you drop (delete) the view, drop the database the view is in, or drop a table used in constructing a view. To drop a view, you use the DROP VIEW command:

```
SQL. DROP VIEW Orders;
```

The message **View ORDERS dropped** appears.

Defining Table and View Synonyms

Synonyms are alternate names that can be substituted for the names of tables and views defined in SQL. You can substitute a synonym for the name of a base table or view for use with any SQL command. For example, you can use synonyms to simplify and shorten table entries when entering commands that perform a query or update.

For example, to create a synonym for the Customer table, you could type:

```
SQL. CREATE SYNONYM C1 FOR Customer;
```

Once you've defined a synonym, you can use it to refer to a table or view in the SELECT command, the CREATE INDEX command, INSERT, UPDATE, DELETE, and DROP commands.

NOTE
Synonyms may also be used in place of table or view names in the DROP TABLE and DROP VIEW commands. Be careful to avoid destroying a table or view inadvertently.

To drop the synonym defined for the Customer table, you would type:

```
SQL. DROP SYNONYM C1;
```

SQL Indexes

One of the most important features of any database system is the ability to find stored information quickly. Indexes are one of the tools that dBASE IV SQL uses to find and retrieve data.

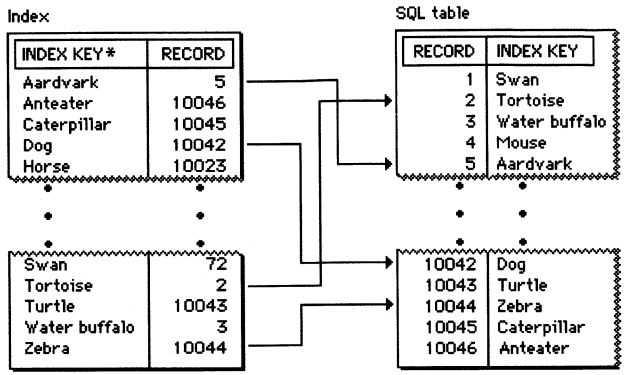
dBASE IV SQL uses indexes to keep track of the location of rows within a database table. Knowing where data is stored helps speed up data retrieval, similar to the way a book's index helps you find the pages where certain words or topics are located.

You build SQL indexes to match the order in which you might want to see the data. This is usually determined by the SELECT, ORDER BY, and GROUP BY statements you plan to execute. However, unlike the operation of indexes in dBASE mode, while you can specify the indexes you want to create, you don't specify which index to use when you retrieve data. dBASE IV SQL automatically selects the most efficient index.

Index files can only be built for tables, not for views. However, when you use a view, dBASE IV SQL is usually able to use indexes based on the tables underlying the view.

The indexes that dBASE IV SQL creates are defined as tags within a dBASE .mdx index file with the same name as the SQL table. When you specify particular columns on which to build an index, the index creates a list of the location of rows, in the order in which the rows should appear. You can create up to 47 different indexes for each database table.

You can think of an index tag as a table having two columns. The first column contains the values of columns in the related database table. The second column contains the storage locations of the corresponding rows. Figure 2-7 shows how an index might appear for an SQL table.



*The index key columns contain the values of columns for which the index is created. Index key values are arranged in ascending or descending order, depending on whether you specify ASCending (the default) or DESCending order when creating the index.

Figure 2-7 Indexing a table

While indexes speed up the retrieval of data from tables, they slow down the performance of updates. Indexes also take up disk space. When you perform updates to a database table, each index based on that table also needs to be updated. You should therefore try to limit the indexes you create.

You should delete indexes if you no longer need them, if the table being indexed is small, or if you do not access data in the same order as the index was built. Note that indexes are automatically dropped if you drop the table on which they are based.

Creating Indexes

You create indexes with the CREATE INDEX command. The syntax of this command is:

```
CREATE [UNIQUE] INDEX <index name>
    ON <table name> (<column name> [ASC/DESC]
    [,<column name> [ASC/DESC]...);
```

The columns specified are the ones by which rows in the index are ordered. These columns are called *index keys*. The maximum combined length allowed for columns used to create an index key is 100.

The UNIQUE keyword specifies a unique index. This means that specified index columns in each row of a table will contain unique values. When you insert new rows in the associated table, only unique indexed column values will be accepted. The unique index also ensures that indexed column values will remain unique. When you update a database table, a unique index ensures that you do not inadvertently create duplicate column entries.

NOTE
Use unique indexes sparingly, since they require checking every row you insert or update against existing values in the index. Also, if you create a unique SQL index, the dBASE database file associated with the SQL table will only be available as a read-only file in dBASE mode.

The ASC/DESC keywords specify the order in which rows will be arranged by value of the data in the indexed column. Ascending value indexes are the default, so you don't have to include the ASC keyword. You may create indexes on all column types except logical.

Figure 2-8 shows a display of the Customer database table, listing the clients in alphabetical order by last name. If a customer file is large, you would probably want to create an index ordered alphabetically by last name. To create this index, you would type:

```
SQL. CREATE INDEX Lastname
     ON Customer (Lastname);
```

The message **Index LASTNAME created** appears.

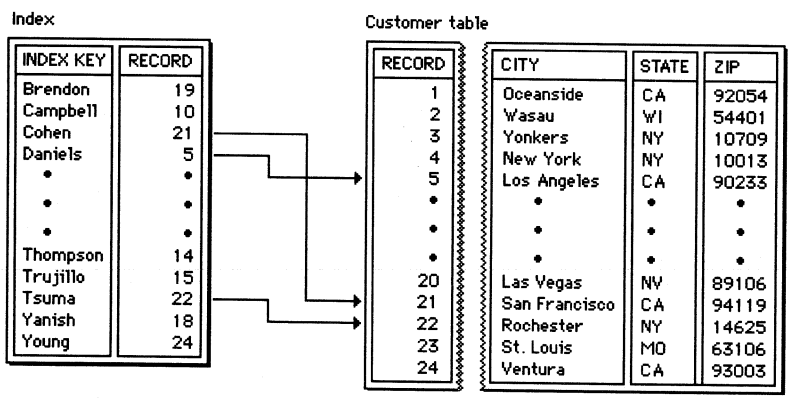


Figure 2-8 Indexing the Customer table

You can similarly define a descending order index. You can also build indexes based on more than one column as long as you do not specify both ascending and descending ordered columns within the same index.

To create a unique index to prevent entries of duplicate order numbers in the Sales table, you could type:

```
SQL. CREATE UNIQUE INDEX Order_No
     ON Sales (Order_No);
```

The index created forces data values in the Order_No column of the Sales table to be unique. If you try to insert a new order number duplicating any other existing order number, it will not be accepted. Also, if you update order numbers in the table, the index will make sure that you do not inadvertently duplicate existing order numbers. If you try to create a unique index and the table already contains duplicate entries, an error message appears.

Dropping Indexes

You drop an index with the DROP INDEX command. For example, to remove the Order_No index, type:

```
SQL. DROP INDEX Order_No;
```

The message **Index ORDER_NO dropped** appears.

SQL System Catalogs

dBASE IV SQL keeps track of all SQL *objects* (tables, views, synonyms, indexes) in a database using catalog tables. It also keeps track of authorization privileges assigned to different users. dBASE IV SQL uses the information in the catalog tables when performing queries or updates. As the size of your database becomes larger, the catalog tables play an increasingly important role. dBASE IV SQL uses the catalog information to perform database operations as fast and reliably as possible.

A set of system catalog tables is constructed for each database you create with the CREATE DATABASE command. There are ten tables in each database catalog. (See Chapter 8 for a description of the catalog tables.) A master catalog table, Sysdbs, is located in the SQL *home* directory (the directory in which the SQL system files are installed). The Sysdbs master catalog table contains the name of each database, the log-in name (user ID) of the person who created it, the date it was created, and the full DOS path location of the database directory.

You can display information from the catalog tables just like any other SQL table. For example, to display all the tables and views in the current database stored in the Systabls table, you could type:

```
SQL. SELECT Tbname, Tbtype
     FROM Systabls;
```

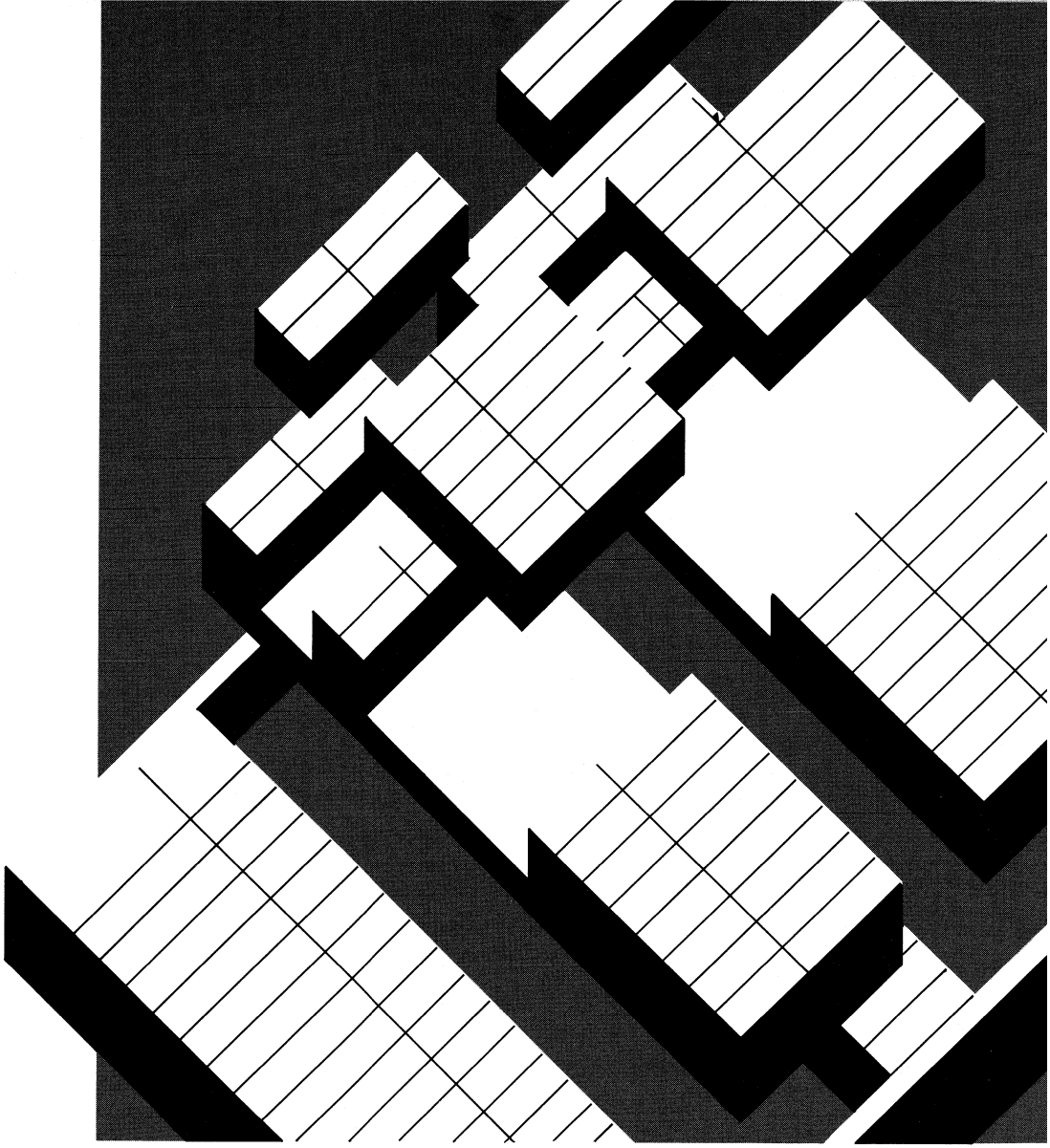
This SELECT statement displays both tables and views in the current database. The Tbtype column indicates whether files are base tables (T), views (V), or temporary tables (D or K).

You can display the definition of columns within these tables by querying the Syscols table. For example:

```
SQL. SELECT Colname, Coltype, Collen
     FROM Syscols
     WHERE Tbname="CUSTOMER";
```

While you can view the contents of the catalog tables, you cannot normally update them, unless you've installed PROTECT and the SQLDBA superuser ID, and are logged in as the SQLDBA user. (See Chapter 5 for a description of how to use PROTECT for SQL operations.)

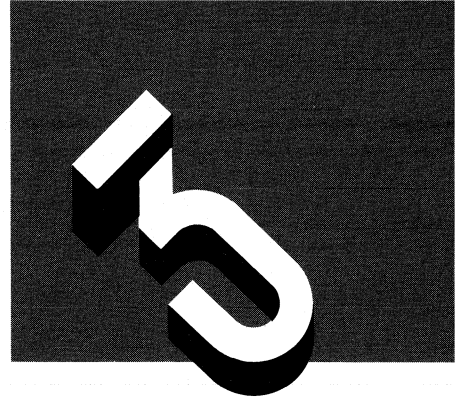
Queries and Updates



i 1 2 3 4 5 6 7 8 A

B C D In

SQL Queries



In this chapter you'll learn about SQL commands that retrieve data. Most of these operations involve the SELECT command. The variations and flexibility of the SELECT command's syntax allow you to create single statements to retrieve, insert, update, and delete data. This chapter begins with the simpler forms of the SELECT command and progresses through more complex examples that use all the options of the SELECT command.

The examples in this chapter show how you can use SELECT to query data from single tables. In Chapter 4, you'll learn how to use the SELECT command in operations combining data from more than one table.

What This Chapter Covers

The topics covered in this chapter are:

- Entering simple queries and queries with conditions
- Defining and using expressions
- Using SQL aggregate functions
- Using the BETWEEN, IN, and LIKE predicates
- Ordering and grouping rows
- Combining queries with the UNION clause

Preparing for This Chapter

The examples in this chapter use SQL sample tables provided with dBASE IV. (Refer to Appendix D in this book for a listing of the tables.) To run any examples in this chapter, start SQL in the interactive mode and activate the Samples database with the START DATABASE command. (You may also start the Samples database by adding SQLDATABASE = SAMPLES to your Config.db file.) All example results assume that you're working with an unmodified copy of the SQL tables in the database. To ensure that you're using data from the original sample tables, DROP the Samples database and then re-CREATE it. Use DBSETUP to copy the original sample files into the Samples database directory.

SELECT Overview

The SELECT command is used in both interactive and embedded SQL modes to retrieve data. With a single SELECT statement, you can instruct SQL to return any set of data from one or more tables. The complete syntax of the SQL command includes the clauses listed below. You can combine these clauses in the order listed to form SQL queries.

SELECT < clause >
[INTO < clause >]
FROM < clause >
[WHERE < clause >]
[GROUP BY < clause >]
[HAVING < clause >]
[UNION subselect]...
[ORDER BY < clause > /FOR UPDATE OF < clause >]
[SAVE TO TEMP < clause >];

NOTE

1. *The INTO and FOR UPDATE OF clauses are used to construct embedded SQL mode queries and are described in Chapter 6, "Embedding SQL Commands."*

2. *The term* subselect *in the UNION clause refers to the SELECT, FROM, WHERE, GROUP BY, and HAVING clauses (shown in bold type in the syntax listed above). A subselect can be joined with another SELECT in the UNION clause or as a subquery, or can appear in combination with other commands such as DELETE, INSERT, and UPDATE.*

Simple Queries

The simplest form of the SELECT command that displays specified columns from a single table is:

SELECT < columns >
  FROM < tables > ;

For example, to display the Company, Firstname, and Lastname columns from the Customer table, you would type:

```
SQL. SELECT Company, Firstname, Lastname
     FROM Customer;
```

The system displays the result:

```
COMPANY                 FIRSTNAME       LASTNAME
Leonard Design Services Rick            Leonard
Ace Furniture           Lisa            Martin
Custom Furniture        Daniel          Pollock
The Office              Dominique       LeClerc
     .                      .               .
     .                      .               .
     .                      .               .
Classic Interiors       Eric            Lawson
Commercial Interiors LTD Sandy          Young
```

In this example, you specify in the SELECT clause the column names to appear in the display: Company, Firstname, and Lastname. (You also dictate the order in which the columns appear.) The FROM clause indicates that the columns are from the Customer table.

The output from a SELECT statement is called a *result table*. The result table is not like other tables you create. It is not stored, nor can you reference the table after the SELECT statement has finished executing. You can, however, use the SAVE TO TEMP clause to save the result table as a temporary SQL table to use during the current session. You can also save the result table as a dBASE .dbf file with the KEEP option.

The headings that appear over individual columns indicate the columns of the table from which the corresponding data is taken. When the SELECT statement displays information from more than one table (specified in the FROM clause), each column is prefaced by the name of the table in which the column is defined.

NOTE

1. *When displaying large result tables, you can specify SET PAUSE ON to display the information one screen at a time. You can also press* **Ctrl-S** *to alternately halt or resume the scrolling of data.*
2. *You can compress the display of columns by specifying alias or synonym names in place of table names in the FROM clause. Use the SET HEADINGS OFF command if you want to remove column headings from the display.*

Selecting All Columns

SQL allows you to include a special symbol, the asterisk (*), in a SELECT statement as a shorthand method of selecting all columns in a table.

```
SQL. SELECT *
     FROM Staff;
```

would display the result:

```
STAFF_NO  LASTNAME  FIRSTNAME  HIREDATE  LOCATION     SUPERVISOR  SALARY  COMMISSION
000001    Zambini   Rick       02/15/80  LOS ANGELES  000000      6000          5.0
000003    Vidoni    Cheryl     03/06/80  NEW YORK     000000      5780          5.0
000004    Coudray   Sandy      06/06/80  LOS ANGELES  000001      6237          5.0
000006    Thomas    Pat        01/08/81  NEW YORK     000003      5875          5.0
000008    McLester  Debbie     04/12/81  LOS ANGELES  000001      4792          5.0
000011    Michaels  Delores    05/05/82  CHICAGO      000012      4927          7.0
000012    Charles   Ted        02/02/83  CHICAGO      000000      5945          5.0
000013    Marin     Mark       06/05/83  LOS ANGELES  000001      4802         11.0
000015    Roddick   Mary       02/13/84  NEW YORK     000003      5493          8.0
000016    Long      Nicole     08/18/84  NEW YORK     000003      5190          7.0
000019    Rolfes    Chuck      09/09/84  LOS ANGELES  000001      4586          6.0
000020    Sanders   Kathy      03/23/85  CHICAGO      000012      3783          5.0
```

In this example, the asterisk indicates that all columns appear in the result table in the order in which you created them in the SQL table.

SELECT with DISTINCT

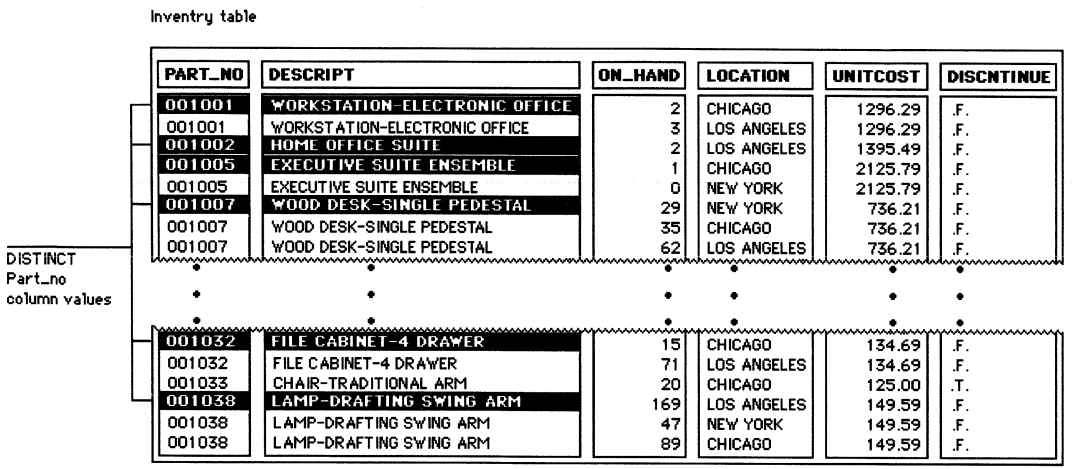
The sample Inventry table contains a combined inventory of items stored in three different locations: Chicago, Los Angeles, and New York. Typing **SELECT * FROM Inventry** would display listings of the same part number for each location stocking it. To display only distinct rows within a table, you use the *DISTINCT* keyword in the SELECT clause. For example, to display each part number in the Inventry table only once, you could type:

```
SQL. SELECT DISTINCT Part_no, Descript
     FROM Inventry;
```

The system would display the result:

```
PART_NO   DESCRIPT
001001    WORKSTATION-ELECTRONIC OFFICE
001002    HOME OFFICE SUITE
001005    EXECUTIVE SUITE ENSEMBLE
001007    WOOD DESK-SINGLE PEDESTAL
    .
    .
    .
001033    CHAIR-TRADITIONAL ARM
001038    LAMP-DRAFTING SWING ARM
```

The display now shows each part number only once. Figure 3-1 shows how rows with DISTINCT column values are selected from the Inventry table.



PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
001001	WORKSTATION-ELECTRONIC OFFICE	3	LOS ANGELES	1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001007	WOOD DESK-SINGLE PEDESTAL	35	CHICAGO	736.21	.F.
001007	WOOD DESK-SINGLE PEDESTAL	62	LOS ANGELES	736.21	.F.
001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
001032	FILE CABINET-4 DRAWER	71	LOS ANGELES	134.69	.F.
001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
001038	LAMP-DRAFTING SWING ARM	47	NEW YORK	149.59	.F.
001038	LAMP-DRAFTING SWING ARM	89	CHICAGO	149.59	.F.

Figure 3-1 Selection of DISTINCT rows

SELECT with a WHERE Clause

So far, the examples have described options of the SELECT clause used to pick columns displayed in a query. Sometimes, though, you'll want to specify or qualify the rows that appear in a query by specifying a condition in the WHERE clause. The syntax then becomes:

```
SELECT <columns>
  FROM <tables>
  WHERE <condition>;
```

Using the sample Inventry table, you can construct a query to display the parts kept at any single location. For example, to list all the parts stocked in Chicago, you can type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     WHERE Location = "CHICAGO";

PART_NO  DESCRIPT                      LOCATION        ON_HAND
001001   WORKSTATION-ELECTRONIC OFFICE CHICAGO             2
001005   EXECUTIVE SUITE ENSEMBLE      CHICAGO             1
001009   CHAIR-ADJUSTABLE SWIVEL       CHICAGO           124
001024   LAMP-BRASS TABLE              CHICAGO           140
001032   FILE CABINET-4 DRAWER         CHICAGO            15
001033   CHAIR-TRADITIONAL ARM         CHICAGO            20
001007   WOOD DESK-SINGLE PEDESTAL     CHICAGO            35
001013   CHAIR-MODERN PNEUMATIC        CHICAGO           115
001038   LAMP-DRAFTING SWING ARM       CHICAGO            89
001027   DESK-EXECUTIVE-6 FOOT         CHICAGO            20
001031   CHAIR-EXECUTIVE SWIVEL/TILT   CHICAGO            44
```

In this example, the SELECT clause identifies the columns to appear on the display, and the FROM clause identifies the Inventry table. The line containing the WHERE clause specifies a *search condition* (Location = "CHICAGO"). The search condition defines a criterion that selected rows must meet. The *equals* operator compares the value of the column in the specified condition with the value you specify. Figure 3-2 shows the selection of rows from the Inventry table.

PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.38	.T.
001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
001024	LAMP-BRASS TABLE	140	CHICAGO	230.79	.F.
001025	DESK-EXECUTIVE-5 FOOT	63	LOS ANGELES	985.00	.F.
001029	FILE CABINET-2 DRAWER	200	NEW YORK	89.95	.T.
001031	CHAIR-EXECUTIVE SWIVEL/TILT	79	LOS ANGELES	420.00	.F.
001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Figure 3-2 Selection of rows

When the SELECT statement has a WHERE clause, SQL uses the search condition to test each row. When the search condition is true, the row is selected and included in the result table. When the search condition is false, the row is ignored and is not included in the result table. When specifying a search condition, the columns included in the condition can be different from those specified in the SELECT clause.

Comparison Operators

SQL provides several operators you can use to define search conditions. These operators are listed in Table 3-1.

Table 3-1 Comparison operators

Operator	Description
=	Equals
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
< >	Not equals (# also supported for dBASE compatibility)
!	Negation of <, >, and = comparison operators, for example, !<, !>, or !=

The two values being compared must be of compatible data types (character to character, numeric to numeric, and so on). However, data types SMALLINT, INTEGER, DECIMAL, NUMERIC, and FLOAT are also all compatible with each other.

You must enclose character type constants in either single or double quotes. Also, comparisons on character data types are done differently than, say, comparisons on date or numeric data types. Comparisons of character data types use the ASCII representation of characters or letters. Thus, when you do a comparison on character types, the result of the comparison depends on whether characters are in upper or lower case. That is, the letter *a* is not equal to *A*. *A* is less than *a* and *Z* is also less than *a*.

NOTE

1. *Logical data type columns can be specified in a WHERE clause both with a comparison operator or without one. For example, if Shipped is a logical data type column, you could construct the following equivalent queries:*

   ```
   SELECT * FROM Orders WHERE SHIPPED = .T.;
   SELECT * FROM Orders WHERE Shipped;
   ```

2. *You can also use dBASE functions to convert column data type values in a WHERE clause search condition. For example, you can use the UPPER() function to convert characters to upper case, or use the CTOD() function to convert a date string to compare with a date column. (See Appendix C for a list of functions you can use with SQL commands.)*

Combining Conditions

You can combine search conditions in a WHERE clause. For example, if you wanted to find only the parts stocked in Los Angeles of which you have fewer than 20, you would enter:

```
SQL. SELECT Part_no, Descript, On_hand, Location
     FROM Inventry
     WHERE Location = "LOS ANGELES" AND On_hand <20;

PART_NO  DESCRIPT                          ON_HAND  LOCATION
001002   HOME OFFICE SUITE                       2  LOS ANGELES
001001   WORKSTATION-ELECTRONIC OFFICE           3  LOS ANGELES
```

The result shows only those inventory items that match both conditions in the WHERE clause. SQL supports three different operators for logically combining search conditions. These connecting operators are listed in order of precedence in Table 3-2.

Table 3-2 SQL logical operators

Operator	Description
NOT	Selects rows to include in the result table that do not meet the condition
AND	Both conditions must be true to qualify the row
OR	One or both of the conditions must be true

When you have multiple conditions in a WHERE clause, the individual conditions are evaluated and then combined with the logical operators to come up with a single result, true or false. This final evaluation determines whether a row is included in the result table.

The precedence of the logical operators dictates the order in which conditions in the WHERE clause are evaluated and combined. As listed in Table 3-2, the NOT operator has highest precedence, followed by AND, and then OR. If, for example, a WHERE clause contains multiple conditions combined using the same type of logical operator, the conditions are simply evaluated left to right.

Parentheses can be used to clarify or override the order in which conditions are evaluated. Sometimes you may have to include parentheses to get the result you want.

For example, suppose you wanted to expand the conditions in the previous example to include inventory in Los Angeles or New York, and in both cases exclude items in short supply if they cost less than $1,000. You might type the following:

```
SQL. SELECT Part_no, Descript, Location, On_hand, Unitcost
     FROM Inventry
     WHERE Location = "LOS ANGELES" OR Location = "NEW YORK"
     AND On_hand <20 AND Unitcost <1000;

PART_NO  DESCRIPT                    LOCATION        ON_HAND   UNITCOST
001002   HOME OFFICE SUITE           LOS ANGELES           2    1395.49
001008   WORKSTATION-STAND           LOS ANGELES          22     275.66
001015   CREDENZA-OAK SLIDING DOOR   NEW YORK             15     745.00
001022   TABLE-WALNUT OCCASIONAL     NEW YORK              5     414.95
   .            .                       .                .        .
   .            .                       .                .        .
   .            .                       .                .        .

001032   FILE CABINET-4 DRAWER       LOS ANGELES          71     134.69
001001   WORKSTATION-ELECTRONIC OFFICE LOS ANGELES         3    1296.29
001029   FILE CABINET-2 DRAWER       LOS ANGELES         145      89.95
```

You can see that the result is not what you intended. As it turns out, SQL first evaluates *Location = "NEW YORK" AND On_hand < 20*, then evaluates the AND expression *Unitcost < 1000*, and finally evaluates the OR expression *Location = "LOS ANGELES"*.

To get the results you wanted, retype the command (or press ↑ to display the last command and edit the SELECT statement) to include the parentheses.

```
SQL. SELECT Part_no, Descript, Location, On_hand, Unitcost
     FROM Inventry
     WHERE (Location = "LOS ANGELES" OR Location = "NEW YORK")
     AND (On_hand <20 AND Unitcost <1000);

PART_NO  DESCRIPT                    LOCATION        ON_HAND   UNITCOST
001015   CREDENZA-OAK SLIDING DOOR   NEW YORK             15     745.00
001022   TABLE-WALNUT OCCASIONAL     NEW YORK              5     414.95
```

Now, you see the results you wanted. First SQL evaluates the expression *(Location = "LOS ANGELES" OR Location = "NEW YORK")*. Then it evaluates the expression *(On_hand < 20 AND Unitcost < 1000)*. Finally, it checks the result of the two separate expressions to determine whether the row is included in the result.

Defining and Using Expressions

Expressions are statements like those found in the WHERE clauses shown previously. For example, "LOS ANGELES" is a constant-valued expression and *Unitcost*10* is a variable expression. When evaluated, an expression always returns a single value, that is, a condition (true or false), a numeric value, or a character string.

An expression may contain a combination of columns, arithmetic operators (+, −, /, *, **, and ^), constants (numeric, logical, and character), memory variables, and dBASE functions. Expressions you can define and use in SQL are similar to those you can define and use with dBASE commands. Table 3-3 lists the arithmetic operators you can use with SQL. Figure 3-3 shows all the elements you can use in constructing an expression.

Table 3-3 Arithmetic operators

Operator	Description
** and ^	Exponentiation
+ and −	Unary operator (plus is the default; minus for negative values)
* and /	Multiplication/division
+ and −	Addition/subtraction

You can use all these operators with numeric values. Only the addition and subtraction operators can be used with character strings (for concatenation).

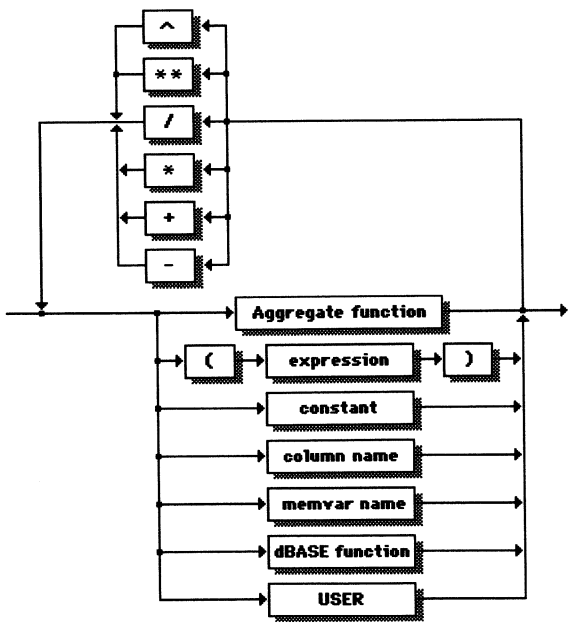


Figure 3-3 SQL expressions

Expressions can be included in SELECT statements to change the resulting display. Some of the other ways you can use expressions with dBASE IV SQL statements are listed below.

- To insert descriptive column information into a SELECT result table
- To define new calculated columns in the SELECT clause
- To define conditions in a WHERE clause
- To define expressions for the HAVING clause

Expressions in the SELECT Clause

There are two different ways you can use expressions in the SELECT clause. The first lets you insert descriptive column information next to the columns displayed by a SELECT statement.

The second way to use expressions in the SELECT clause is to define a new column, known as a *calculated column*. This column does not exist in the SQL table but may be based in some way on existing columns. Instead of the column from the SQL table appearing in the result table, an expression that defines the new column appears. For example, if you wanted to display employee salaries per year instead of per month, you could type:

```
SQL. SELECT Staff_no, Firstname, Lastname,
       Salary*12, "(Yearly Salary)"
     FROM Staff;

STAFF_NO  FIRSTNAME      LASTNAME        EXP1  EXP2
000001    Rick           Zambini      72000.00  (Yearly Salary)
000003    Cheryl         Vidoni       69360.00  (Yearly Salary)
000004    Sandy          Coudray      74844.00  (Yearly Salary)
000006    Pat            Thomas       70500.00  (Yearly Salary)
000008    Debbie         McLester     57504.00  (Yearly Salary)
000011    Delores        Michaels     59124.00  (Yearly Salary)
000012    Ted            Charles      71340.00  (Yearly Salary)
000013    Mark           Marin        57624.00  (Yearly Salary)
000015    Mary           Roddick      65916.00  (Yearly Salary)
000016    Nicole         Long         62280.00  (Yearly Salary)
000019    Chuck          Rolfes       55032.00  (Yearly Salary)
000020    Kathy          Sanders      45396.00  (Yearly Salary)
```

A calculated column appears in the result table according to the order in which columns or calculated columns are listed in the SELECT clause. Note that SQL displays the headings *EXP1* and *EXP2* over the two columns specified as expressions in the SELECT clause.

Columns that are derived from expressions (including calculated columns, dBASE functions, constants, and memory variables) are shown with the heading EXP followed by a number indicating the position of the expression in a SELECT clause.

Expressions in the WHERE Clause

Conditions specified in the WHERE clause (and the predicates BETWEEN, IN, and LIKE) may use expressions. These expressions evaluate to a logical true or false value. You can use logical operators to combine several conditions. This is illustrated by the SELECT statement shown below.

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     WHERE Location = "LOS ANGELES" AND On_hand < 75;

PART_NO  DESCRIPT                    LOCATION      ON_HAND
001002   HOME OFFICE SUITE           LOS ANGELES         2
001008   WORKSTATION-STAND           LOS ANGELES        22
001025   DESK-EXECUTIVE-5 FOOT       LOS ANGELES        63
001007   WOOD DESK-SINGLE PEDESTAL   LOS ANGELES        62
001013   CHAIR-MODERN PNEUMATIC      LOS ANGELES        35
001032   FILE CABINET-4 DRAWER       LOS ANGELES        71
001001   WORKSTATION-ELECTRONIC OFFICE LOS ANGELES       3
```

In this example, the character column Location is compared with a simple character-string constant, and the numeric column On_hand is compared with a simple numeric constant.

dBASE IV SQL allows you to include more complex expressions in the WHERE clause just as you can in the SELECT clause. These can further qualify the data displayed by a SELECT statement. You can perform calculations in specifying a condition, and also specify comparisons based on the values of more than one column.

For example, to display employees from the staff whose annual salary is more than $50,000, you could type the following query:

```
SQL. SELECT Staff_no, Firstname, Lastname, Salary*12
     FROM Staff
     WHERE Salary*12 > 50000;

STAFF_NO  FIRSTNAME      LASTNAME          EXP1
000001    Rick           Zambini       72000.00
000003    Cheryl         Vidoni        69360.00
000004    Sandy          Coudray       74844.00
000006    Pat            Thomas        70500.00
000004    Sandy          Coudray       74844.00
000008    Debbie         McLester      57504.00
000011    Delores        Michaels      59124.00
000012    Ted            Charles       71340.00
000013    Mark           Marin         57624.00
000015    Mary           Roddick       65916.00
000016    Nicole         Long          62280.00
000019    Chuck          Rolfes        55032.00
```

This comparison could also be made using a memory variable. That way, you could write a single SELECT statement that would provide different results based on the value in the memory variable. If you assigned the memory variable *msalary* a numeric value of 50,000, you could type:

```
SQL. SELECT Staff_no, Firstname, Lastname, Salary*12
     FROM Staff
     WHERE Salary*12 > msalary;
```

TIP

This technique of using memory variables to influence the result of a SELECT statement is most useful in programs. You'll learn more about using dBASE memory variables with SQL in Chapter 6.

Using dBASE Functions

Besides using constants, column values, and arithmetic operators in expressions, you can also use dBASE functions. dBASE IV provides over 100 functions that can be used with SQL (see Appendix C).

As an example, you could use the dBASE DATE() function to display all employees in the Staff table employed for longer than five years from the current date (September 30, 1988). You would type:

```
SQL. SELECT Firstname, Lastname, Hiredate
     FROM Staff
     WHERE (DATE()-hiredate) > 365*5;

FIRSTNAME    LASTNAME     HIREDATE
Rick         Zambini      02/15/80
Cheryl       Vidoni       03/06/80
Sandy        Coudray      06/06/80
Pat          Thomas       01/08/81
Debbie       McLester     04/12/81
Delores      Michaels     05/05/82
Ted          Charles      02/02/83
Mark         Marin        06/05/83
```

The expression *(DATE()-hiredate)* calculates the number of days since an employee was hired.

dBASE functions are also useful in defining character-string comparisons. dBASE IV provides such functions as UPPER(), LOWER(), and SUBSTR(), to name a few. (See Appendix C for a complete list of dBASE functions that can be used with SQL.) For example, to locate all customers in the Customer table who live in San Francisco, you might construct the following query:

```
SQL. SELECT Company, City, State
     FROM Customer
     WHERE UPPER(City) = "SAN FRANCISCO";

COMPANY                  CITY           STATE
Black's Furniture Store  San Francisco  CA
A1 Office Supply Store   San Francisco  CA
Cohen's Furniture        San Francisco  CA
```

With this query, the dBASE UPPER() function transforms values from columns in the Customer table. The columns in the Customer table will match even if the characters were entered in upper and lower case.

You may also specify dBASE functions in a SELECT clause to transform the data displayed in a result table. Changing the example shown above, you could type *SELECT Company, UPPER (City), State FROM Customer WHERE UPPER (City) = "SAN FRANCISCO"*. The result table column heading for the city will now be shown as *UPPER(CITY)*.

NOTE
When including functions in an SQL command, you must specify the full name of each dBASE function.

SQL Aggregate Functions

Besides the functions that the dBASE language provides, SQL provides five special functions called *aggregate* functions that can be used with the SELECT command. These are:

- COUNT() — Counts the number of rows selected
- SUM() — Sums the values of a numeric column
- MIN() — Finds the minimum value of a character, date, or numeric column
- MAX() — Finds the maximum value of a character, date, or numeric column
- AVG() — Calculates the average of the values in a numeric column

NOTE
When the MIN() and MAX() functions are used with character column arguments, they return column values with the lowest and highest character constant values, respectively. ASCII character values in each column are compared starting with the first character on the left.

Character type constants must be enclosed in either single or double quotes. Also, comparisons on character data types are done differently than, say, comparisons on date or numeric data types. Comparisons of character data types use the ASCII representation of characters or letters. Thus, when you do a comparison on character types, the result of the comparison depends on whether characters are in upper or lower case. That is, the letter *a* is not equal to *A*. *A* is less than *a* and *Z* is also less than *a*.

The aggregate functions operate on all the values of a column (or all columns) in a table, view, or intermediate result table, or values based on a group of rows qualified by a GROUP BY or HAVING clause. This differs from the operation of dBASE functions allowed in SQL mode, which operate on the value of a single column. Figure 3-4 shows how you can use the SQL functions and illustrates the arguments you can supply to them.

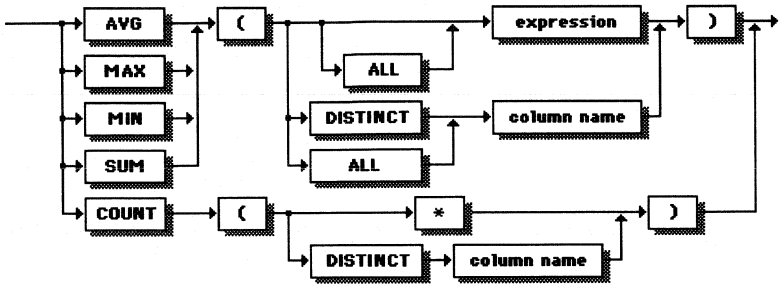


Figure 3-4 SQL aggregate functions

Only expressions including a column name can be used as arguments to the SQL functions. You cannot use a constant, memory variable, or logical column as an argument to an SQL function. Specifying DISTINCT with an SQL function prevents duplicate values within a table or group of rows from being used in the evaluation of the function's result.

Using Aggregate Functions in a SELECT Clause

Aggregate functions can be used in place of a column name in the SELECT clause. They operate on the values of a result table or a group and return a single value. For example, to find the number of employees in the Staff table, you could type:

```
SQL. SELECT COUNT(*)
     FROM Staff;

            COUNT1
             12.00
```

NOTE

1. *The asterisk symbol (*) indicates that the COUNT() function operates on the entire row, not on any single column.*
2. *Columns that display the result of SQL aggregate functions are shown with a corresponding heading AVG, COUNT, MAX, and SUM, followed by a number indicating the position of the aggregate function in the SELECT clause.*

Using the SUM() function, you can calculate the total monthly payroll from the Staff table. You would type:

```
SQL. SELECT SUM(Salary)
     FROM Staff;

          SUM1
      63410.00
```

Using the MAX(), MIN(), and AVG() functions you can, for example, find the highest, lowest, and average salaries in the Staff table. You would type:

```
SQL. SELECT MAX(Salary), MIN(Salary), AVG(Salary)
     FROM Staff;

          MAX1          MIN2          AVG3
       6237.00       3783.00       5284.17
```

Remember that aggregate functions operate on the values of columns in all rows of a result table or group. You cannot mix aggregate functions and regular columns in the SELECT clause, except as allowed by the GROUP BY clause (see Chapter 4).

The aggregate functions allow you to specify the DISTINCT keyword. This keyword causes these functions to recognize only unique values in a column. For example, to total the number of cities represented in the Customer table, you could type:

```
SQL. SELECT COUNT(DISTINCT City)
     FROM Customer;

          COUNT1
           15.00
```

Aggregate Functions Combined with a WHERE Clause

You can also use aggregate functions in a SELECT statement that has a WHERE clause. The system will select rows using the WHERE clause and then apply the aggregate function to the result table. For example, to find the number of customers from the Customer table who live in the state of New York, you would type:

```
SQL. SELECT COUNT(*)
     FROM Customer
     WHERE State = "NY";

          COUNT1
            5.00
```

In this example, the SELECT command first creates a result table containing rows where the State column matches the value "NY". The COUNT() function then calculates and displays the number of rows in the result table. Figure 3-5 shows how aggregate functions use column values from selected rows.

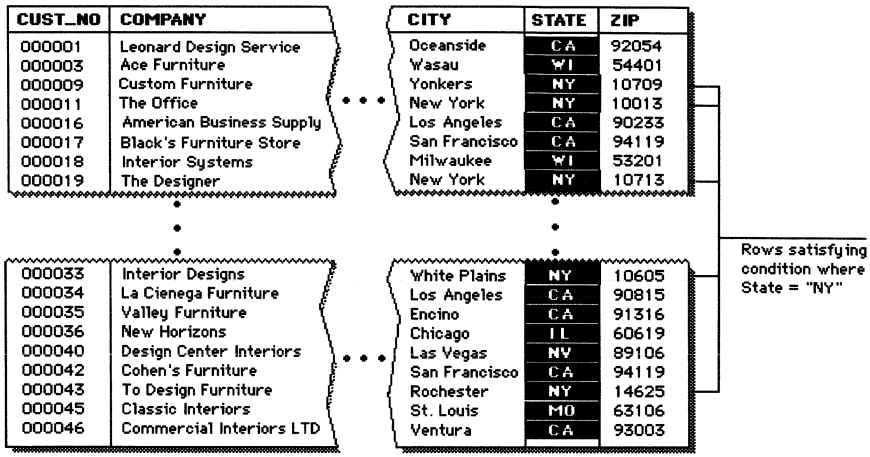


Figure 3-5 Aggregate functions with a WHERE clause

SELECT with BETWEEN, IN, and LIKE Predicates

The SELECT command allows you to specify search conditions that use the *predicates* BETWEEN, IN, and LIKE in the WHERE clause. BETWEEN, IN, and LIKE are called predicates because evaluation of their WHERE clause is based (or predicated) on the result of these operations.

- BETWEEN tests for a value within a specified range
- IN tests for a value matching an item in a list
- LIKE compares a character column to a specified string

The BETWEEN Predicate

The BETWEEN predicate tests whether a column value is between two specified values. BETWEEN simplifies the writing of a search condition and makes the resulting search condition easier to read and understand.

Using BETWEEN, you can replace a condition like:

WHERE salary > = 1500 AND salary < = 2500; with:

WHERE salary BETWEEN 1500 AND 2500;

The range specified with BETWEEN is *inclusive*, that is, rows with column values that match either of the two values specified in the range will appear in the result table, as well as those between the specified range.

You can use the BETWEEN predicate with character, date, and numeric type columns. For character columns, the range values should be entered with upper- and lower-case letters to match the case of entries you expect in the column. You also can use dBASE functions to convert range values or the data returned from columns.

The IN Predicate

The IN predicate tests whether a column value matches one of those specified in a list of values. Like BETWEEN, the IN predicate simplifies the writing of a search condition and makes the resulting search condition easier to read and understand.

Using IN, you can replace a condition like:

```
WHERE state = "AZ" OR state = "MO" OR state = "WI"
```

with:

```
WHERE state IN ("AZ","MO","WI")
```

For example:

```
SQL. SELECT Company, Address, City, State, Zip
     FROM Customer
     WHERE State IN ("AZ","MO","WI");

COMPANY                  ADDRESS           CITY        STATE  ZIP
Ace Furniture            1960 Lindley Ave. Wasau       WI     54401
Interior Systems         899 Kenwood St.   Milwaukee   WI     53201
Baker Furniture          6700 Tyler St.    Phoenix     AZ     85012
Modern Furniture Store   366 Shirley Ave.  Phoenix     AZ     85004
Al's Furniture & Supplies 40555 Brentwood  St. Louis   MO     63121
Contemporary Designs     5670 Colorado Blvd. Milwaukee WI     53220
Classic Interiors        2015 Edmonton     St. Louis   MO     63106
```

You can use the IN predicate to specify lists for date and numeric data type searches, besides the character types illustrated in the example. You can include dBASE functions, memory variables, other column names, and the USER keyword in place of values. Also, although expressions cannot be specified in a value list, you can specify a single expression to replace the value list (as long as it does not contain an SQL aggregate function).

The LIKE Predicate

The LIKE predicate selects rows by comparing a character column's value with a specified character string, the USER keyword, or a character-type memory variable. Besides exact character string searches, the LIKE predicate allows you to specify wildcard characters in the search string. This means that column values matching the specified pattern will also appear in the result table. The wildcard characters you can use are:

- underscore (_) — match any single character
- percent symbol (%) — match any number of characters

You can use the single-character wildcard, for example, to display customers from states starting with the letter N:

```
SQL. SELECT Company, City, State, Zip
     FROM Customer
     WHERE State LIKE "N_";

COMPANY                  CITY          STATE   ZIP
Custom Furniture         Yonkers       NY      10709
The Office               New York      NY      10013
The Designer             New York      NY      10713
Las Vegas Furniture      Las Vegas     NV      89106
Accent Furniture Designs Las Vegas     NV      89108
Interior Designs         White Plains  NY      10605
Design Center Interiors  Las Vegas     NV      89106
To Design Furniture      Rochester     NY      14625
```

You can use the multiple-character wildcard symbol, for example, to display parts that contain the text string "DESK" as part of their description:

```
SQL. SELECT DISTINCT Part_no, Descript
     FROM Inventry
     WHERE Descript LIKE "%DESK%";

PART_NO   DESCRIPT
001007    WOOD DESK-SINGLE PEDESTAL
001025    DESK-EXECUTIVE-5 FOOT
001027    DESK-EXECUTIVE-6 FOOT
```

Combining BETWEEN, LIKE, and IN Predicates

Just as with simple comparisons in the WHERE clause, you can combine BETWEEN, IN, and LIKE predicates with the AND, OR, and NOT logical operators. You can also enter parentheses to clarify or change the precedence when combined comparisons are evaluated.

Ordering Displays

In previous examples, the rows of the result tables appeared in the same order in which you added to the SQL tables. To change the order, you can use the ORDER BY clause of the SELECT statement. The syntax of the SELECT statement now becomes:

```
SELECT <columns>
    FROM <tables>
    WHERE <condition>
    ORDER BY <column/integer> [ASC/DESC]
        [, <column/integer> [ASC/DESC]...];
```

The columns you use in the ORDER BY clause must also be included in the SELECT clause. Also, you may specify integer values in the ORDER BY clause to indicate ordering by a derived column. Integer values specified in the ORDER BY clause correspond to column positions of the SELECT statement's result table.

TIP
To speed data retrievals, you should construct indexes that match the columns on which you often order your data or those you often specify in a WHERE clause.

Ordering SELECT Results on a Single Column

With the ORDER BY clause, you specify one or more columns on which to order the display. Rows are arranged in ascending or descending order based on the values in specified columns. For example, to display the part descriptions in the Inventry table in alphabetical order, you could type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     ORDER BY Descript ASC;
```

The ASC keyword specifies that the table should be displayed in ascending order. This keyword is optional since ascending order is the default. However, you may want to include it anyway to clearly indicate your choice.

Ordering SELECT Results on More than One Column

To order the display using more than one column, enter each column followed by either the ASC or DESC keyword (ASC optional). Separate each column entry with a comma. For example, to order the display of rows in the Inventry table by location and then by description, you would type:

```
SQL. SELECT Part_no, Descript, Location, ON_hand
     FROM Inventry
     ORDER BY Location ASC, Descript ASC;

PART_NO  DESCRIPT                      LOCATION      ON_HAND
001009   CHAIR-ADJUSTABLE SWIVEL       CHICAGO           124
001031   CHAIR-EXECUTIVE SWIVEL/TILT   CHICAGO            44
001013   CHAIR-MODERN PNEUMATIC        CHICAGO           115
001033   CHAIR-TRADITIONAL ARM         CHICAGO            20
001027   DESK-EXECUTIVE-6 FOOT         CHICAGO            20
001005   EXECUTIVE SUITE ENSEMBLE      CHICAGO             1
    .            .                       .               .
    .            .                       .               .
    .            .                       .               .

001031   CHAIR-EXECUTIVE SWIVEL/TILT   LOS ANGELES        79
001013   CHAIR-MODERN PNEUMATIC        LOS ANGELES        35
001025   DESK-EXECUTIVE-5 FOOT         LOS ANGELES        63
    .            .                       .               .
    .            .                       .               .
    .            .                       .               .

001001   WORKSTATION-ELECTRONIC OFFICE LOS ANGELES         3
001008   WORKSTATION-STAND             LOS ANGELES        22
001031   CHAIR-EXECUTIVE SWIVEL/TILT   NEW YORK           76
001015   CREDENZA-OAK SLIDING DOOR     NEW YORK           15
001025   DESK-EXECUTIVE-5 FOOT         NEW YORK           47
001027   DESK-EXECUTIVE-6 FOOT         NEW YORK           56
001005   EXECUTIVE SUITE ENSEMBLE      NEW YORK            0
001029   FILE CABINET-2 DRAWER         NEW YORK          200
    .            .                       .               .
    .            .                       .               .
    .            .                       .               .

001019   TABLE-BOARD ROOM              NEW YORK           12
001022   TABLE-WALNUT OCCASIONAL       NEW YORK            5
001007   WOOD DESK-SINGLE PEDESTAL     NEW YORK           29
```

When you specify more than one column to order by, the rows are arranged in order by the first column entry. Then those with the same value are arranged according to the second column entry.

Using the ORDER BY clause, you can also specify some columns to appear in ascending order and others to appear in descending order. Still using the Inventry table, you might want to list parts on hand with the location in alphabetical order and the number of parts on hand in decreasing order. You would type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     ORDER BY Location ASC, On_hand DESC;

PART_NO  DESCRIPT                    LOCATION     ON_HAND
001024   LAMP-BRASS TABLE            CHICAGO          140
001009   CHAIR-ADJUSTABLE SWIVEL     CHICAGO          124
001013   CHAIR-MODERN PNEUMATIC      CHICAGO          115
001038   LAMP-DRAFTING SWING ARM     CHICAGO           89
001031   CHAIR-EXECUTIVE SWIVEL/TILT CHICAGO           44
001007   WOOD DESK-SINGLE PEDESTAL   CHICAGO           35
    .            .                      .             .
    .            .                      .             .
    .            .                      .             .

001038   LAMP-DRAFTING SWING ARM     LOS ANGELES      169
001029   FILE CABINET-2 DRAWER       LOS ANGELES      145
001031   CHAIR-EXECUTIVE SWIVEL/TILT LOS ANGELES       79
001032   FILE CABINET-4 DRAWER       LOS ANGELES       71
001025   DESK-EXECUTIVE-5 FOOT       LOS ANGELES       63
    .            .                      .             .
    .            .                      .             .
    .            .                      .             .

 001019  TABLE-BOARD ROOM            NEW YORK          12
 001022  TABLE-WALNUT OCCASIONAL     NEW YORK           5
 001021  MANAGERS OFFICE ENSEMBLE    NEW YORK           3
 001005  EXECUTIVE SUITE ENSEMBLE    NEW YORK           0
```

In this example, inventoried items are displayed in ascending order based on the Location column. Rows for items with the same location column value are arranged in descending order based on values of the On_hand column.

ORDER BY with the WHERE Clause

If you use the WHERE clause, the only rows that appear will be those that satisfy its conditions. By adding the ORDER BY clause, you can choose the rows to display and arrange those rows in a specific order in a single SELECT statement.

For example, using the previous query on the Inventry table, you could type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     WHERE Location = "CHICAGO"
     ORDER BY Part_no;

PART_NO   DESCRIPT                        LOCATION      ON_HAND
001001    WORKSTATION-ELECTRONIC OFFICE   CHICAGO             2
001005    EXECUTIVE SUITE ENSEMBLE        CHICAGO             1
001007    WOOD DESK-SINGLE PEDESTAL       CHICAGO            35
001009    CHAIR-ADJUSTABLE SWIVEL         CHICAGO           124
001013    CHAIR-MODERN PNEUMATIC          CHICAGO           115
001024    LAMP-BRASS TABLE                CHICAGO           140
001027    DESK-EXECUTIVE-6 FOOT           CHICAGO            20
001031    CHAIR-EXECUTIVE SWIVEL/TILT     CHICAGO            44
001032    FILE CABINET-4 DRAWER           CHICAGO            15
001033    CHAIR-TRADITIONAL ARM           CHICAGO            20
001038    LAMP-DRAFTING SWING ARM         CHICAGO            89
```

Grouping Records

Two additional clauses of the SELECT statement, *GROUP BY* and *HAVING*, allow you to arrange rows into groups. You can then perform operations on those groups, usually using the aggregate functions. The GROUP BY and HAVING clauses change the scope of these functions from all the rows in a table to all the rows within a group.

The GROUP BY Clause

The GROUP BY clause combines rows from a SELECT statement's result table into groups in which specified columns have the same value. Each group is reduced to a single row in the result table. The columns in the row either identify the group or are the result of an SQL aggregate function acting on the group.

There are a couple of restrictions on using the GROUP BY clause. First, each column named in the SELECT clause must also be included in the GROUP BY clause. (Conversely, each column named in a GROUP BY clause must also be specified in the SELECT clause.) Second, you may not specify a derived column in a GROUP BY clause except for column names used in aggregate functions in the SELECT clause.

For example, you can use the GROUP BY clause with the Inventry table to count the number of parts available in all locations. You would type:

```
SQL. SELECT Part_no, Descript, SUM(On_hand)
     FROM Inventry
     GROUP BY Part_no, Descript;

G_PART_NO  G_DESCRIPT                          SUM1
001001     WORKSTATION-ELECTRONIC OFFICE       5.00
001002     HOME OFFICE SUITE                   2.00
001005     EXECUTIVE SUITE ENSEMBLE            1.00
   .               .                           .
   .               .                           .
   .               .                           .

001029     FILE CABINET-2 DRAWER             345.00
001031     CHAIR-EXECUTIVE SWIVEL/TILT       199.00
001032     FILE CABINET-4 DRAWER              86.00
001033     CHAIR-TRADITIONAL ARM              20.00
001038     LAMP-DRAFTING SWING ARM           305.00
```

In this example, the rows in the Inventry table are all grouped by the Part_no and Descript columns. The SUM() function then adds the quantity of each item (the On_hand column) in each group. This process is shown in Figure 3-6.

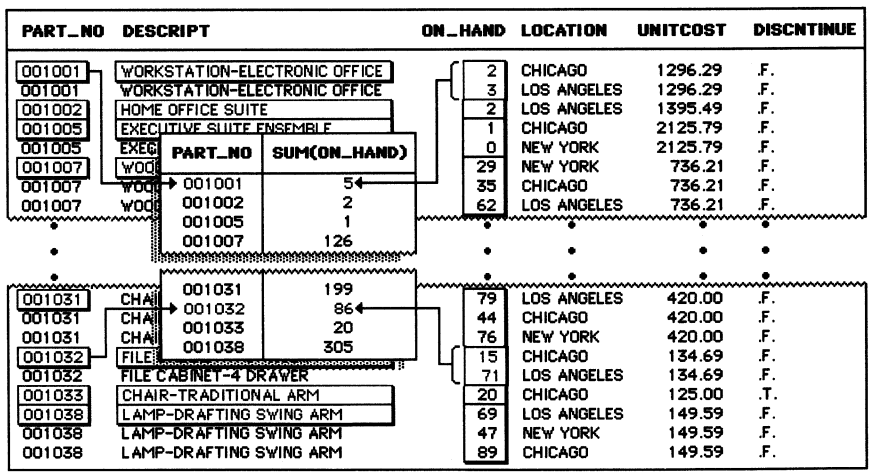


Figure 3-6 Grouping rows and the aggregate function

You could also use this method to count the number of salespeople in each office location. You would type:

```
SQL. SELECT Location, COUNT(*)
     FROM Staff
     GROUP BY Location;

G_LOCATION      COUNT1
CHICAGO           3.00
LOS ANGELES       5.00
NEW YORK          4.00
```

The GROUP BY clause can also be used in a SELECT statement that has an ORDER clause. This is useful because the GROUP BY clause does not automatically put the result table in order by the columns used to form the groups.

For example, to produce an inventory of all parts alphabetically grouped by the description of each part, you could type:

```
SQL. SELECT Part_no, Descript, SUM(On_hand)
     FROM Inventry
     GROUP BY Descript, Part_no
     ORDER BY Descript;

G_PART_NO  G_DESCRIPT                        SUM1
001009     CHAIR-ADJUSTABLE SWIVEL         124.00
001031     CHAIR-EXECUTIVE SWIVEL/TILT     199.00
001013     CHAIR-MODERN PNEUMATIC          150.00
001031     CHAIR-EXECUTIVE SWIVEL/TILT     199.00
001033     CHAIR-TRADITIONAL ARM            20.00
001015     CREDENZA-OAK SLIDING DOOR        15.00
001025     DESK-EXECUTIVE-5 FOOT           110.00
001027     DESK-EXECUTIVE-6 FOOT            76.00
001005     EXECUTIVE SUITE ENSEMBLE          1.00
001029     FILE CABINET-2 DRAWER           345.00
001032     FILE CABINET-4 DRAWER            86.00
001002     HOME OFFICE SUITE                 2.00
001024     LAMP-BRASS TABLE                196.00
001038     LAMP-DRAFTING SWING ARM         305.00
001021     MANAGERS OFFICE ENSEMBLE          3.00
001019     TABLE-BOARD ROOM                 12.00
001022     TABLE-WALNUT OCCASIONAL           5.00
001007     WOOD DESK-SINGLE PEDESTAL       126.00
001001     WORKSTATION-ELECTRONIC OFFICE     5.00
001008     WORKSTATION-STAND                22.00
```

The HAVING Clause

You can select the groups that appear in the result using the HAVING clause. This is similar to the way the WHERE clause selects rows. The HAVING clause specifies a condition that each group must satisfy before it can appear in the result table.

For example, to produce an alphabetically ordered inventory listing of all parts costing more than $500 of which you have more than 10 on hand, you could type:

```
SQL. SELECT Descript, Part_no, SUM(On_hand), Unitcost
     FROM Inventry
     WHERE Unitcost > 500
     GROUP BY Descript, Part_no, Unitcost
     HAVING SUM (On_hand) > 10
     ORDER BY Descript;

G_DESCRIPT                  G_PART_NO      SUM1      G_UNITCOST
CREDENZA-OAK SLIDING DOOR   001015        15.00        745.00
DESK-EXECUTIVE-5 FOOT       001025       110.00        985.00
DESK-EXECUTIVE-6 FOOT       001027        76.00       1475.00
TABLE-BOARD ROOM            001019        12.00       4250.00
WOOD DESK-SINGLE PEDESTAL   001007       126.00        736.21
```

NOTE
The HAVING clause is normally used with the GROUP BY clause. If you do not specify a GROUP BY clause, the HAVING clause operates on the entire result table as if it were a single group.

The UNION Clause

SQL provides one other method to combine queries, the UNION statement. It combines the result tables of two or more different SELECT statements and removes any duplicate rows. If rows from one SELECT are duplicated in another, only one will appear in the final result table. Figure 3-7 illustrates the operation of the UNION clause.

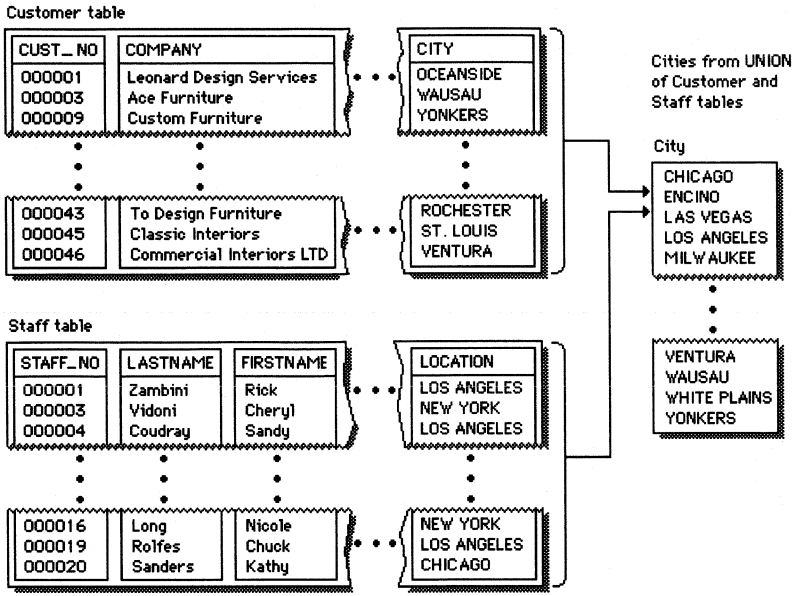


Figure 3-7 Combining queries with the UNION clause

The result tables from different queries being joined must be compatible. That is, the queries must have the same number of columns and the data types of columns in each table must match. (The column *names* in each query can be different.) For character columns, the column widths must be the same. For numeric columns, the columns must define the same total number of digits, be the same type of number (fixed or floating point), and have the same number of decimal point digits.

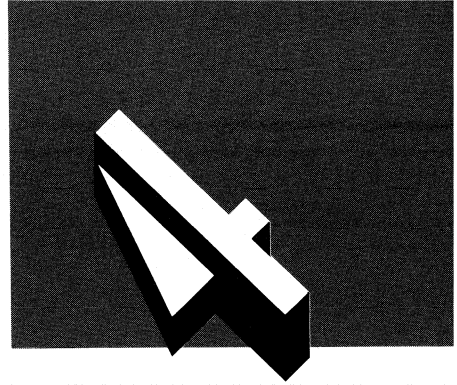
You can order the result of a UNION by placing an ORDER BY clause in the last SELECT statement of the UNION. The ORDER BY clause must use integers (corresponding to column positions) to identify the columns by which the result is ordered.

To select all the cities in which you have customers, staff, or inventory, you could type:

```
SQL. SELECT City
     FROM Customer
        UNION
     SELECT Location
     FROM Staff
        UNION
     SELECT Location
     FROM Inventry
       ORDER BY 1;
```

In this example, the cities in rows of three different tables are combined. In the Staff and Inventry tables, the city is stored in a column with a different name than that in the Customer table. However, the type and width of the field is the same, so the values can be combined.

Joins and Subqueries



You've already learned how to use the SELECT command to display data from a single table. Often, though, the information you need will be in several different tables. Most people build individual tables to contain specific types of information.

The Samples database is a good example. The Inventry table contains information about available stock items. The Sales table contains basic information about sales, most notably the order number and the entry date. Individual items that are part of a sale are entered in the Items table. The Customer and Staff tables contain information regarding customers and salespeople, respectively.

In this chapter, you'll learn to construct queries and do other database operations on more than one SQL table. You'll use the SQL SELECT command for queries that combine and select data from one or more tables.

What This Chapter Covers

The topics covered in this chapter are:

- Joins — querying data combined from more than one table.
- Simple subqueries — *nesting* one query within another. Each *inner* nested SELECT query is evaluated and its result used as part of an *outer* SELECT query.
- Correlated subquery — forming a subquery in which *inner* nested SELECT queries are evaluated once for each row in an *outer* SELECT query.

Preparing for This Chapter

The examples in this chapter also use the sample SQL tables provided with dBASE IV. (Refer to Appendix D for a listing of the tables.) To run the examples in this chapter, start SQL in the interactive mode. Activate the SQL Samples database with the START DATABASE command. Remember that the same commands used in the examples can also be embedded in dBASE IV SQL programs.

Joins

You can use a single-statement SELECT command to get information from multiple tables. This operation is commonly called a *join*. Joins are one of the features that characterize the operation of a relational database.

Joins are useful because you can specify a query to display data from multiple tables without having to specify a procedure or write a program to get the data you want. Also, any time you want to change the query, you can simply change the SELECT statement instead of writing a custom program. You can add the same clauses to a SELECT statement for a join as you would for a single table. These include the WHERE, GROUP BY, HAVING, and ORDER BY clauses.

When using the SELECT command to do a join, you specify the columns from each table you want to appear in the result. You need to prefix columns with the name of the table to which they belong if a column with the same name exists in more than one table. For example, Sales.Order_no would be the Order_no column from the Sales table. Use the asterisk wildcard (for example, Sales.*) to display all columns from a joined table.

You also may specify *join conditions* using a WHERE clause to indicate the rows from each table that you want to appear in the result.

For example, to do a join of the Sales table with the Staff table, you could type:

```
SQL. SELECT Order_no, Staff.Staff_no, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no;

SALES->ORDER_NO  STAFF->STAFF_NO  STAFF->LASTNAME
020002           000008           McLester
020003           000006           Thomas
020004           000019           Rolfes
   .                .                .
   .                .                .
   .                .                .

020023           000003           Vidoni
020024           000012           Charles
020025           000003           Vidoni
020026           000004           Coudray
```

The result table from this SELECT statement contains the column for order number from the Sales table, and the Staff_no and Lastname columns from the Staff table. The headings that appear over individual columns indicate both the column name and the table or view from which the corresponding data is taken. The join condition (WHERE clause) specifies that rows from the Sales table are joined with those rows of the Staff table where the values of Staff_no are the same.

NOTE

1. *When displaying large result tables, you can specify SET PAUSE ON to display the information one screen at a time. You can also press* **Ctrl-S** *to alternately halt or resume the scrolling of data.*
2. *You can compress the display of columns by specifying alias or synonym names in place of table names in the FROM clause. Use the SET HEADINGS OFF command if you want to remove column headings from the display.*

To show the columns that join the Sales and Staff tables to form the result table, you can display the value of the column Staff_no in each of the two tables by typing:

```
SQL. SELECT Order_no, Sales.Staff_no, Staff.Staff_no, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no;

SALES->ORDER_NO  SALES->STAFF_NO  STAFF->STAFF_NO  STAFF->LASTNAME
020002           000008           000008           McLester
020003           000006           000006           Thomas
020004           000019           000019           Rolfes
   .                .                .                .
   .                .                .                .
   .                .                .                .

020023           000003           000003           Vidoni
020024           000012           000012           Charles
020025           000003           000003           Vidoni
020026           000004           000004           Coudray
```

This example shows that in forming a join, all other rows in which the join condition is not met are excluded from the result table.

NOTE

1. *When the equals operator is specified in the join condition, the join is called an* equijoin. *If one of the two columns on which an equijoin is formed (in the example, the Staff_no column) is eliminated from the result table, the join is called a* natural *join.*
2. *The names of the columns specified in the WHERE clause (the join condition) do not have to match. However, they must be of compatible data types.*

How dBASE IV SQL Joins Tables

In a join, the tables are combined and then conditions are applied to the rows to form the result table. Figure 4-1 shows the logical steps taken to complete the SELECT statement in the example.

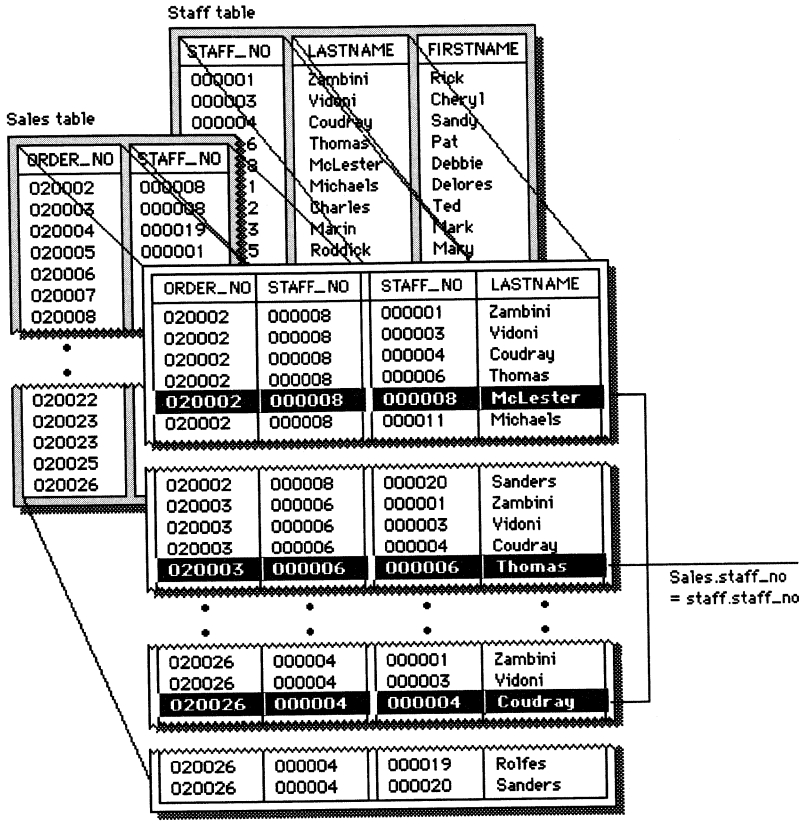


Figure 4-1 Natural join

You can see that the entire combined table could be large. dBASE IV SQL therefore uses statistical information stored in the system catalog files to build the most efficiently combined join table. Then, it reduces the table to include only the rows you see in the result table. A built-in *query optimizer* decides how dBASE IV SQL can most quickly arrive at the final result table.

Selecting Data from a Join

You can specify additional conditions other than the join condition in the WHERE clause. For example, while joining the Sales and Staff tables, you can further limit the result table to include only orders taken by the salesperson whose last name is Charles.

```
SQL. SELECT Order_no, Staff.Staff_no, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no
       AND Lastname = "Charles";

SALES->ORDER_NO  STAFF->STAFF_NO  STAFF->LASTNAME
020006           000012           Charles
020009           000012           Charles
020013           000012           Charles
020024           000012           Charles
```

NOTE
You can also use dBASE functions in the WHERE clause. For example, UPPER() can be used to match character strings whether they are entered in upper or lower case.

Here's another example of a join. This one combines information from the Inventry and Items tables. The result table contains the columns you typically might print on an order invoice form.

```
SQL. SELECT DISTINCT Items.Part_no, Qty, Unitcost, (Qty * Unitcost)
     FROM Items, Inventry
     WHERE Inventry.Part_no = Items.Part_no
       AND Order_no = "020021";

ITEMS->PART_NO  ITEMS->QTY  INVENTRY->UNITCOST      EXP1
001013                   8              275.80   2206.40
001024                   6              230.79   1384.74
001025                   8              985.00   7880.00
```

The SELECT statement includes ordered items and quantity from the Items table, and the description and cost of an ordered part number from the Inventry table. One of the columns included in the result table is a calculated column: *Qty * Unitcost*. The DISTINCT keyword eliminates duplicate rows, as shown in Figure 4-2.

PART_NO	QTY	UNITCOST	QTY*UNITCOST
001013	8	275.80	2206.40
001013	8	275.80	2206.40
001024	6	230.79	1384.74
001024	6	230.79	1384.74
001025	8	985.00	7880.00
001025	8	985.00	7880.00

Figure 4-2 The DISTINCT keyword used in a join

NOTE

1. *Columns that are derived from expressions (including calculated columns, dBASE functions, constants, and memory variables) are shown with the heading EXP followed by a number indicating the position of the expression in a SELECT clause.*
2. *Columns that display the result of SQL aggregate functions are shown with a corresponding heading AVG, COUNT, MAX, and SUM followed by a number indicating the position of the aggregate function in the SELECT clause. When you specify a dBASE function as a column in a SELECT clause, the corresponding column heading displays the function name.*
3. *Headings for columns specified in a GROUP BY clause display each column name preceded by G_.*

Greater-Than and Less-Than Join Operators

You can specify other operators other than *equals* to form a join. For example, you can specify either a greater-than join (using the > operator in the join condition) or a less-than join (using the < operator). For example, you could join the Inventry and Items tables to display the locations of part numbers for which quantities on hand are sufficient to ship the parts for a certain order.

```
SQL. SELECT Items.Part_no, Qty,
       On_hand, Location
     FROM Items, Inventry
     WHERE Items.Part_no = Inventry.Part_no AND
       On_hand > Qty AND Items.Order_no = "020021";

ITEMS->PART_NO  ITEMS->QTY  INVENTRY->ON_HAND    INVENTRY->LOCATION
001013                   8                115    CHICAGO
001013                   8                 35    LOS ANGELES
001025                   8                 63    LOS ANGELES
001025                   8                 47    NEW YORK
001024                   6                140    CHICAGO
001024                   6                 56    NEW YORK
```

The ORDER BY Clause

As in a single-table query, you can use the ORDER BY clause to assure that the rows in the result table appear in a specific order. You can arrange columns in ascending (ASC) or descending (DESC) order, or a combination of the two. For example, you could type:

```
SQL. SELECT Order_no, Sale_date, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no
     ORDER BY Sale_date DESC, Lastname;

SALES->ORDER_NO  SALES->SALE_DATE  STAFF->LASTNAME
020024           09/25/87          Charles
020026           09/25/87          Coudray
020022           09/25/87          Coudray
020021           09/25/87          McLester
020025           09/25/87          Vidoni
020023           09/25/87          Vidoni
020020           09/24/87          Marin
020019           09/24/87          Marin
020018           09/24/87          Michaels
020017           09/24/87          Thomas
020013           09/23/87          Charles
020016           09/23/87          Roddick
   .                 .                 .
   .                 .                 .
   .                 .                 .
020008           09/22/87          Vidoni
020002           09/21/87          McLester
020004           09/21/87          Rolfes
020003           09/21/87          Thomas
020005           09/21/87          Zambini
```

In this example, rows in the result table are displayed in descending order according to the date of each sale. For sales on the same date, the rows are displayed in ascending order by the last name of the salesperson. Note that if you reversed the columns specified in the ORDER BY clause (*ORDER BY Lastname, Sale_date DESC*), the rows would appear in alphabetical order by salesperson, with sales appearing from the most current to the earliest date of sale.

The GROUP BY and HAVING Clauses

You can also use the GROUP BY clause in a join. The GROUP BY clause reduces the display of grouped rows to a single row in the result table. Rows are grouped together when all the grouping column values for one row are duplicated in the corresponding columns of another row. The columns specified in the SELECT clause are either grouping columns or are the result of an SQL aggregate function. Each column in the SELECT clause must also be specified in the GROUP clause, unless the column is named in an SQL aggregate function. The total width of columns specified in a GROUP BY clause is 100 bytes.

For example, you can use the GROUP BY clause to group the sales for each salesperson.

```
SQL. SELECT Staff.Staff_no, Lastname, COUNT(*)
     FROM Staff, Sales
     WHERE Staff.Staff_no = Sales.Staff_no
     GROUP BY Staff.Staff_no, Lastname;

G_STAFF_NO  G_LASTNAME        COUNT1
000001      Zambini             2.00
000003      Vidoni              3.00
000004      Coudray             2.00
000006      Thomas              2.00
000008      McLester            3.00
000011      Michaels            2.00
000012      Charles             4.00
000013      Marin               2.00
000015      Roddick             4.00
000019      Rolfes              1.00
```

The COUNT (*) aggregate function operates on the rows for each group (the sales each salesperson made) and produces a count that is displayed next to the Staff_no and Lastname column values for each salesperson.

You can select the groups that appear in the result with the HAVING clause. You can also use the ORDER BY clause to arrange the rows in the final result table. For example, you can display the salespeople who made more than two sales in order by the number of sales each one made.

```
SQL. SELECT Staff.Staff_no, Lastname, COUNT(*)
     FROM Staff, Sales
     WHERE Staff.Staff_no = Sales.Staff_no
     GROUP BY Staff.Staff_no, Lastname
     HAVING COUNT(*) > 2
     ORDER BY 3;

G_STAFF_NO  G_LASTNAME        COUNT1
000003      Vidoni              3.00
000008      McLester            3.00
000012      Charles             4.00
000015      Roddick             4.00
```

Figure 4-3 shows the operation of the GROUP BY and HAVING clauses in the example.

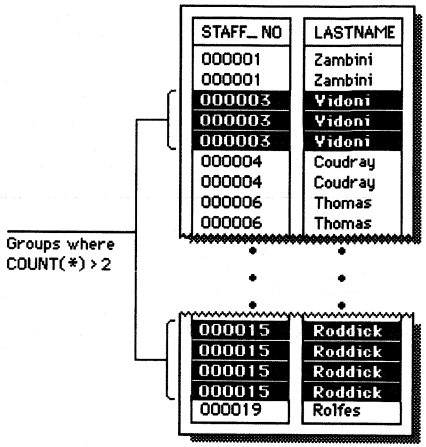


Figure 4-3 GROUP BY and HAVING clauses

Joining More than Two Tables

You can join more than two tables in a SELECT statement. With each new file you join, you'll want to specify one or more additional join conditions. The WHERE clause specifies the conditions determining which rows are included in the result table. For example, to join the Staff, Sales, and Customer tables, you could type:

```
SQL. SELECT Staff.Lastname, Order_no, Company
     FROM Staff, Sales, Customer
     WHERE Staff.Staff_no = Sales.Staff_no
       AND Sales.Cust_no = Customer.Cust_no
     ORDER BY Staff.Lastname;

STAFF->LASTNAME  SALES->ORDER_NO  CUSTOMER->COMPANY
Charles          020006           New Horizons
Charles          020009           Interior Systems
Charles          020013           New Horizons
Charles          020024           Classic Interiors
Coudray          020022           A1 Office Supply Store
Coudray          020026           Black's Furniture Store
   .                .                    .
   .                .                    .
   .                .                    .

Thomas           020017           Contemporary Designs
Vidoni           020008           The Office
Vidoni           020023           Design Center Interiors
Vidoni           020025           The Designer
Zambini          020005           American Business Supply
Zambini          020014           Leonard Design Services
```

The result table joins the Staff and Sales tables on a common staff number. The Sales and Customer tables are joined on a common customer number. In addition, the result table is ordered by the last names of salespeople in the Staff table. The result is a list of the salesperson and the corresponding customer (company) for each order in the Sales table.

Joining a Table with Itself

Using the SQL SELECT command, you can join a table to a copy of itself. This operation is called a *self-join*. Self-joins allow you to correlate information between different rows of the same table and combine this information in rows of a result table. For example, you can use a self-join to print a list of supervisors and the employees who work for them.

To do a self-join, you define different names or *aliases* for the same table. dBASE IV SQL then operates as if the aliases referred to different tables. Aliases allow a single query to operate simultaneously on different parts of the table.

For example, using the Staff table, you could list the supervisor for each employee by typing:

```
SQL. SELECT Super.Lastname,"Supervises:", Emp.Lastname
     FROM Staff Emp, Staff Super
     WHERE Super.Staff_no = Emp.Supervisor
     ORDER BY Super.Lastname DESC;

SUPER->LASTNAME  EXP1         EMP->LASTNAME
Zambini          Supervises:  Coudray
Zambini          Supervises:  McLester
Zambini          Supervises:  Marin
Zambini          Supervises:  Rolfes
Vidoni           Supervises:  Thomas
Vidoni           Supervises:  Roddick
Vidoni           Supervises:  Long
Charles          Supervises:  Michaels
Charles          Supervises:  Sanders
```

In this example, Emp and Super are aliases for the Staff table. However, they function as if they were completely different tables. Figure 4-4 shows the result table formed by the self-join and the rows that satisfy the WHERE join condition.

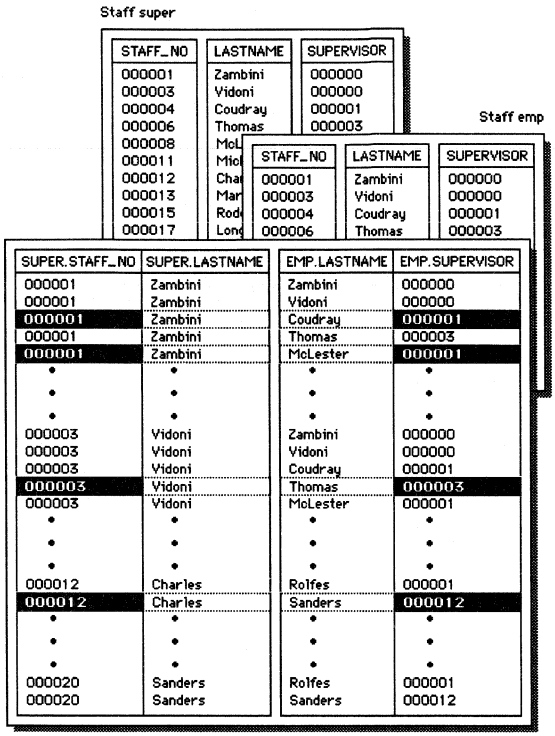


Figure 4-4 A self-join

Subqueries

Just as you can use joins to combine tables in a query, you can use subqueries to combine one query with another. In a subquery, a SELECT statement is *nested* within the WHERE clause (or in the HAVING clause) of another SELECT statement. The nested SELECT statement, called an *inner* SELECT query, supplies values for the search condition of the SELECT statement containing it. The SELECT statement containing the nested query is called an *outer* SELECT query. The outer SELECT builds the final result table based on the values the nested subquery provides.

The construction of the outer SELECT statement depends on the number of values that the SELECT subquery statement returns. There are two cases to consider:

- Subqueries that return a single value
- Subqueries that return multiple values (more than one row, with values in a single column)

There are two basic types of subqueries, *simple* and *correlated.* In a simple subquery, the inner subquery is evaluated first, independently of the outer query. The inner SELECT statement is evaluated and its result used as part of an outer SELECT query. In a correlated subquery, the values returned by the inner subquery are dependent on values supplied to it by the outer query. The inner SELECT statement is evaluated once for each row in an outer SELECT query.

This section first describes various simple subqueries and then describes correlated subqueries.

Subqueries Returning a Single Value

The simplest type of subquery, a nested SELECT, is one that returns a single value. You can use the result of the SELECT in place of a value in a WHERE clause. For example, you could enter:

```
SQL. SELECT Order_no, Sale_date
     FROM Sales
     WHERE Cust_no =
       (SELECT Cust_no
       FROM Customer
       WHERE Company = "American Business Supply")
     ORDER BY Sale_date;

ORDER_NO  SALE_DATE
020005    09/21/87
020019    09/24/87
```

This example displays the orders where *American Business Supply* is the customer. The inner query looks through the Customer table to find the customer number for that company. The outer query uses this value of customer number in the search condition *WHERE Cust_no =*.

You can also use the SQL aggregate functions in a subquery to return a single value from a SELECT statement, as shown in the following example.

```
SQL. SELECT Lastname, Salary
     FROM Staff
     WHERE Salary >
       (SELECT AVG(Salary)
       FROM Staff)
     ORDER BY Lastname;

LASTNAME        SALARY
Charles           5945
Coudray           6237
Roddick           5493
Thomas            5875
Vidoni            5780
Zambini           6000
```

This example displays the last names of people with salaries larger than the average salary computed from the Staff table. The inner query returns a single value computed by the SQL AVG() function.

You can also specify the logical operators AND, OR, and NOT in the WHERE clause to specify multiple subquery SELECT statements. For example:

```
SQL. SELECT Order_no, Sale_date
     FROM Sales
     WHERE Cust_no =
       (SELECT cust_no
       FROM Customer
       WHERE company = "American Business Supply")
         OR Staff_no =
       (SELECT Staff_no
        FROM Staff
        WHERE Lastname = "Zambini")
     ORDER BY Order_no;

     ORDER_NO  SALE_DATE
     020005    09/21/87
     020014    09/23/87
     020019    09/24/87
```

This example displays the orders where *American Business Supply* is the customer or the staff person is *Zambini.*

Subqueries Returning Multiple Values

For subqueries that return more than one row, you need to use a WHERE clause that can accept more than one value. You can use the predicate keywords IN, ANY, and ALL to create such a query. If a subquery contains IN, ANY, or ALL, it can only return a single column.

Using the IN Predicate

For example, you can follow the IN predicate with subquery values in place of the value list that is normally specified. This is shown below.

```
SQL. SELECT Order_no, Cust_no, Sale_date
     FROM sales
     WHERE Cust_no IN
       (SELECT Cust_no
        FROM Customer
        WHERE Zip LIKE "9%")
     ORDER BY Order_no;

ORDER_NO  CUST_NO  SALE_DATE
020004    000034   09/21/87
020005    000016   09/21/87
020012    000027   09/22/87
020014    000001   09/23/87
020019    000016   09/24/87
020021    000046   09/25/87
020022    000027   09/25/87
020026    000017   09/25/87
```

The subquery SELECT statement returns customer numbers from rows with zip code column values in which the first digit is 9. The customer number values are then substituted for the list of values used by the IN predicate. The WHERE clause in the outer query is evaluated as true if a column matches any of the values returned by the subquery. Figure 4-5 illustrates the operation of the subquery.

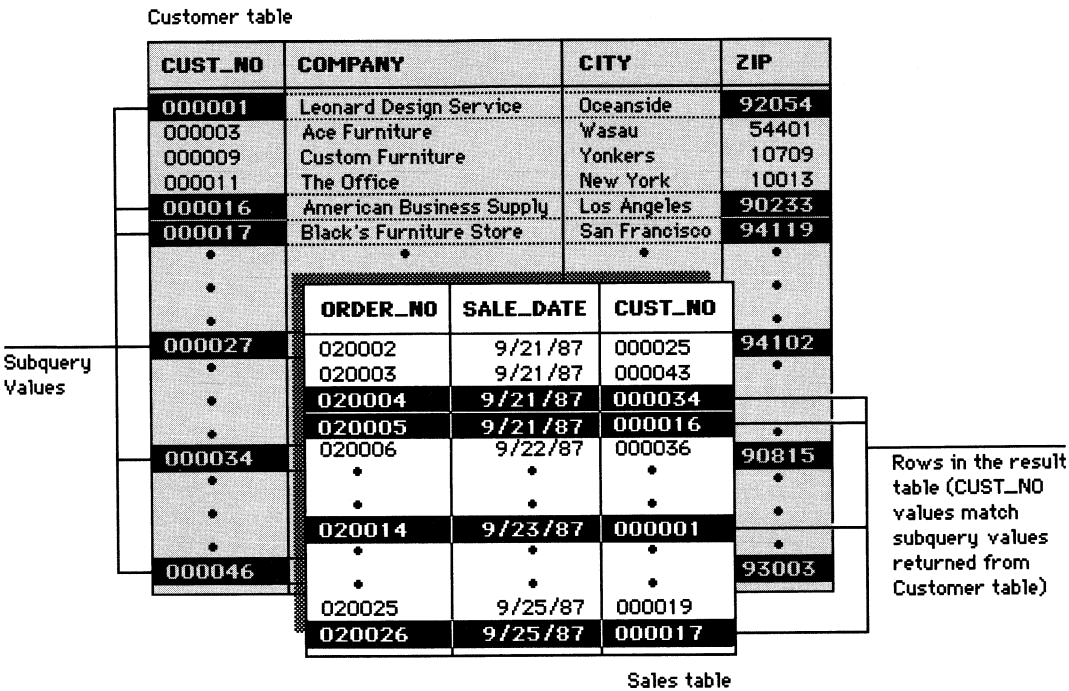


Figure 4-5 Subquery returning multiple values

NOTE
Nested SELECT queries cannot contain ORDER BY clauses or a UNION clause. Subqueries may contain a GROUP BY or HAVING clause, but there can be only one in the entire query statement, counting those in both inner and outer queries.

Using the ANY Keyword

You can use the ANY keyword to determine whether the WHERE condition is true for *any* of the values returned from a subquery. For example, you could type:

```
SQL. SELECT Staff_no, Firstname, Lastname, Hiredate, Location, Commission
     FROM Staff
     WHERE Commission  <  ANY
       (SELECT Commission
        FROM Staff
        WHERE Location = "CHICAGO")
     ORDER BY Staff_no;

STAFF_NO  FIRSTNAME  LASTNAME    HIREDATE  LOCATION     COMMISSION
000001    Rick       Zambini     02/15/80  LOS ANGELES         5.0
000003    Cheryl     Vidoni      03/06/80  NEW YORK            5.0
000004    Sandy      Coudray     06/06/80  LOS ANGELES         5.0
000006    Pat        Thomas      01/08/81  NEW YORK            5.0
000008    Debbie     McLester    04/12/81  LOS ANGELES         5.0
000012    Ted        Charles     02/02/83  CHICAGO             5.0
000019    Chuck      Rolfes      09/09/84  LOS ANGELES         6.0
000020    Kathy      Sanders     03/23/85  CHICAGO             5.0
```

This example displays the salespeople from any location who get a lower commission than any salespeople in Chicago.

Using the ANY keyword, the WHERE condition can contain any of the comparison operators, such as equals, less-than, and greater-than. If you use the equality operator (=), the WHERE clause operates like the WHERE...IN form of subquery.

Using the ALL Keyword

You can use the ALL keyword to determine whether the WHERE condition is true for all the values returned from a subquery. The following example is similar to the last one. However, it displays only those salespeople who receive a *higher* commission than anyone at the Chicago location.

```
SQL. SELECT Staff_no, Firstname, Lastname, Hiredate, Location, Commission
     FROM Staff
     WHERE Commission > ALL
       (SELECT Commission
        FROM Staff
        WHERE Location = "CHICAGO")
     ORDER BY Staff_no;

STAFF_NO  FIRSTNAME  LASTNAME    HIREDATE  LOCATION     COMMISSION
000013    Mark       Marin       06/05/83  LOS ANGELES        11.0
000015    Mary       Roddick     02/13/84  NEW YORK            8.0
```

Multiple Subqueries

You can nest subqueries within another subquery. For example, you could type:

```
SQL. SELECT Cust_no, Company, Firstname, Lastname, City
     FROM Customer
     WHERE Cust_no IN
        (SELECT Cust_no
         FROM Sales
         WHERE Order_no IN
            (SELECT Order_no
             FROM Items
             WHERE Part_no = "001025"))
     ORDER BY Cust_no;

CUST_NO  COMPANY                 FIRSTNAME   LASTNAME   CITY
000011   The Office              Dominique   LeClerc    New York
000019   The Designer            Luke        Hobbs      New York
000025   Modern Furniture Store  Robert      Hamilton   Phoenix
000040   Design Center Interiors Chuck       Gilbert    Las Vegas
000046   Commercial Interiors LTD Sandy      Young      Ventura
```

This example displays all customers who have ordered a specified part. The innermost query returns order numbers that contain the specified part number. The next outer query returns the customer numbers for each order containing the specified part. Finally, the outermost query retrieves company names related to the order numbers.

You could have expressed this same query as a join by typing:

```
SELECT Customer.Cust_no, Company,
   Firstname, Lastname, City
FROM Customer, Sales, Items
WHERE Customer.Cust_no = Sales.Cust_no
   AND Sales.Order_no = Items.Order_no
   AND Items.Part_no = "001025 "
ORDER BY 1;
```

All queries with subqueries can be expressed as joins. However, the expression of a subquery is often easier to understand than the equivalent join. Thus, you might prefer to use subqueries rather than joins as a way to write queries.

NOTE
While you can express all subqueries as an equivalent join, not all joins can be expressed as subqueries.

The EXISTS Predicate

The EXISTS predicate specifies a condition that operates on the result of a subquery as a whole, rather than on individual values returned by a subquery. When specified in a WHERE clause, it returns a true or false value depending on whether the subquery returns any values. If the nested subquery returns a value, the EXISTS search condition is true. Otherwise, the EXISTS search condition is false. NOT EXISTS reverses the logic of the search condition. If you specify NOT EXISTS, the search condition will be false if the associated subquery returns a value.

The following example illustrates the use of the EXISTS predicate.

```
SQL. SELECT Cust_no, Company, City, State
     FROM Customer
     WHERE EXISTS
       (SELECT *
       FROM Sales
       WHERE Customer.cust_no = Sales.Cust_no)
     ORDER BY Cust_no;

CUST_NO  COMPANY                  CITY           STATE
000001   Leonard Design Services  Oceanside      CA
000011   The Office               New York       NY
000016   American Business Supply Los Angeles    CA
000017   Black's Furniture Store  San Francisco  CA
000018   Interior Systems         Milwaukee      WI
000019   The Designer             New York       NY
000025   Modern Furniture Store   Phoenix        AZ
000027   Al Office Supply Store   San Francisco  CA
000031   Al's Furniture & Supplies St. Louis     MO
000032   Contemporary Designs     Milwaukee      WI
000034   La Cienega Furniture     Los Angeles    CA
000036   New Horizons             Chicago        IL
000040   Design Center Interiors  Las Vegas      NV
000043   To Design Furniture      Rochester      NY
000045   Classic Interiors        St. Louis      MO
000046   Commercial Interiors LTD Ventura        CA
```

In this example, EXISTS is true for rows in the Customer table where the customer numbers are also entered in the Sales table. Because of the WHERE EXISTS clause, the result table of the query only contains those rows where *Customer.Cust_no = Sales.Cust_no.*

You can display the opposite information, displaying those customers who do *not* have a current order, by substituting WHERE NOT EXISTS for WHERE EXISTS.

You can use more than one EXISTS or NOT EXISTS predicate in the same SQL statement, for example, to construct a subquery in which an EXISTS or NOT EXISTS query is itself nested in another EXISTS or NOT EXISTS query.

NOTE
*The EXISTS predicate is the only one in which SELECT * is allowed in a subquery.*

Correlated Subqueries

The most powerful form of the subquery is the correlated subquery. In this kind of subquery, an outer query supplies a value from each of its rows to the inner query. The inner query is evaluated once for each of these values. If both the inner and outer queries access the same table, you need to assign aliases for each table reference, since the inner and outer queries need to access different rows within the same table at the same time.

The example below shows how a correlated query works. The query displays employee information for the people at each location with the highest salary.

```
SQL. SELECT Outer.Firstname, Outer.Lastname, Outer.Location,
     Outer.Hiredate, Outer.Salary
     FROM Staff Outer
     WHERE Salary =
        (SELECT MAX(Salary)
        FROM Staff Inner
        WHERE Inner.Location = Outer.Location)
     ORDER BY Outer.Lastname;

OUTER->FIRSTNAME  OUTER->LASTNAME  OUTER->LOCATION  OUTER->HIREDATE  OUTER->SALARY
Ted               Charles         CHICAGO          02/02/83                  5945
Sandy             Coudray         LOS ANGELES      06/06/80                  6237
Pat               Thomas          NEW YORK         01/08/81                  5875
```

In this example, the outer query supplies the values of the location column for each row in the Staff table to the inner query, one at a time. Using these values, the inner query determines the largest salary within the group of employee salaries at that location. The outer query then compares the value of the salary column within each row of the Staff table with the maximum salary for the same group. If the salary matches the maximum, the row is selected for the result table. Figure 4-6 shows how the final result is obtained.

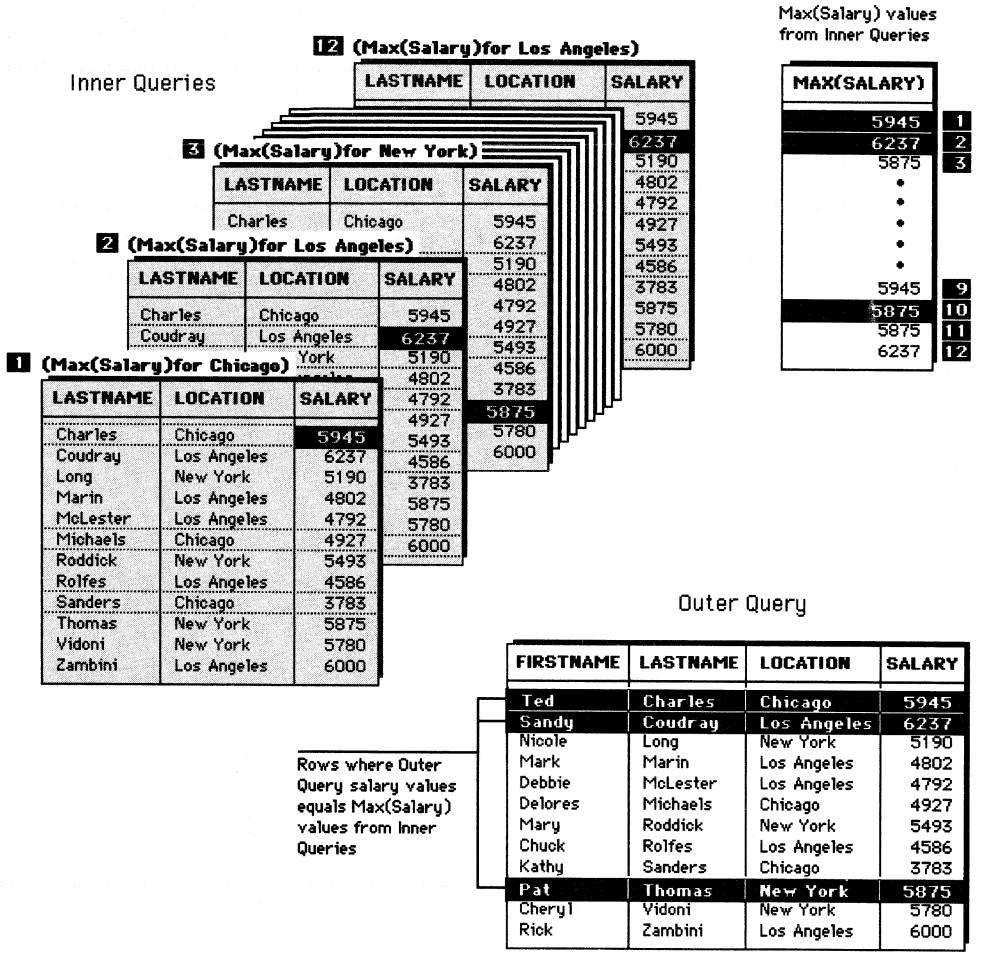


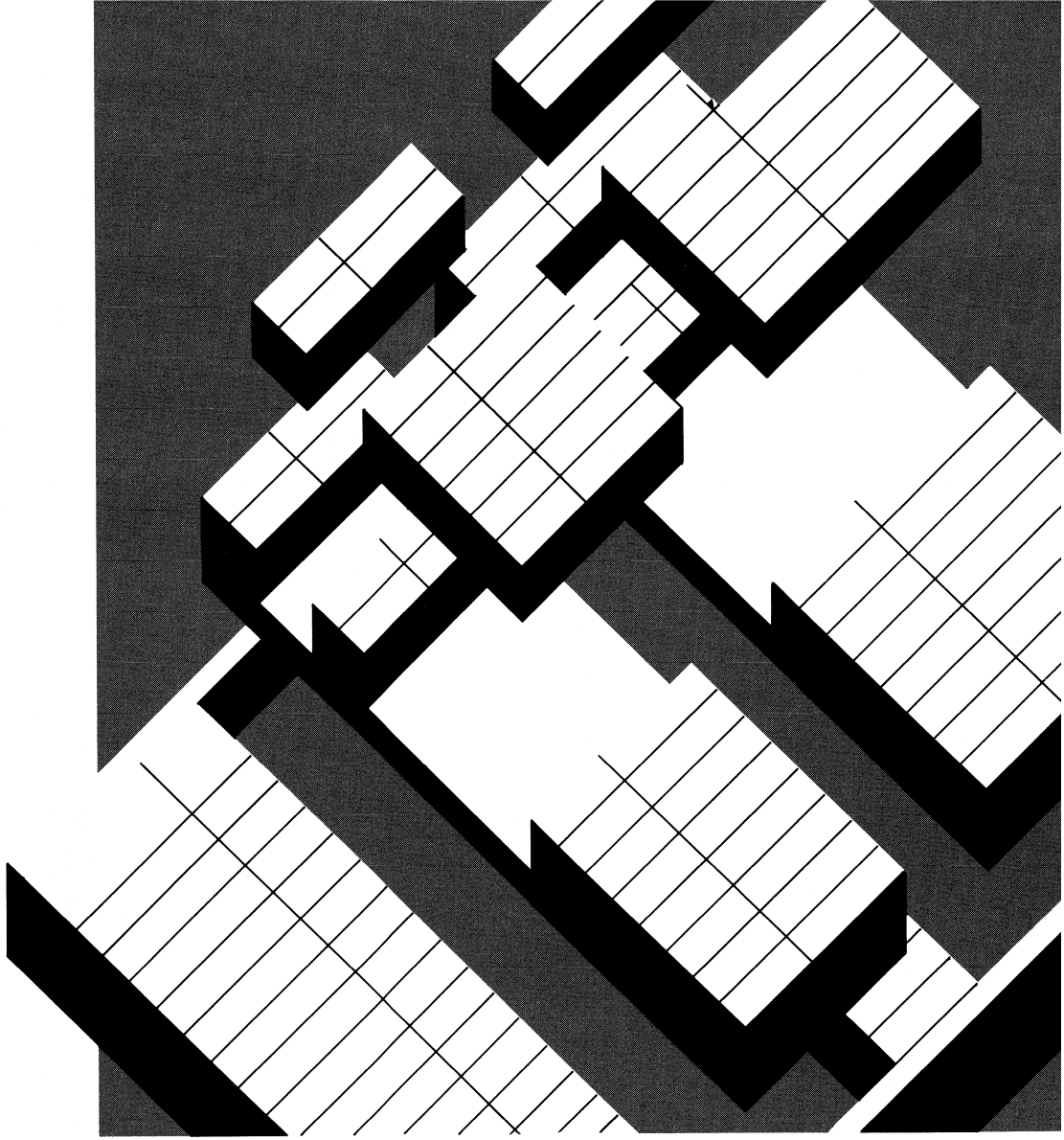
Figure 4-6 A correlated subquery example

Here's another example. To display all orders for customers who have more than one current order, you could type:

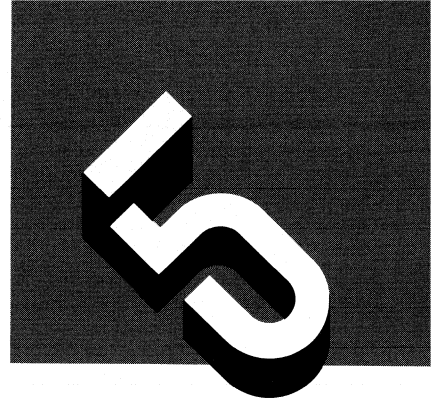
```
SQL. SELECT S1.Cust_no, S1.Order_no
     FROM Sales S1
     WHERE S1.Cust_no IN
       (SELECT S2.Cust_no
       FROM Sales S2
       WHERE S1.Order_no <> S2.Order_no)
     ORDER BY S1.Cust_no, S1.Order_no;

S1->CUST_NO    S1->ORDER_NO
000011         020008
000011         020016
000016         020005
000016         020019
000019         020007
000019         020015
000019         020025
000027         020012
000027         020022
000031         020010
000031         020020
000036         020006
000036         020013
000036         020018
000040         020011
000040         020023
```

The outer part of this query passes customer values for each order to the inner query, through the reference of alias table S1. The inner query checks alias table S2 to see if it contains any other orders with the same customer value.

SQL and dBASE



Combining SQL and dBASE



In earlier chapters, you learned how to use SQL commands to define SQL tables and views, and how to insert, retrieve, and update data. In this chapter, you'll see how to use dBASE commands in the interactive SQL mode.

This chapter also provides a bridge to using SQL and dBASE commands to design database application programs. In the interactive environment, you can design your SQL database and then test the operations you plan to incorporate in your application. Almost all the dBASE and SQL commands used in the interactive environment can also be used in an application. dBASE IV also provides other tools (a text editor, a debugger, and a compiler) that you can use interactively to help build your application.

What This Chapter Covers

The topics covered in this chapter are:

- Using dBASE commands and functions in SQL mode
- Saving queries; creating reports and labels
- Accessing dBASE database files
- Importing and exporting data
- Securing SQL data
- Using SQL on a Local Area Network (LAN)

Preparing for This Chapter

In this chapter, you'll continue to execute commands in the interactive SQL mode. Before starting this chapter, make sure that the SQL prompt appears on your screen, and that you've activated the SQL Sample database using the START DATABASE command.

Using dBASE Commands and Functions in SQL Mode

dBASE IV allows you to combine SQL with other dBASE commands and functions to perform various database tasks. Using only SQL commands, you can define or delete a table and its indexes, and insert, update, delete, or query (access) data. In addition, you can set privileges that restrict access to the data. With the set of functions SQL provides, you can do simple statistical operations (SUM(), COUNT(), AVG(), MIN(), and MAX()).

dBASE IV includes a full set of dBASE commands and functions that you can use with SQL. Interactively, for example, you can use dBASE commands to create and print reports from data selected by SQL queries. Other commands and functions allow you to convert or format data in SQL queries, set options, and develop programs.

dBASE IV has separate SQL and dBASE modes because of differences in the way the two systems access data (set versus record-pointer orientation). You can, however, effectively use both dBASE and SQL commands to operate on the same data, using SQL commands to retrieve data, and dBASE commands to process and format it.

The complete sets of dBASE commands and functions you can use in the SQL mode are listed in Table 5-1 and Table 5-2, respectively.

NOTE
Refer to Appendix C for a brief description of all dBASE commands and functions.

Table 5-1 dBASE commands allowed in SQL mode

!	ERASE
?, ??, or ???	EXIT
&&	FUNCTION
*	HELP
@...CLEAR...	IF...ELSE...ENDIF
@...DOUBLE/PANEL/NONE...	INPUT
@...FILL TO...	LIST FILES
@...SAY...GET	LIST HISTORY
ACCEPT	LIST MEMORY
ACTIVATE MENU	LIST STATUS
ACTIVATE POPUP	LIST USERS
ACTIVATE SCREEN	LOAD
ACTIVATE WINDOW	LOGOUT
BEGIN...END TRANSACTION	LOOP
CALL	MODIFY COMMAND/FILE
CANCEL	MOVE GETS
CLEAR GETS	MOVE WINDOW
CLEAR MENUS	NOTE
CLEAR POPUPS	ON ERROR/ESCAPE/KEY
CLEAR TYPEAHEAD	ON PAD
CLEAR WINDOWS	ON PAGE
CLOSE ALTERNATE	ON READERROR
CLOSE FORMAT	ON SELECTION PAD
CLOSE PROCEDURE	ON SELECTION POPUP
COMPILE	PARAMETERS
COPY FILE	PLAY MACRO
DEACTIVATE MENU	PRINTJOB...ENDPRINTJOB
DEACTIVATE POPUP	PRIVATE
DEACTIVATE WINDOW	PROCEDURE
DEBUG	PROTECT
DECLARE	PUBLIC
DEFINE BAR	QUIT
DEFINE BOX	READ [SAVE]
DEFINE MENU	RELEASE
DEFINE PAD	RELEASE MENUS
DEFINE POPUP	RELEASE MODULE
DEFINE WINDOW	RELEASE POPUPS
DELETE FILE	RELEASE WINDOWS
DIR/DIRECTORY	RENAME
DISPLAY FILES	RESTORE FROM
DISPLAY HISTORY	RESTORE MACROS
DISPLAY MEMORY	RESTORE MENU
DISPLAY STATUS	RESTORE WINDOW
DISPLAY USERS	RESUME
DO	RETRY
DO CASE...CASE... OTHERWISE...ENDCASE	RETURN
DO WHILE	RUN
EJECT	SAVE MACROS
	SAVE TO

(*continued*)

Table 5-1 dBASE commands allowed in SQL mode (*continued*)

SAVE WINDOW
SET
SET ALTERNATE ON/OFF
SET ALTERNATE TO
SET AUTOSAVE ON/OFF
SET BELL TO
SET BELL ON/OFF
SET BORDER
SET CENTURY ON/OFF
SET CLOCK ON/OFF
SET CLOCK TO
SET COLOR OF
SET COLOR ON/OFF
SET COLOR TO
SET CONFIRM ON/OFF
SET CONSOLE ON/OFF
SET CURRENCY LEFT/RIGHT
SET CURRENCY TO
SET DATE
SET DEBUG ON/OFF
SET DECIMALS TO
SET DEFAULT TO
SET DELETED ON/OFF
SET DELIMITERS ON/OFF
SET DELIMITERS TO
SET DEVELOPMENT ON/OFF
SET DEVICE TO
SET DISPLAY TO
SET DOHISTORY ON/OFF
SET ECHO ON/OFF
SET ENCRYPTION ON/OFF
SET ESCAPE ON/OFF
SET EXACT ON/OFF
SET EXCLUSIVE ON/OFF
SET FIXED ON/OFF
SET FORMAT TO
SET FUNCTION
SET HEADING ON/OFF
SET HELP ON/OFF
SET HISTORY ON/OFF
SET HISTORY TO
SET HOURS TO
SET INTENSITY ON/OFF
SET LOCK ON/OFF
SET MARGIN TO
SET MARK TO
SET MENUS ON/OFF
SET MESSAGE TO
SET ODOMETER TO
SET PATH TO
SET PAUSE ON/OFF
SET POINT TO
SET PRECISION TO
SET PRINTER ON/OFF
SET PRINTER TO
SET PROCEDURE TO
SET REFRESH TO
SET REPROCESS TO
SET SAFETY ON/OFF
SET SCOREBOARD ON/OFF
SET SEPARATOR TO
SET SPACE ON/OFF
SET SQL ON/OFF
SET STATUS ON/OFF
SET STEP ON/OFF
SET TALK ON/OFF
SET TRAP ON/OFF
SET TYPEAHEAD TO
SHOW MENU
SHOW POPUP
STORE
SUSPEND
TEXT...ENDTEXT
TYPE
WAIT

NOTE
dBASE commands and functions that open a database file (USE), position a database file record pointer (RECNO(), GOTO, and SKIP) and seek or display data (FIND, SEEK, and DISPLAY) are not in the set of commands allowed in SQL mode. SQL commands are set-oriented and handle the operations to open tables and find data in them automatically.

Table 5-2 dBASE functions allowed in SQL mode

&*	FLOOR()	PROGRAM()*
ABS()	GETENV()*	PROMPT()*
ACOS()	IIF()*	PROW()*
ASC()	INKEY()*	RAND()
ASIN()	INT()	READKEY()*
AT()	ISALPHA()*	REPLICATE()
ATAN()	ISCOLOR()*	RIGHT()
ATN2()	ISLOWER()*	ROLLBACK()*
BAR()*	ISUPPER()*	ROUND()
CALL()*	LASTKEY()*	ROW()*
CDOW()	LEFT()	RTOD()
CEILING()	LEN()	RTRIM()
CHR()	LIKE()*	SET()*
CMONTH()	LINENO()*	SIGN()
COL()*	LOG()	SIN()
COMPLETED()*	LOG10()	SOUNDEX()
COS()	LOWER()	SPACE()
CTOD()	LTRIM()	SQRT()
DATE()	MAX()*	STR()
DAY()	MDY()	STUFF()
DIFFERENCE()	MEMORY()*	SUBSTR()
DISKSPACE()*	MENU()*	SUM()
DMY()	MESSAGE()*	TAN()
DOW()	MIN()*	TIME()
DTOC()	MOD()	TRANSFORM()
DTOR()	MONTH()	TRIM()
DTOS()	NETWORK()*	TYPE()*
ERROR()*	OS()*	UPPER()
EXP()	PAD()*	USER()
FILE()*	PCOL()*	VAL()
FIXED()	PI()	VARREAD()*
FKLABEL()*	POPUP()*	VERSION()*
FKMAX()*	PRINTSTATUS()*	YEAR()
FLOAT()		

NOTE
dBASE functions that are allowed in SQL mode but not in an SQL statement are noted with an asterisk.

Entering dBASE Commands

When you use one of the dBASE commands listed in Table 5-1 at the SQL dot prompt, enter the entire command on one line *without* a semicolon following the command. Then press ↵ to execute the command. For example, say you wanted to change the way dates appear when you enter an SQL SELECT statement. To display dates in the French date format, you would type:

```
SQL. SET DATE TO FRENCH
```

The next time you displayed date information from a table, it would appear in the new format.

If you want, you can specify additional SET options. For example, you could route SQL statements and the results they produce to the printer by typing:

```
SQL. SET PRINTER ON
```

Now everything you type will echo to the printer after you press ↵ to execute a command. To stop echoing to the printer, you would SET PRINTER OFF.

You can capture the SQL commands you type and their results in a text file by using the SET ALTERNATE TO and SET ALTERNATE ON commands.

```
SQL. SET ALTERNATE TO Session
SQL. SET ALTERNATE ON
```

The SET ALTERNATE TO command in this example opens a text file named Session.txt, and the SET ALTERNATE ON command starts recording keyboard entries and screen displays. You can stop recording in the file by entering SET ALTERNATE OFF and then close the file with CLOSE ALTERNATE. You can then use the TYPE command to display the contents of the Session.txt file, or use the command MODIFY COMMAND to edit it.

Using dBASE Functions

dBASE functions listed in Table 5-2 can be used with SQL statements (as you've seen in the examples in previous chapters). You can use dBASE functions in the SELECT clause to transform or combine column data before it appears in a result, or within a WHERE clause to evaluate conditions.

Some examples in Chapter 3 used functions to define condition expressions in the WHERE clause. Here is an additional example that shows functions used in both the SELECT and WHERE clause.

```
SQL. SELECT RTRIM(Firstname) + SPACE(1) + Lastname
     FROM Staff
     WHERE MONTH(Hiredate) = 6;
```

This statement removes trailing blanks following the first name and then concatenates the first names and last names of staff members hired in the month of June. The SPACE() function adds a single space between each first and last name displayed.

The SAVE TO TEMP Clause

dBASE IV provides a SAVE TO TEMP clause with the SQL SELECT command to allow you to save the result of SQL queries. The SAVE TO TEMP clause follows the last clause of a SELECT query. The syntax is:

SAVE TO TEMP <table name> [(column list)] [KEEP];

The SAVE TO TEMP clause is useful in two ways. First, it lets you break up complex SELECT statements into smaller, more manageable operations. By dividing an operation such as a complicated join or subquery, you can do part of the query and check the intermediate table to see if you're getting the correct results. Then, when you do the rest of the query on a smaller table, you can be more confident that the final result will be the one you want.

The second use of the SAVE TO TEMP clause is to capture the results from an SQL query to a .dbf database file. You can then use the database file to produce reports and labels with dBASE IV's report and label design screens, or process data in the file with other dBASE IV commands. To save the temporary table to a .dbf database file, you need to specify the KEEP clause.

Unless you save them, tables created with the SAVE TO TEMP clause are temporary and are only available while the current database remains active, during a current session at the SQL dot prompt, or during execution of the highest level .prs program file. To save a query to use in later sessions, you can create a view with the CREATE VIEW command, and use the SELECT statement to define rows and columns in the view. Or, if you've specified the KEEP option to create a dBASE database file, you can use DBDEFINE to define an SQL table using the database file.

The example below shows how you can use the SAVE TO TEMP clause. Expanding on an example from Chapter 4, you could create a single query statement that generates the detail for orders you want to appear on invoice forms. You generate the necessary information by joining the Customer, Sales, Items, and Inventry tables. You can save the final result table and use it to print a report. Because the invoice requires joining four different tables, you might want to break the join into two more manageable and easily understood query statements.

Your first step would be to join information from two tables, Items and Inventry, to generate the detail lines for each ordered item. For example, you would type:

```
SQL. SELECT DISTINCT Order_no, Items.Part_no, Descript,
       Qty, Unitcost, (Qty * Unitcost)
     FROM Items, Inventry
     WHERE Inventry.Part_no = Items.Part_no
     ORDER BY Items.Order_no
     SAVE TO TEMP Detail (Order_no, Part_no, Descript,
       Qty, Unitcost, Total);
```

This SELECT statement returns a table with all the detail lines needed to generate invoices for all order numbers. The SAVE TO TEMP clause saves the result table as a temporary SQL table named Detail. All the columns have the same names as in the original table except the calculated column, which is renamed Total. The ORDER BY clause arranges the table according to order number, which helps you see if the query you've constructed is correct so far. To check, you could type:

```
SQL. SELECT *
     FROM Detail;
```

NOTE
Saving information to a database file (with the KEEP option) creates fields based on the widths of columns in the original tables (see the SELECT command in Chapter 7 for more information).

Next, you can finish building the table needed to generate invoices by joining the Detail table with the Sales and Customer tables.

```
SQL. SELECT DISTINCT Lastname, Sale_date, Detail.Order_no, Part_no, Descript,
       Qty, Unitcost, Total
     FROM Customer, Sales, Detail
     WHERE Sales.Cust_no = Customer.Cust_no
     AND Detail.Order_no = Sales.Order_no
     ORDER BY Detail.Order_no
     SAVE TO TEMP Invoice (Lastname, Inv_Date, Order_no, Part_no,
       Descript, Qty, Unitcost, Total) KEEP;
```

This query finishes the join operation needed to collect all the information to generate invoices. It then saves the result table as a database file named Invoice (because of the KEEP keyword). You can again check the table by typing:

```
SQL. SELECT *
     FROM Invoice;
```

The only step remaining is to create and print the invoice. You can use dBASE IV's report generator, but first you need to switch back to the dBASE dot prompt to open (USE) the database file you've created. You also must specify the layout of the report, adding sales tax, an invoice price total, and whatever other detail you want. When you use the report generator, you reference the columns as *fields* to be included in the report. You can group the report by order number to print each invoice on a separate page. (Refer to Chapter 9 of *Using the Menu System* for details on how to create and print reports.)

From the last example, you can see how to use SQL commands to retrieve data, and dBASE commands to format and process it. The effectiveness of this approach is more evident when you use SQL and dBASE together in an application program, as you'll see in the next chapter.

Accessing dBASE .dbf and .mdx Files

dBASE IV data is maintained in .dbf database and .mdx index files that both SQL and more traditional commands can access. However, when you create tables and indexes using SQL commands, additional information about the tables and indexes is stored in SQL catalogs. SQL uses the catalog information to access data from SQL tables. Thus, to use SQL commands with a .dbf database file created outside of SQL mode, you need to update SQL catalog tables with the DBDEFINE command to include information about the database file and any associated indexes you want to use with it. Once a database file has been defined as an SQL table, you cannot modify its structure using dBASE commands.

Conversely, in dBASE mode, you can use dBASE commands to access most SQL tables without doing any advance conversion or preparation, since SQL tables are already stored as .dbf database files.

You can add, update, or delete data in these database files using dBASE commands, with only a few restrictions.

1. If you've installed PROTECT, you cannot access an SQL-created database file in dBASE mode. (You can use the SET ENCRYPTION and COPY commands to create unencrypted copies of the files.)
2. You cannot use the MODIFY STRUCTURE command on SQL-created database files. Also, you cannot use the CONVERT TO command with any SQL-created database file.
3. When using the CREATE and CREATE FROM commands, you cannot create a database file with the same name as an existing SQL-created database file.
4. For .dbf files defined as SQL tables (using CREATE TABLE or DBDEFINE), you cannot add or delete index tags from the .mdx file that has the same name as the database file.
5. Updates are not allowed on an SQL-created database file for which a unique SQL index has been defined (with the CREATE INDEX command). Also, no updates are allowed on SQL catalog table .dbf files.

The DBDEFINE Utility Command

The DBDEFINE command creates catalog entries for .dbf database files and their associated .mdx indexes. The catalog information allows you to access these files with SQL commands. The syntax for the DBDEFINE command is:

DBDEFINE [< .dbf filename >];

DBDEFINE updates the catalog tables of the active database for a single database file or for all database files in the current database directory.

To move all the database files for a dBASE application into a new SQL database, you need to do the following:

1. Create an SQL database for the application using the CREATE DATABASE command. (If you want to add the database files to an existing SQL database, you can skip this step.)
2. Copy the database, memo, and index files into the database directory to which you want the files transferred. (You can use the COPY FILE command in SQL mode.)
3. Activate the database using the START DATABASE command.
4. Run the DBDEFINE command.

   All the database files copied into the database directory will be entered in the database catalog tables. (To create catalog entries only for particular database files in the database, specify the individual database filenames with the DBDEFINE command.)

The DBDEFINE command can only define tables for .dbf files that are unencrypted. To unencrypt a dBASE-encrypted file before executing DBDEFINE, first SET ENCRYPTION OFF and then use the COPY TO command to create an unencrypted version of the file. (You must have sufficient PROTECT privileges to access the original file.)

NOTE

1. *You cannot update an SQL catalog to include definitions of dBASE views. However, you can create an SQL definition of the database files represented in a view, and then create an SQL view equivalent to the query that created the dBASE view file.*
2. *You cannot update an SQL catalog for .ndx index files. You need to convert any .ndx files you want to use in SQL mode to .mdx tags and then perform the DBDEFINE operation.*
3. *After updating the catalogs, DBDEFINE leaves the text file Dbdefine.txt in the same directory as the current database. This file records the table definitions used to define SQL tables for the dBASE database files added to the current database. DBDEFINE also lists the database files for which SQL tables were created and those which it could not successfully define.*
4. *If .mdx files are available when a .dbf file is DBDEFINEd, the catalog tables are updated to reflect the statistics of the most recent REINDEX operation. To ensure that the statistics reflect the actual state of the current .dbf file, you should REINDEX a file before entering SQL mode to DBDEFINE it, or execute the RUNSTATS command afterward.*

To see how the DBDEFINE command works, add the Vendors sample database file to the current SQL database. To do this, type:

```
SQL. DBDEFINE Vendors;
```

To see some of the information that SQL records for SQL tables, you can look at the definitions made in two SQL catalog tables.

```
SQL. SELECT *
     FROM Systabls;
```

The Systabls catalog table contains one row for each SQL tables in the same database. To display the columns that the DBDEFINE cocmmand has entered for the Vendors table, you can type:

```
SQL. SELECT Colno, Colname, Coltype, Collen
     FROM Syscols
     WHERE Tbname = "Vendors";
```

This command verifies that column names and column type definitions entered for the Vendors table are the same as those defined in the database file structure (displayed with the DISPLAY STRUCTURE command).

```
Structure for database:  VENDORS.DBF
Number of data records:        8
Date of last update    :  05/25/88
Field   Field Name   Type       Width   Dec   Index
    1   VENDOR_ID    Character      4             Y
    2   VENDOR       Character     30             N
    3   ADDRESS1_V   Character     30             N
    4   ADDRESS2_V   Character     30             N
    5   CITY_V       Character     20             N
    6   STATE_V      Character      2             N
    7   ZIP_V        Character     10             N
    8   PHONE_VEND   Character     13             N
    9   CONTACT_V    Character     30             N
   10   PHONE_EXT    Character      4             N
   11   TERMS_V      Character     10             N
   12   DISCOUNT_V   Numeric        2             N
** Total **                      186
```

The DBCHECK Utility Command

The DBCHECK command verifies the SQL catalog entries for database files and their associated indexes. The syntax for the DBCHECK command is:

DBCHECK [< table name >];

The DBCHECK command checks the current definitions of SQL tables and indexes against the actual structure of dBASE database files in the current SQL database directory. DBCHECK returns error messages for every database file that does not match its catalog definition.

NOTE
When DBCHECK displays errors, you need to follow the steps below:

1. *Copy the database files and indexes mentioned in DBCHECK error messages to another directory (besides the current SQL database directory). If you do not make copies of the files, they will be lost when you DROP the corresponding tables.*
2. *DROP the tables from the SQL database.*
3. *Copy the database files and indexes (with names < tablename.mdx >) back into the database directory.*
4. *Run the command DBDEFINE < filename > to redefine the dBASE database files as SQL tables.*

Importing and Exporting Data

dBASE IV SQL provides two commands, LOAD DATA and UNLOAD DATA, that allow you to transfer data between SQL databases and external non-SQL data files. These commands support transfer of the same file types as supported by the dBASE APPEND FROM and COPY TO commands. These file types are:

- dBASE II, dBASE III, dBASE III PLUS, and dBASE IV database files
- RapidFile database files (RPD)
- Framework II database and spreadsheet files (FW2)
- Delimited format ASCII files (DELIMITED)
- System data format ASCII files (SDF)
- VisiCalc format files (DIF)
- Multiplan spreadsheet format files (SYLK)
- Lotus 1-2-3 format files (WKS)

The LOAD and UNLOAD commands can be run in both interactive and embedded SQL modes. Data is imported to or exported from SQL tables in the current database. The external file may be in the current user's directory, or in a different directory specified by a path preceding the filename.

The LOAD Utility Command

The LOAD command appends data from an external file to the end of an existing SQL table in the current database. The syntax of the LOAD command is:

```
LOAD DATA FROM [path] < filename > INTO TABLE < table name >
    [[TYPE] SDF/DIF/WKS/SYLK/FW2/RPD/DBASEII/
    DELIMITED [WITH BLANK/WITH < delimiter > ]];
```

If you do not specify a file type with the LOAD command, dBASE IV assumes import from a dBASE database file. For dBASE files, data is imported only for those fields also defined as columns in the specified SQL table. (Memo fields are not copied.) To copy dBASE II files, you need to specify the DBASEII file type.

If you're importing data from spreadsheets or character-delimited files, the SQL table structure into which you're loading data will match the format of the file from which you're importing. Spreadsheet data will be stored in *row major* order as opposed to *column* order.

The SQL table must contain as many columns as are expected from the file being imported. Column definitions should correspond to the length and type of data expected from the imported file. The LOAD command will truncate data that does not fit into columns when it is imported.

NOTE
The LOAD command operates like the APPEND FROM command in dBASE mode. Refer to the APPEND FROM command in the Language Reference *manual for additional guidelines on importing data.*

The UNLOAD Utility

The UNLOAD command copies data from an SQL table to a non-SQL format file. The syntax for this command is:

```
UNLOAD DATA TO [path] < filename > FROM TABLE  < table name >
    [[TYPE] SDF/DIF/WKS/SYLK/FW2/RPD/DBASEII/
    DELIMITED [WITH BLANK/WITH  < delimiter > ]];
```

As with the LOAD command, if you do not specify a file type with the LOAD command, dBASE IV assumes export to a .dbf database file. (Use the DBASEII file type to copy data to dBASE II files.) Also, for .dbf files, fields in the new database file are created with names and definitions corresponding to columns in the SQL table being exported. If you're exporting data to spreadsheets, column names are written as column headers in the resulting file.

If tables are secured with PROTECT passwords, you need to be the creator of a table or have SELECT privileges to UNLOAD it. Data is unencrypted automatically and copied to a file in the current user's directory (unless a path is specified).

NOTE
Refer to the dBASE COPY TO command in the Language Reference *manual for additional guidelines on copying data to* foreign format *files.*

SQL Security and Authorization

Security in the context of databases is the way a database system protects data from unauthorized access, modification, or destruction. dBASE IV provides security through a password-protected log-in system, assignment of access privileges to users, and encryption of data files.

Database protection in dBASE IV is set up by a system administrator using the dBASE PROTECT command. When a protection scheme is in place, all users must first *log in* with a password to gain access to dBASE IV. After users successfully log in to dBASE IV, the operations they can perform are dictated by the privileges assigned to them. In dBASE mode, file privileges are specified with the PROTECT command. In SQL mode, privileges are assigned using the SQL GRANT and REVOKE commands. (See the *Networking with dBASE IV* manual for a description of the PROTECT command.) After PROTECT is installed, database files defined as SQL tables may only be accessed using SQL commands.

User Authorization

Before you can use the GRANT and REVOKE commands in SQL mode, you must use PROTECT to create group names, user IDs (log-in names), and passwords. One of the user IDs that you should create is SQLDBA. This user ID has administration privileges to perform any SQL operation on any table or view. Otherwise, when tables are created, exclusive privileges for operations on tables and views are restricted to their creator. After a table or view is created, only its creator or the SQLDBA user ID can assign or restrict user privileges, or *authorization*, for operations on the table or view.

NOTE

1. *The SQLDBA user ID has full privileges, including DROP, on all tables, views, indexes, and synonyms in each database.*
2. *User ID information created by PROTECT for use with SQL commands is stored in the file Dbsystem.sql. This file is created in the same directory in which its dBASE equivalent, Dbsystem.db, is stored.*

All privileges assigned by GRANT and REVOKE are recorded in the catalog tables for the currently active database.

Data Encryption

An additional facet of security that dBASE IV provides is data encryption. Data encryption protects against unauthorized access by storing the data on disk in an encoded fashion, so only authorized users with a decryption key can read it. Even if unauthorized people get a copy of the data, they still will not be able to read or interpret it without a key. In SQL mode, tables are encrypted when you create them. Thereafter, only authorized users can access the tables.

NOTE

Data in tables created in SQL mode cannot be accessed in dBASE mode unless you UNLOAD data from the table (or use the SAVE TO TEMP clause in a SELECT statement). Similarly, you cannot use the DBDEFINE or LOAD command with database files encrypted in dBASE mode. To use dBASE-encrypted files, SET ENCRYPTION OFF and make a copy (with the COPY FILE command). Then, use that copy for DBDEFINE or LOAD operations. To create an unencrypted copy of a dBASE-encrypted file, you must have read-access privileges to the file (assigned by PROTECT).

The GRANT Command

The GRANT command grants privileges to users for specified tables or views. The syntax for this command is:

```
GRANT ALL [PRIVILEGES]/ < privilege list >
ON [TABLE] < table list >
TO PUBLIC/ < user list >
[WITH GRANT OPTION];
```

GRANT allows you to specify privileges that correspond to ALTER, CREATE INDEX, DELETE, INSERT, SELECT, and UPDATE operations on tables. GRANT privileges are cumulative; that is, you can expand existing privileges by adding privileges to those a user already has. Also, if users are granted GRANT privileges (the WITH GRANT OPTION), they can pass on those same privileges, or a subset of them, to other users. Conversely, if a user's privileges are revoked, they are also revoked for any users who received them from that user.

You can grant all privileges to a user with the ALL option, or specify one or more privileges separated by commas. Available privileges are:

- ALTER — ability to add columns to a table
- DELETE — ability to delete rows from a table or view
- INDEX — ability to use the CREATE INDEX command
- INSERT — ability to add rows to a table or view
- SELECT — ability to display rows from a table or view
- UPDATE [(column list)] — ability to update all or only specified columns of rows in a table or view

You can specify that the GRANT statement applies to all users (PUBLIC), or specify a list of user IDs created by PROTECT. For example, to assign all privileges to a user ID Janice on the Staff table, you could type:

```
SQL. GRANT ALL ON TABLE Staff
     TO Janice;
```

Rather than assigning all privileges, you could specify individual privileges. For example:

```
SQL. GRANT INSERT, UPDATE, DELETE ON Staff
     TO PUBLIC;
```

The REVOKE Command

The REVOKE command revokes access privileges previously granted on specified tables or views. The syntax of this command is:

```
REVOKE ALL [PRIVILEGES]/ < privilege list >
ON [TABLE] < table list >
FROM PUBLIC/ < user list > ;
```

You can revoke all privileges from specified users with the ALL option, or specify one or more privileges separated by commas. Privileges you can specify are the same as those listed for the GRANT command.

For example, if you had assigned full privileges for the Staff table to three users (John, Janice, and Greg), you could revoke INSERT and UPDATE privileges of one of the users by typing the following command:

```
SQL. REVOKE UPDATE, INSERT ON Staff
     FROM John;
```

NOTE
REVOKE from PUBLIC only affects privileges previously assigned by a GRANT TO PUBLIC command since users may also have privileges granted to them individually.

dBASE IV SQL on a Local Area Network (LAN)

On a local area network, where users share access to files, there are often conflicts between users when they attempt to change the same information at the same time. If users were able to change data simultaneously, the results could be disastrous. dBASE IV provides three different methods of resolving these conflicts: exclusive use of files, locking of files (for the duration of an operation), and locking of records.

When it is crucial that a user control an entire file in dBASE mode (for example, during a multiple record update operation), all other users can be prevented from updating the same file. In contrast, if many users need to have access to the same file, users can share a file and lock individual records only when they need to update them.

While these concepts are important for SQL users to understand, SQL does not require commands to lock file or records. File and record locking for SQL commands is automatic. When a user executes an SQL command to insert, update, or delete data in an SQL table or view, dBASE IV determines the kind of lock needed.

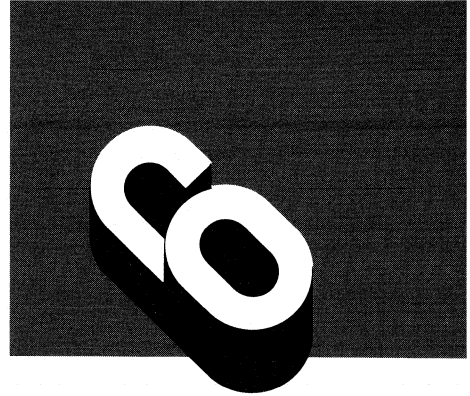
Occasionally, dBASE IV is unable to complete a lock operation because a resource is already locked. Another user may be executing SQL commands at another machine, or dBASE commands have locked records or a database file. In cases where data is not immediately available, dBASE IV continues to try to lock for a period of time you can specify with the SET REPROCESS command.

NOTE

1. *You may want to check the status setting for SET EXCLUSIVE. To maximize concurrent access of data, SET EXCLUSIVE should be OFF (the default setting).*

2. *You may want to use the BEGIN and END TRANSACTION commands to ensure that update operations are performed successfully. If an operation is not completed, you will be able to restore tables to their original state using the ROLLBACK command. The next chapter illustrates how transaction processing can be used with SQL statements embedded in an application program.*

Embedding SQL Commands



In this chapter, you'll learn how to create dBASE IV SQL programs. You use the same commands in SQL programs as you can execute interactively, plus some commands allowed exclusively in programs.

Embedded SQL refers to the commands that allow SQL to be used with a host programming language. In mainframes and minicomputer environments, SQL is often used to write database applications with languages such as COBOL, FORTRAN, or C. In dBASE IV, dBASE commands provide the tools to build the application.

In SQL programs, the dBASE language provides commands and functions needed to do such tasks as building menus and data entry forms, and printing reports. It provides constructs such as DO WHILE, DO CASE, IF...ELSE, ON ERROR, and program memory variables to control the processing and flow within an application. dBASE also provides commands to process keyboard input and set up the environment, for example, to customize and set options for the display of data.

What This Chapter Covers

The topics covered in this chapter are:

- Embedding SQL commands
- Creating and running SQL programs
- Embedding SQL data definition and SELECT statements
- Using SQL cursor commands
- Embedding SQL INSERT, UPDATE, and DELETE commands
- Multi-user programming and transaction processing
- SQL application programming

Preparing for This Chapter

This chapter assumes that you've already started dBASE IV and have the dot prompt displayed on your screen. You'll use the dBASE IV program file editor, MODIFY COMMAND, to create SQL program files in this chapter.

Embedding SQL Commands

dBASE IV provides two different methods of accessing data, each with different advantages. The first method is based on traditional dBASE commands such as USE, SELECT, and FIND. Using these commands, you *open* database files, select *work areas*, and move record pointers within each open file to *access* individual records.

SQL commands in dBASE IV provide a second method involving a set-oriented approach. Instead of opening files and manipulating record pointers to find data, you specify the set of data you want to retrieve from a table, and the system determines how to find the information. A similar set-oriented approach is used for inserting, updating, or deleting data.

You can choose either the dBASE or SQL approach to accessing data, or you can use both. However, because certain commands in dBASE and SQL have the same name, and because of the differences in the ways these two techniques access data, you need to clearly identify program files that contain SQL commands.

Programs in which SQL commands are used must have a .prs (program source) file extension. Program files that use dBASE commands to access data normally have a .prg file extension. You can use both types of program files — dBASE .prg and embedded SQL .prs files — in the same application. A dBASE .prg file can call an embedded SQL .prs file (with the DO command) and, similarly, a .prs file can call a .prg file. This allows you to pick the best method for your particular programming application.

You cannot include SQL commands in program files in which you use dBASE commands to open and access database files. The reverse is also true. In programs that contain SQL commands, you cannot also use dBASE commands and functions that reference an open file or work area (for example, USE), or relate to use of the record pointer (for example, the RECNO() and EOF() functions). Appendix C provides a quick summary of dBASE IV commands and functions, and indicates whether each command or function can be used in SQL mode.

There is one other difference between the two types of files. In a .prg file, the semicolon character is used only as a *continuation* character. The semicolon indicates that a dBASE command continues on the next line. In a .prs (SQL) file, all SQL statements must end with a semicolon; for allowed dBASE commands, a semicolon may be used to indicate the continuation of a dBASE command on more than one line. SQL and dBASE statements can both contain up to 1,024 characters.

The following sections describe how to use SQL commands in programs to develop applications. You'll also see how dBASE and SQL programs can use and share common dBASE IV resources. With few exceptions, SQL can use all the tools and resources that dBASE IV provides.

dBASE Memory Variables

You can specify dBASE memory variables in SQL expressions in WHERE clause search conditions and with SQL predicates such as IN or LIKE. They can also be used in commands to transfer data to columns in tables with the INSERT command. When used with the FETCH command, memory variables receive data values from table columns. Values from dBASE memory variables can also pass data between dBASE and SQL programs.

dBASE Functions

dBASE functions can also be used in SQL expressions, in the SELECT, WHERE, and HAVING clauses. Most dBASE functions, in contrast to SQL functions, operate on a single value at a time. Appendix C describes the dBASE functions that can be used with SQL statements.

You can use user-defined functions (UDFs) in SQL mode; however, you cannot include a user-defined function within an SQL statement. Also, UDFs cannot include SQL statements. UDFs used in SQL mode can only contain dBASE commands and functions allowed in SQL mode.

SQL Status and Error Handling

When executing SQL commands from a program file, you can specify error handling using the ON ERROR command. (Refer to Appendix A for a list of the errors trapped by the ON ERROR command in SQL mode.) You can create specific error handling routines depending on the different error numbers that an SQL program returns.

SQL provides two system memory variables, *Sqlcnt* and *Sqlcode* that indicate the status of SQL operations. *Sqlcode* holds a value that indicates the outcome of the last SQL operation. It takes on one of the three different values listed in Table 6-1.

Table 6-1 Sqlcode values

Value	Description
0	*Success* code: indicates successful completion of an SQL command
−1	*Error* code: indicates that an execution error occurred or that the current user did not have sufficient privileges to execute the command
+100	*Warning* code: indicates that SELECT in an INSERT statement returned no rows, UPDATE or DELETE statement did not update or delete any rows, or FETCH was attempted past the last row of a result table

You can use the value of *Sqlcode* to check whether an SQL operation was completed successfully. For example, using the FETCH command, you can check whether the last operation returned any rows you need to process.

The *Sqlcnt* variable holds a value indicating the number of rows affected by the last SQL operation. For example, if an UPDATE operation's WHERE clause limited updated rows to six, *Sqlcnt* would hold a value of six following completion of the command. You may want to use the values returned by *Sqlcnt*, for example, to determine when you need to run the RUNSTATS command to update catalog table statistics after updates to a table.

dBASE IV Work Areas

A total of ten work areas are available in dBASE IV. When dBASE IV switches to SQL mode, any work areas in use remain open. To minimize the possibility of running out of work areas, you should close any unnecessary open files before switching to SQL mode. The following guidelines describe SQL's use of work areas for execution of SQL commands.

- One work area is required for each table referenced in a SELECT query, subquery, or self-join.
- An additional work area is required for each open cursor (specified by the OPEN < cursor name > command). Work areas used by open cursors remain open during subsequent execution of program files that switch to dBASE mode. Work areas are released when the cursors are released.
- An additional work area is required for each GROUP BY or ORDER BY clause.
- An additional work area is required for each use of a SAVE TO TEMP clause.
- When a transaction has been defined (with the BEGIN...END TRANSACTION commands), additional work areas are required for each system catalog table opened.

NOTE
dBASE IV automatically closes any .dbf database files in use that reference the same files as those accessed by an SQL command.

Creating and Running SQL Programs

A dBASE IV program is a sequence of SQL and dBASE statements in one or more ASCII text files. These files are separate from the database files or tables maintained by dBASE IV, which are referenced within the programs.

Creating SQL Programs

To create program files in dBASE IV, you can use the dBASE MODIFY COMMAND text editor or any other editor that produces ASCII text files. (See Chapter 11 in *Using the Menu System* for a description of the MODIFY COMMAND editor.) If you're creating an embedded SQL program, remember to label the file with a .prs file extension.

NOTE
You can substitute a different editor for the built-in MODIFY COMMAND editor by using the TEDIT configuration command in your Config.db file.

Depending on how you structure your dBASE IV application, you'll need to create one or more program files that contain dBASE and SQL statements. You can use the same techniques to design .prs program files containing SQL commands as you use to create traditional .prg program files. SQL programs are structured the same, use nearly the same commands and functions, and define and use memory variables in the same way. (See Appendix C for a list of commands and functions you can use with SQL commands.) You also can use similar methods to define error handling and set up transaction processing, and to operate on local area networks.

NOTE
You may want to refer to Programming with dBASE IV *to learn how to create a database application, and how to use the many commands provided by dBASE IV for designing programs.*

Compiling and Executing Programs

Once you've created the program files for your application, you can run your application by typing DO <filename>. The first time you run an application, the DO command first compiles the program files in the application before beginning to execute it. dBASE IV then loads the specified main program into memory and begins executing its commands. During the execution of an application program, dBASE program files can call SQL program files and vice versa. dBASE IV automatically switches modes depending on the extension of the file it encounters. When dBASE IV switches from dBASE mode to SQL (or SQL to dBASE mode) to execute a called program or return, it restores the environment (an active SQL database, work areas, and open files) previously set in that mode.

SQL program files can also contain a set of SQL-defined procedures within the same file, as can new .prg program files. Thus, when a DO command is executed and a file extension is not specified, dBASE IV searches first for a procedure with the specified name within the currently executing program file, then searches within any other already active program files and, finally, searches for a .prg or .prs file with the specifed name.

There are additional dBASE commands you can use after you've created your application program. If you encounter errors in executing your application, you can use the test and debugging environment provided by the SET TRAP and DEBUG commands. You can use the COMPILE command to compile dBASE programs into dBASE *object* code format without executing the resulting code (unlike the DO command). To link program object modules into a single file, use DBLINK, the DOS-level linker. The RunTime program lets you package a run-time version of dBASE IV with your application program object files. Finally, the BUILD DOS-level command combines operation of the compiler, linker, and RunTime program. It allows you to create an executable version of your application, complete with RunTime program modules, and copy the files to a floppy disk.

Embedding Data Definition Statements

The largest group of SQL statements that can be embedded in an SQL program is the group of data definition statements. These statements define or drop tables, views, synonyms, and indexes. In a program, database objects are normally defined first. You can then query or update them. You may want to define tables and views, for example, as part of an installation or maintenance procedure, or perhaps to generate reports. Table 6-2 lists the data definition statements.

Table 6-2 Data definition statements

Command	Description
ALTER TABLE	Add columns to a table
CREATE INDEX	Create an index based on the columns in a table
CREATE SYNONYM	Create an alternate name for a table or view
CREATE TABLE	Create a base table
CREATE VIEW	Create a view based on one or more tables or views
DBDEFINE	Creates an SQL table from an existing .dbf database file
DROP INDEX	Drop an index
DROP SYNONYM	Drop a synonym
DROP TABLE	Drop a table and any views or synonyms based on that table
DROP VIEW	Drop a view

The syntax of these commands in an embedded SQL program is the same as when you execute the commands interactively. Related to the data definition statements are the two authorization commands, GRANT and REVOKE, that define privileges for each of the defined data objects. After you create data objects, you may want to specify privileges to users and have your program check those privileges when users perform an operation. (This assumes that you've defined user IDs with PROTECT.)

NOTE
There are a few restrictions on using SQL data definition commands in programs. Refer to the "Developing SQL Applications" section later in this chapter.

Embedding SELECT Statements

The most powerful embedded SQL statement is the SELECT statement. When embedded in SQL programs, the SELECT statement can do the following:

- Display data from a table or view
- Transfer selected column values from a single row to dBASE memory variables
- Transfer selected column values from selected rows to dBASE memory variables, a row at a time, using SQL cursor commands
- Update selected column values or delete selected rows using the SQL cursor commands

Embedding SELECTs to Display Data

You can use a SELECT statement in an SQL program to display data on the screen, just as you would if you executed the SELECT command interactively. The following code example shows how to display different queries of the Staff table:

```
DO CASE
   *
   CASE mchoice = 1
      *
      WAIT "Press any key to display the employee with highest salary"
      *
      SELECT Lastname, Salary
      FROM Staff
      WHERE Salary =
         (SELECT MAX(Salary)
         FROM Staff);
   CASE mchoice = 2
      .
      .
      .
ENDCASE
```

You can use dBASE memory variables in the SELECT statement anywhere that you can use a constant (for example, in the WHERE clause). Using a dBASE memory variable in the WHERE clause, you could specify a query matching rows in a table against the value in the memory variable (*Mlocation = "NY"*), as in the following example:

```
SELECT *
FROM Customer
WHERE State = mlocation
ORDER BY Company;
```

Transferring a Single Row

For single-row SELECTs, the SELECT statement has an additional INTO clause that you can use to transfer a single row of column values into corresponding dBASE memory variables. The SELECT statement should only select one row. If more than one row is selected, only the first row's values are stored.

For example, to retrieve client information from the Customer table (passing those values to a dBASE program), you could execute the following program code segment:

```
ACCEPT "Enter customer's last name: " TO mcustomer
SELECT Firstname, Lastname, Address, City, State, Zip
INTO mfname, mlname, maddr, mcity, mstate, mzip
FROM Customer
WHERE UPPER(Lastname) = UPPER(Mcustomer);
*
DO Subprg1 WITH mfname, mlname, maddr, mcity, mstate, mzip
```

In this example, the ACCEPT statement stores the entry of a customer's last name in a memory variable named *mcustomer*. The value of this memory variable is used inside the SELECT statement's WHERE clause to find a customer with the same name. That information can then be passed to a subprogram using values stored in memory variables.

NOTE
If a SELECT...INTO statement does not return any rows, memory variables named in the INTO clause are not created. You may want to initialize the values of these variables or check the value of Sqlcnt before processing the memory variables named in the INTO clause.

Returning Multiple Rows

The most widely used embedded SELECT statement is one that returns a set of rows to a database program through memory variables. SQL uses a *cursor* to return column values from selected rows, one row at a time. The SQL cursor is similar to a dBASE record pointer. However, the SQL cursor only points to rows in the SELECT statement's result table. Also, the SQL cursor can only be advanced in a forward direction, one row at a time.

Column values of the current row are placed in memory variables that correspond to columns named in the SELECT clause by a FETCH statement. The four cursor command statements are listed below.

- DECLARE < cursor name > CURSOR defines a cursor. Associated with each cursor is the embedded SELECT statement that returns multiple rows. The SELECT statement defines the rows in a result table that the cursor will step through.
- OPEN < cursor name > executes the SELECT statement associated with the cursor to create a result table. It sets the cursor to the position before the first row of the result table (or at the end of file if the result table is empty).
- FETCH < cursor name > INTO < memvar list > advances the cursor to the next selected row, then transfers column values from the row to corresponding program memory variables.
- CLOSE < cursor name > closes the cursor and releases the work area and memory it uses. You may reopen a cursor after it has been closed. If you reopen the cursor, its associated SELECT statement is executed again to produce a new result table.

NOTE
As with the SELECT...INTO statement, if a SELECT statement defined by DECLARE CURSOR does not return any rows, memory variables specified by a corresponding FETCH command are not created. You may want to initialize their values or check the value of Sqlcnt in your program before processing the memory variables named in the FETCH command.

For example, you can use SQL cursors and an embedded SELECT statement to identify rows of orders in the Sales table that have not been invoiced. You can pass the order information to an SQL subprogram that joins the row with information from the Item and Inventry tables to generate all the information for an invoice.

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no
FROM Sales
WHERE NOT Invoiced;
   .
   .
   .
OPEN Inv;
   .
   .
   .
DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no;
   IF SQLCODE = 0
            .
            .
            .
      DO Invoice WITH morder_no, msale_date, mstaff_no, mcust_no
            .
            .
            .
   ELSE
     EXIT
   ENDIF
ENDDO
CLOSE Inv;
```

You can also delete or update information using cursors. For example, you could use the UPDATE...WHERE CURRENT OF command to change the value of the logical Invoiced column to true, or use the DELETE...WHERE CURRENT OF command to delete an entire row after invoicing it. When updating rows using a cursor, you must specify a FOR UPDATE OF clause in the DECLARE CURSOR's query statement.

Embedding UPDATE Statements

The SQL UPDATE statement changes values of columns in selected rows of a table or view. There are two ways you can embed the UPDATE statement. You can use the same UPDATE command as allowed at the SQL dot prompt, or you can use UPDATE with SQL cursor commands.

For example, to update the Commission column for all rows of the Staff table where the salesperson is located in New York, you could type:

```
ACCEPT "Enter percent commission increase" TO mcomm
UPDATE Staff
SET Commission = Commission + mcomm
WHERE State = "NY";
```

NOTE
You can use dBASE memory variables with the UPDATE statement anywhere that you can use a constant, for example, in the WHERE clause. Using a dBASE memory variable in the WHERE clause, you could update rows in a table where a particular column value matches the value of the memory variable.

To use UPDATE with SQL cursors, you use a form of the command UPDATE...WHERE CURRENT OF, which allows you to update the row pointed to by the cursor. The DECLARE CURSOR's query statement must include a FOR UPDATE OF clause to specify updatable columns. For example, you can modify the code segment used previously to generate invoices for uninvoiced orders:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced
FOR UPDATE OF Invoiced;
   .
   .
   .
OPEN Inv;
DO WHILE .T.
    FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
    IF SQLCODE = 0
       * Print an invoice for order number in the current row
       * If successful, change Minvoiced memory variable to .T.
       *
       DO Invoice WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
       *
       IF minvoiced
          UPDATE Sales
          SET Invoiced = .T.
          WHERE CURRENT OF Inv;
             *
       ENDIF
    ELSE
      EXIT
   ENDIF
ENDDO
CLOSE Inv;
```

In this example, the Invoice program generates an invoice for the order number in the row pointed to by the cursor. After completing the invoice, the Invoiced column in the current row is updated to true. *WHERE CURRENT OF Inv* selects the current row of the specified cursor's result table.

Embedding DELETE Statements

The SQL DELETE statement deletes selected rows of a table or view. As with the SQL UPDATE statement, there are two ways you can embed the DELETE statement. You can use the same DELETE commands as allowed at the SQL dot prompt, or you can use DELETE with an SQL cursor.

For example, to delete all rows in which the order has been invoiced and was placed before 9/22/87, you could type:

```
DELETE
FROM Sales
WHERE Invoiced AND Sale_date < {09/22/87};
```

NOTE
You can use dBASE memory variables in the DELETE statement anywhere that you can use a constant. For example, using a dBASE memory variable in the WHERE clause, you could delete rows in a table in which specified columns match the value in the memory variable.

To use the DELETE statement with SQL cursors, you use a form of the command DELETE...WHERE CURRENT OF, which will delete the row pointed to by the cursor. For example:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced;
   .
   .
   .
OPEN Inv;
   .
DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
      * Print an invoice for order number in the current row
      * If successful, change Minvoiced memory variable to .T.
   *
      DO Invoices WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
      *
         IF minvoiced .AND. Sale_date < {09/22/87}
              DELETE FROM Sales
              WHERE CURRENT OF Inv;
      ENDIF
   ELSE
     EXIT
   ENDIF
```

Rows deleted with the DELETE...WHERE CURRENT OF command are not removed from the corresponding .dbf file until the associated cursor is closed. If SET DELETED is OFF, subsequent SQL queries include deleted rows (preceded by an asterisk) in result table displays. IF SET DELETED is ON, deleted rows do not appear in result table displays and SQL operations do not affect deleted rows.

Embedding INSERT Statements

You can also embed the INSERT statement in programs. For example, the following program code segment shows how you could create a dBASE data entry form and insert validated new rows to the Inventry table.

```
DO WHILE .T.
   *
   @ 3,20 SAY "Add items to Inventry table"
   @ 5,8 SAY "Enter 5-digit part number" GET mpart_no
   @ 6,8 SAY "Enter a part description" GET mdescript
   @ 7,8 SAY "Enter the quantity on hand" GET mon_hand
   @ 8,8 SAY "Enter the stock location" GET mlocation
   @ 9,8 SAY "Enter the part's unit cost" GET munitcost
   @10,8 SAY "Enter 0 in the part number field to exit"
   *
   READ
   *
   IF mpart_no = 0
     EXIT
   ENDIF
   SELECT COUNT(*)
   INTO l_match
   FROM Inventry
   WHERE Location = mlocation AND Part_no = mpart_no;
   *
   *   Add new row if location is new for existing part
   *
   IF l_match = 0
      INSERT INTO Inventry
      (Part_no, Descript, On_hand, Location, Unitcost)
      VALUES (mpart_no, mdescript, mon_hand, mlocation, munitcost);
      *
      * Part number and location are not unique
   ENDIF
ENDDO
```

In this example, new rows are added in two different situations: when the part number you've entered is new, or when you enter a new stock location for an existing part. Otherwise, the program loops back to display the entry form without entering a row.

Multi-User and Transaction Programming

The embedded SQL programs you create will automatically run on workstations operating dBASE IV on a local area network (LAN). When more than one person is running dBASE IV, corresponding .dbf database files are automatically locked during SQL operations that insert, update, or delete data. You can specify a value with the SET REPROCESS command so that dBASE IV will retry an operation automatically if a database table is locked because someone else is inserting, updating, or deleting data.

Occasionally, dBASE IV is unable to complete a lock operation because a resource is already locked. Another user may be executing SQL commands at another machine, or other users may have locked records or a database file. To prevent situations in which you cannot complete an SQL statement, you may want to use the BEGIN and END TRANSACTION commands. You can then make sure that an operation is completed or, if it is not completed, restore a table to its original state. The following example shows the setup of a transaction.

```
ON ERROR DO Recover
SET REPROCESS TO 15
BEGIN TRANSACTION
    UPDATE Staff
    SET Commission = Commission + mcomm
    WHERE State = "NY";
END TRANSACTION
ON ERROR
IF COMPLETED()
   @ 21,15 SAY "Transaction successfully completed"
ENDIF
```

NOTE

1. *SQL commands not allowed in transactions are CREATE, DROP, ALTER, GRANT, DBCHECK, DBDEFINE, and REVOKE.*
2. *Rows deleted in a transaction with the DELETE command are not removed from the corresponding .dbf file until the END TRANSACTION command is executed. Rows deleted with the DELETE...WHERE CURRENT OF command are not removed from the SQL table until the end of a transaction in which the associated cursor is also closed.*

The ON ERROR command is normally used to handle situations where the SQL transaction produces an error. For example, you might set up the following error recovery procedure.

```
PROCEDURE Recover
   @21,15 SAY "Your transaction has encountered an error condition"
   @22,15 SAY "Do you want to RETRY? (Y/N)" GET choice PICTURE "!"
   READ
   @ 21,15 TO 22,65 CLEAR
   IF choice = "Y"
     RETRY
   ELSE
     @21,15 SAY "Rolling back your transaction. Please wait."
     ROLLBACK;
   ENDIF
RETURN
```

In this example, dBASE IV attempts to perform the SQL UPDATE command specified in the transaction. If successful, it displays the message **Transaction successfully completed**. However, if an error occurs, the Recover procedure is called. The Recover procedure allows the user two options: one to retry and proceed with the update, and a second option to discontinue the operation and roll back any updates made to the table before the error occurred.

Developing SQL Applications

dBASE IV makes certain decisions at compile time about references to database objects in an SQL program file, mostly to improve performance and maintain security of SQL files. As a result, you need to make an extra effort to modularize SQL program files and observe a few special rules. This section describes the rules you need to follow when building an SQL application.

Reference to Database Objects Before Creation

Unlike traditional .prg programs, SQL requires that all database objects already exist at compile time or are defined before being referenced by another SQL statement. In defining an SQL program, a statement that defines or creates a database object must therefore precede any statements such as DELETE, DECLARE CURSOR, GRANT, INSERT, REVOKE, SELECT, or UPDATE that reference the corresponding object.

SQL commands that define SQL objects are DBDEFINE, CREATE DATABASE, CREATE INDEX, CREATE SYNONYM, CREATE TABLE, CREATE VIEW, and SELECT...SAVE TO TEMP. To make sure these commands are executed before their corresponding objects are referenced, place them at the beginning of a program file that executes SQL commands (or contains SQL procedures), or define a separate initialization program file executed before any other SQL program files.

Repeated Definition of SQL Objects with the Same Name

The dBASE IV compiler does not allow you to repeat the definition of an SQL object. Thus, you cannot specify more than one SQL statement in a .prs file that creates the same database object (unless the object is dropped), even if at run time only one statement is executed. For example, compilation of the following statements will generate an error:

```
IF <condition>
     CREATE TABLE Newtable (Col1 char(5));
ELSE
     CREATE TABLE Newtable (Col1 char(10));
ENDIF
```

Most applications do not require you to create tables or other SQL objects at run time. However, you might encounter a similar situation when using the SELECT...SAVE TO TEMP command. For example, you might want to specify two different queries that generate an SQL temporary table:

```
IF <condition>
     SELECT Lastname, Firstname, Hiredate
     FROM Staff
     WHERE Salary > 4000 and Salary <= 5000
     SAVE TO TEMP Service KEEP;
ELSE
     SELECT Lastname, Firstname, Hiredate
     FROM Staff
     WHERE Salary > 5000 AND Salary <= 6000
     SAVE TO TEMP Service KEEP;
ENDIF
```

When compiling these statements, dBASE IV generates an error since both SQL statements specify creation of a temporary table with the same name.

The program in which the statements above are executed can be structured in another way. If, for example, the purpose of the code above is to generate a report using the .dbf file generated by the SAVE TO TEMP clause, you can change the code so that the program can successfully compile.

```
IF <condition>
     SELECT Lastname, Firstname, Hiredate
     FROM Staff
     WHERE Salary > 4000 and Salary <= 5000
     SAVE TO TEMP Service KEEP;
     *.
     DO REPORT1
     DROP TABLE Service;
     *
ELSE
     SELECT Lastname, Firstname, Hiredate
     FROM Staff
     WHERE Salary > 5000 AND Salary <= 6000
     SAVE TO TEMP Service KEEP;
     *
     DO REPORT1
     DROP TABLE Service;
     *
ENDIF
```

Another way to restructure the original query is shown below. In this example, conditions for the original query are stored in memory variables that are substituted into the SQL query statement:

```
IF <condition>
     mlower = 4000
     mupper = 5000
*
ELSE
     mlower = 5000
     mupper = 6000
     *
ENDIF
     SELECT Lastname, Firstname, Hiredate
     FROM Staff
     WHERE Salary > mlower AND Salary <= mupper
     SAVE TO TEMP Service KEEP;
     *
     DO REPORT1
     DROP TABLE Service;
```

Specifying the Current Database

The START DATABASE command specifies a database that subsequent SQL statements identify as the location of referenced database objects such as tables and views. It remains the current (active) database until it is closed with STOP DATABASE, or until a new database is specified with another START DATABASE command or a CREATE DATABASE command. In a program file, dBASE IV treats the last database specified in an SQL CREATE or START DATABASE statement as the active database for SQL commands that follow. For example, consider the following SQL statements.

```
IF <condition>
     START DATABASE Mktg;
ELSE
     START DATABASE Sales;
ENDIF
UPDATE Staff SET Salary = 8000;
```

Because the dBASE IV compiler cannot evaluate the IF condition, the UPDATE command will always operate on the Staff table in the Sales database.

You can restructure the SQL statements above so the correct Staff table will be updated, depending on the condition specified by the IF command. You could replace the SQL statements with the following:

```
IF <condition>
     START DATABASE Mktg;
     *
     UPDATE Staff SET Salary = 8000;
     *
ELSE
     *
     START DATABASE Sales;
     *
     UPDATE Staff SET Salary = 8000;
     *
ENDIF
```

SQL Optimization

The last aspect to consider when compiling an application is SQL optimization. SQL optimization determines the best method to use when performing SQL queries, based on the size and structure of data in SQL tables.

The methods chosen to perform queries are selected based on the data in the SQL catalog tables at the time the compiler is run. If data in an application changes drastically from when it was compiled, the methods chosen to perform queries may no longer be optimal. You should recompile applications periodically to allow dBASE IV to re-evaluate and optimize queries to match current data.

In performing optimization, dBASE IV also selects from current indexes available when an application is compiled. If you add an index to speed up queries, for example, you should also recompile your application. You should also recompile your application if you drop an index, since optimization may have used the dropped index in performing queries.

Creating RunTime Applications

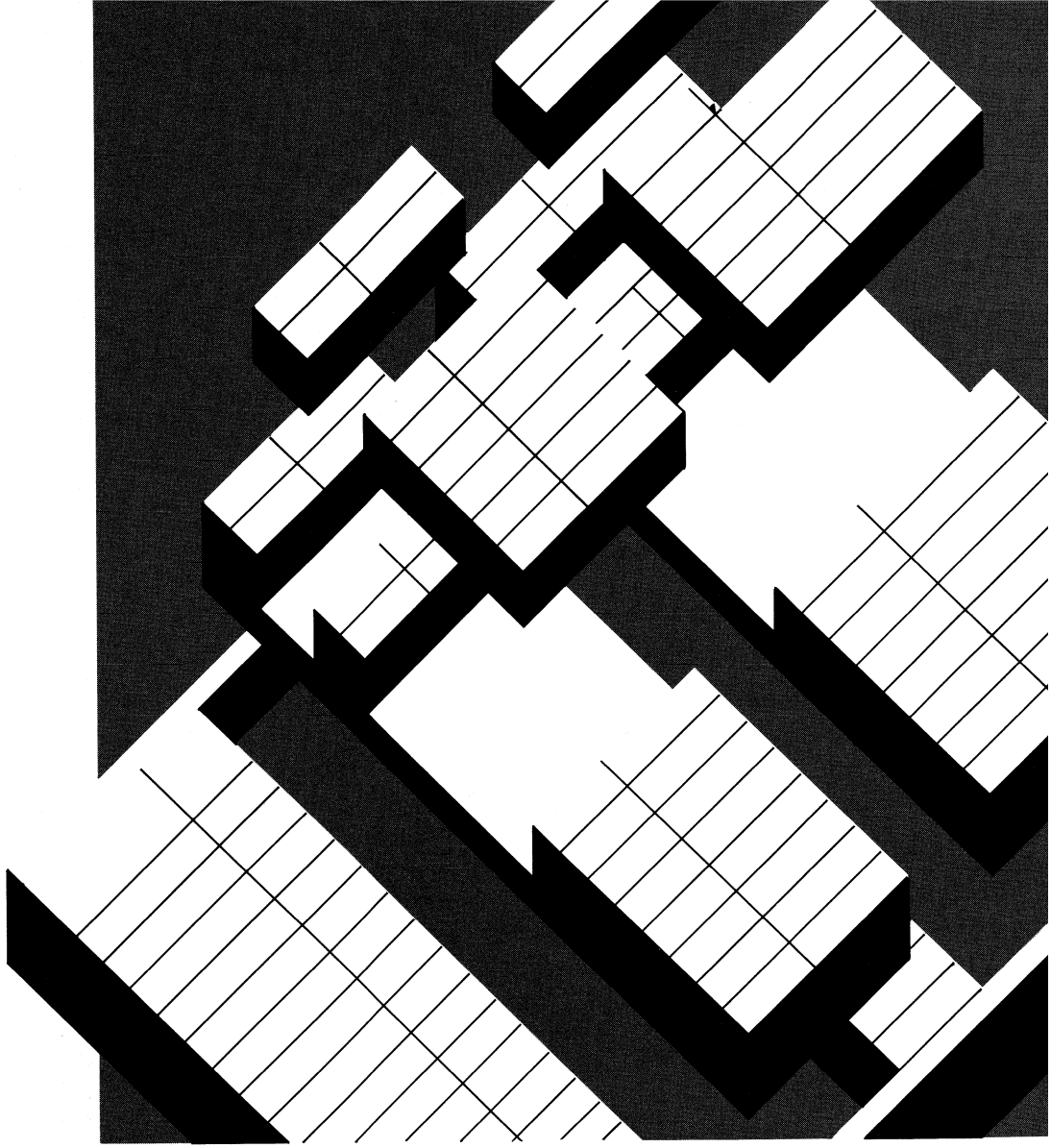
The RunTime program lets you package a run-time version of dBASE IV with your application program object files. The BUILD DOS-level command combines operation of the compiler, linker, and RunTime program. It creates an installed copy of your application and a run-time version of dBASE IV on floppy disk.

To prepare an SQL application for RunTime distribution, you copy the files from each SQL database directory (including SQL catalog table files), plus your application's .prg and .prs files, to a floppy disk. If you installed password protection using PROTECT, you should also copy the Dbsystem.db and Dbsystem.sql files.

If your application uses more than one SQL database, you must duplicate the same directory structure on the machine to which you're installing, and copy your application files to the same directories on the new machine.

To avoid having to duplicate the directory structure on a new machine, design your application to use a single database. Then, you can specify START DATABASE in your SQL application program without specifying a database name. To transfer a single-user application, transfer the contents of the database directory in which you developed your application, including the SQL catalog table files in that directory, and copy the Dbsystem.db and Dbsystem.sql files.

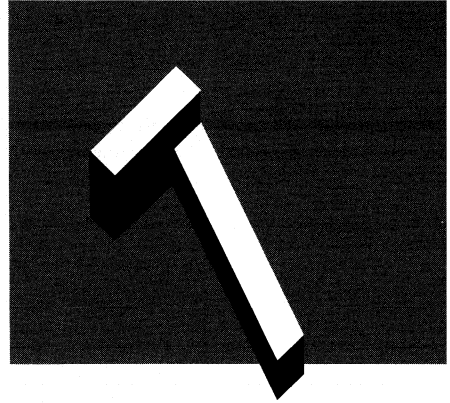
SQL Reference



i 1 2 3 4 5 6 7 8 A

B C D In

SQL Commands



This section describes the syntax of all SQL commands provided by dBASE IV. The commands are arranged in alphabetical order.

Symbols and Conventions

The syntax for SQL commands is provided in both written and graphic form. Figure 7-1 illustrates how to interpret each form of the command syntax.

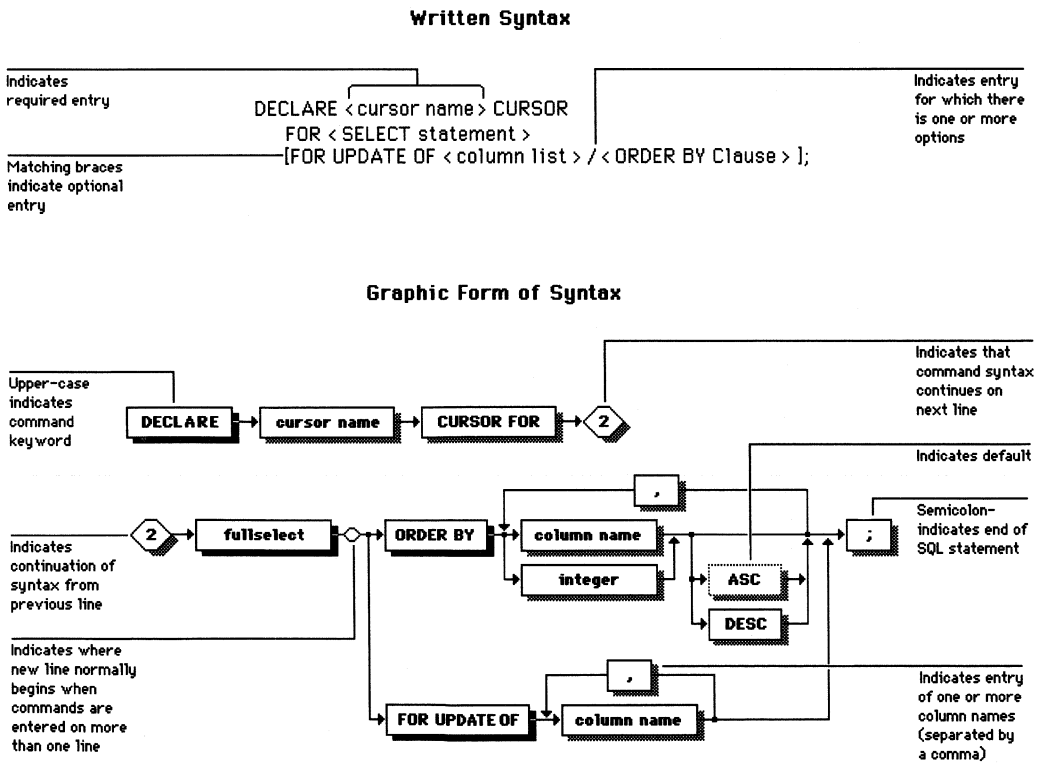


Figure 7-1 Command syntax

Reserved Words

The reserved words listed in Table 7-1 have special meaning and should not be used as names of tables, views, synonyms, indexes, columns, or memory variables.

Table 7-1 SQL Reserved Words

?	DECIMAL	LOG10	SIN
ABS	DECLARE	LOGICAL	SMALLINT
ACOS	DELETE	LOWER	SOUNDEX
ADD	DELIMITED	LTRIM	SPACE
ALL	DESC	MAX	SQRT
ALTER	DIF	MDY	START
AND	DIFFERENCE	MIN	STOP
ANY	DISTINCT	MOD	STR
AS	DMY	MONTH	STUFF
ASC	DOW	NOT	SUBSTR
ASIN	DROP	NUMERIC	SUM
AT	DTOC	OF	SYLK
ATAN	DTOR	ON	SYNONYM
ATN2	DTOS	OPEN	TABLE
AVG	EXISTS	OPTION	TAN
BETWEEN	EXP	OR	TEMP
BLANK	FETCH	ORDER	TIME
BY	FIXED	PAYMENT	TO
CDOW	FLOAT	PI	TRANSFORM
CEILING	FLOOR	PRIVILEGES	TRIM
CHAR	FOR	PUBLIC	TYPE
CHECK	FROM	PV	UNION
CHR	FV	RAND	UNIQUE
CLOSE	FW2	REAL	UNLOAD
CLUSTER	GRANT	REPLICATE	UPDATE
CMONTH	GROUP	REVOKE	UPPER
COS	HAVING	RIGHT	USER
COUNT	IN	ROLLBACK	USING
CREATE	INDEX	ROUND	VAL
CTOD	INSERT	RPD	VALUES
CURRENT	INT	RTOD	VIEW
CURSOR	INTEGER	RTRIM	WHERE
DATA	INTO	RUNSTATS	WITH
DATABASE	KEEP	SAVE	WKS
DATE	LEFT	SDF	WORK
DAY	LEN	SELECT	YEAR
DBASEII	LIKE	SET	
DBCHECK	LOAD	SHOW	
DBDEFINE	LOG	SIGN	

NOTE
Do not use the names of dBASE commands and functions as names for SQL tables, views, synonyms, indexes, columns, or memory variables.

Classes of Commands

SQL commands in dBASE IV can be categorized by the operation they perform. The SQL commands in each category are listed below.

Creation/Startup of SQL Databases

CREATE DATABASE	Creates a database directory and a set of SQL catalog tables for the new database
SHOW DATABASE	Lists the currently available databases
START DATABASE	*Opens* (activates) a database
STOP DATABASE	Closes the current database

Creation/Modification of Objects

ALTER TABLE	Adds a new column to an existing table
CREATE INDEX	Creates an index based on one or more columns in a table or view
CREATE SYNONYM	Defines an alternate name for a table or view
CREATE TABLE	Creates a new table, defining the columns within that table
CREATE VIEW	Creates a virtual table based on the columns defined in other tables or views

Database Security

GRANT	Grants user privileges for table access and update
REVOKE	Revokes table access and update privileges

Deletion of Objects

DROP DATABASE	*Drops* (deletes) an SQL database and deletes all tables, views, synonyms, and indexes from the database directory
DROP INDEX	Drops (deletes) an existing index
DROP SYNONYM	Drops (deletes) a synonym
DROP TABLE	Drops (deletes) a table
DROP VIEW	Drops (deletes) a view

Embedded SQL

CLOSE	Closes (releases) an SQL cursor
DECLARE CURSOR	Defines a cursor and a SELECT operation that defines the rows available to the cursor
FETCH	Advances the cursor pointer and copies the values of the selected row into dBASE memory variables
OPEN	Opens a cursor and executes the associated SELECT statement

Query and Update of Data

DELETE	Deletes specified rows from a table
INSERT	Adds new rows to a table or view
SELECT	Retrieves data from rows of one or more tables
UPDATE	Changes the data in selected rows of a table

Utilities

DBCHECK	Checks whether SQL catalog tables contain current information for SQL tables
DBDEFINE	Adds SQL catalog information to define .dbf database files as SQL tables
LOAD	Imports data from a non-SQL file into an SQL table
RUNSTATS	Updates catalog table statistical entries to optimize SQL access of tables
UNLOAD	Exports data from an SQL table to a foreign non-SQL file

ALTER TABLE

This command adds one or more columns to a table in the current database. If dBASE IV is password-protected, you must be the creator of, or have ALTER privileges for, the table you wish to alter.

Syntax

ALTER TABLE < table name >
    ADD (< column name > < data type >
    [, < column name > < data type > ...]);

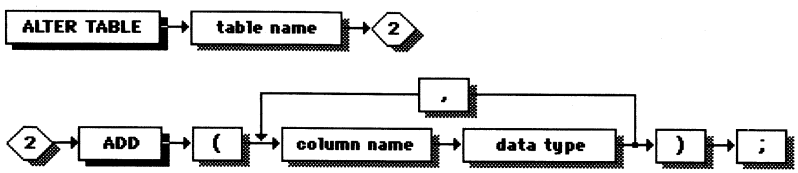


Figure 7-2 ALTER command

Usage

When you add a column to a table, you specify its column name and data type. For character and some numeric type columns, you also specify the column length and number of decimal places. You can specify one of the data types shown in Table 7-2. (See CREATE TABLE for a more detailed description of SQL data types.)

Table 7-2 SQL data types

Data Type	Description
CHAR(n)	Character string data
DATE	Dates (with default format *mm/dd/yy*)
DECIMAL(x,y)	Fixed decimal point numbers
FLOAT(x,y)	Floating point numbers
INTEGER	11-digit integer values
LOGICAL	Logical values (true or false)
NUMERIC(x,y)	Fixed decimal point numbers
SMALLINT	6-digit integer values

ALTER TABLE

NOTE

1. *When adding new columns, remember that the total number of columns in a table cannot exceed 255 and that the total width of a table in bytes cannot exceed 4,000.*
2. *dBASE IV SQL does not allow you to create or use dBASE memo fields.*
3. *New character columns added to existing rows are initialized with blank characters. Numeric columns are set to zero. Logical columns are set to false.*

You can only add columns to a table; you cannot add columns to a view. You also cannot remove a column or change the definition of an existing column.

TIP

To remove a column from a table or to modify a column's data type, create a new table with new column definitions and then insert data from the desired columns of the old table.

Example

To add two new columns, Phone_no and Last_order, to the Customer table, you would type:

```
SQL. ALTER TABLE Customer
     ADD (Phone_no CHAR(13), Last_order DATE);
```

See Also

CREATE TABLE, DROP TABLE, INSERT

CLOSE deactivates an open cursor in embedded SQL mode.

Syntax

CLOSE < cursor name > ;

Figure 7-3 CLOSE command

Usage

The CLOSE command deactivates the named cursor and releases all memory storage associated with the cursor. (This is the memory that holds the result table from the associated DECLARE CURSOR's SELECT statement.)

To use the CLOSE command, the specified cursor must already be open. A closed cursor can be reopened again by issuing another OPEN command. When you reactivate a cursor, the associated SELECT statement in the DECLARE CURSOR statement is re-evaluated to obtain a new result table.

The cursor is automatically closed at the end of execution of the SQL program module in which the cursor was opened, execution of the SET SQL OFF command, or a change in the active database specified by a CREATE DATABASE, START DATABASE, or STOP DATABASE command. A cursor will remain open, however, when an SQL program branches to execute a subprogram. When execution returns to the SQL program from which the call was made, the cursor points to the same row as when the program was left.

TIP
If you want to re-evaluate the SELECT statement defined with the cursor declaration, you may close and then open the cursor as often as you like. A cursor must be CLOSEd before it can be OPENed again.

Example

The following program example includes a statement to close the cursor Inv after fetching all records from an SQL cursor:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced;
   .
   .
   .
OPEN Inv;
   .
   .
   .
DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
     * Print an invoice for order number in the the current row
     * If successful, change Minvoiced memory variable to .T.
     *
     DO Invoices WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
     *
     IF minvoiced .AND. Sale_date < CTOD("09/22/87")
       DELETE FROM sales
       WHERE CURRENT OF Inv;
     ENDIF
   ELSE
     EXIT
   ENDIF
ENDDO
CLOSE Inv;
```

See Also

DECLARE CURSOR, FETCH, OPEN CURSOR

CREATE DATABASE

This command creates a new database to hold related tables and files. No authorization is required to create a database.

Syntax

CREATE DATABASE [< path >] < database > ;

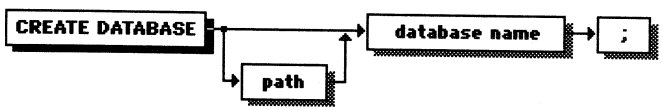


Figure 7-4 CREATE DATABASE command

Usage

The CREATE DATABASE command creates an SQL database and defines a set of SQL catalog tables in the directory defined for the new database. All information about new tables, views, synonyms, and indexes is recorded in the database directory's catalog tables. The name and DOS path of the database directory are stored in a row entry of the master catalog table Sysdbs. Each SQL database name must be unique.

You can specify an explicit path of up to 64 characters. (Do not enter space characters between the path and the name of the database.) If the specified database directory does not already exist, a new directory is created. The new database will be created in the directory specified by the path (relative paths are not allowed). If you do not specify a path, the database directory is created as a subdirectory of your current directory. If a database directory already exists, SQL defines it as a new database and copies a set of catalog tables into that directory.

After you create an SQL database, it is automatically activated. You may also activate a database with the START DATABASE command. To automatically start a database when you enter SQL mode or execute an SQL program, set SQLDATABASE to the name of an SQL database in your Config.db file. If dBASE IV is password-protected, only the creator of a database or the SQLDBA user ID may DROP that database.

Examples

To create the database Mydata as a subdirectory of your current directory, you would type:

```
SQL. CREATE DATABASE Mydata;
```

To create the database Myapp as a subdirectory of an existing directory named C:\DBASE, you would type:

```
SQL. CREATE DATABASE C:\Dbase\Myapp;
```

CREATE INDEX

This command builds an index based on one or more columns of a table in the current database. If dBASE IV is password-protected, you must be the table's creator or have INDEX privileges for the table on which the index is based.

Syntax

CREATE [UNIQUE] INDEX <index name>
ON <table name>
(<column name> [ASC/DESC]
[, <column name> [ASC/DESC]...]);

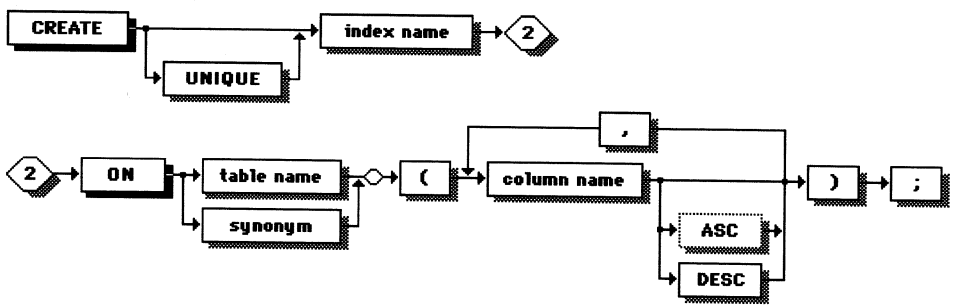


Figure 7-5 CREATE INDEX command

Usage

You use the CREATE INDEX command to create an SQL index. Indexes keep track of the location of rows within an SQL table. The indexes you create should match the order in which you normally retrieve rows from a table. While you specify the indexes you want to create, you don't specify the indexes to use when you retrieve data. dBASE IV SQL automatically selects the most efficient index to use to retrieve data each time you execute an SQL statement.

The column or columns used to create an index are called an *index key* or *tag*. You can create up to 47 different index keys or tags for any table. However, as you increase the number of indexes, the speed of updating data in tables (INSERT, UPDATE, and DELETE operations) will decrease.

CREATE INDEX

SQL indexes are maintained as index tags in a dBASE .mdx file having the same name as the table on which the index is based. Whenever you INSERT, UPDATE, or DELETE columns specified in an index key or tag, the appropriate index tags in the .mdx index file are updated.

The [ASC/DESC] keyword specifies the order of rows in the index, ascending (ASC) or descending (DESC), by values of the data in the indexed column. Ascending order indexes are the default, so you don't have to include the ASC keyword when creating them. Indexes may be created on any column data type except Logical. ASCending and DESCending columns cannot be combined in the same index.

NOTE

1. *Index files can only be built for tables, not for views. However, when you use a view, dBASE IV SQL will use any indexes based on the tables underlying the view.*
2. *Rows for indexes based on character columns are arranged according to the ASCII values of columns. Keep in mind that the ASCII value of characters distinguishes letters in lower case from those in upper case.*
3. *You can view the indexes available for tables by displaying columns from the Sysidxs catalog table.*

If you specify the UNIQUE keyword when you create an index, dBASE IV checks the values of columns specified in the index and verifies that the data in those columns is unique. If the values in index columns are not unique, the index will not be created.

After you create a unique index, dBASE IV SQL checks every INSERT or UPDATE operation to ensure that the values of columns specified in the index remain unique. (Note that these checks slow performance.) Changes to the table that would result in non-unique index key values are rejected.

WARNING
Database files for SQL tables that have unique indexes defined will be available only as read-only files in dBASE mode. To edit or append data in those tables, you can export the data to another dBASE file using the UNLOAD command or the SAVE TO TEMP clause of the SELECT command.

If dBASE IV is password-protected, only the creator of an index or the SQLDBA user ID can drop that index. However, indexes are dropped automatically when the tables on which they are based are dropped.

Examples

To create an ascending order index on the Lastname column in the Customer table, you would type:

```
SQL. CREATE INDEX Lastname
     ON Customer (Lastname ASC);
```

To create a descending order index on the Unitcost column in the Inventry table, you would type:

```
SQL. CREATE INDEX Expense
     ON Inventry (Unitcost DESC);
```

To create a unique index based on the Order_no column of the Sales table, you would type:

```
SQL. CREATE UNIQUE INDEX Order_No
     ON Sales (Order_No ASC);
```

See Also

DROP INDEX

CREATE SYNONYM

CREATE SYNONYM defines a synonym or alternate name for a table or view in the current database. No authorization is required to create a synonym. However, if dBASE IV is password-protected, your privileges for use of the synonym are the same as those you have on the table or view for which it is defined.

Syntax

CREATE SYNONYM <synonym name>
FOR <table name>;

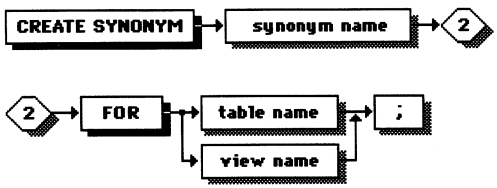


Figure 7-6 CREATE SYNONYM command

Usage

The CREATE SYNONYM statement defines an alternate name for a table or view. For example, you could use this command to create an abbreviated name for a table or view with an inconveniently long name. New synonym names must be unique within the current database. If dBASE IV is password-protected, only the creator of a synonym or the SQLDBA user ID can drop it. Synonyms are automatically dropped when the tables or views on which they are based are dropped.

NOTE

1. *If the creator of a table or view changes your privileges for a specific table or view, those changes also apply to associated synonyms.*
2. *You can view the synonyms already defined in a database by displaying columns from the Syssyns catalog table.*

Examples

To create the synonym C1 for the Customer table, you would type:

```
SQL. CREATE SYNONYM C1 FOR Customer;
```

Then, to use the synonym, you could type:

```
SQL. SELECT * FROM C1;
```

See Also

DROP SYNONYM

CREATE TABLE

The CREATE TABLE command defines a new SQL table in the current database. No authorization is required to create an SQL table. If dBASE IV is password-protected, tables are encrypted after creation.

Syntax

CREATE TABLE < table name >
    (< column name > < data type >
    [, < column name > < data type > ...]);

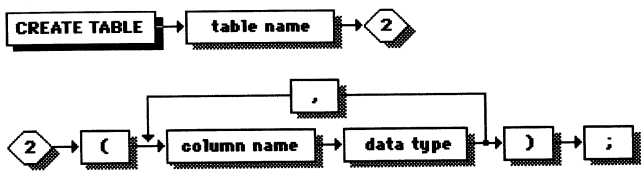


Figure 7-7 CREATE TABLE command

Usage

You use this command to create a new table. You provide the name of the new table and the names, data types, and sizes of columns in the new table. < table name > is the name of the table you are creating. The current database cannot already contain another table, view, or synonym with the same name.

When you create a table, SQL creates a .dbf database file with the same name as the table. If a .dbf file with the same name already exists in the current database directory, the new table will overwrite the existing file, and create a new .dbf file. Fields in the structure of the .dbf file are created to correspond with specified columns of the SQL table. To define the columns in the table, you specify each column's name and its data type as shown in Table 7-3.

Table 7-3 SQL data types

Data Type	Description
SMALLINT	Holds an integer with up to six digits (including sign). Values entered may range from −99,999 to 999,999 (if a plus sign is not specified). This column type maps to a NUMERIC(6,0) field in the .dbf file.
INTEGER	Holds an integer containing up to 11 digits (including sign). Values entered may range from −9,999,999 to 99,999,999,999 (if a plus sign is not specified). This column type maps to a NUMERIC(11,0) field in the .dbf file.
DECIMAL(x,y)	Holds a signed fixed decimal point number with *x* total digits (including sign) and *y* decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 19 and *y* may range from 0 to 18. This column type maps to a NUMERIC(x + 1,y) field in the .dbf file.
NUMERIC(x,y)	Holds a signed fixed decimal point number with *x* total digits (including sign and decimal point) and *y* decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 20 and *y* may range from 0 to 18. This column type maps directly to the same data type in the .dbf file, NUMERIC(x,y).
FLOAT(x,y)	Holds a signed floating point number with *x* total digits (including sign and decimal point) and y decimal places (significant digits to the right of the decimal point). *x* may range from 1 to 20 and *y* may range from 0 to 18. The range of numbers you may store is $0.1\ 10^{-307}$ to $0.9\ 10^{+308}$. A number may be specified using scientific (exponential) notation, for example, −9.99E + 235. This column type maps to a FLOAT(x,y) field in the .dbf file.
CHAR(n)	Holds a character string of up to *n* characters. *n* may range from 1 to 254. Values may be entered from character columns, character type memory variables, or a character string. This column maps directly to a CHAR(n) field of the same length in the .dbf file.

(*continued*)

Table 7-3 SQL data types (*continued*)

Data Type	Description
DATE	Holds a date in the format specified by the SET DATE and SET CENTURY commands. The default format is *mm/dd/yy*. Values are entered from date columns, date type memory variables, or date strings converted with the dBASE CTOD() function, for example, *CTOD("02/15/86")* or a date delimited by {}, for example, *{02/15/86}*. This column type maps directly to a date field in the .dbf file.
LOGICAL	Holds logical true or false values. .T. represents a true value and .F. represents a false value. Values are entered from dBASE logical memory variables or columns, or by the constants .T., .t., .Y., .y., .F., .f., .N., and .n.. This column type maps directly to a logical field in the .dbf file.

NOTE

1. *The total number of columns in a table cannot exceed 255 and the total width of a table in bytes cannot exceed 4,000.*
2. *dBASE IV SQL does not allow you to create or use dBASE memo fields.*

Examples

To define a table, Shipment, to record shipments leaving a firm, you would type:

```
SQL. CREATE TABLE Shipment
     (Ship_no       CHAR(6),
      Shipdate      DATE,
      Order_no      CHAR(6),
      Shipper       CHAR(25),
      Weight        DECIMAL(4,1),
      Value         DECIMAL(10,2));
```

See Also

ALTER TABLE, DBDEFINE, DROP TABLE, LOAD DATA

CREATE VIEW

The CREATE VIEW command creates a special *virtual* table from columns defined in one or more tables or views (including catalog tables). If dBASE IV is password-protected, you must be the creator of, or have SELECT privileges on, every table referenced in the view's definition.

Syntax

CREATE VIEW < view name > [(< column name > , < column name > ...)]
AS < subselect >
[WITH CHECK OPTION];

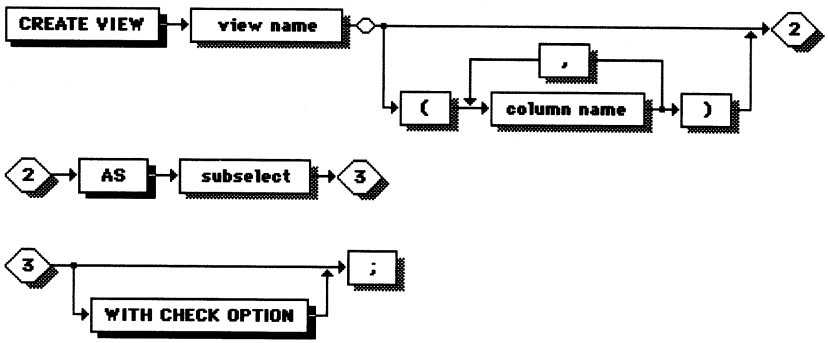


Figure 7-8 CREATE VIEW command

Usage

You can create a view on one or more tables or views for which you have SELECT privileges. (You may also specify synonyms that name tables or views.) You can then use the view to retrieve and, in some cases, insert, update, or delete data. Once you create the view, your authorization to use it is based on your privileges for its underlying tables and whether the view is updatable. When retrieving data, the view always contains the most current data from the tables on which the view is defined.

Once a view is created, only the creator of the view or the SQLDBA user can delete it. However, if any of the tables on which the view is defined are dropped, the view is automatically dropped.

The definition of a view is stored in the SQL catalog table Sysviews. When you assign a name to a view, the current database cannot contain another table, view, or synonym with the same name. (The view name can duplicate the name of an existing index, but this is not recommended.)

The view is created with column values from rows specified by a SELECT statement. Data is transferred from the columns specified in the AS < subselect > clause to the corresponding columns defined in the view definition. You may specify (with the exceptions listed below) any valid SELECT statement, even if the result returns no rows.

For views that can be updated, you can specify the WITH CHECK OPTION. This option specifies that inserted or updated rows are checked against conditions set in the WHERE clause of the SELECT statement that defines rows in the view. If inserted rows or updated column values would create a row not allowed by the view definition, those rows are not added or changed in the underlying tables.

Naming Columns in the View

[(< column name > , < column name > ...)] defines a column list for the view. You may specify column names in the view to change the column names from those specified in the SELECT statement. If you do not enter a column list, the view's column names will be the same as those in the underlying tables specified in the SELECT clause. All columns, however, must be explicitly named if the column names in the SELECT statement are derived (that is, if they are a constant, character string, memory variable, expression, or an SQL or dBASE function).

NOTE
Naming conventions for view columns are the same as those defined for table columns. The size limits of views (number of rows and columns) are the same as those defined for tables.

There are some restrictions on the SELECT statements you can use to create a view.

1. You may not specify a UNION as part of the SELECT statement that defines the view.
2. The SELECT statement cannot include FOR UPDATE OF, INTO, ORDER BY, or SAVE TO TEMP clauses.
3. If you specify a GROUP BY clause in the SELECT statement used to create a view, you cannot later use that view in the FROM clause of another statement to join with another table or view.

Updatable Views

In addition to table authorization privileges, the definition of a view by the SELECT statement also determines whether you can update underlying tables of views. You will not be able to update underlying tables of views if you include any of the following in the SELECT statement defining views:

- An SQL aggregate function in the SELECT clause
- A FROM clause naming another non-updatable view or more than one table or view
- The DISTINCT keyword
- A GROUP BY clause
- A nested subquery with a FROM clause referring to the same base table on which the view is based

If a column of a view is derived from an arithmetic expression or constant, you cannot update that particular column of the view.

Examples

To create a simple view that includes all columns from the Sales table and includes only those rows for non-invoiced orders, you could type:

```
SQL. CREATE VIEW Billings
     AS SELECT *
     FROM Sales
     WHERE NOT Invoiced;
```

To create a view that contains only a subset of columns from the Staff table, you could type:

```
SQL. CREATE VIEW La
     AS SELECT Staff_no, Lastname, Hiredate
     FROM Staff
     WHERE Location = "LOS ANGELES";
```

To create a view that combines the Lastname and Firstname columns of the Customer table, you could create the following view:

```
SQL. CREATE VIEW Address
     (Fullname, City, State)
     AS SELECT Firstname+Lastname, City, State
     FROM Customer;
```

To create a view that combines data from columns in two different tables, the Staff table and the Sales table, you could type:

```
SQL. CREATE VIEW Orders
     (Order_no, Sale_date, Seller)
     AS SELECT Sales.Order_no, Sale_date, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no;
```

See Also

DROP VIEW, SELECT

The DBCHECK command checks the catalog table entries for one or more SQL tables in the current SQL database to see if the catalog table entries are consistent with the tables' underlying .dbf and .mdx file structures.

Syntax

DBCHECK [< table name >];

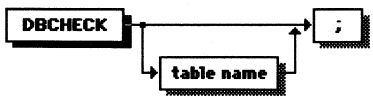


Figure 7-9 DBCHECK command

Usage

This command is used to verify the structure of .dbf and .mdx files with entries in SQL catalog tables. If you specify a table name with the DBCHECK command, it only verifies the .dbf and .mdx files for that table. Otherwise, DBCHECK verifies the files for all SQL tables and their associated indexes in the current database. DBCHECK cannot check files encrypted using the PROTECT command in dBASE mode.

DBCHECK returns error messages for every table for which the .dbf or .mdx file structures do not match the definitions in the SQL database's catalog tables. If you receive these error messages, you should perform the following steps:

1. Copy the .dbf and .mdx files named in the error messages from the current database directory to another directory or backup disk.
2. In SQL mode, DROP the tables for which errors were displayed.
3. Copy the .dbf and .mdx files back into the database directory.
4. In SQL mode, run the DBDEFINE command for each .dbf file you want to redefine as an SQL table. (Associated indexes will be redefined as part of the DBDEFINE command's operation.)

See Also

DBDEFINE

DBDEFINE

This command creates catalog table entries for one or more dBASE database files (and their associated .mdx index files) in the currrent SQL database. No authorization is needed to run DBDEFINE; however, all files specified with this command must be unencrypted. If dBASE IV is password-protected, the current user ID will be named as the creator of any tables defined with this command. These new tables will also be encrypted following creation with a special SQL encryption key.

Syntax

DBDEFINE [< .dbf file >];

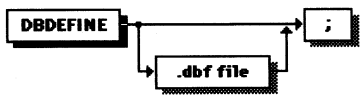


Figure 7-10 DBDEFINE command

Usage

If you specify a .dbf file with DBDEFINE, SQL catalog entries in the current database are created for that file only (and for the .mdx file of the same name). Otherwise, DBDEFINE creates catalog entries for all .dbf files and .mdx files in the current SQL database that are not already defined as SQL tables or indexes. After DBDEFINE executes successfully, it creates a documentation text file named Dbdefine.txt. This file contains the SQL CREATE TABLE and CREATE INDEX statements used to define tables and indexes in the current database.

NOTE

1. *The DBDEFINE command ignores memo fields. Associated memo fields remain with the database file, but no catalog table entries are made for them. You cannot access memo fields while in SQL mode.*
2. *If .mdx files are available when a .dbf file is DBDEFINEd, the catalog tables are updated to reflect the statistics of the most recent REINDEX operation. To assure that the statistics reflect the actual state of the current .dbf file, you should REINDEX a file before entering SQL mode to DBDEFINE it, or execute the RUNSTATS command afterward.*
3. *You cannot DBEFINE an index created in dBASE mode that specifies the UNIQUE option of the INDEX command.*

DBDEFINE displays error messages for any files for which it cannot successfully define tables or indexes. Messages also display when DBDEFINE cannot read a .dbf file, or when catalog entries already exist for specified files.

To redefine SQL tables or indexes you should perform the following steps:

1. Copy the .dbf and .mdx files named in the error messages from the current database directory to another directory or backup disk.
2. In SQL mode, DROP the tables for which errors were displayed.
3. Copy the .dbf and .mdx files back into the database directory.
4. In SQL mode, run the DBDEFINE command for each .dbf file you want to redefine as an SQL table. (Associated indexes will be redefined as part of the DBDEFINE command's operation.)

See also

CREATE DATABASE, DBCHECK, START DATABASE

DECLARE CURSOR

The DECLARE CURSOR command defines a cursor. Cursors allow you to execute a SELECT statement and process rows from its result table, one at a time, while you're operating in embedded SQL mode. If dBASE IV is password-protected, you must have sufficient privileges to perform the SELECT statement in the DECLARE CURSOR statement.

DECLARE < cursor name > CURSOR
    FOR < SELECT statement >
    [FOR UPDATE OF < column list > / < ORDER BY clause >];

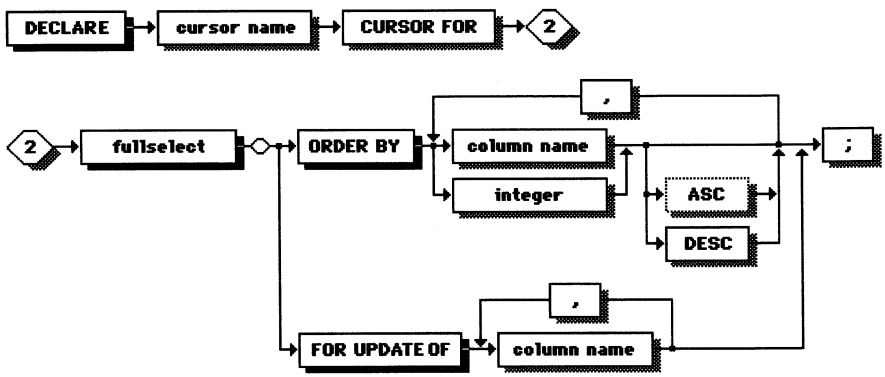


Figure 7-11 DECLARE CURSOR command

Usage

SQL cursors provide a mechanism similar to a record pointer. Unlike a record pointer, however, the cursor points to rows of a result table rather than to rows within a base table. The data in columns of a row to which the cursor is pointing can be transferred to dBASE memory variables to be processed.

The DECLARE CURSOR command defines a cursor that has an associated SELECT statement. The cursor definition remains declared until you leave SQL mode (for example, by executing the statement SET SQL OFF, exiting the program in which the cursor was declared, or exiting dBASE IV). Different users can define the same cursor definition, but they do not share the same cursor when the DECLARE CURSOR statement is executed.

The SELECT statement defined with the DECLARE CURSOR command is not executed until an OPEN < cursor > command is executed. The OPEN command executes the SELECT statement, producing a result table. The FETCH command advances the cursor pointer one row and retrieves the data transferring it to dBASE memory variables specified with the FETCH statement.

The DECLARE CURSOR command can also set up operations to update and delete data in rows pointed to by the cursor. To update data, you need to add the FOR UPDATE clause to the SELECT statement. Then, after fetching a particular row, you can update the column values with a form of the UPDATE command that includes a SET...WHERE CURRENT OF clause. When you specify the FOR UPDATE clause, you cannot also include the ORDER BY or UNION clauses.

To delete data from a table in which cursors are used to select rows, you use a form of the DELETE command that also includes the clause WHERE CURRENT OF.

NOTE
The result table associated with an SQL cursor follows rules for updates similar to those defined for views (see CREATE VIEW). That is, rows cannot be updated if the DECLARE CURSOR's SELECT statement contains aggregate functions or DISTINCT, GROUP BY, HAVING or UNION clauses. Rows also cannot be updated if the SELECT statement's FROM clause lists more than one table, a view that is not updatable, or a table that is also specified in a subquery within the SELECT statement.

Examples

To declare a cursor to display rows from the Sales table for uninvoiced orders, you could enter the following statement in a program:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced;
```

DECLARE CURSOR

To declare a cursor in a program to allow updating rows in a table, you could enter the following program:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced
FOR UPDATE OF Invoiced;
   .
   .
   .
OPEN Inv;

DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
         .
         .
         .
      IF minvoiced
         UPDATE Sales
         SET Invoiced = .T.
         WHERE CURRENT OF Inv;
      ENDIF
   ELSE
      EXIT
   ENDIF
ENDDO
CLOSE Inv;
```

See Also

CLOSE, FETCH, OPEN

The DELETE command deletes specified rows from a table. If dBASE IV is password-protected, you must have DELETE privileges on the table or view from which you want to delete rows.

There are two forms of the command. The first form can be run in both interactive and embedded SQL modes. The second form includes the WHERE CURRENT OF clause and is used exclusively in embedded SQL mode with a declared cursor.

Syntax

DELETE (1)
FROM <table name> [<alias name>]
[<WHERE clause>];

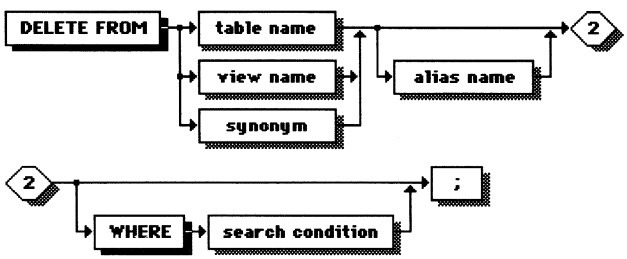


Figure 7-12 DELETE ... WHERE <search condition> command

DELETE (2)
FROM <table name>
WHERE CURRENT OF <cursor name>;

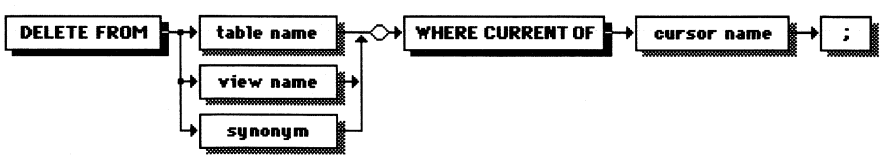


Figure 7-13 DELETE ... WHERE CURRENT OF command

Usage

If dBASE IV is password-protected, you must be the creator of, or have DELETE privileges on, the table or view (or synonym that names a table or view) from which you want to delete rows.

To delete rows from a view, the view must be constructed from a single table. The view's definition also cannot include the DISTINCT keyword, a GROUP BY or HAVING clause, or an SQL aggregate function.

DELETE with WHERE Clause

The first form of the command allows you to specify a WHERE clause to select the rows of a table or view to be deleted. The WHERE clause may contain simple search conditions, combined search conditions, subselects, and non-correlated subqueries (see the SELECT command).

TIP
If you specify a WHERE < search condition > clause with the DELETE command, you may want to verify that the WHERE condition is selecting and deleting the correct rows. You can verify this by first using the SELECT command with the same WHERE clause you intend to use to delete rows.

Each column named in the WHERE search condition must name a column of the table from which you are deleting rows. Also, if you specify a subselect in the WHERE clause, the table or view from which you're deleting rows cannot be referenced in the subselect.

WARNING
Be careful when you use the DELETE command. A warning message appears if you specify the DELETE command without a WHERE clause. If you choose to continue and execute the command, all rows in the table or view will be deleted. Unlike the DELETE command executed in dBASE mode, which only marks records for deletion, the SQL DELETE command actually removes the rows from the table.

If you execute the DELETE command within a transaction (using the BEGIN...END TRANSACTION commands), deleted rows are not removed from the SQL table until the transaction ends. They are displayed in result tables preceded by an asterisk. SET DELETED ON to ignore deleted rows in subsequent SQL operations and remove them from the display of result tables. This is especially important when working on a local area network.

DELETE with WHERE CURRENT OF Clause

The second form of the DELETE command is only used in embedded SQL mode. You can use the DELETE command to delete rows pointed to by a cursor.

With this form of the DELETE command, the cursor referenced in the WHERE CURRENT OF clause must already be defined in a DECLARE CURSOR statement. Also, the table or view from which you're deleting must be referenced in the cursor's SELECT statement. When the DELETE statement is executed, the cursor must be open. The DELETE statement will delete the row to which the cursor is pointing.

In a program using a cursor, you can FETCH columns into dBASE memory variables and determine from those values whether you want to use the DELETE command. For example, you could use the IF...ELSE program construct to DELETE...WHERE CURRENT OF if you want to delete the current row, otherwise (else) FETCH the next row.

If you set up operations to delete multiple rows in an SQL program file, particularly one that will run on a local area network, you should include transaction processing (using BEGIN and END TRANSACTION commands) in your program. Rows deleted in a transaction with DELETE...WHERE CURRENT OF are not removed from the table until the END TRANSACTION command is executed and the corresponding cursor is CLOSEd within the transaction.

Example

To delete a row from the Staff table, you could type:

```
SQL. DELETE FROM Staff
     WHERE Lastname = "Long";
```

The system displays the message **1 row(s) deleted.**

To delete all rows from the Sales table where the Sale_date column contains a date prior to 9/22/87, you could type:

```
SQL. DELETE
     FROM Sales
     WHERE Sale_date < {"09/22/87"};
```

The system displays the message **4 row(s) deleted.**

DELETE

To use the DELETE statement with an SQL cursor, you can use DELETE... WHERE CURRENT OF, which will delete the row pointed to by the cursor. For example:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced;
   .
   .
   .
OPEN Inv;
   .
   .
   .
ON ERROR DO Recovery
DO WHILE .T.
   BEGIN TRANSACTION
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
         .
         .
         .
      * Print an invoice for order number in the the current row
      * After invoicing, change Minvoiced memory variable to .T.
      *
      DO Invoices WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
      *
      IF minvoiced .AND. Sale_date < {09/22/87}
         DELETE FROM Sales
         WHERE CURRENT OF Inv;
      ENDIF
   ELSE
      EXIT
   ENDIF
   END TRANSACTION
ENDDO
CLOSE Inv;
```

See Also

DECLARE CURSOR, FETCH, SELECT

DROP DATABASE

The DROP DATABASE command drops (deletes) a database. If dBASE IV is password-protected, only the creator of a database or the SQLDBA user can drop that database.

Syntax

DROP DATABASE < database name > ;

Figure 7-14 DROP DATABASE command

Usage

The DROP DATABASE command drops a database by deleting all .dbf and .mdx files in the database directory for which there are entries in the Systables and Sysidxs catalog tables. It also deletes the database's entry in the Sysdbs catalog table, but does not delete the directory.

When you issue this command, a warning prompt appears. You can then cancel the DROP DATABASE operation by pressing **Esc**.

NOTE
To close a database that you have activated, use the STOP DATABASE command. On a local area network, you cannot DROP a database that another user has activated.

See Also

CREATE DATABASE, SHOW DATABASE, START DATABASE, STOP DATABASE

DROP INDEX

The DROP INDEX command is used to drop (delete) an index. If dBASE IV is password-protected, only the creator of an index or the SQLDBA user can drop that index.

Syntax

DROP INDEX <index name>;

Figure 7-15 DROP INDEX command

Usage

The DROP INDEX command drops the specified indexes. Indexes are automatically dropped whenever the table (or a synonym for a table) on which they are defined is dropped.

See Also

CREATE INDEX, DROP TABLE

DROP SYNONYM

The DROP SYNONYM command drops (deletes) a table or view synonym. If dBASE IV is password-protected, only the creator of a synonym or the SQLDBA user can drop that synonym.

Syntax

DROP SYNONYM <synonym name>;

Figure 7-16 DROP SYNONYM command

Usage

The DROP SYNONYM command drops a synonym name defined for a table or view.

Synonyms are automatically dropped if the table or view for which they are defined is dropped.

See Also

CREATE SYNONYM

DROP TABLE

The DROP TABLE command drops (deletes) a table. If dBASE IV is password-protected, only the creator of a table or the SQLDBA user can drop that table.

Syntax

DROP TABLE <table name>;

Figure 7-17 DROP TABLE command

Usage

When you drop a table using the DROP TABLE command, the corresponding .dbf database file is deleted and references to the table are removed from the SQL catalog tables. In addition, all indexes, views, and synonyms based on the table are also dropped.

WARNING
Be careful when you use the DROP TABLE command. Once a table is dropped, you cannot restore it without re-creating the table and inserting the data into the table again.

See Also

CREATE TABLE, DELETE

DROP VIEW

The DROP VIEW command drops (deletes) a view. If dBASE IV is password-protected, only the creator of a table or the SQLDBA user can drop that view.

Syntax

DROP VIEW <view name>;

Figure 7-18 DROP VIEW command

Usage

When you use the DROP VIEW command to drop a view, only the view definition is deleted. No underlying tables are deleted. In addition, when you drop a view, all synonyms and views based on that view are also dropped.

A view and any synonyms of the view are automatically dropped if the underlying tables on which the view is defined are dropped.

See Also

CREATE VIEW, DROP TABLE

FETCH

The FETCH command positions a cursor on the next row of the result table and transfers values from that row into corresponding dBASE memory variables. You can only use this command in embedded SQL mode. A cursor must have been declared and opened prior to executing this command.

Syntax

FETCH < cursor name >
    INTO < variable list > ;

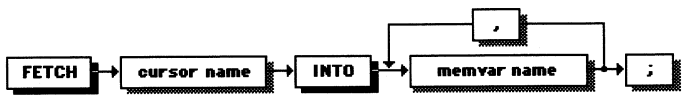


Figure 7-19 FETCH command

Usage

The FETCH command allows you to process rows of a result table. It is used in conjunction with the DECLARE CURSOR command, which specifies a SELECT statement to generate a result table. The OPEN < cursor > command executes the SELECT statement. The FETCH statement advances the cursor through each row of the result table, one row at a time. If dBASE IV is password-protected, user privileges are checked when the DECLARE CURSOR and OPEN commands are executed.

The cursor specified in the FETCH statement must already be declared (with the DECLARE CURSOR command) and opened (with the OPEN < cursor > command). The cursor's SELECT statement will then have been executed to produce a result table. (See DECLARE CURSOR for more information on setting up a cursor to use with this command.) The first time a FETCH statement is executed, it positions the cursor to the first row of the result table. Each time the statement is executed thereafter, the cursor advances to the next row until reaching the last row of the result table.

NOTE
The cursor can only be advanced in a forward direction. However, if you want, you can reset the cursor by closing the cursor with the CLOSE command. You can then issue the OPEN statement to re-execute the SELECT statement and position the cursor back at the first row of the new result table.

Information from columns in the current row is transferred into the memory variables specified in the FETCH command's INTO clause in the same order as the columns specified in the cursor's SELECT statement. Therefore, when specifying memory variables of an INTO clause, you need to specify the same number of memory variables as appear in the cursor's SELECT clause.

Information is transferred into memory variables each time a FETCH statement is executed unless the cursor was already on the last row of the cursor's result table (or the cursor's SELECT statement didn't return any rows).

You can use two SQL system status variables, *Sqlcnt* and *Sqlcode*, to determine when you've processed all rows in the cursor's result table. After executing an OPEN statement that executes the SELECT statement associated with the cursor, *Sqlcnt* contains the number of rows in the result table. (If the result table is empty, the *Sqlcnt* will contain 0 and *Sqlcode* will contain a value of + 100.)

NOTE
If the SELECT statement defined by DECLARE CURSOR does not return any rows, memory variables specified in the corresponding FETCH command are not created. You may want to initialize their values or check the value of Sqlcnt before processing memory variables named in the FETCH command.

After each FETCH statement the *Sqlcode* variable indicates the status of that operation. A value of zero indicates the FETCH was successful — the FETCH advanced the cursor and successfully transferred the row. If there was an error, *Sqlcode* contains a value of − 1. If the cursor was on the last row when the FETCH statement was executed, *Sqlcode* contains a value of 100 (indicating there are no more rows in the result table to process).

Examples

To FETCH Order_no, Sale_date, Staff_no, Cust_no, and Invoiced columns from selected rows of the Sales table, you could enter the following statements in an SQL program:

```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced;
   .
   .
   .
OPEN Inv;
   .
DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
         .
         .
         .
   ELSE
      EXIT
   ENDIF
ENDDO
```

See Also

DECLARE CURSOR, DELETE, FETCH, UPDATE

The GRANT command grants access privileges on tables and views (or synonyms naming tables or views) in the current database. GRANT works with user IDs created with the PROTECT command (recording user ID information in the Dbsystem.sql file).

Syntax

```
GRANT ALL [PRIVILEGES]/ < privilege list >
    ON [TABLE] < table list >
    TO PUBLIC/ < user list >
    [WITH GRANT OPTION];
```

NOTE
The WITH GRANT OPTION cannot be assigned for a privilege previously GRANTed to PUBLIC.

The privilege list consists of one or more of the following:

```
ALTER
DELETE
INDEX
INSERT
SELECT
UPDATE [( < column list > )]
```

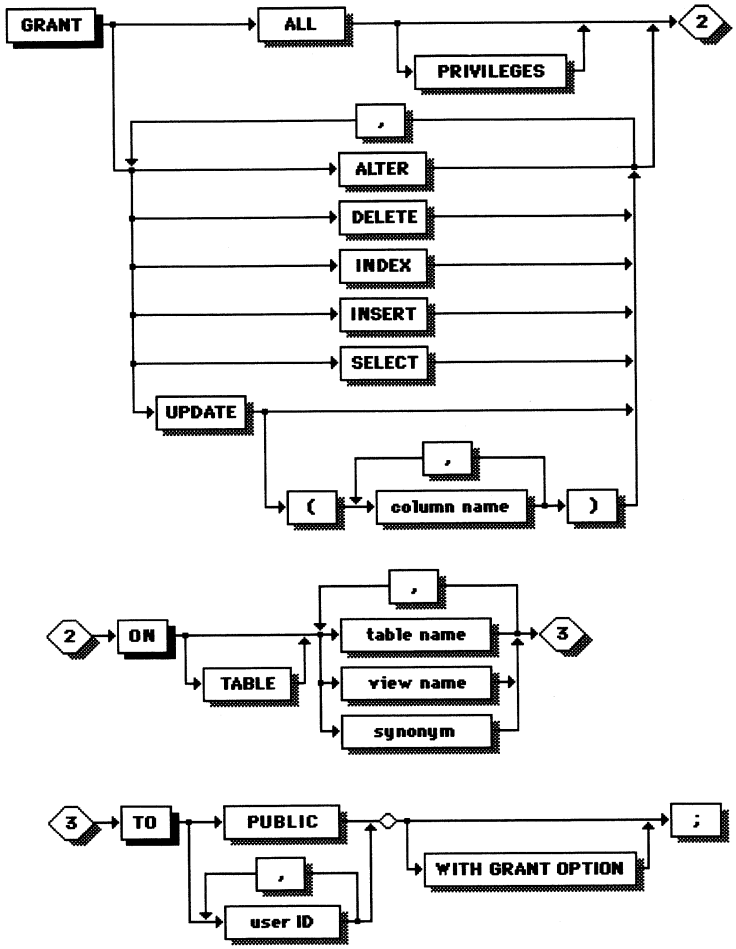


Figure 7-20 GRANT command

Usage

SQL provides two commands, GRANT and REVOKE, to assign or restrict user privileges to tables or views. All privileges or *authorization* assigned by GRANT and REVOKE are recorded in the catalog tables of the current database.

When a user switches to SQL mode, user privileges are determined by the GRANT and REVOKE commands. The GRANT and REVOKE commands assign privileges to user IDs created with the PROTECT command.

When a protection scheme is in place (set up by a database administrator using the PROTECT command), all users must first log in to gain access to dBASE IV. Once users have logged in and changed into SQL mode, the operations they can perform in a database are dictated by the privileges assigned to their log-in user ID by GRANT and REVOKE. (See *Language Reference* for a description of the PROTECT command.)

The GRANT command allows you to specify privileges that correspond to display, update, and modification operations on tables or views. You can GRANT all privileges to a user with the ALL option (except on views), or specify one or more privileges in a < privilege list >, each privilege entry separated from the next by a comma. The privileges you can assign are:

- ALTER — ability to add columns to a table (cannot be granted on a view)
- DELETE — ability to delete rows from a table or view
- INDEX — ability to use the CREATE INDEX command (cannot be specified for a view)
- INSERT — ability to add rows to a table or view
- SELECT — ability to display rows from a table or view
- UPDATE — ability to update rows in a table or view, or only update specific columns

You can grant a set of privileges to a single table or view, or a list of tables and views. You may substitute a synonym for a table or view to assign privileges. If you specify privileges on a collection of tables or views, separate each table or view by a comma.

NOTE
When a table is created (and a protection scheme is in place), only the creator of the table and the SQLDBA user have privileges to use the table. Both the creator and SQLDBA user can GRANT privileges on tables. Only the creator of a table or the SQLDBA user can drop that table. The SQLDBA user has all privileges on all tables, views, synonyms, and indexes in every database.

You can grant privileges to a single user or to a list of users in the same GRANT statement. The names of users you specify in a user list must correspond to user IDs assigned in PROTECT. To grant privileges to all users, specify the PUBLIC keyword. GRANT privileges are cumulative, that is, you can add privileges to those a user already has. The same privilege may be granted to the same user more than once.

You can grant privileges for tables and views that you've created or those that you've been assigned a WITH GRANT OPTION privilege. Users who are provided the WITH GRANT OPTION can pass on those same privileges to other users. However, if a user's privileges are revoked, they are also revoked for any users to whom they were passed. Note, however, that the WITH GRANT OPTION cannot be specified in the same statement granting privileges to PUBLIC.

Examples

To assign all privileges for the Staff table to a user ID Janice, you could type:

```
SQL. GRANT ALL ON Staff
     TO Janice;
```

To assign specific privileges for the Staff table to a user ID Greg, you could type:

```
SQL. GRANT INSERT, UPDATE, DELETE ON Staff
     TO Greg;
```

See Also

REVOKE

The INSERT command inserts rows to a table or updatable view. There are two forms of the command; each can be run interactively or embedded in an SQL program. If dBASE IV is password-protected, you must have INSERT privileges on the specified table or view in which you want to insert rows.

Syntax

INSERT INTO <table name> (1)
[(<column list>)]
VALUES (<value list>);

INSERT INTO <table name> (2)
[(<column list>)]
<subselect>;

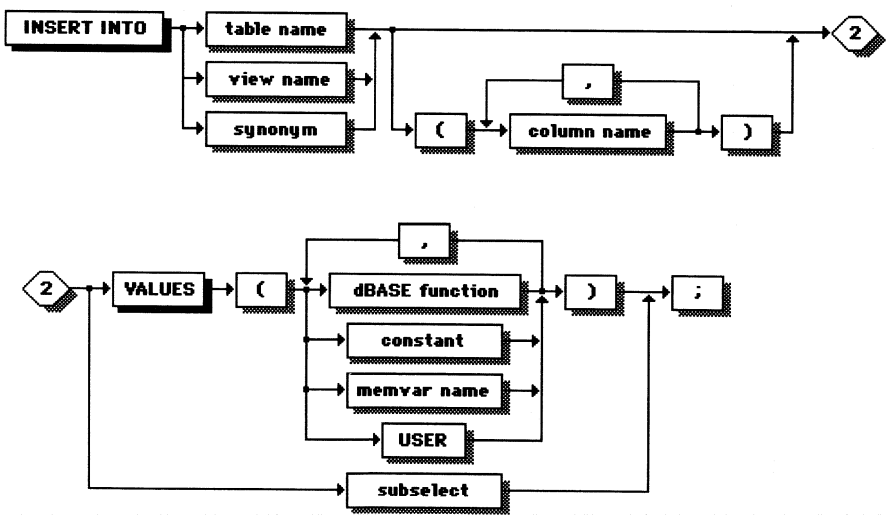


Figure 7-21 INSERT Command

Usage

Rows may be inserted into a table or an updatable view based on a single table (see CREATE VIEW). You may also specify a synonym that names a table or view. Values inserted into rows are either specified in a VALUES list or are taken from rows of a SELECT statement's result table. If you specified the WITH CHECK OPTION as part of a view's definition, row values that do not meet the conditions specified in the view definition will not be inserted.

Specifying a columns list in the INSERT statement is optional. You can use the columns list to specify the order in which values are entered in the VALUES list or are retrieved from a SELECT statement's result table. If you do not name the columns, the values are assumed to be in the same order as the table or view's data definition. You do not have to specify values for each column in a row you're inserting; however, if you only specify a partial VALUE list, indicate the columns in which the values should be inserted. When values are not specified for each column, unspecified character columns are initialized with blanks. Numeric fields are initialized with the value zero.

You may specify column values in a VALUES list as constants, character strings (enclosed in matching single or double quotes), dBASE memory variables, dBASE functions that return a value, or the SQL keyword USER. Each column value in a VALUES list must be separated by a comma.

The data types of values entered in a VALUES list must be compatible with those of the table or view data in which you're inserting the values. For example, the character string *"truck"* is compatible with a character type column, *CTOD("04/15/88")* or *{04/15/88}* is compatible with a date type column, and the value *15.4* is compatible with a numeric type column.

If you INSERT values into a table from rows retrieved by a SELECT statement, the SELECT statement must also return rows with columns compatible with the table or view in which values are inserted.

Examples

To insert a single row in the Staff table, you could type:

```
SQL. INSERT INTO Staff
     VALUES ("000013", "Ross", "Terry", CTOD("07/15/87"),
     "LOS ANGELES", "000001", 5678, 3.5);
```

The system displays the message **1 row(s) inserted.**

To insert multiple rows into a table from another table having the same columns as the Staff table, you could enter the following:

```
SQL. INSERT INTO LA
     SELECT *
     FROM Staff
     WHERE Location = "LOS ANGELES";
```

See Also

CREATE TABLE, CREATE VIEW, SELECT

The LOAD DATA command imports data from a non-SQL file and appends it to an existing SQL table in the current database. If dBASE IV is password-protected, you must be the creator of, or have INSERT privileges for, the table in which data is LOADed.

Syntax

LOAD DATA FROM [path] < filename >
INTO TABLE < table name >
[[TYPE] SDF/DIF/WKS/SYLK/FW2/RPD/DBASEII/
DELIMITED [WITH BLANK/WITH < delimiter >];

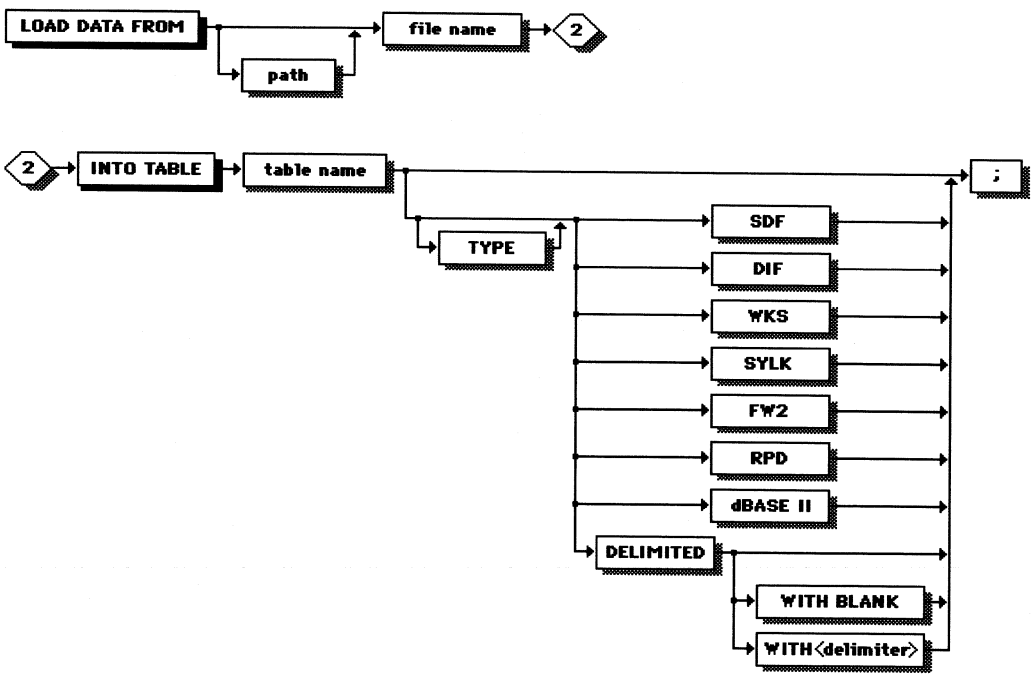


Figure 7-22 LOAD DATA command

LOAD DATA

Usage

You can use the LOAD DATA command to import data from non-SQL files into SQL tables. You can import the following types of files:

- dBASE II, dBASE III, dBASE III PLUS, and dBASE IV .dbf database files
- RapidFile database files (RPD)
- Framework II database and spreadsheet files (FW2)
- Delimited format ASCII files (DELIMITED WITH)
- System data format ASCII files (SDF)
- VisiCalc format files (DIF)
- MultiPlan spreadsheet format files (SYLK)
- Lotus 1-2-3 format files (WKS)

The LOAD DATA command supports the same file formats as the dBASE APPEND FROM command. (Refer to the APPEND FROM command in *Language Reference* for additional guidelines on importing data.)

The file from which data is loaded may be in the current DOS directory, or may be specified by an explicit DOS path. If you do not specify a file TYPE with the LOAD DATA command, import from a .dbf database file is assumed.

To use the LOAD DATA command, the SQL table to which data is appended must already exist. The table must have as many columns as are expected from the file being imported. Column definitions should correspond with the length and type of data expected from the imported file. The LOAD DATA command may truncate data that does not fit into columns when it is imported.

If you're importing data from a .dbf database file, the SQL table must have the same column name, length, and type definitions for the fields you want to import. For .dbf files, data is imported only for those fields that are also defined as columns in the specified SQL table. (Memo fields are not copied.) Also, to be used with the LOAD DATA command, the .dbf file must not be encrypted.

The DELIMITED option specifies transfer from a delimited format ASCII text (.txt) file. By default, all fields in the file are separated by commas and, in addition, character columns are enclosed in double quotes. Each record ends with a carriage return and line feed. If you specify DELIMITED WITH BLANK, each field is separated by a single space character instead of a comma. Also, character fields are treated like any other field and are not enclosed by any delimiter character. Finally, if you specify DELIMITED WITH <delimiter> each field is again separated by commas, and character fields are separated by a delimiter character of your choice (double quote is the default).

If you're importing data from spreadsheets or character delimited files, the SQL table into which you're loading data must match the format of the file you're importing. Spreadsheets should be stored in *row-major* order (as opposed to *column-major* order).

See Also

CREATE TABLE, UNLOAD DATA

OPEN

The OPEN command opens a previously declared cursor, executing the associated SELECT statement and positioning the cursor before the first row in the result table. This command is only run in embedded SQL mode.

Syntax

OPEN < cursor name > ;

Figure 7-23 OPEN command

Usage

OPEN executes the SELECT statement specified in the cursor's declaration. If dBASE IV is password-protected, you must have sufficient privileges to execute the SELECT statement in the associated DECLARE CURSOR statement. (See DECLARE CURSOR for more information on setting up a cursor to use with this command.)

After opening a cursor, use the FETCH command to retrieve data from the SELECT statement's result table. If you decide to update or delete data in rows retrieved from the open cursor, you will also need UPDATE or DELETE privileges on the table or view specified in the cursor declaration.

After executing an OPEN statement that executes the SELECT statement associated with the cursor, the system variable *Sqlcnt* contains the number of rows in the result table. (If the result table is empty, *Sqlcnt* will contain 0 and the system variable *Sqlcode* will contain a value of 100.)

Once the cursor is OPEN, you use the FETCH command to process rows of the cursor's result table. You can use *Sqlcode* to determine when there are no more rows to fetch. After each FETCH statement, the *Sqlcode* variable indicates the status of that operation. An *Sqlcode* value of zero indicates the FETCH was successful — the FETCH advanced the cursor and successfully transferred the row. If there was an error, *Sqlcode* contains a value of − 1. If the cursor was already on the last row when the FETCH statement is executed, *Sqlcode* contains a value of + 100 (indicating there are no more rows in the result table to process).

The FETCH command can only advance in a forward direction. However, closing the cursor with the CLOSE command and reopening it with another OPEN statement lets you use the FETCH command to begin processing rows again, starting with the first row.

NOTE
A DECLAREd cursor may be OPENed and CLOSEd as many times as you want. However, a cursor must be CLOSEd before it can be OPENed again.

Example

To open a cursor following a DECLARE CURSOR statement, you would type:

```
DECLARE x_ptr CURSOR
FOR SELECT *
FROM Staff
WHERE Hiredate > {9/01/86};
      .
      .
      .
OPEN CURSOR x_ptr;
```

See Also

CLOSE, DECLARE CURSOR, FETCH

REVOKE

The REVOKE command revokes access privileges previously granted on a table or view (or on synonyms naming tables or views). REVOKE works with user IDs created with the PROTECT command.

Syntax

REVOKE ALL [PRIVILEGES]/ < privileges list >
    ON [TABLE] < table list >
    FROM PUBLIC/ < user list > ;

The privilege list consists of one of more of the following keyword entries:

    ALTER
    DELETE
    INDEX
    INSERT
    SELECT
    UPDATE [(< column list >)]

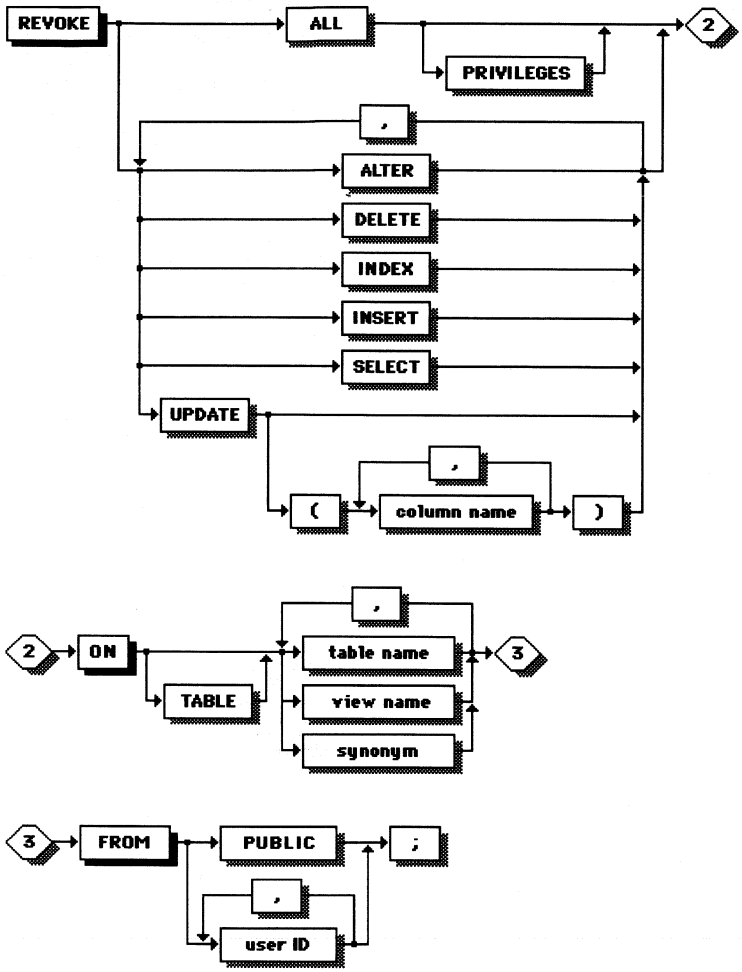


Figure 7-24 REVOKE command

Usage

SQL provides two commands, GRANT and REVOKE, to assign or restrict user privileges to tables or views in a database. All privileges or *authorization* assigned by GRANT and REVOKE are recorded in the catalog tables of the current database.

When a user switches to SQL mode, user privileges are determined by the GRANT and REVOKE commands. The GRANT and REVOKE commands assign privileges to user IDs created with the PROTECT command.

When dBASE IV is password-protected (set up by a database administrator using the PROTECT command), each user must first log in to gain access to dBASE IV. Once users have logged in and changed into SQL mode, the operations they can perform in a database are dictated by the privileges assigned to their log-in user ID by GRANT and REVOKE. (See *Language Reference* for a description of the PROTECT command.)

You can revoke all privileges by specifying the ALL keyword (except for views), or specify one or more privileges, separating each privilege keyword by a comma. The privileges you can revoke are:

- ALTER — ability to add columns to a table (cannot be specified for a view)
- DELETE — ability to delete rows from a table or view
- INDEX — ability to use the CREATE INDEX command (cannot be specified for a view)
- INSERT — ability to add rows to a table or view
- SELECT — ability to display rows from a table or view
- UPDATE — ability to update rows in a table or view, or only update specific columns

You can revoke privileges for a single table or view, or specify a list of tables and views. Also, you can revoke privileges from a single user, a list of users (separating each individual user ID with a comma), or from all users by entering the PUBLIC keyword.

You can only revoke privileges which you've previously granted (see note below). A user granted the WITH GRANT OPTION may only revoke privileges they previously granted. Privileges cannot be revoked from the creator of a table or view, or from the SQLDBA user ID.

NOTE
The SQLDBA user ID can revoke any privileges granted to users, including those it did not assign.

To remove the WITH GRANT OPTION, you need to revoke ALL privileges on a table or view and then grant them again without the WITH GRANT OPTION. The REVOKE command links privileges that users may have passed on to other users with the WITH GRANT OPTION. When you revoke privileges previously granted to users, they are also revoked from any users to whom they were passed.

The same privilege may be granted to the same user more than once. If there is one grantor (person granting privileges), the privileges are all revoked at once. If there is more than one grantor, the privilege must be revoked by each grantor. Also, REVOKE...FROM PUBLIC only revokes those privileges previously granted to PUBLIC, not those granted to individual user IDs.

Examples

To revoke privileges previously GRANTed to PUBLIC for the Staff table, you could type:

```
SQL. REVOKE ALL ON Staff
     FROM PUBLIC;
```

If you had previously assigned privileges to three users, you could revoke the INSERT and UPDATE privileges of one of the users by typing the following command:

```
SQL. REVOKE INSERT, UPDATE ON Staff
     FROM John;
```

See Also

GRANT

ROLLBACK

The ROLLBACK command restores a table or view to its contents prior to execution of the commands within a block specified by BEGIN...END TRANSACTION commands.

Syntax

ROLLBACK [WORK];

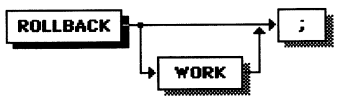


Figure 7-25 ROLLBACK command

Usage

You can only use this command to roll back changes from commands specified within a block defined by the two commands BEGIN TRANSACTION and END TRANSACTION. The WORK keyword is optional, and included only for compatibility with IBM's DB2 implementation.

NOTE
You may not specify the SQL commands ALTER, CREATE, DBCHECK, DBDEFINE, DROP, GRANT, or REVOKE in a transaction. Also, STOP DATABASE is not recommended since it commits *any current changes made in a transaction.*

Examples

To use the ROLLBACK command, you could first set up the following transaction in an SQL program file:

```
ON ERROR DO Recover
SET REPROCESS TO 15
BEGIN TRANSACTION
    UPDATE Staff
    SET Commission = Commission * 1.5
    WHERE State = "NY";
END TRANSACTION
ON ERROR
IF COMPLETED()
   @ 21,15 SAY "Transaction successfully completed"
ENDIF
```

The ON ERROR command is normally used to handle situations where the SQL transaction produces an error. To correspond to the previous example, you can set up a ROLLBACK operation in the following error recovery procedure:

```
PROCEDURE Recover
   @21,15 SAY "Your transaction has encountered an error condition"
   @22,15 SAY "Do you want to RETRY? (Y/N)" GET choice PICTURE "!"
   READ
   @ 21,15 TO 22,65 CLEAR
   IF choice = "Y"
     RETRY
   ELSE
     @21,15 SAY "Rolling back your transaction. Please wait."
     ROLLBACK;
   ENDIF
RETURN
```

RUNSTATS

This command updates database statistics in the SQL system catalog tables in the current database.

Syntax

RUNSTATS [< table name >];

Figure 7-26 RUNSTATS command

Usage

The RUNSTATS command updates statistical information stored in the SQL catalog tables for tables and indexes in the current database. This information helps SQL to determine the best ways to carry out database operations. Normally, you should run the RUNSTATS command after you've added or made changes to a table definition (using ALTER TABLE and CREATE TABLE), created new indexes (with CREATE INDEX), or added or changed data in more than ten percent of the rows in a table (using INSERT, DELETE, or UPDATE commands).

You can execute the RUNSTATS command to update the catalog table statistics for a single table or for all tables in the current database. If you do not specify a table, RUNSTATS will update the statistics for all tables in the current database.

NOTE
Refer to Chapter 8 for more information on the SQL catalog tables and specific details about the statistics updated with the RUNSTATS command.

The SELECT command produces a result table. It selects rows and columns from tables or views for display or input to another SQL statement. If dBASE IV is password-protected, you must be the creator of, or have SELECT privileges for, all tables and views referenced in the SELECT statement. You can execute SELECT statements interactively or embedded in SQL programs.

Syntax

SELECT < clause >
[INTO < clause >]
FROM < clause >
[WHERE < clause >]
[GROUP BY < clause >]
[HAVING < clause >]
[UNION < subselect >]...
[ORDER BY < clause > /FOR UPDATE OF < clause >]
[SAVE TO TEMP < clause >];

NOTE
The term subselect in the UNION clause refers to the SELECT, FROM WHERE, GROUP BY, and HAVING clauses (shown in bold type in the syntax listed above). A subselect can be joined with another subselect in the UNION clause or can appear in combination with other commands such as DELETE, INSERT, and UPDATE.

The following sections describe each clause of the SELECT command.

SELECT Clause

SELECT is a required clause that specifies the columns, SQL aggregate functions, or expressions to be included in a SELECT statement's result table and the headings over respective columns (unless SET HEADING is OFF). The expanded syntax of the SELECT clause is the following:

SELECT [ALL/DISTINCT] < column list > /*

Each column referenced in the SELECT clause must originate from tables or views (or synonyms naming a table or view) specified in the FROM clause. You must also separate each item specified in the column list with a comma.

Where the same column name is defined in more than one table of the FROM clause, you can qualify a column by the name of the table or view from which you want data to appear.

An asterisk (*) is used to specify that all columns from the tables or views specified in the FROM clause appear in the result. You can also qualify the asterisk with a table or view name, for example, *Clients.**. If alias names are specified in the FROM clause, these may replace the name of a table or view name to qualify a specified column.

ALL is an optional keyword (also the default) that specifies that all rows are displayed. It does not eliminate duplicate rows.

DISTINCT is an optional keyword that eliminates all but one of each set of duplicate rows (based on specified columns) from the SELECT statement's result table. When DISTINCT is used, the total concatenated length of the columns listed in the SELECT clause may not exceed 100 bytes. Also, you may only specify DISTINCT once in a SELECT statement.

When a SELECT statement is executed, headings are displayed over each column of the result table. The headings correspond to columns specified in the SELECT clause. Column headings specify the column from which data is taken; for multiple-table queries, they also indicate the table from which the column appears.

In addition to headings for columns that come directly from SQL tables, dBASE IV also displays column headings for other types of displayed data:

1. Columns derived from expressions (including calculated columns, dBASE functions, constants, and memory variables) are shown with the heading EXP followed by a number indicating the position of the expression in a SELECT clause.
2. Columns that display the result of SQL aggregate functions are shown with a corresponding heading (AVG, COUNT, MAX, MIN, or SUM) followed by a number indicating the position of the aggregate function in the SELECT clause. When you specify a dBASE function as a column in a SELECT clause, the corresponding column heading displays the function name.
3. Headings for columns specified in a GROUP BY clause are preceded by G_.

NOTE

1. *To compress the display of data in columns, particularly from multiple-table queries (joins), specify shorter alias names for tables in the FROM clause. You can also SET HEADINGS OFF to remove the headings display.*
2. *When a query displays a long result table, you can press* **Ctrl-S** *to stop or resume the scrolling and display of data. To display the data of a result table one screen at a time, SET PAUSE ON.*

INTO Clause

The INTO clause is an option used only in special cases (and only in embedded SQL mode) where a SELECT statement usually returns a single row. The expanded syntax of the INTO clause is the following:

INTO < memvar1 > [, < memvar2 > ,... < memvar*n* >]

Starting with the first column on the left, values from columns in the result table's single row are transferred into specified dBASE memory variables.

If you specify INTO, you cannot also specify the GROUP BY, HAVING, UNION, ORDER BY, FOR UPDATE OF, or SAVE TO TEMP clauses. If the SELECT statement you specify returns more than a single row, the first row is selected.

NOTE
If the SELECT statement does not return any rows, the memory variables specified in the INTO clause are not created. When using the SELECT...INTO command in a program, you may want to check the value of the Sqlcode before processing the memory variables named in the INTO clause.

FROM Clause

FROM is a required clause that identifies the tables or views (or synonyms that name tables and views) from which the result table is formed. The FROM clause also defines alias names for tables or views. Alias names may be referenced in both SELECT and WHERE clauses as is necessary in a self-join or correlated subquery (see Chapter 4). The expanded syntax of the FROM clause is the following:

FROM < table > / < view > [alias] [[, < table > / < view > [alias]]...]

Alias names are necessary, for example, to distinguish tables in correlated subqueries and in self-joins. They may be up to ten characters long, starting with a letter and otherwise containing letters, numbers, and underscores. For example, in *FROM Staff S*, *S* is defined as an alias for Staff.

NOTE
You may not specify the single-letter alias names of A through J.

WHERE Clause

The WHERE clause is an optional clause that specifies a search condition to restrict the rows that appear in the result table. The expanded syntax of the WHERE clause is the following:

WHERE [NOT] < search condition >
[AND/OR [NOT] < search condition > ...]

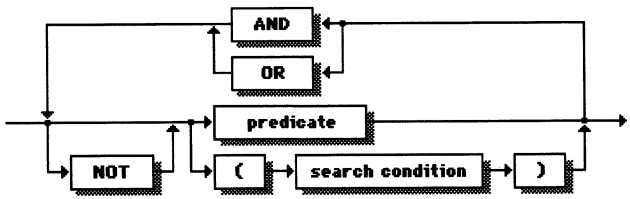


Figure 7-27 Search conditions

The search condition defines one or more conditions producing a combined result of true or false for a specific row. In constructing a search condition, you can specify any valid expression (which may include columns, dBASE functions, dBASE memory variables, operators, parentheses, and constants). For example, a simple search condition is *Name = "Zambini"*.

Besides *equals*, you may specify the following comparison operators in constructing a search condition:

Operator	Description
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
=	Equals
< > or #	Not equal
!	Reverses comparison logic of equals (=), greater-than (>), and less-than (<) operators

Comparisons in search conditions must be made between expressions of compatible data types. For example, a numeric column must be compared with another numeric value, a character type column with a character value or character string, and so on. Character strings specified in conditions must be enclosed in matching single or double quotes.

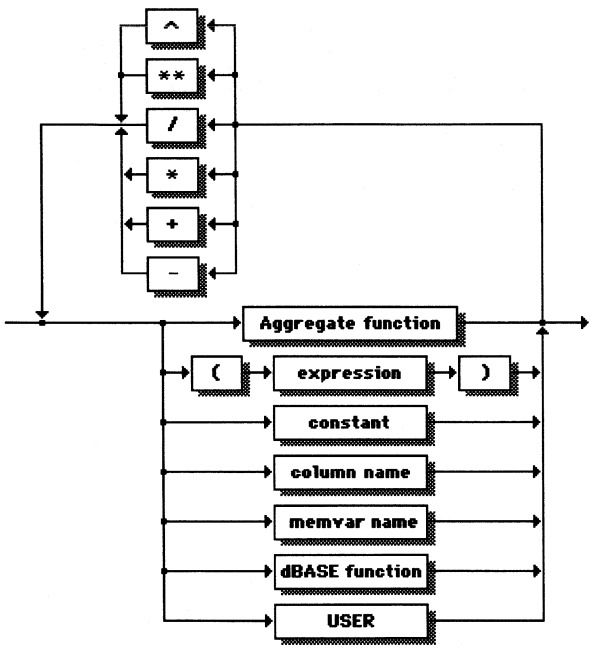


Figure 7-28 SQL expressions

You can use the logical operators AND, OR, and NOT to combine conditions and build more complex search conditions. For example, you could build a search condition *Name < > "Zambini" AND Age < 34*. In selecting rows, both conditions have to be satisfied for the overall condition to be true.

When multiple conditions are combined, individual conditions are evaluated in an order dictated by the precedence of the operators that connect them. NOT is applied first, followed by AND, and then OR. You may use parentheses to change the order in which conditions are evaluated.

NOTE
Comparison of character-type data (in columns or in a string) is made using the ASCII value of characters evaluated from left to right. Because ASCII values are used and ASCII upper-case letters have lower values than lower-case letters, the letter Z*, for example, will be evaluated as less than the value of letter* a*, and the letter* Z *will not equal* z*.*

Search conditions can specify predicates ALL, ANY, BETWEEN, EXISTS, IN, and LIKE. Also, particularly in combination with predicates, you can specify SQL aggregate functions, dBASE functions, or additional subselect statements in the WHERE clause. If you specify a subselect clause, it is evaluated first (unless the SELECT is correlated) so the WHERE clause can be tested. Figure 7-29 summarizes the forms in which predicates can be used.

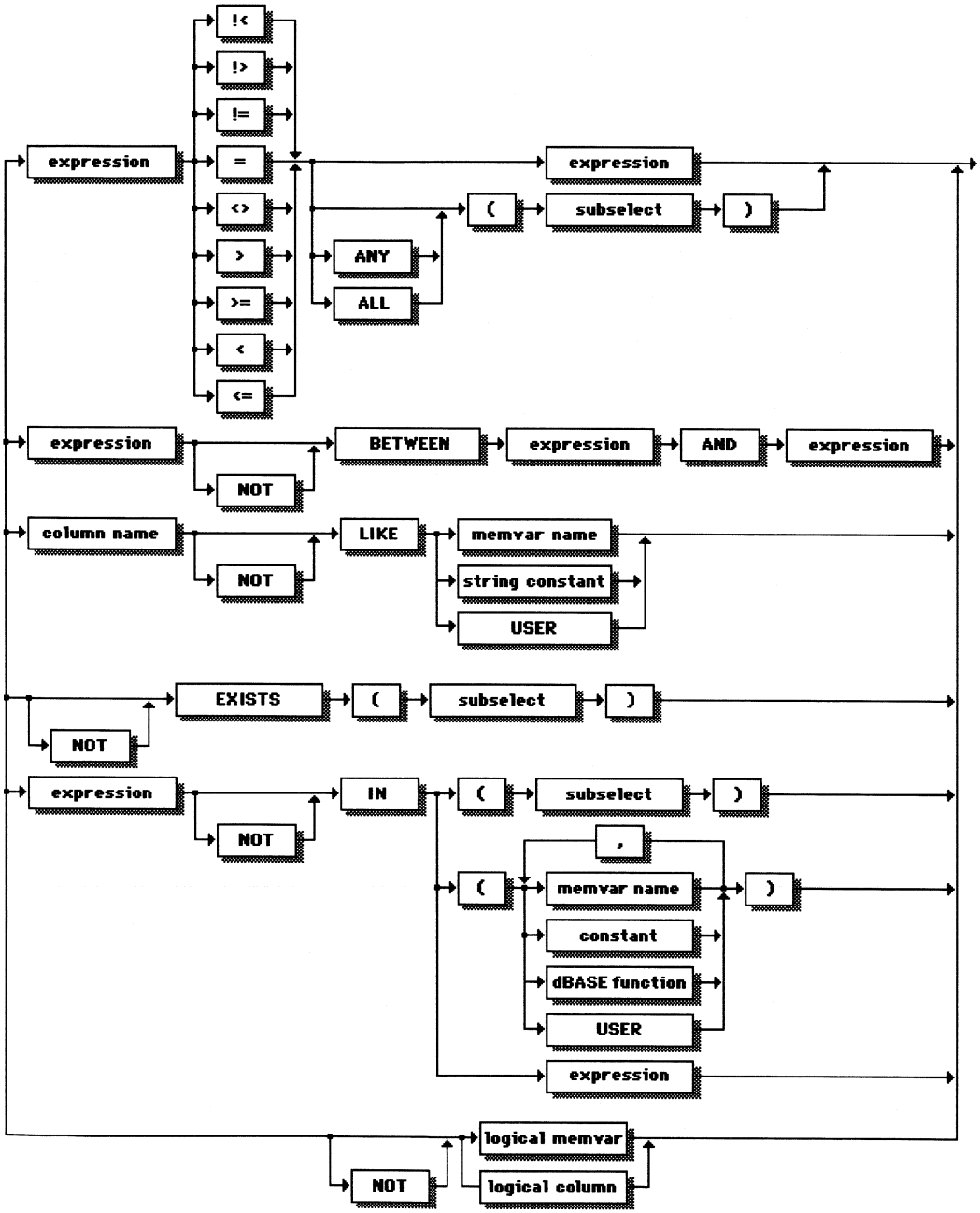


Figure 7-29 SQL predicates

GROUP BY Clause

The GROUP BY clause is an optional clause that groups rows in a result table by columns that have the same value. Each group is reduced to a single row. The columns in each grouped row either identify the group or are the result of an SQL aggregate function acting on the group. The expanded syntax of the GROUP BY clause is the following:

GROUP BY < column > [, < column > ...]

You must separate each item specified in the column list with a comma. Every column named in the associated SELECT clause must also be specified in the GROUP BY clause or as an argument to an SQL aggregate function. The total concatenated length of columns listed in the GROUP BY clause may not exceed 100 bytes. Also, you cannot list derived columns in the GROUP BY clause.

When a GROUP BY clause is added to a SELECT statement that includes SQL aggregate functions, any aggregate functions in the SELECT clause apply only to column values within each group, rather than to every column value within the table or view.

You cannot use the GROUP BY clause in SELECT statements that have an INTO clause. It also cannot be used in a subselect in which the FROM clause includes a view that has a GROUP BY or HAVING clause in its definition. You can specify the GROUP BY clause in a subquery, but in any single SELECT statement, there can be only one GROUP BY clause.

HAVING Clause

The HAVING clause specifies a search condition to restrict grouped rows that appear in the result table. The expanded syntax of the HAVING clause is the following:

HAVING [NOT] < search condition >
    [AND/OR [NOT] < search condition > ...]

The search condition specifies one or more conditions that evaluate to a true or false value for each group. Groups are those specified in a GROUP BY clause or, if GROUP BY is not specified, the entire result table is one group. If you do not specify a GROUP BY clause, all column names appearing in the SELECT clause must be named in SQL aggregate functions.

The search condition for the HAVING clause must specify an SQL aggregate function. Also, no correlated references are allowed in a HAVING clause subquery. The FROM clause of a subquery may not reference the same table as is named in the FROM clause of the outer query in which the HAVING clause is specified.

UNION Clause

The UNION clause is an optional clause that combines result tables of two or more subselects to produce a single result table. In producing the final result table, the UNION clause eliminates duplicate rows. The complete syntax of a query using the UNION clause is the following:

```
< subselect > [UNION < subselect > ]...]
   [ORDER BY < clause > ]
   [SAVE TO TEMP < clause > ];
```

Each subselect combined in a UNION statement must generate a result table with the same number of columns. Columns in each result table must have compatible data types (character, number, date, logical) and the same widths. The column names may be different. No memory variables or dBASE functions are allowed in the SELECT clauses of statements joined with UNION.

You can order the result of a UNION by placing an ORDER BY clause after the last subselect statement. In this case, you must specify integer numbers corresponding to the position of columns in the final result table to identify columns by which to order the result.

You may also follow a UNION with the SAVE TO TEMP clause. This option creates a temporary table available during the current SQL session and allows you to save the final result table as a dBASE .dbf file.

ORDER BY Clause

The ORDER BY clause is an optional clause that determines the order of rows in the SELECT statement's result table. You can specify ascending order (ASC, the default), descending order (DESC), or a combination of the two. The expanded syntax of the ORDER BY clause is the following:

```
[ORDER BY < column > / < integer > [ASC/DESC]
    [, < column > / < integer > [ASC/DESC]...];
```

The ORDER BY clause must specify columns that also appear in the SELECT statement's SELECT clause. The total concatenated length of columns listed in the ORDER BY clause may not exceed 100 bytes.

You may specify columns in the ORDER BY clause by their column name or by an integer value. If you specify an integer, it represents the relative position of the column in the SELECT clause. This allows you to order results derived from a constant, dBASE and SQL functions, or other expressions (including calculated columns).

Rows in the result table from an ordered SELECT statement are arranged by values in specified columns. If you specify more than one column in the ORDER BY clause, rows of the result table are first arranged by values in the first column. Rows within the same value in the first column are then arranged by values in the second column, and so on. You can specify the values to be placed in either ascending or descending order. Or, you can specify that ordering occur on some columns in ascending order and descending in others.

FOR UPDATE OF Clause

The FOR UPDATE OF clause is an optional clause (in SELECT statements specified as part of a cursor definition) that is used to specify columns that may be updated by issuing the UPDATE WHERE CURRENT OF command. The FOR UPDATE OF clause is used exclusively in embedded SQL mode. It may be included in interactive SQL statements, but will have no effect.

FOR UPDATE OF < column > [, < column > ...]

The columns named must be defined in a table or view specified in the FROM clause of the associated SELECT statement. You cannot use the FOR UPDATE OF clause with the INTO, ORDER BY, or SAVE TO TEMP clauses.

SAVE TO TEMP Clause

dBASE IV SQL provides the SAVE TO TEMP clause as an extension to SQL SELECT command. The SAVE TO TEMP clause saves the result table generated by a SELECT statement as a temporary table (that you can use during the remainder of a session in SQL mode) or saves the result table as a permanent dBASE file (in the current DOS directory). Unless you save them, tables created with the SAVE TO TEMP clause are temporary and are only available while the current database remains active during a current session at the SQL dot prompt or during execution of the highest level .prs program file.

You might find the SAVE TO TEMP clause useful in breaking up a complex operation such as a join or subquery into several more manageable operations. In addition, you can use the SAVE TO TEMP clause to create a permanent dBASE .dbf file that contains the rows and columns of the result table. The expanded syntax of the SAVE TO TEMP clause is the following:

SAVE TO TEMP < table name > [(< column > [, < column > ...])] [KEEP]

Specifying a list of columns is optional, unless the SELECT clause includes columns derived from an expression, dBASE or SQL function, memory variable, or constant. In that case, you must explicitly define a complete list of column names for the new table. You can also specify a column list to change the names of columns from their definition in the result table. If you do not specify a column list, the file is created with the same columns as those in the result table.

The data types and widths of SAVE TO TEMP columns are based on the corresponding data types and widths of columns or expressions specified in the SELECT clause.

The KEEP option saves the result table generated by the associated SELECT statement as an unencrypted .dbf database file, which you can then use in dBASE mode. For example, you can save the results of an SQL operation to a database file and then use the .dbf file to produce a report. Or, you could use DBDEFINE to define the .dbf file as a new SQL table.

NOTE
You cannot use the SAVE TO TEMP clause if the associated SELECT statement contains either FOR UPDATE OF or INTO clauses.

Usage

SELECT is the most widely used command in the SQL language. It can be used alone or combined with other operations such as INSERT, UPDATE, and DELETE. The syntax for specific operations of the SELECT statement in interactive and embedded SQL modes is described in the following three sections pertaining to different forms of the command. These forms are:

- Interactive and Display-Only SELECT
- Embedded Single-Row SELECT
- Embedded SELECT used with DECLARE CURSOR

NOTE
Refer to Chapters 3 and 4 for descriptions and examples of various applications using the SELECT command.

Interactive and Display-only SELECT

In interactive and embedded SQL modes, you can use the SELECT command to query and display (without updating) data from one or more tables. The expanded syntax of the SELECT command for these types of operations is the following:

```
SELECT [ALL/DISTINCT] < column list >
    FROM  < table > / < view >  [alias]
        [, < table > / < view >  [alias]...]
    [WHERE [NOT] < search condition >
        [AND/OR [NOT]       < search condition > ...]]
    [GROUP BY  < column >  [, < column > ...]]
    [HAVING [NOT] < search condition  >
        [AND/OR [NOT]  < search condition > ...]]
    [ORDER BY  < column > / < integer >  [ASC/DESC]
        [, < column > / < integer >  [ASC/DESC]...]]
    [SAVE TO TEMP  < table name >
         < column >  [, < column > ...]) [KEEP]];
```

Combined with UNION, the syntax is the following:

```
< subselect >  [UNION  < subselect > ]...
    [ORDER BY  < column > / < integer >  [ASC/DESC]
        [, < column > / < integer >  [ASC/DESC]...]
    [SAVE TO TEMP  < table name >
        ( < column >  [, < column > ...]) [KEEP] ];
```

NOTE
You can also include the FOR UPDATE clause in SELECT statements executed interactively. This allows you to build and test the results of a query before integrating it into an SQL program. You cannot, however, specify both the ORDER BY and FOR UPDATE OF clauses in the same SELECT statement.

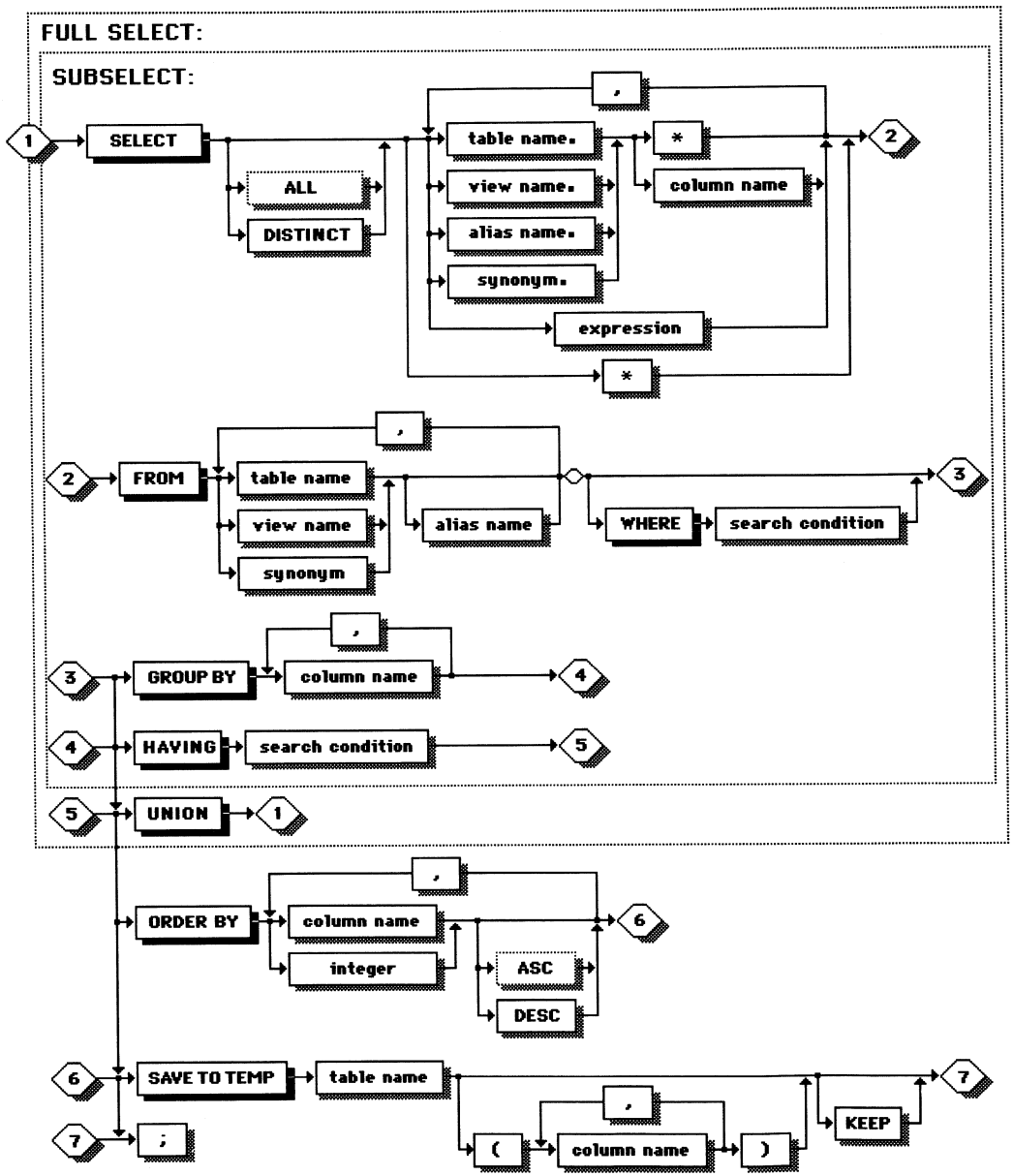


Figure 7-30 Interactive and Display-only SELECT

In addition to forming queries and using the SELECT statement to display data, you can use a subselect (SELECT, FROM, WHERE, and GROUP BY clauses) in a number of other applications.

- In the WHERE clause of a DELETE or UPDATE statement
- With the INSERT command to select rows to insert in a table
- In the WHERE or HAVING clauses of another SELECT statement
- In a CREATE VIEW command

Evaluation of Clauses

The SELECT command produces a result table by performing a series of operations on the tables and views specified in the FROM clause. Although dBASE IV SQL may actually perform the steps in a different order, you can view the operations as being performed in the following order:

1. FROM forms the Cartesian product of all rows from all tables and views.
2. WHERE eliminates rows that do not satisfy specified conditions.
3. GROUP BY groups selected rows and calculates results for all aggregate functions.
4. HAVING eliminates groups of rows that do not satisfy conditions placed on groups.
5. SELECT defines columns and performs calculations to be included in the result table.
6. UNION combines the result tables from each subselect and eliminates duplicate rows from the result table.
7. ORDER BY orders rows in the result table according to the values of specified columns.
8. SAVE TO TEMP stores the information in a temporary table with definitions in the current database's catalog table and optionally saves the result table as a permanent .dbf file.

Embedded Single-Row SELECT (with INTO)

In embedded mode, you can use this form of the SELECT command with the INTO clause to transfer column values into dBASE memory variables, where the SELECT statement only returns a single row. The INTO clause is used in special cases where the SELECT statement only returns a single row. Usually SQL aggregate functions are used. The expanded syntax of this form of the SELECT statement is the following:

```
SELECT [ALL/DISTINCT] < column list >
    INTO < memvar list >
    FROM < table > / < view > [alias]
        [, < table > / < view > [alias]...]
    [WHERE [NOT] < search condition >
        [AND/OR [NOT] < search condition > ...]]
```

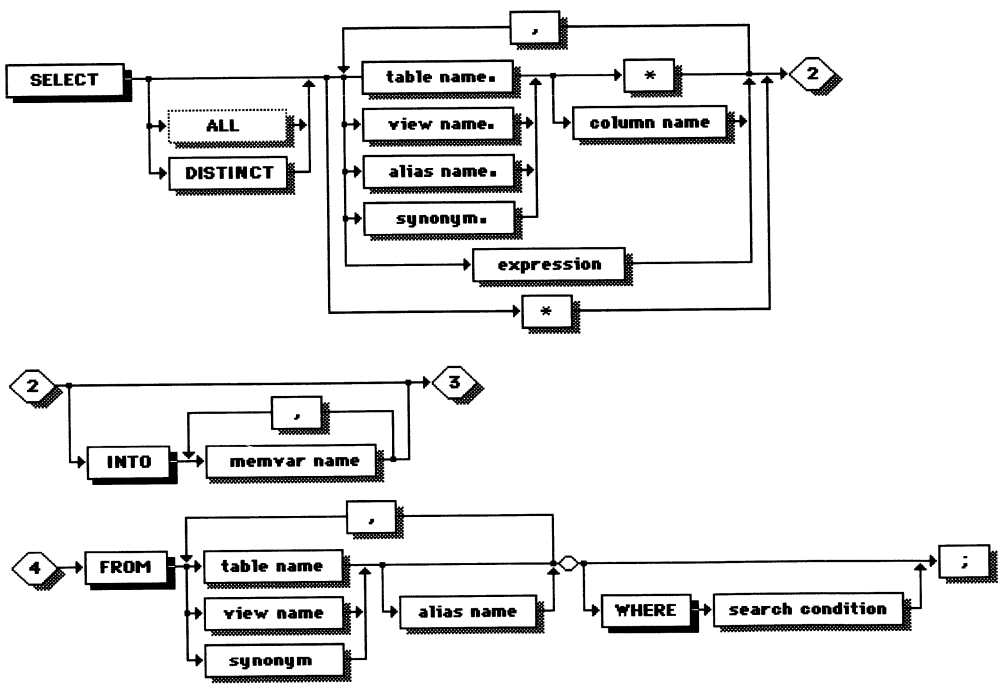


Figure 7-31 Embedded SELECT (with INTO)

NOTE
If you specify INTO, you cannot also specify the FOR UPDATE OF, GROUP BY, HAVING, UNION, ORDER BY, or SAVE TO TEMP clauses. Also, if the SELECT statement you specify returns more than a single row, only the first row is returned.

Starting with the first column on the left, values from columns in the result table's single row are transferred into dBASE memory variables. If the memory variables defined in the INTO statement do not already exist, they are created with the same data type as the column data being transferred into them.

Embedded SELECT Used with DECLARE CURSOR

This form of the SELECT command is used with the DECLARE CURSOR command in embedded mode. Rows returned by the SELECT statement are transferred into dBASE memory variables, one row at a time, using the FETCH command. You can also update or delete individual rows of the tables referenced in the SELECT statement using the DELETE and UPDATE commands.

```
SELECT [ALL/DISTINCT] < column list >
    FROM < table > / < view > [alias]
        [, < table > / < view > [alias]...]
    [WHERE [NOT] < search condition >
        [AND/OR [NOT] < search condition > ...]]
    [GROUP BY < column > [, < column > ...]]
    [HAVING [NOT] < search condition >
        [AND/OR [NOT] < search condition > ...]]
    [ORDER BY < column > / < integer > [ASC/DESC]
        [, < column > / < integer > [ASC/DESC]...]]/
    [FOR UPDATE OF < column > [, < column > ...]];
```

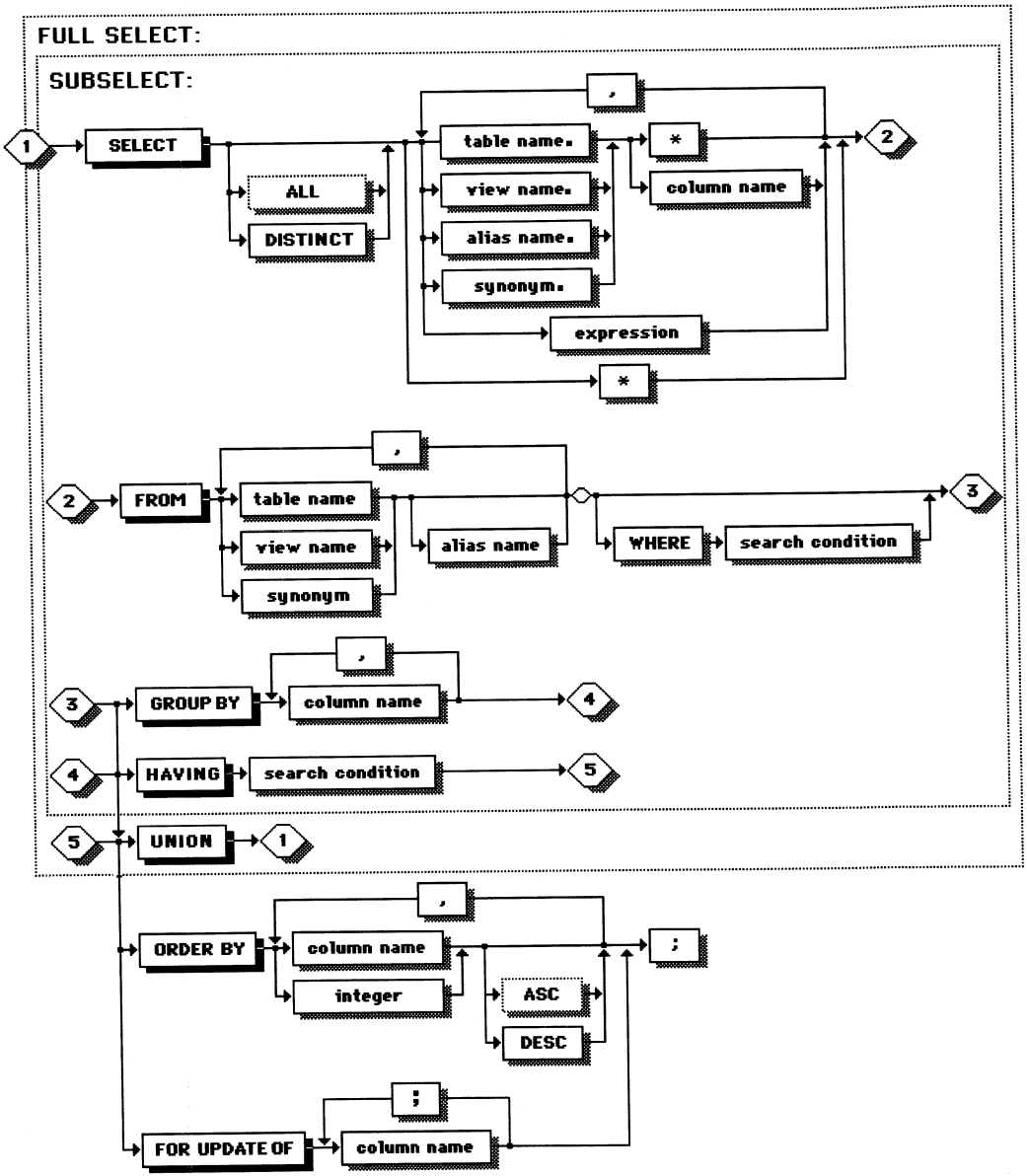


Figure 7-32 Embedded SELECT (with DECLARE CURSOR)

You can define a SELECT clause combined with the DECLARE CURSOR command that transfers column values from selected rows into dBASE memory values, one row at a time (using the FETCH command). In addition to transferring the data into dBASE memory variables, you can:

- Update column values using the WHERE CURRENT OF clause of the UPDATE command
- Delete rows using the WHERE CURRENT OF clause of the DELETE command

Examples

To display *all columns* from the Sales table, you could type:

```
SQL. SELECT *
     FROM Sales;
```

To display *specific columns* from the Customer table, you could type:

```
SQL. SELECT Company, Firstname, Lastname
     FROM Customer;
```

To display only *distinct columns* from the Inventry table, you could type:

```
SQL. SELECT DISTINCT Part_no, Descript, Unitcost
     FROM Inventry;
```

To display only *selected rows* for the part number data in the inventory for a Chicago location, you could type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     WHERE Location = "CHICAGO";
```

To display *rows from a table specifying multiple conditions* in the WHERE clause and using parentheses, you could type:

```
SQL. SELECT Part_no, Descript, Location, On_hand, Unitcost
     FROM Inventry
     WHERE (Location = "LOS ANGELES" OR Location = "NEW YORK")
     AND (On_hand < 20 AND Unitcost < 1000);
```

To *combine columns with expressions and a calculated field* in a query, you could type:

```
SQL. SELECT Staff_no, Firstname, Lastname,
     Salary*12, "(Yearly Salary)"
     FROM Staff;
```

To specify a dBASE function in the WHERE clause of a query, you could type:

```
SQL. SELECT Firstname, Lastname, Hiredate
     FROM Staff
     WHERE (DATE()-hiredate) > 365*5;
```

To *order rows* in Inventry using the ORDER BY clause, you could type:

```
SQL. SELECT Part_no, Descript, Location, On_hand
     FROM Inventry
     ORDER BY Descript ASC;
```

To use an *aggregate function* in a query, you could type:

```
SQL. SELECT COUNT(*)
     FROM Customers
     WHERE State = "NY";
```

To *group rows* of a query, you could type:

```
SQL. SELECT Part_no, SUM(On_hand)
     FROM Inventry
     GROUP BY Part_no;
```

To *combine data from two tables* with a join, you could type:

```
SQL. SELECT Order_no, Sale_date, Staff.Staff_no, Lastname
     FROM Sales, Staff
     WHERE Sales.Staff_no = Staff.Staff_no;
```

To *insert data* into a table using a SELECT statement, you could type:

```
SQL. INSERT INTO LA
     SELECT *
     FROM Staff
     WHERE Lastname = "LOS ANGELES";
```

See Also

DECLARE CURSOR, DELETE, INSERT, UPDATE

SHOW DATABASE

The SHOW DATABASE command lists SQL databases available to SQL users. This command can be run without a database in use.

Syntax

SHOW DATABASE;

Figure 7-33 SHOW DATABASE command

Usage

The SHOW DATABASE command lists the name of each database, the user ID of the person who created each database (on password-protected systems), the date each database was created, and the DOS path of the database directory. This database information is stored in the master catalog table Sysdbs, maintained in the dBASE IV SQL *home* directory (C:\DBASE\SQLHOME is the default) to keep track of each SQL database.

NOTE
The SQLHOME = setting in Config.db specifies the full DOS path for the SQL home *directory. (SQLHOME = C:\DBASE\SQLHOME is the default.)*

Example

To show all currently available databases, you can type:

```
SQL. SHOW DATABASE;
```

See Also

CREATE DATABASE, DROP DATABASE, START DATABASE, STOP DATABASE

START DATABASE

The START DATABASE command activates a database. All subsequent SQL commands use the tables and views (or synonyms that name a table or view) in that database.

Syntax

START DATABASE [< database name >];

Figure 7-34 START DATABASE command

Usage

Normally, START DATABASE is the first statement you execute before entering SQL commands interactively (unless you specify a default database in the Config.db file with SQLDATABASE = < database name >). Often, it is also the first SQL statement to execute after you start executing statements in an SQL program file.

Only one database may be active at a time. To change databases, execute the STOP DATABASE command to deactivate a database, then use START DATABASE to activate another. When you start a database, all cursors and temporary tables associated with the previous database will be dropped.

When you create a database, it is automatically activated.

NOTE
If you create a single-database SQL application, you can execute the START DATABASE command without specifying a database. (See "Developing SQL Applications" in Chapter 6.)

See Also

CREATE DATABASE, DROP DATABASE, SHOW DATABASE, STOP DATABASE

STOP DATABASE

The STOP DATABASE command closes the current database.

Syntax

STOP DATABASE;

Figure 7-35 STOP DATABASE command

Usage

You need to use this command before you can drop a database with the DROP DATABASE command. Executing the START DATABASE command automatically closes the current database.

See Also

CREATE DATABASE, DROP DATABASE, SHOW DATABASE, START DATABASE

UNLOAD DATA

The UNLOAD DATA command exports data from an SQL table to a non-SQL file. If dBASE IV is password-protected, you must be the creator of, or have SELECT privileges for, the table from which data is UNLOADed.

Syntax

UNLOAD DATA TO [path] < filename >
FROM TABLE < table name >
[[TYPE] SDF/DIF/WKS/SYLK/FW2/RPD/DBASEII/
DELIMITED [WITH BLANK/WITH < delimiter >]];

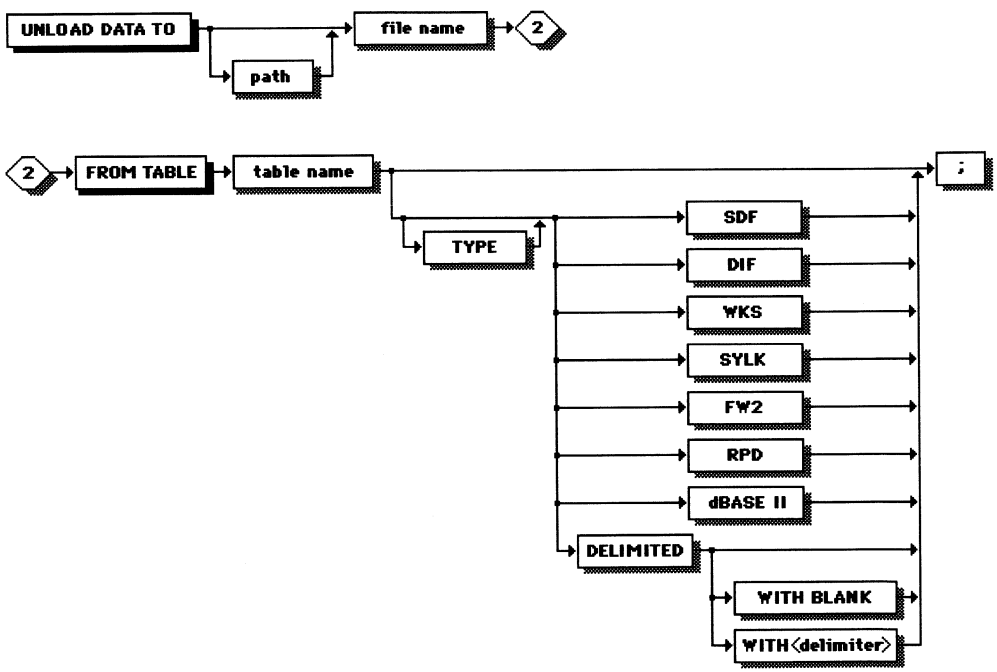


Figure 7-36 UNLOAD DATA command

Usage

The UNLOAD DATA utility command creates a new non-SQL file and copies data to it from an SQL table. You can export data to the following types of files:

- dBASE II, dBASE III, dBASE III PLUS, and dBASE IV .dbf database files
- RapidFile database files (RPD)
- Framework II database and spreadsheet files (FW2)
- Delimited format ASCII files (DELIMITED)
- System data format ASCII files (SDF)
- VisiCalc format files (DIF)
- MutiPlan spreadsheet format files (SYLK)
- Lotus 1-2-3 format files (WKS)

The UNLOAD DATA command supports the same file formats as the dBASE COPY TO command. Refer to the COPY TO command in *Language Reference* for additional guidelines on importing data.

The UNLOAD DATA command exports the SQL table to a file created in the current user directory (not the current database directory) unless you specify an explicit path.

Unless you specify a file TYPE with the UNLOAD DATA command, export to a dBASE database file is assumed. Fields in new files are created with names and definitions corresponding to columns in the SQL table. For data exported to spreadsheets, column names appear as column headers in the resulting file.

The DELIMITED option specifies transfer to a delimited format ASCII text (.txt) file. By default, all fields in the file are separated by commas and, in addition, character columns are enclosed in double quotes. Each record ends with a carriage return and line feed. If you specify DELIMITED WITH BLANK, each field is separated by a single space character instead of comma. Also, character fields are treated like any other field and are not enclosed by any delimiter character. Finally, if you specify DELIMITED WITH < delimiter >, each field is again separated by commas, and character fields are separated by a delimiter character of your choice (double quote is the default).

See Also

LOAD DATA

UPDATE

The UPDATE command changes the values of columns within specified rows of a table or view. If you specify a view, it must be updatable (see CREATE VIEW).

There are two forms of this command. The first runs in both interactive and embedded SQL modes. The second, used only in embedded SQL mode with a declared cursor, has a WHERE CURRENT OF clause that updates the row corresponding to the current location of the cursor.

Syntax

UPDATE < table name > / < view name > (1)
SET < column name > = < expression >
[, < column name > = < expression > ...]
[WHERE < search condition >];

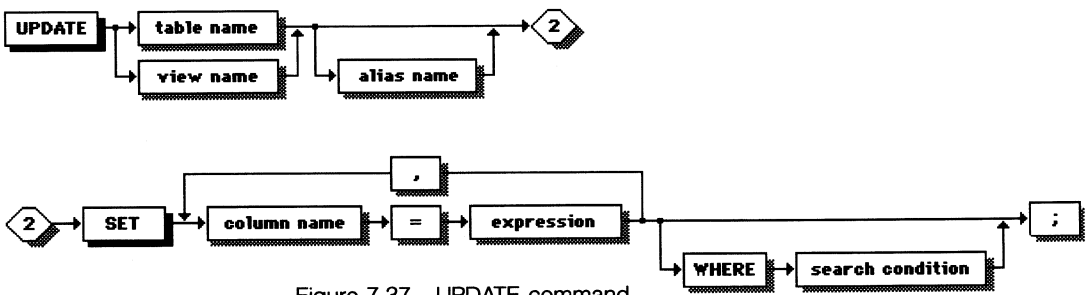


Figure 7-37 UPDATE command

UPDATE < table name > (2)
SET < column name > = < expression >
[, < column name > = < expression >, ...]
WHERE CURRENT OF < cursor name >;

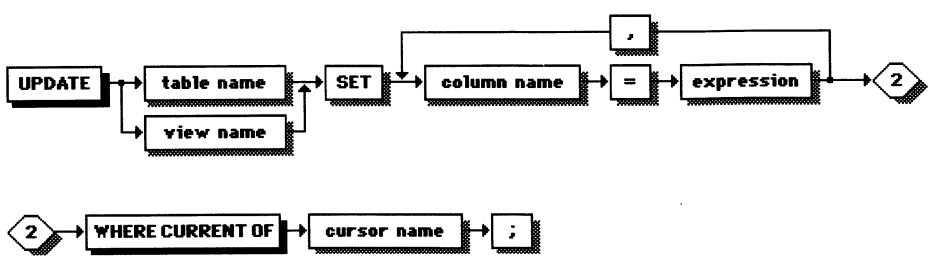


Figure 7-38 UPDATE command

Usage

If dBASE IV is password-protected, you must be the creator, or have UPDATE privileges for, each column in the table or view you want to update. When you update columns in rows of a view, the rows are also updated in the table on which the view is based.

UPDATE with WHERE Clause

You can specify UPDATE with a WHERE clause to select the rows you want to update. The WHERE clause may contain simple search conditions, combined search conditions, subselects, or subqueries. (See Chapter 3 for a description of search conditions.) However, if you specify a subselect in the WHERE clause, the table or view that you're updating cannot be referenced in the subselect.

NOTE
If you attempt to update a view defined with the WITH CHECK OPTION, column values will only be updated if the updated column values meet the conditions specified in the WHERE clause that defines the view.

UPDATE with the WHERE CURRENT OF Clause

You use the WHERE CURRENT OF clause in embedded SQL mode to update rows pointed to by a cursor. To update rows in a view using a cursor, the view specified in the SELECT statement defined with the DECLARE CURSOR must be updatable. See CREATE VIEW for details on updatable views.

To use this form of the UPDATE command, the cursor referenced in the WHERE CURRENT OF clause must already be defined in a DECLARE CURSOR statement. When the UPDATE statement is executed, the cursor must be open. The UPDATE statement will update columns in the row to which the cursor is pointing (the row last FETCHed).

TIP

1. *You can verify that the WHERE clause you specify with UPDATE updates the correct rows by first using it with the SELECT command.*
2. *In a program using a cursor, you can FETCH columns into dBASE memory variables and determine from those values whether you want to update the row. You can use the IF...ELSE program construct to test the FETCH command. You can then test whether you want to update the current row, or skip to the next one.*
3. *If you set up operations to update multiple rows in an SQL program file (particularly one that will run on a local area network), you may want to consider including transaction processing code in your program. You can use the Sqlcode status variable to check whether an operation completes successfully. If an operation fails, you can use the ROLLBACK command to return the table to its previous state.*

Example

To update the Commission column of the Staff table, you could type:

```
SQL. UPDATE Staff
     SET Commission = Commission * 1.25
     WHERE Hiredate < {07/05/82};
```

To use UPDATE with SQL cursors in embedded SQL mode, you can use UPDATE...WHERE CURRENT OF, which allows you to perform an update of the row pointed to by the cursor. For example, you could type the following example to generate invoices for previously uninvoiced orders:

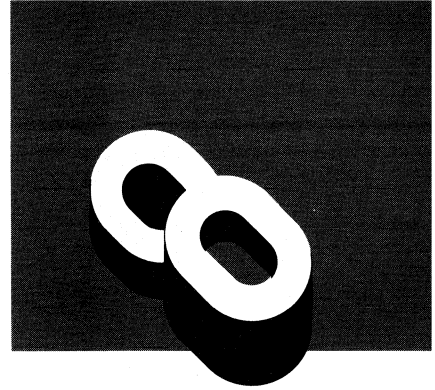
```
DECLARE Inv CURSOR FOR
SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
FROM Sales
WHERE NOT Invoiced
FOR UPDATE OF Invoiced;
   .
   .
   .
OPEN Inv;
   .
DO WHILE .T.
   FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
   IF SQLCODE = 0
         .
         .
         .
      * Print an invoice for order number in the the current row
      * If successful, change Minvoiced memory variable to .T.
      *
      DO Invoice WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
      *
      IF minvoiced
         UPDATE Sales
         SET Invoiced = .T.
         WHERE CURRENT OF Inv;
      ENDIF
   ELSE
      EXIT
   ENDIF
ENDDO
CLOSE Inv;
```

In this example, the Invoice program generates an invoice for the order number in the row pointed to by the cursor. After completing the invoice, the Invoiced column in the current row is updated to .T..

See Also

DECLARE CURSOR, FETCH, OPEN, SELECT

SQL Catalogs



dBASE IV SQL keeps track of all *objects* (tables, views, synonyms, indexes) in a database using *catalog tables*. dBASE IV SQL uses the information in the catalog tables when performing queries or updates.

A set of catalog tables is constructed for each database you create with the CREATE DATABASE command. A master catalog table, Sysdbs, is maintained in the SQL *home* directory (default is \DBASE\SQLHOME) to keep track of each SQL database. The Sysdbs table contains the name of each database, the user ID of the person who created the database, the date it was created, and the full DOS path to the database. You can use the SHOW DATABASE command to list the databases in the Sysdbs table.

The catalog for each database consists of ten SQL tables described below.

Table Name	Description
SYSAUTH	Describes the privileges held by users on tables and views — one row is entered for each user for each table for which a successful grant of one or more privileges has been made
SYSCOLAU	Describes the privileges held by users to update columns in a table or view — one row is entered for each GRANT operation for each grantee for each column for which UPDATE privileges were granted
SYSCOLS	Describes each column in every table and view including the catalog tables — one row per column (for every table or view in the current database)
SYSIDXS	Describes every index in the database and the table for which it's defined — one row per index
SYSKEYS	Describes every column (index key) used in each index — one row for each column for every index in the current database
SYSSYNS	Contains all synonym definitions for each table and view — one row for every synonym defined in the current database
SYSTABLS	Contains information describing every table and view — one row for every table or view defined in the current database

(continued)

Table Name	Description
SYSTIMES	Contains information used in multi-user environments to ensure that users have the latest copies of SQL catalog tables
SYSVDEPS	Describes the relation between views and tables in the current database (so that if tables are dropped, views based on those tables will also be dropped) — one row for each table used in a view definition in the current database
SYSVIEWS	Contains the definition of views and the limitations on use of each view — one or more rows for each view definition

NOTE
The catalog tables that SQL uses are similar to those defined for IBM's DB2 database product, with variations due to the differences between the mainframe and microcomputer environments.

You can display information from a system catalog table using the SELECT command, just like any other SQL table. For example, to display all the tables and views in the current database as stored in the Systabls table, you could type:

```
SQL. SELECT Tbname, Creator, Tbtype
     FROM Systabls;
```

This SELECT statement displays both tables and views in the current database. The Tbtype column indicates whether files are tables (T) or views (V). Letters D and K specify temporary tables created with the SELECT...SAVE TO TEMP command. (K indicates that the KEEP option was specified.)

While you can view the contents of the catalog tables, you cannot update them.

NOTE
The SQLDBA user ID has special privileges and can also directly update catalog tables with the UPDATE command. The SQLDBA user cannot, however, use the ALTER command to change the structure of the SQL catalog tables.

Updating the Catalog Tables

The catalog tables contain two types of information, SQL object definitions and statistics related to stored data. SQL object definitions specify information about SQL objects, databases, tables, views, synonyms, and indexes and are automatically updated by SQL commands. When you create or add objects to a database, or add or change table and view authorization privileges, the SQL catalog tables are automatically updated. Catalog tables are also updated for specific tables when you CREATE INDEXes or run the DBDEFINE command.

Catalog tables are not automatically updated for changes to data in tables, for example, those made with INSERT, UPDATE, or DELETE. Because SQL uses statistical information on tables and associated indexes to optimize its performance, you need to update the catalog tables periodically with the RUNSTATS utility command. Normally, you should run the RUNSTATS command when you've made changes to a table definition, just DBDEFINEd a .dbf file with an associated production .mdx file, or added or changed more than 10% of the rows in a table. You should not run any other database operations while using the RUNSTATS command.

Run the RUNSTATS utility command just as you would any other command. The syntax of this command is:

RUNSTATS [< table name >];

If you specify a table name, only statistics for that table are updated. Otherwise, statistics for all tables in the current database will be updated.

RUNSTATS checks the catalogs against the actual database tables and views and any other objects in the database, and updates the catalog tables for these changes. After RUNSTATS has finished, the message **RUNSTATS successful** displays if all updates were made. If a problem occurred, the message **RUNSTATS completed without catalog updates. Error/warning found** appears. If this message appears, follow the suggested remedies for messages preceding the RUNSTATS message and run RUNSTATS again.

The columns that RUNSTATS updates are the following:

Column	Table
COLCARD	Syscols
HIGH2KEY	Syscols
LOW2KEY	Syscols
FIRSTKCARD	Sysdxs
FULLKCARD	Sysidxs
NLEAF	Sysidxs
NLEVEL	Sysidxs
UPDATED	Systabls
CARD	Systabls
NPAGES	Systabls

NOTE
COLCARD, HIGH2KEY, and LOW2KEY are updated only for columns of numeric type (Decimal, Float, Integer, Numeric, or Smallint).

Description of Catalog Tables

The following sections describe each SQL catalog table and the information stored in each column of the catalog table.

Sysauth Table

The Sysauth table contains information about the privileges held by users on tables and views. This table is updated by the GRANT and REVOKE commands. One row of 84 bytes is added for every assignment of privileges to a user (IDs assigned by PROTECT or the PUBLIC user keyword) for each table in the current database. The number of rows generated by each successful GRANT is the product of the number of grantees specified in the TO list and the number of tables specified in the ON list.

Column	Type	Description
GRANTOR	Char(10)	User ID granting these privileges.
TNAME	Char(10)	Table to which privileges apply.
USERID	Char(10)	User ID of person to whom privileges are granted.
TBTYPE	Char(1)	T for tables and V for views. D and K specify temporary tables created with the SELECT... SAVE TO TEMP command (K indicates that the KEEP option was specified).
PSELECT	Numeric(8)	Integer value time stamp of the date when the privilege was granted.
PINSERT	Numeric(8)	Integer value time stamp of the date when the privilege was granted.
PDELETE	Numeric(8)	Integer value time stamp of the date when the privilege was granted.
PALTER	Numeric(8)	Integer value time stamp of the date when the privilege was granted.
PINDEX	Numeric(8)	Integer value time stamp of the date when the privilege was granted.
PUPDATE	Char(4)	Scope of privileges granted for UPDATE: ALL (all columns), SOME (specified columns), or NONE (if no privileges granted).
GRANTOPT	Char(1)	Set to G if a person is given GRANT privileges (allowing user to grant privileges to another user); otherwise set to Y.

Syscolau Table

The Syscolau table contains information about each user's UPDATE privileges for each column in a table or view in the current database. This catalog table is updated by the GRANT and REVOKE commands. One row of 50 bytes is added for each GRANT operation for each grantee (user ID or PUBLIC) and every column for which UPDATE privileges are granted. The number of rows generated by each successful GRANT is the product of the number of grantees specified in the TO list and the number of columns for which the UPDATE privilege is granted.

Column	Type	Description
GRANTOR	Char(10)	User ID of person assigning UPDATE privileges (or creator of view)
TNAME	Char(10)	Table or view to which privilege applies
GRANTEE	Char(10)	User ID to whom privilege is granted
COLNAME	Char(10)	Name of column for which UPDATE privilege is granted
AUTHTIME	Numeric(8)	Integer value time stamp of the date when the privilege was granted
GRANTOPT	Char(1)	Set to G if WITH GRANT privileges are assigned to user (allows user to grant privileges to another user); otherwise set to Y

Syscols Table

The Syscols table contains information that describes each column in every table and view in the current database. One row of 68 bytes is added for each column defined in each table or view in the database (including all columns in the catalog tables). New rows are automatically added when new tables or views are added, or when existing tables are altered to include additional columns.

Column	Type	Description
COLNAME	Char(10)	Column name
TBNAME	Char(10)	Table or view in which the column is defined
TBCREATOR	Char(10)	User ID of person who created the table
COLNO	Numeric(3)	Relative position (starting with one) of column within the table or view
COLTYPE	Char(1)	dBASE data type of column: C (character), N (numeric), D (date), F (float), or L (logical); SQL data types Smallint, Integer, Decimal, and Numeric are all recorded as type N
COLLEN	Numeric(3)	Length of column
COLSCALE	Numeric(2)	Number of places to the right of the decimal point
NULLS	Char(1)	Reserved for future use

(continued)

Column	Type	Description
COLCARD	Numeric(10)	Number of distinct values in the column (updated by CREATE INDEX and RUNSTATS for columns with COLTYPE of Numeric, Float, and Date)
UPDATES	Char(1)	Set to Y if updatable; N if not
HIGH2KEY	Char(8)	Second highest value of the column if the column is part of an index key (updated by CREATE INDEX and RUNSTATS for columns with COLTYPE of Numeric, Float, and Date)
LOW2KEY	Char(8)	Second lowest value of the column if the column is part of an index key (updated by CREATE INDEX and RUNSTATS for columns with COLTYPE of Numeric, Float, and Date)

Sysdbs Table

The Sysdbs catalog table is a *master* catalog table that keeps track of all SQL databases created with the CREATE DATABASE command. This table is different from the other catalog tables in that there is only one Sysdbs table, whereas a set of the other catalog tables are created for every new database. The Sysdbs catalog table is located in the SQL *home* directory (default directory \DBASE\SQLHOME). One row of 91 bytes is added for each SQL database created.

Column	Type	Description
NAME	Char(8)	Name of database
CREATOR	Char(10)	User ID of person who created the database
CREATED	Date	Date the database was created
PATH	Char(64)	DOS path of database directory

Sysidxs Table

The Sysidxs table describes every SQL index and the table for which it's defined. One row of 65 bytes is added for every index created in the current database.

Column	Type	Description
IXNAME	Char(10)	Name of index
CREATOR	Char(10)	User ID of person who created the index
TBNAME	Char(10)	Name of table on which index is defined
TBCREATOR	Char(10)	User ID of person who created table (may be different from CREATOR entry above)
UNIQUERULE	Char(1)	Indicates whether index was created as UNIQUE. Set to U if UNIQUE; otherwise D (to indicate duplicate columns allowed)
COLCOUNT	Numeric(3)	Number of columns in index key
CLUSTERING	Char(1)	Reserved for future use
CLUSTERED	Char(1)	Statistical entry
FIRSTKCARD	Numeric(5)	Statistical entry
FULLKCARD	Numeric(5)	Statistical entry
NLEAF	Numeric(5)	Statistical entry
NLEVELS	Numeric(3)	Statistical entry

Syskeys Table

The Syskeys table describes each column defined as an index key. One row of 38 bytes is added for each column specified in the index key of an index in the current database.

Column	Type	Description
IXNAME	Char(10)	Name of index
IXCREATOR	Char(10)	User ID of person who created the index
COLNAME	Char(10)	Name of column that defines index
COLNO	Numeric(3)	Relative position of column within table
COLSEQ	Numeric(3)	Relative position of index key within an index (.mdx) file
ORDERING	Char(1)	Set to A if Ascending (ASC) order; set to D if Descending (DESC)

Syssyns Table

The Syssyns table describes synonyms defined in the current database. One row of 41 bytes is added for each table or view synonym definition in the current database.

Column	Type	Description
SYNAME	Char(10)	Name of synonym
CREATOR	Char(10)	User ID of person who created synonym
TBNAME	Char(10)	Name of table or view for which synonym is defined
TBCREATOR	Char(10)	User ID of person who created the table or view for which the synonym is defined

Systabls Table

The Systabls table describes tables and views defined in the current database. One row of 74 bytes is added for each table or view in the current database.

Column	Type	Description
TBNAME	Char(10)	Name of table or view
CREATOR	Char(10)	User ID of person who created the table or view
TBTYPE	Char(1)	Set to T if object is a base table, V if object is a view, D or K if object is a temporary table (created with SELECT... SAVE TO TEMP command)
COLCOUNT	Numeric(3)	Indicates the number of columns in the table or view
CLUSTERRID	Numeric(10)	Reserved for future use
INDXCOUNT	Numeric(3)	Indicates the number of indexes defined for base tables; set to 0 for views
CREATED	Date	Date the table or view was created
UPDATED	Date	Last date that CREATE INDEX or RUNSTATS command was run; for views, contains the same date as the CREATED column
CARD	Numeric(10)	Statistical entry
NPAGES	Numeric(10)	Statistical entry

Systimes Table

The Systimes table contains information used only in multi-user environments to ensure that user copies of tables reflect the most recent catalog updates. One row of 27 bytes is maintained for each catalog table. (No new rows are ever added.) The DATEUP and TIMEUP columns are updated whenever a catalog table in the current database is updated.

Column	Type	Description
NAME	Char(10)	Catalog table name
DATEUP	Date	Date of last catalog table update
TIMEUP	Char(8)	Time of last catalog table update

Sysvdeps Table

The Sysvdeps table describes the tables (or other views) on which a view is defined. One row of 36 bytes is added for each table or view on which a view is defined in the current database.

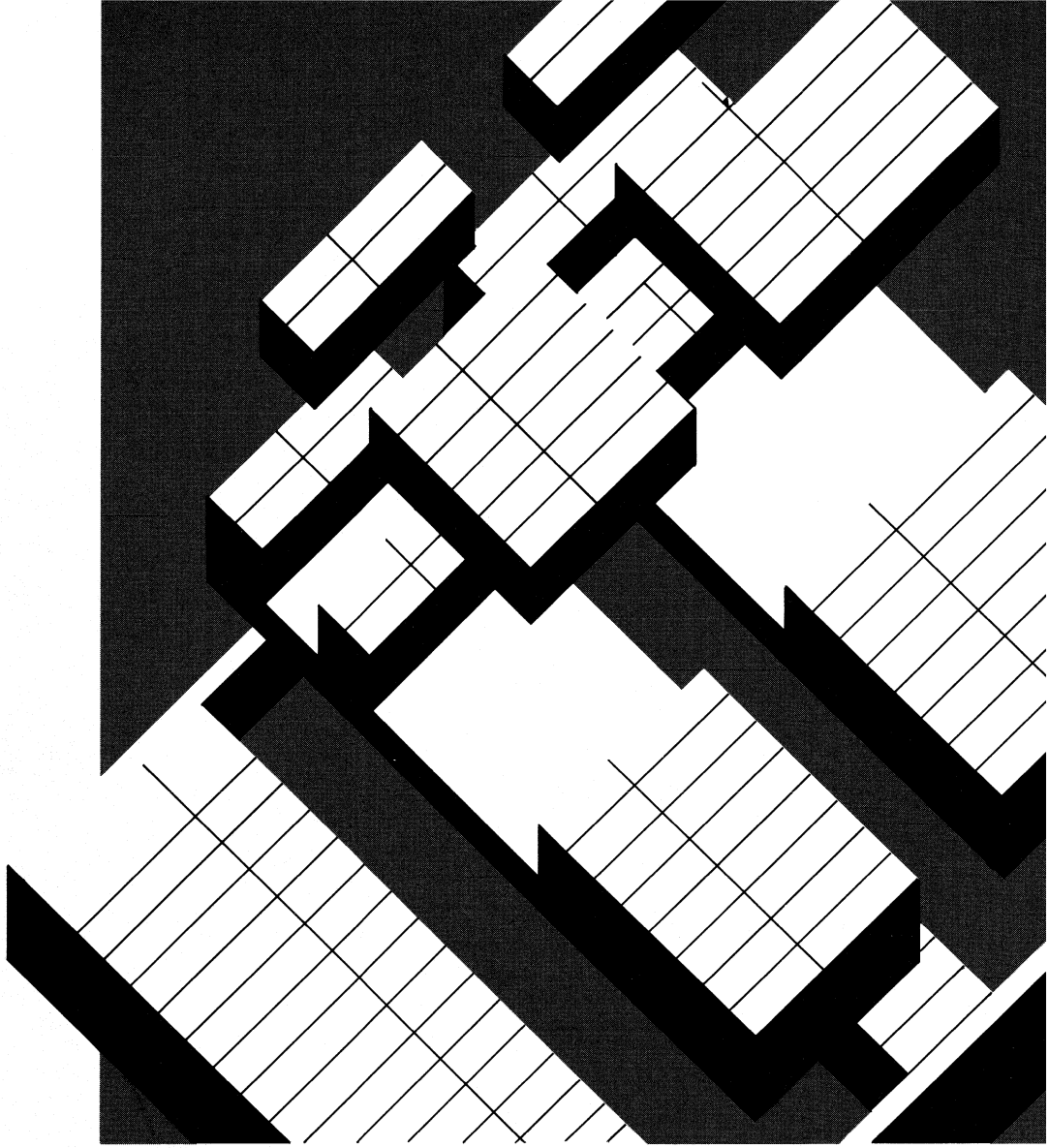
Column	Type	Description
VIEWNAME	Char(10)	Name of view
TBLNO	Numeric(5)	Relative position (starting with one) of table in view definition
TBNAME	Char(10)	Name of table or view on which the view is defined
CREATOR	Char(10)	User ID of person who created the view

Sysviews Table

The Sysviews table describes each view. One or more rows of 225 bytes is added for each new view defined in the current database. Multiple rows are entered in Sysviews when the length of the view definition text exceeds 200 bytes. When multiple rows are entered, the Seqno column value indicates the rows' relative position in the view definition.

Column	Type	Description
VIEWNAME	Char(10)	Name of view
CREATOR	Char(10)	User ID of person who created the view
SEQNO	Numeric(1)	Relative position (starting with 1) of this row in relation to other rows that also describe the same view
VCHECK	Char(1)	Set to Y if CHECK option enabled; otherwise set to N
READONLY	Char(1)	Set to Y if view is not updatable; otherwise set to N
JOIN	Char(1)	Set to Y if the view can be used in a join; otherwise set to N (if view definition contains GROUP BY clause)
SQLTEXT	Char(200)	Text of CREATE VIEW statement that defines the view

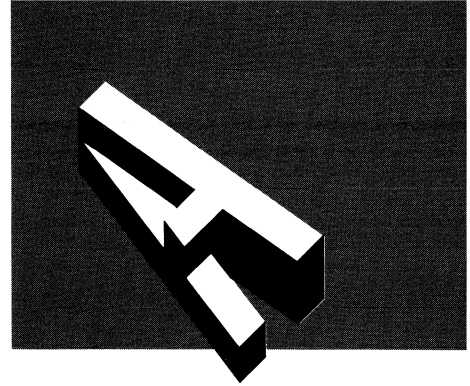
Appendixes



i 1 2 3 4 5 6 7 8 A

B C D In

SQL Error Messages



Below is the list of error messages for SQL commands in dBASE IV. The messages are arranged alphabetically.

Error numbers for messages that can be trapped with the ERROR() function are listed to the right of the error message in square brackets.

NOTE
If errors occur during the execution of an SQL program, follow the advice pertaining to the displayed error message. If the error still occurs, you can do the following:

1. *Run DBCHECK to check that database file and index structures match entries in corresponding SQL catalog tables.*
2. *Run RUNSTATS to update statistical information in the SQL catalog tables.*
3. *Recompile the programs that caused an error.*

*** allowed for COUNT function only**
An asterisk has been used as the argument to a function other than COUNT. The argument for functions AVG(), MAX(), MIN(), and SUM() must be a column name.

ADD clause expected in ALTER TABLE
ADD clause missing, misspelled, or misplaced in ALTER TABLE command.

Aggregate function not allowed in WHERE clause
The SQL aggregate functions AVG(), COUNT(), MAX(), MIN(), and SUM() are not allowed in a WHERE clause (except as part of the SELECT clause in a subselect). You may want to redefine the operation using a subselect. For example, *SELECT * FROM Staff WHERE Salary > AVG(Salary)* can be replaced by *SELECT * FROM Staff WHERE Salary > (SELECT AVG(Salary) FROM Staff)*.

Alias name already exists: < alias name >
You have already applied this alias name to another table at some earlier point in the statement. Use a different alias name.

All SELECT columns must be inside an aggregate function
When HAVING is used without GROUP BY, all SELECT columns must either be constants, or be inside an SQL aggregate function. Place all column names inside an aggregate function, or remove them from the SELECT clause.

All SELECT items must be GROUP BY columns or aggregate functions
A SELECT clause includes both aggregate functions (AVG(), COUNT(), MAX(), MIN(), and SUM()) and column names not included in functions. All columns not included in functions must be included in GROUP BY clause. Add a GROUP BY clause to the command. If the GROUP BY clause is already part of the command, add the missing SELECT clause column names to it.

ALTER privilege not granted
Grantor does not possess the ALTER privilege, or did not receive it WITH GRANT OPTION.

ALTER privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has been revoked already.

Ambiguous column name: < column name >
A column with the same name appears in two tables, both referenced in the current statement. Prefix ambiguous columns with their table name and a period, as in < table name > . < column name > .

An illegal table is referenced in a subselect FROM clause: < table name >
The table upon which the INSERT, UPDATE, or DELETE command is carried out cannot also be referenced in the FROM clause of a subselect.

Argument too long in CREATE INDEX, GROUP BY, ORDER BY, or SELECT DISTINCT
An index expression of more than 100 characters has caused an error on a CREATE INDEX command, or on a SELECT command including clauses (such as GROUP BY or ORDER BY) that use indexes. Specify a shorter index key for a CREATE INDEX command. If this error occurred on a SELECT command, it is an internal error. You can remove the GROUP BY or ORDER BY clause or the DISTINCT keyword.

BY clause is not supported and will be ignored
This is a warning, not an error. dBASE IV SQL does not support a BY clause in the GRANT and REVOKE statements. No action needed.

Cannot ALTER views: < view name >
Only base tables and their synonyms may be used in an ALTER TABLE statement. You can DROP a view, then redefine it to include additional columns.

Cannot CREATE INDEX/GROUP BY/ORDER BY/SELECT DISTINCT on a LOGICAL column
Index cannot be built on a column of LOGICAL data type and all these operations involve either user-defined or internal SQL indexes. Remove the LOGICAL type column.

Cannot DROP open database: < database name >
You tried to DROP the active database. First use STOP DATABASE and then use the DROP DATABASE command.

Cannot GRANT or REVOKE a privilege to yourself
A GRANT or REVOKE command specifies your own user ID in the TO or FROM clause. You may only GRANT and REVOKE privileges of other users. Check the user ID list and make sure it does not contain your own user ID. Make sure you logged in with your own user ID.

Cannot LOAD or UNLOAD DATA for views
LOAD and UNLOAD DATA commands must be used with SQL base tables or their synonyms. You may use a SELECT * statement with SAVE TO TEMP clause on the view and then UNLOAD from the temporary table.

Cannot mix ASC and DESC options in index key
If the keyword DESC is specified, it must be specified for all keys in the index.

Cannot read the header of the following file: < filename >
An error occurred during execution of a DBCHECK, DBDEFINE, or RUNSTATS because the header of a .dbf file could not be read. If the error occurred on DBDEFINE, remove the file with the bad file header from the database directory before re-executing DBDEFINE. If the error occurred on DBCHECK or RUNSTATS, you must DROP the table in SQL before you can successfully re-execute the command.

Catalog table Sysdbs does not exist
The Sysdbs table must be in the SQL home directory. Make sure that you have not deleted the Sysdbs.dbf file and that the SQLHOME directive in Config.db is correct. If SQLHOME has been changed from its Install setting, make sure the complete set of SQL system tables, including Sysdbs, has been transferred to the new SQL home directory.

Catalog tables are read-only: < table name >
Direct modification of the catalog tables is only allowed on a password-protected system by the SQLDBA superuser ID.

Catalog table(s) locked by another user: < table name > [1140]
Locks on the SQL catalog tables prevent completion of an operation. Wait and retry operation. If this message appears often, SET REPROCESS to a higher number of retries.

CHECK OPTION cannot be used with current view
The WITH CHECK OPTION clause cannot be used with a view that cannot be updated. See the CREATE VIEW command entry in Chapter 7 for rules on updating views.

Column data type doesn't match for table, column: < table >
An error occurred on a DBCHECK or RUNSTATS command because the data type indicated in the Coltype column of the Syscols catalog table doesn't match the data type of the actual column. First, copy the .dbf file and its indexes to another directory and DROP the SQL table. Then, copy the .dbf file and indexes back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Column is not updatable: < column name >
An INSERT or UPDATE command specifies a derived column or a column in a non-updatable view. See the CREATE VIEW command entry in Chapter 7 for rules on updating columns.

Column name already exists: < column name >
A command that names new columns has repeated the same name twice, or has attempted to use a name that already exists for some column in the table. Use a different column name.

Column name missing in AVG, MAX, MIN, or SUM function
The SQL aggregate functions AVG(), MAX(), MIN(), and SUM() require a column name as their argument.

Column name or number expected in ORDER BY
Only column names, or integers can appear in the ORDER BY list. After the keyword ORDER BY, either give the column(s) by which you want the result table ordered, or substitute integers indicating the columns according to their place in the SELECT clause.

Column not found in catalog table Syscols for table column: < table and column names >
An error occurred on DBCHECK or RUNSTATS command because a field in the .dbf file was not found as a column in the Syscols catalog table. First, copy the .dbf file and its .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Column/field names must be specified in SAVE TO TEMP clause
The SELECT returns columns derived from functions or constants, so SAVE TO TEMP column/field names must be specified. Add a list of column/field names enclosed in parentheses after the filename in the SAVE TO TEMP clause.

Column's position doesn't match for table, column: < table >
An error occurred on DBCHECK or RUNSTATS command because the information in the Syscols catalog table about a column's position in a table doesn't match the actual position of the column in the table. First, copy the .dbf file and its .mdx file to another directory and DROP the SQL table. Then, copy the .dbf file and its .mdx file back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Comma or right parenthesis expected
A list has been specified incorrectly. Check for commas between the items and for a right parenthesis at the end of the list.

Command cannot be executed within a transaction [2001]
SQL commands ALTER, CREATE, DBCHECK, DBDEFINE, DROP, GRANT, REVOKE, and RUNSTATS are not allowed in transactions. Run these commands outside of transactions.

Comparison operator or keyword expected
A comparison operator or keyword must follow the first column name or constant in a WHERE clause. The comparison operators are: =, >, <, > =, < =, < >, !=, !<, or !>. The comparison keywords are NOT, LIKE, IN, and BETWEEN.

Correlated subquery not allowed in HAVING clause
A subselect in a HAVING clause cannot reference the same table as the outer query. Restructure the query into two or more simpler queries, using SAVE TO TEMP clauses if necessary.

Cursor already open: < cursor name >
An OPEN command in a .prs program specifies a cursor that has already been OPENed. Precede the OPEN command with a CLOSE command.

Cursor declaration does not include the FOR UPDATE OF clause
An error has occurred during execution of an UPDATE WHERE CURRENT OF statement because the DECLARE CURSOR command that defined the cursor used for the UPDATE did not include a FOR UPDATE OF clause. Include the FOR UPDATE OF clause in the DECLARE CURSOR statement of the cursor used for the UPDATE. Note that the ORDER BY clause cannot be used when the FOR UPDATE OF clause is included.

Cursor name previously declared: < cursor name >
A DECLARE CURSOR statement contains a cursor name that has already been used in another DECLARE CURSOR statement in the same .prg program. Choose a new name for the cursor.

Cursor not declared: < cursor name >
No DECLARE CURSOR statement defines this cursor. Include a DECLARE CURSOR statement in your .prs program. Make sure the DECLARE CURSOR statement precedes the first statement that references the cursor.

Cursor not open: < cursor name >
A FETCH or CLOSE CURSOR command in a .prs program cannot be executed because the specified cursor has not been OPENed. Precede the FETCH or CLOSE command with an OPEN command.

Cursor not updatable: < cursor name >
The DECLARE CURSOR statement that defined the cursor included a SELECT DISTINCT, a UNION, aggregate functions, or included more than one table, or a non-updatable view. Use the cursor only in SELECTs. DECLARE another cursor for use in DELETE/UPDATE WHERE CURRENT OF and INSERT statements.

Data type keyword expected
In a CREATE TABLE or ALTER TABLE command, the data type of a column must be specified after the column name. The SQL data types are: SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, CHAR, LOGICAL, and DATE. If data type is already specified, check for misspelling, or misplacement of keyword.

Data type mismatch of corresponding columns in UNION operation
The corresponding columns of SELECT statements joined by the UNION keyword are not of matching data types. Check the data types of corresponding columns: SMALLINT, INTEGER, DECIMAL, NUMERIC, and FLOAT types are considered matching for UNION operations; however, they must specify the same length and number of decimal places. Other data types must match exactly.

Database does not exist **[2002]**
A database may have been DROPped since the program was compiled.

Database name already exists: < database name >
The database name used in a CREATE DATABASE command already exists as an SQL database. Use a different name for the new database.

DBCHECK and RUNSTATS must be used with base tables
A DBCHECK or RUNSTATS command specifies a view name or a non-.dbf filename instead of a base table name. To display the list of non-view tables in the active database, use the following query: *SELECT Tbname FROM Systabls WHERE Tbtype = "T".*

DBDEFINE completed; errors/warnings found [2010]
At least one file was not converted to an SQL table because of inconsistencies in the SQL catalog tables, or because of problems in the file's structure. Informational messages preceding the error indicate which tables have been successfully DBDEFINEd and which have not.

DELETE privilege not granted
Grantor does not possess the DELETE privilege, or did not receive it WITH GRANT OPTION.

DELETE privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has been revoked already.

Delimiter must be one character long or keyword BLANK
After the keywords DELIMITED WITH in a LOAD or UNLOAD utility, you must specify BLANK or a single character as delimiter.

Different table name is specified in cursor declaration: < cursor name >
The table specified in the DECLARE CURSOR statement is not the same as the table specified by the UPDATE/DELETE WHERE CURRENT OF statement using that cursor. Check table and cursor names to make sure both are identified correctly in both the DECLARE CURSOR and UPDATE/DELETE WHERE CURRENT OF statements.

DISTINCT must be followed by a column name not of type LOGICAL
When the keyword DISTINCT is used with the SQL aggregate functions, it must be followed by a non-LOGICAL type column name.

Duplicate user ID
The GRANT or REVOKE command user ID list contains duplicate user IDs. Remove duplicate user IDs.

Duplicated value in unique index — aborted [2000]
UPDATE or INSERT was not performed because it would have duplicated a value in a unique index key. Change the new value so that it is distinct from all current values. You may also DROP the index, but consider carefully before doing so, since unique indexes are used to insure database integrity.

Entry already exists in catalog table Syscols for the following table, column: < table and column names >
DBDEFINE has attempted to define a table for which an entry already exists in the Syscols catalog table. Error indicates severe inconsistencies in the database directory catalog tables; CREATE a new database directory, copy all non-catalog .dbf and .mdx files to a new database, and use DBDEFINE.

Entry already exists in catalog table Sysidxs for the following table, index: < table and index names >
DBDEFINE has attempted to define an index for which an entry already exists in the Sysidxs catalog table. Error indicates severe inconsistencies in the database directory catalog tables: CREATE a new database directory, copy all non-catalog .dbf files and related .mdx files to a new database, and use DBDEFINE.

Entry already exists in catalog table Syskeys for the following table, index, column: < table, index, column names >
DBDEFINE has attempted to define an index key for which an entry already exists in the Syskeys catalog table. Error indicates severe inconsistencies in the database directory catalog tables; CREATE a new database directory, copy all non-catalog .dbf and related .mdx files to the new database and use DBDEFINE.

Equal sign expected
Equal sign missing or misplaced after the keyword SET in an UPDATE statement.

Expression not allowed in GROUP BY/ORDER BY
Only column names are allowed in GROUP BY lists. Only column names or integers are allowed in ORDER BY lists. Remove all expressions from the clause.

File encryption error
File encryption error. Check the encrypted file. You can SET ENCRYPTION OFF and use an SQL LOAD or dBASE COPY TO command to create an unencrypted copy of a file.

File has invalid SQL encryption
DBCHECK or RUNSTATS has encountered a file that has invalid SQL encryption. Try to copy an unencrypted version of the file using the SQL UNLOAD command. If this is successful, erase the encrypted version, use DBDEFINE < filename > and then re-execute the DBCHECK or RUNSTATS command. If UNLOAD is not successful, the table must be DROPped in SQL before DBCHECK or RUNSTATS will execute successfully.

File is encrypted
DBDEFINE cannot be used with an encrypted file. If the file is dBASE-encrypted, SET SQL OFF. Use the dBASE COPY TO command to create an unencrypted copy of the file. Erase the encrypted version before re-executing DBDEFINE. If file is SQL-encrypted, SET ENCRYPTION OFF, and use the SQL UNLOAD command to create an unencrypted copy of the file. Erase the encrypted version before re-executing DBDEFINE.

File is not legal dBASE/SQL: < database filename >
An error has occurred during DBCHECK, DBDEFINE, or RUNSTATS execution because file header information indicates it is not a dBASE or SQL file. Make sure that you have not inadvertently copied a non-dBASE or SQL file with a .dbf extension. Remove the file causing the error from the SQL database, before re-executing the DBCHECK, DBDEFINE, or RUNSTATS utility command.

File is not SQL encrypted
DBCHECK or RUNSTATS has encountered a file that is encrypted, but not under the SQL encryption key. File must be decrypted by user and/or removed before re-executing the command.

Filename is same as existing synonym
Error occurs on DBDEFINE when the .dbf file specified in the command has the same name as an existing SQL synonym. Rename the .dbf file before re-executing DBDEFINE <filename>.

File not found in the current database
An error occurred during execution of DBCHECK, DBDEFINE, or RUNSTATS command because the file was not found in the currently active database directory. Check that you have not misspelled a filename, and that the named .dbf file is in the active database directory.

File open error: <filename>
File open error.

File read error: <filename>
File read error.

File seek error: <filename>
File seek error.

File write error: <filename>
File write error.

First argument of LIKE clause must be a CHAR column
The column name preceding the keyword LIKE in a WHERE clause does not identify a CHARACTER type column. Do not use LIKE with a non-CHARACTER type column.

Float value out of range
A float value greater than 10^{+38} or smaller than 10^{-38} cannot be entered.

GRANT OPTION ignored for UPDATE with column list specified
This is a warning, not an error. The grantee will not be able to GRANT the UPDATE privilege to others because the UPDATE privilege was GRANTed only for certain columns. No action needed. If the UPDATE privilege is GRANTed without a column list, then the WITH GRANT OPTION will become effective.

GRANT OPTION ignored when GRANT is TO PUBLIC
This is a warning, not an error. The GRANT OPTION is ignored because the privileges are being GRANTed to PUBLIC and so no further GRANTs will be necessary. No action needed.

GROUP BY clause needed
The SELECT clause includes both SQL aggregate functions (AVG, COUNT, MAX, MIN, or SUM) and column names. All column names that are not part of the aggregate functions must be included in a GROUP BY clause.

GROUP BY column(s) not specified in the SELECT clause
A column has been specified in the GROUP BY clause that is not included in the SELECT clause. Include the GROUP BY column(s) in the SELECT clause. All non-GROUP BY columns in the SELECT clause must be columns derived from aggregate functions.

GROUP BY, HAVING, ORDER BY, UNION, or FOR UPDATE OF clause not allowed with INTO
A SELECT statement in a .prs program includes both the INTO clause and GROUP BY, HAVING, ORDER BY, UNION, or FOR UPDATE OF clauses. A SELECT statement including the INTO clause should only return a single row and cannot include any of these clauses.

HAVING clause must include aggregate functions
The HAVING clause specifies a search condition for grouped data, and usually follows a GROUP BY clause. You must specify at least one of the aggregate function(s) AVG(), COUNT(), MAX(), MIN(), or SUM() in the HAVING clause.

Host variable count in INTO clause is not equal to number of SELECT items
A SELECT statement in a .prs program contains an INTO clause with a number of variables that does not match the number of columns in the SELECT clause.

In UNION, ORDER BY column(s) must be specified by integers
You may use an ORDER BY clause to order the result table of two or more SELECT statements joined by a UNION, but you must use integers instead of column names since column names may differ in the two SELECT clauses. Replace column names with integers (for example, to ORDER BY the first column, use 1, the second column, use 2, and so on).

Incompatible data types in comparison
The data types of columns, constants, or expressions in a comparison do not match. Check the data types of columns used. Check for missing quotes on string constants.

Incompatible data types in expression
Columns or constants with incompatible data types have been used in an expression. CHAR, LOGICAL, and DATE types cannot be mixed with the data types that hold numeric values, or with each other. Check the data types of columns used in the expression to make sure they are compatible with other columns, constants, or functions used in it.

Incomplete SQL statement
A name, keyword, operator, comma, or semicolon is missing or misspelled. Check for a missing semicolon at the end of the SQL command. Check the correct syntax of command.

Incorrect data type for arguments in dBASE function
Refers to a dBASE function used in an SQL statement. Check *Language Reference* for the correct data types in the dBASE function.

Incorrect number of arguments in dBASE function
Refers to a dBASE function used in an SQL statement. For example, this error might occur when the CTOD() function is used with an invalid value for the date, as in CTOD("99/99/99"). Check *Language Reference* for correct syntax and usage of the dBASE function.

Incorrect number of INSERT items
The number of values from the VALUES clause or the subselect doesn't match the number of columns specified in the column list (or in the table if no column list was given). Check that the column list includes the exact columns for which data is to be INSERTed. Check the VALUES list or subselect SELECT clause to make sure the correct number of values are provided.

Index name already exists
The index name used in a CREATE INDEX command already exists as an SQL index. Use a different index name for the new index.

INDEX privilege not granted
Grantor does not possess the INDEX privilege, or did not receive it WITH GRANT OPTION.

INDEX privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has already been revoked.

INSERT privilege not granted
Grantor does not possess the INSERT privilege, or did not receive it WITH GRANT OPTION.

INSERT privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has been revoked already.

Insufficient memory
Insufficient memory available for allocation. Simplify queries by breaking long statements into several shorter ones.

Insufficient privilege
You do not have the privilege to perform the requested operation on the table or view specified. Have the creator of the table/view GRANT you privileges on it. Anyone who received the privileges WITH GRANT OPTION may also GRANT them to you.

Insufficient privilege to CREATE VIEW
To CREATE a view, you must have SELECT privileges for every table on which the view is based. Delete the table for which you lack SELECT privileges from the view definition. Or, have someone GRANT you SELECT privileges on the table.

Integer expected
A non-integer value was encountered. Replace with an integer value.

Internal SQL error: < error number >
Simplify or restate your query and retry the operation.

Internal SQL utility error: < error number >
Internal SQL error occurred during a DBCHECK, DBDEFINE, or RUNSTATS operation.

INTO clause not allowed in cursor declaration
The SELECT in a DECLARE CURSOR statement cannot include the INTO clause. Delete the INTO clause.

Invalid argument for aggregate function
You have used an asterisk (*) or a column of a disallowed data type in the AVG(), MAX(), MIN(), or SUM() function. Columns of data type Logical cannot be used in SQL functions. The argument of SUM() and AVG() functions must be a column or expression that yields a numeric value.

Invalid character
Only letters, digits, and underscores are allowed in object and column names. Numbers may contain only digits and decimal points. Delete any disallowed characters.

Invalid character length
The length of a character string either equals zero, or exceeds the maximum length of 255 for dBASE character strings.

Invalid column number in ORDER BY clause
A column number is not an integer, or is greater than the number of columns returned by the SELECT clause. Make sure the column number corresponds to the column's placement in the SELECT clause (for example, the first column is designated by the number 1, the second by 2, and so on).

Invalid constant
An SQL statement is expecting a constant but no value is specified. Check for empty parentheses in values lists.

Invalid COUNT argument
Count argument must be (*) or (DISTINCT/ALL < column name >).

Invalid decimal length
The CREATE TABLE command incorrectly specifies the length of a Numeric or Decimal type column. The width and scale of Numeric and Decimal type columns is specified as (x,y). Both column types may be specified as (1,0). If y is not 0, then y may not be greater than x − 2. For both types, y may not exceed 18.

Invalid file type specified
In the LOAD and UNLOAD statements file type may be specified as SDF, DIF, WKS, SYLK, FW2, RPD, DBASEII,or DELIMITED. Check the file type. If the file LOADed from or UNLOADed to is a .dbf file, the type need not be specified.

Invalid filename
An invalid filename appears in the LOAD or UNLOAD statement. Check that the filename (and path) are correctly specified. When the file type is .dbf, the extension need not be specified.

Invalid INSERT item
Items in the value list following the keyword VALUES cannot be column names or compound expressions involving arithmetic operators. Check the value list to make sure it contains only the following items: constants, dBASE functions, memory variables, or USER.

Invalid logical predicate
This message occurs when you use comparison operators or keywords other than the unary equal (=) or not equal (! = , < > , or #) operators in a predicate evaluating a logical type column, memory variable, or array element.

Invalid password: < user ID >
Invalid password in file header.

Invalid string operator
Operators other than plus and minus cannot be used with strings. Remove disallowed operator. Also check that you have not mistakenly identified something as a string by inadvertently starting a name with a quotation mark.

Invalid SYSTIME.MEM
The SYSTIME.MEM time stamp in the active database directory is invalid. Make sure you have not accidentally overwritten the Systime.mem file. Try copying it from a backup of the database.

Invalid unary operator
An operator has been inappropriately applied to a non-numeric value. Check for typographical errors or operators in front of CHAR, LOGICAL or DATE columns.

Keys not unique, index not created **[2006]**
A CREATE INDEX command specified the UNIQUE option, but the column(s) on which the index was to be built contain duplicate values. Add a column to the index key, or DELETE rows containing duplicate index key values before re-executing the CREATE INDEX with UNIQUE option.

Keyword AND expected
The keyword AND was expected after first argument in BETWEEN clause. Check that the keyword AND is not missing, misspelled, or misplaced.

Keyword AS expected
The keyword AS is missing, misspelled, or misplaced in CREATE VIEW command. AS should follow the view name (and optional column list) and precede the subselect.

Keyword ASC or DESC, comma, or right parenthesis expected
The CREATE INDEX command is incomplete. Add the ASC or DESC option if desired after the column name(s) in the column list (use commas to separate items in the list) and be sure a right parenthesis ends the column list.

Keyword ASC or DESC, comma, or semicolon expected
An ORDER BY clause is incomplete. ASCending, or DESCending order may be specified after the column name(s) in the ORDER BY clause. ASC is the default. The statement must be terminated with a semicolon (;).

Keyword BY expected
The keyword ORDER must be followed by the keyword BY. Check that BY is not missing, misspelled, or misplaced.

Keyword CHECK expected
The keyword CHECK is missing, misspelled, or misplaced in the WITH CHECK OPTION clause of a CREATE VIEW command.

Keyword CURSOR expected
The keyword CURSOR is missing, misplaced, or misspelled in the DECLARE CURSOR statement in a .prs program.

Keyword DATA expected
Missing keyword DATA in LOAD or UNLOAD statement. Check that the word DATA is directly after the word LOAD or UNLOAD and that it is spelled correctly.

Keyword DATABASE expected
The keyword DATABASE is missing, misplaced, or misspelled in a START, STOP, or SHOW DATABASE command.

Keyword DATABASE, TABLE, INDEX, SYNONYM, or VIEW expected
A keyword is missing or misspelled after CREATE or DROP keywords. Use DATABASE, TABLE, INDEX, SYNONYM, or VIEW depending on the kind of object to be CREATEd or DROPped.

Keyword FOR expected
The keyword FOR is missing, misplaced, or misspelled in the CREATE SYNONYM or DECLARE CURSOR command.

Keyword FROM expected
The keyword FROM is missing, misplaced, or misspelled in a LOAD, UNLOAD, SELECT, REVOKE, or DELETE command.

Keyword GRANT expected
The keyword GRANT is missing, misplaced, or misspelled in the WITH GRANT OPTION clause of a GRANT statement.

Keyword INDEX expected
The keywords CREATE UNIQUE must be followed by the word INDEX. Check that the keyword INDEX is not missing, misspelled, or misplaced.

Keyword INTO expected
The keyword INTO is missing, misplaced, or misspelled in a FETCH, INSERT, or LOAD DATA statement.

Keyword not allowed in interactive mode
A keyword used during an interactive SQL session can only be used in .prs programs. All keywords related to the use of SQL cursors are used only in programs. Replace the command with an interactive SQL command.

Keyword OF expected
The keyword OF is missing, misplaced, or misspelled in a FOR UPDATE OF or WHERE CURRENT OF clause.

Keyword ON expected
The keyword ON is missing, misspelled, or misplaced in a CREATE INDEX, GRANT, or REVOKE command.

Keyword OPTION expected
The keyword OPTION is missing, misspelled, or misplaced in the WITH GRANT OPTION clause of a GRANT command, or in the WITH CHECK OPTION clause of a CREATE VIEW statement.

Keyword SELECT expected
The keyword SELECT is missing, misplaced, or misspelled after the keyword UNION, or in a CREATE VIEW command after the keyword AS. The missing SELECT keyword is the first word of the required subselect used in these statements.

Keyword SELECT missing in DECLARE CURSOR statement
The keyword SELECT is missing, misplaced, or misspelled in a DECLARE CURSOR statement. A SELECT statement must follow the keyword FOR.

Keyword SET expected
The keyword SET is missing, misplaced, or misspelled in UPDATE statement.

Keyword TABLE expected
The keyword TABLE is missing, misspelled, or misplaced in an ALTER TABLE, LOAD, or UNLOAD command.

Keyword TEMP expected
The keyword TEMP is missing, misplaced, or misspelled in the SAVE TO TEMP clause of a SELECT statement.

Keyword TO expected
The keyword TO is missing, misspelled, or misplaced in a GRANT or UNLOAD command or the SAVE TO TEMP clause of a SELECT command.

Keyword UPDATE expected
The keyword UPDATE is missing, misspelled, or misplaced in the FOR UPDATE OF clause in the DECLARE CURSOR statement.

Keyword VALUES or SELECT expected
The INSERT statement must include either a VALUES clause or a subselect. Check that one of the keywords is included and that it is not misspelled or misplaced.

Keyword WITH expected
The keyword WITH is missing, misplaced, or misspelled in the LOAD or UNLOAD utility where delimiters are specified. When data is LOADed to or UNLOADed from dBASE files, the TYPE and WITH clauses need not be specified.

Keywords BETWEEN, LIKE, or IN expected
A WHERE clause is incorrectly specified. Check for correct syntax of the desired form of the WHERE clause.

Left parenthesis missing
A left parenthesis is missing before the beginning of a list. Check the syntax of the command for required parentheses.

Length of column doesn't match for table, column: < table and column name >
An error occurred on a DBCHECK or RUNSTATS command because the column length indicated in the Collen column of the Syscols catalog table does not match the actual length of the column. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Memory variable and dBASE function not allowed in SELECT with UNION
All SELECT columns must be named columns or constants when UNION is used. Remove memory variables, dBASE functions, and array references from SELECT clauses.

Memory variable or column name undefined or memory variable of invalid type **[2007]**
A name has been found that is not a column name, and has not been defined as a memory variable, or the memory variable data type does not match that expected in the SQL statement. Check that a column name or memory variable name has not been misspelled. Also, check that columns belong to the referenced table; make sure that values have been STOREd to memory variables and that the values stored to memory variables are compatible with the data types expected in the SQL statement.

Missing end quotes for string
Strings must be enclosed in quotes. Add missing quote ("). Check to make sure that you have not inadvertently used a quote that has been interpreted as a marker to begin a string.

Name already exists: < object name >
The name used in a CREATE TABLE, CREATE SYNONYM, or CREATE VIEW command already names an existing SQL table, synonym, or view. Use a different name for the new table, synonym, or view.

Name, constant, or expression expected
A scalar value was expected. Check that a column name, constant, or expression has not been omitted from the statement. Look for incorrectly placed commas; commas should not be used to separate clauses.

Name expected
An object name is missing, misplaced, or misspelled. Check that database, table, view, synonym, or column names are correctly specified. Also, check for incorrectly placed commas; commas should not be used to separate clauses.

Name longer than 8 characters not allowed
The name for a database, table, index, synonym, or view exceeds eight characters. Use a name of eight characters or less (beginning with a letter and containing only letters, digits, and underscores).

Name longer than 10 characters
SQL column names and memory variables referenced in SQL statements may be up to 10 characters long. (Database, table, view, synonym, and index names may be up to eight characters long.)

Nested function not allowed
An aggregate function contains another nested aggregate function. Remove the nested aggregate function.

No alias defined for self-join: < table name >
When a self-join form of the WHERE clause is used, an alias must be defined in the FROM clause. The table name cannot be referenced twice, nor may a synonym name be used in place of an alias. Use a FROM clause of the form FROM < table name > < alias name >. Then prefix column names in the SELECT and WHERE clauses as necessary.

No ALTER or INDEX privileges for views
These privileges cannot be GRANTed or REVOKEd because views cannot be ALTERed or INDEXed. You cannot GRANT or REVOKE ALL PRIVILEGES on views because ALL includes the ALTER and INDEX privileges. A privilege list for views may contain the following privileges: DELETE, INSERT, SELECT, or UPDATE.

No current row available for UPDATE or DELETE: < table or view name >
This error occurs only in .prs programs when UPDATEs and DELETEs are being performed under SQL cursor control. Make sure a FETCH has been executed before the UPDATE or DELETE is attempted.

No database open
The command requested can only be executed when a database is open. If you have entered SQL with all databases closed, or if you used the STOP DATABASE command, the current command cannot be executed. Use a START DATABASE command before retrieving information.

No .dbf file in the current database
No .dbf files (other than the SQL catalog tables) were found during execution of a DBDEFINE command. Make sure that the .dbf files that you want to define as SQL tables have been copied into the current database directory before executing the DBDEFINE command.

No entry found in catalog table Sysidxs for the following table, index: < table and index names >
An error occurred on a DBCHECK or RUNSTATS command because an index file was found with a name that does not exist in the Sysidxs catalog table. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

No entry found in catalog table Syskeys for following table, index: < table, index, column name >
An error occurred on a DBCHECK or RUNSTATS command because an index key has been found with an index name that does not exist in the Syskeys catalog table. First copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

No record selected
A query returned no rows.

Non-numeric array subscript
All array subscripts must evaluate to integers. If columns or functions are used, check that the data type of the expression is numeric.

Number of columns doesn't match in table: < table name >
An error occurred on a DBCHECK or RUNSTATS command because the actual number of columns in a table does not match the number of columns indicated in the Colcount column of the Systabls SQL catalog table. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Number of columns in index key doesn't match for table, index: < table and index names >
An error occurred on a DBCHECK or RUNSTATS command because the Sysidxs catalog table indicates a different number of columns in the index key from the actual number of columns in the index. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Number of columns must be the same in UNION operation
The SELECT statements joined by the UNION keyword do not generate the same number of columns. Change the SELECT clauses of the SELECT statements so that they return the same number of columns. Columns must also be of matching data type and length.

Number of decimal places doesn't match for table, column: < table and column names >
An error occurred on a DBCHECK or RUNSTATS command because the scale indicated in the Colscale column of the Syscols catalog table doesn't match the scale of the actual column. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Number of indexes doesn't match for table: < table name >
An error occurred on a DBCHECK or RUNSTATS command because the number of indexes indicated by the Systabls catalog table does not match the actual number of indexes. First, copy the .dbf and associated .mdx files in question to another directory, and DROP the table from the current database. Then, copy the .dbf and .mdx files back into the current database and use DBDEFINE to redefine them. Then re-execute the DBCHECK or RUNSTATS command.

Number of SAVE TO TEMP columns does not match number of SELECT columns
The SAVE TO TEMP column list must contain the same number of columns as the SELECT statement results being saved. If all the columns in the result table are named columns (not derived from functions, constants, and so on), then no column list need be specified in the SAVE TO TEMP clause.

Number of view columns does not match number of SELECT columns
In a CREATE VIEW command, the number of columns specified in the view column list is not the same as the number of columns generated in the SELECT clause. If all the columns in the view are named columns (not the result of aggregate functions, expressions, and so on), then the column list may be omitted and the view columns will inherit SELECT column names.

Numeric value too large
A value exceeds dBASE limits on numeric values.

Numeric value too small
A value is smaller than the smallest allowable dBASE numeric value.

Only one DISTINCT allowed in any SELECT clause
The keyword DISTINCT has appeared more than once in a SELECT clause.

ORDER BY clause not allowed in CREATE VIEW
An ORDER BY clause was included as part of the AS SELECT in a CREATE VIEW statement. The ORDER BY clause may only be used in a full SELECT, with UNION, or in the DECLARE CURSOR statement. Remove the ORDER BY clause. You may use an ORDER BY clause when you SELECT from the view.

ORDER BY column(s) not specified in the SELECT clause
A column has been specified in the ORDER BY clause that is not included in the SELECT clause. Include the ORDER BY column(s) in the SELECT clause.

Ordering of index column doesn't match for table, index, column: < table, index, and column names >
An error occurred on a DBCHECK or RUNSTATS command because the Syskeys catalog table indicates an ordering of ASC or DESC that is not the actual ordering of the column in the index. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Path too long
A path name may not exceed 64 characters.

Position of column in index key doesn't match for table, index, column: < table, index, column names >
An error occurred on a DBCHECK or RUNSTATS command because the position of a column in an index key as indicated in the Syskeys catalog table doesn't match the column's actual position in the index. First, copy the .dbf file and its associated .mdx file to another directory and DROP the SQL table. Then, copy the .dbf and .mdx files back to the database directory and use DBDEFINE < filename > to redefine the table and indexes.

Press ESC to abandon operation, any other key to continue
Safety prompt that appears in conjunction with warning messages for certain operations (namely UPDATE or DELETE without WHERE clause and DROP DATABASE commands).

Right bracket missing
A left bracket appears without a matching right bracket. Check array references to make sure brackets match. Square brackets are not used elsewhere in SQL.

Right parenthesis missing
A left parenthesis is present without a matching right parenthesis. Check for missing right parenthesis in subselects, aggregate functions, and so on.

Rows violates view definition — INSERT/UPDATE row rejected **[2005]**
A row cannot be or UPDATEd or INSERTed in a view because the view was created WITH CHECK OPTION and the inserted or updated row violates the view's definition. You can read the view's definition by SELECTing the Sqltext column from the Sysviews catalog table. You can also INSERT into or UPDATE the underlying base table, but such rows will not appear in the view.

RUNSTATS completed without catalog updates error/warning found **[2009]**
RUNSTATS was not successful because of inconsistencies in the database. Note that no updates are made, even for tables for which no error/warning appeared. Check the error messages that preceded this message, follow suggested remedies, and run RUNSTATS again.

SAVE TO TEMP clause not allowed
SAVE TO TEMP clause not allowed as part of a DECLARE CURSOR statement, or when an INTO clause is used in a SELECT statement.

Second argument of LIKE clause must be a character string
LIKE must be followed by a character string, the USER keyword, or a character-type memory variable. Functions and column names are not allowed.

SELECT cannot include both FOR UPDATE OF and ORDER BY clauses
A SELECT or DECLARE CURSOR statement cannot include both the FOR UPDATE OF and ORDER BY clauses. If a cursor defined in a DECLARE CURSOR statement is to be used in an UPDATE WHERE CURRENT OF statement, then the FOR UPDATE OF clause must be chosen. Otherwise, drop it and keep the ORDER BY clause.

SELECT privilege not granted
Grantor does not possess the SELECT privilege or did not receive it WITH GRANT OPTION.

SELECT privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has been revoked already.

Subquery did not return any value **[2011]**
The subquery in a WHERE or HAVING clause did not return any values. (Only the WHERE EXISTS predicate may include a subquery that doesn't return any values.) Modify the search condition subquery. You can test the query separately to verify whether it returns the correct values.

Subquery did not return exactly one value **[2003]**
A subquery following an arithmetic comparison operator in a WHERE clause must return only one value. An error occurred because multiple values were returned. Either precede the subquery with the keyword ANY or ALL, or rewrite the WHERE clause using the keyword IN instead of an operator.

System table entry missing for table **[2004]**
The Systabls catalog table does not contain an entry for this table; it has been DROPped, or does not exist.

Systimes error
The Systimes.dbf file (used to indicate updates to tables in a multi-user environment) has been corrrupted. Have each user currently accessing the associated database CLOSE the affected database. Then, transfer a new copy of the Systimes.dbf file from the SQL home directory to the affected SQL database directory.

Table already exists
The filename following the DBDEFINE keyword already exists as an SQL table, so no new entries can be made in the SQL catalog tables for this file. If you need to redefine this table (because of DBCHECK errors, or for any other reason), you must copy the .dbf file and its associated .mdx file to another directory and DROP it as an SQL table. Copy the .dbf and .mdx files back into the database directory, then use DBDEFINE <filename>.

Table(s) not DBDEFINEd:
Lists files which were not converted to SQL tables. See the **DBDEFINE completed; errors/warnings found** error message.

Table not found in the SQL catalog tables
The table name specified in the DBCHECK or RUNSTATS command does not exist in the SQL catalog tables. Verify the correct table name with a SELECT from Systabls catalog table.

Table not included in current statement: < table name >
A column prefixed with the table name from which it comes is referenced, but its table is not included in a FROM clause. Add the missing table name to the FROM clause.

The following file was not found in the current database: < filename >
On DBCHECK or RUNSTATS there is an entry for a table in the SQL system catalogs, but there is no .dbf file with this name in the database directory. The file may have been erased outside of SQL (from dBASE or DOS).

The following table already exists: < table name >
The filename specified in a DBDEFINE command already exists in the SQL catalog tables.

The following table name is the same as the existing synonym: < table name >
This error occurs when running DBDEFINE if one of the .dbf files to be defined has the same name as an existing SQL synonym. Rename the .dbf file before re-executing DBDEFINE.

The following table was not found in catalog table SYSTABLS: < filename >
An error occurred on a DBCHECK or RUNSTATS command because a .dbf was found with a name that does not appear in the Systabls catalog table. Either remove the .dbf file from the database directory or use the DBDEFINE < filename > command to define the .dbf file as an SQL table before re-executing the DBCHECK or RUNSTATS command.

Too many columns in a table
This message appears if a CREATE TABLE or ALTER TABLE command attempts to create a table with more than 255 columns.

Too many indexes for a table
You cannot create more than 47 indexes for a single table. Drop indexes you no longer need.

Too many unique indexes
A unique index has been used and more than ten unique indexes all use the same column as part of their unique key. You need to drop some of the unique indexes.

Too many values specified in INSERT
A column list was specified that contained more columns than exist in the table. Check the column list for duplicate column names and remove extra columns

Too many work areas open **[2008]**
Either more than ten tables are referenced in one SQL statement, or you have left some work areas open before issuing the SET SQL ON command and the total number of open work areas is greater than ten. Reference fewer tables in the FROM clause, reduce the complexity of the SQL query, or return to dBASE mode and close work areas.

Undefined column name: < column name >
The column name that caused this error is not a column in any of the tables referenced in the command. Either you have referenced the wrong table, forgotten to include a table in the FROM clause, misspelled the column name, or used a column name that doesn't exist.

Undefined database name: < database name >
There is no entry in the Sysdbs catalog table for this database. Check that you have not given an incorrect database name, or use a CREATE DATABASE statement to create the database.

Undefined index name: < index name >
The index named in a DROP INDEX command does not exist as an SQL index. (The index name is not entered in the SQL catalog table SYSIDXS.) Do a SELECT operation on the Sysidxs catalog table to verify the correct name of the index.

Undefined privilege in GRANT or REVOKE statement
You may GRANT and REVOKE the following privileges: SELECT, INSERT, DELETE, UPDATE, INDEX, ALTER. Check for misspellings or keywords such as CREATE that cannot be GRANTed.

Undefined symbol
An unrecognized or inappropriate symbol has been accidentally included in the statement. Check for and delete any symbols such as $ or &, or arithmetic comparison or operator symbols accidentally included in names or keywords.

Undefined synonym name: < synonym name >
The synonym named in a DROP SYNONYM command does not exist as an SQL synonym. (The synonym name is not entered in the Syssyns catalog table.) Check that you have not given a table or view name by mistake. Use a SELECT operation on the Syssyns catalog table to get a listing of all existing synonym names.

Undefined table name: < table name >
The table, synonym, or view requested is not entered in the relevant SQL catalog table. Either the name is misspelled, or a CREATE (or DBDEFINE) command was never used. Check that you have not put a column name or other item where a table name was expected. Do a SELECT from the appropriate catalog table (Systabls, Syssyns, or Sysviews) to list the correct names.

Undefined view name: < view name >
The view named in a DROP VIEW command is not an SQL view. (The view name is not entered in the Sysviews catalog table.) Do a SELECT from the Sysviews catalog table to verify existing view names.

UNION is not allowed in a view definition
The UNION keyword has been included in a CREATE VIEW command. You may SELECT columns from more than one table to CREATE a single view, but the UNION operation is not allowed.

UPDATE may only be REVOKEd from the whole table, not by column
This is a warning, not an error. REVOKE of an UPDATE privilege applies to all columns in a table. After REVOKEing the UPDATE privilege, GRANT it again specifying the columns to which it applies.

UPDATE column(s) not defined in cursor declaration: < column name >
The UPDATE WHERE CURRENT OF statement in a .prs program updates columns that were not included in the column list of the FOR UPDATE OF clause of the associated DECLARE CURSOR statement. Add all columns to be updated to the column list in the FOR UPDATE OF clause of the DECLARE CURSOR statement.

UPDATE privilege not granted
Grantor does not possess the UPDATE privilege, or did not receive it WITH GRANT OPTION.

UPDATE privilege not revoked
The user revoking this privilege never granted it to the user from whom it is to be revoked, or the privilege has been revoked already.

Value exceeds column length
A value is longer than the column into which it is to be entered. Query the Syscols catalog table to determine the length of the column into which you are entering values.

Value must match column data type
An UPDATE or INSERT value doesn't match the data type of the column into which it is to be placed. Check the data type of the column. Be sure dates are correctly entered in the form CTOD(" / / "), or { / / }. Check for missing beginning quotes on character strings.

View column names must be specified
The SELECT clause of a CREATE VIEW statement includes columns derived from aggregate functions or from two different tables. View column names cannot simply be inherited from the SELECT clause. Add a list of column names enclosed in parentheses after the CREATE VIEW keywords.

View defined with GROUP BY cannot be used in a join
A view referenced in the FROM clause of a statement joining views (and tables) has a GROUP BY clause as part of its definition. This view cannot be used in a join.

View defined with GROUP BY cannot be used in a query including a GROUP BY clause
A SELECT statement including a GROUP BY clause references a view that has a GROUP BY clause as part of its definition. Remove the GROUP BY clause in the current SELECT, or do not use the named view in this query.

View is not updatable: < view name >
A DELETE operation was attempted on a non-updatable view. Views with definitions referencing more than one table, or including GROUP BY, SELECT DISTINCT, or aggregate functions are not updatable.

Views cannot be INDEXed: < view name >
A CREATE INDEX command included a view name instead of a table or synonym name. Views cannot be INDEXed.

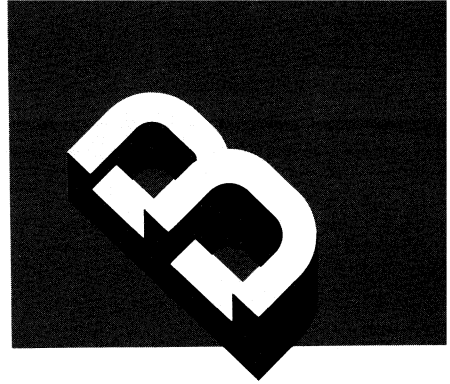
Warning — Cursor < cursor name > not closed
Safety message. An open cursor was not closed by the user before the end of the program or procedure. Upon exiting a program or procedure, SQL automatically closes all cursors opened during it.

Warning — DROP DATABASE will drop all SQL tables
Safety prompt that appears when you enter a DROP DATABASE command. If you continue, all SQL tables, both catalog tables and data tables, will be deleted. The database's entry in the Sysdbs catalog table is also deleted. Press **Esc** to abandon operation. Press any other key to continue DROPping the database.

Warning — No WHERE clause specified in DELETE
Safety prompt that appears if a DELETE command contains no WHERE clause. If executed, this DELETE will delete all rows in the table. Press **Esc** to abandon operation. Press any other key to continue deleting all rows.

Warning — No WHERE clause specified in UPDATE statement
Safety prompt that appears if an UPDATE command contains no WHERE clause. If executed, the command will update all rows in the table. Press **Esc** to abandon operation. Press any other key to continue updating all rows.

Glossary



This section defines terms used in dBASE IV SQL, particularly those that have special meaning when used to describe SQL commands, functions, and operations.

Aggregate function — One of the five SQL functions AVG(), COUNT(), MAX(), MIN(), and SUM(). These functions are different from the dBASE functions of the same name, and operate on SQL grouping of rows.

Alias — An identifier designating a table or view that can be specified in the FROM clause of a SELECT statement. Alias names are used to explicitly reference columns in a SELECT clause that references more than one table or specifies a self-join.

Application — A program or set of programs that performs tasks required by a specific situation. Source program modules containing SQL statements use the .prs file extension. Source program modules that do not include SQL statements use the .prg extension. Both dBASE and SQL compiled programs have the .dbo extension.

Authorization — The system by which privileges are assigned to users by GRANT and REVOKE commands to perform a number of SQL operations such as INSERT, UPDATE, and DELETE. Access privileges assigned by SQL are distinct from those assigned using the PROTECT command.

Base table — A table containing actual data (in contrast to a view).

Catalog — The set of system-controlled tables used to maintain the definition of, and statistics about, an SQL database.

Character string — A sequence of characters, numbers, and symbols defined in the ASCII character set and normally enclosed by single or double quotation marks.

Clause — The portion of an SQL statement that performs a specific function, such as the WHERE clause of the SELECT command.

Column — The vertical component of a table having a name and a specific data type (character, integer, numeric, and so on). Columns correspond to fields in .dbf database files.

Command — An instruction, such as INSERT, that signifies an SQL operation to be performed.

Comparison operator — A symbol (such as <, >, or =) used to test the relationship between two values. For example, comparison operators are used in the WHERE clause to qualify rows that match a specified condition.

Clustering — The degree to which the order of rows in a table matches that in an SQL index.

Concurrency — The shared use and access of information in tables by more than one user.

Condition — An expression that returns a true or false value, as in the WHERE clause which qualifies rows.

Constant — A constant (often called a *literal*) that specifies a character string (a fixed sequence of characters), a date, a logical value, or a number.

Correlated subquery — A subquery in which the WHERE clause compares values of an outer query with values of a nested query.

Creator — A term used when dBASE IV is password-protected to denote the person creating a database, table, view, or index. The creator has full privileges on the created object and the right to drop (delete) those objects. The SQLDBA superuser ID also has full privileges on any created object.

Cursor — A mechanism (similar to a record pointer) that allows SQL operations to be performed on a result table one row at a time.

Data control — Class of SQL commands such as GRANT and REVOKE that restrict access to data in tables or views (often referred to as Data Control Language commands).

Data definition — Class of SQL commands such as CREATE TABLE or DROP TABLE that create or drop SQL objects (often referred to as Data Definition Language commands).

Data manipulation — Class of SQL commands such as INSERT, UPDATE, or DELETE that update data in SQL tables (often referred to as Data Manipulation Language commands).

Data type — Classification assigned to columns reflecting the type of data that can be entered in columns.

Database — A collection of tables, views, and indexes relating the same data (similar to the set of files contained in a .cat file in dBASE mode). In SQL, a directory is created to hold the related files contained in each database created with the CREATE DATABASE command.

Database administrator — Person charged with the responsibility of maintaining certain system information, performing maintenance operations, and granting and revoking privileges to access data. A special user ID (SQLDBA) can be defined (using PROTECT) to provide full privileges (including the ability to drop any database, table, view, or index) for SQL database administrators.

dBASE mode — Default mode for entering commands in which only traditional dBASE commands (not SQL) can be executed.

Deadlock Condition occurring on a network when two or more operations cannot be completed because each has locked data necessary to complete another's operation.

DISTINCT Keyword allowed in a SELECT clause to restrict display of rows to those in which specified columns have unique values.

Embedded SQL In dBASE IV, refers to SQL commands used in program files. It also refers to the mode in which SQL program files are executed.

Expression Combination of data from columns, memory variables, or constants joined by operators or functions that evaluates to a single character, date, logical, or numeric value.

Fetch The method of accessing data from selected rows one row at a time using the DECLARE CURSOR, FETCH, and OPEN commands.

Full-select A query containing one or more subselects joined with UNION, followed by the ORDER and SAVE TO TEMP clauses.

Group Rows of an SQL table arranged or processed by the value(s) of one or more columns in the same table. The GROUP BY clause is used to group rows based on the values of individual columns. Aggregate functions evaluate results based on the column values within groups.

History A dBASE IV feature that retains command lines typed at either the dBASE or SQL dot prompts or executed in a program file. History allows commands to be recalled (and edited) or re-executed. The commands are stored in a memory buffer called the history buffer.

Index A structure that helps to speed the retrieval of data from tables. Indexes arrange the rows in ascending or descending order by the values of columns. A *unique* index also prevents entry of duplicate data in rows of an associated table. SQL indexes differ from dBASE indexes in that you do not use indexes to order the display of data.

Index key The column or columns on which an index is created. An index key is specified when an index is created with the CREATE INDEX command.

Interactive SQL Operating environment in which SQL commands can be executed one at a time. Commands are entered directly at the SQL dot prompt rather than placed in program files.

Keyword One of the words used to form a command, for example, SELECT. SQL keywords are shown in upper-case letters in descriptions of syntax; however, they can be entered in upper- or lower-case letters.

Join The combination of data from more than one table. In SQL, joins are performed using the SELECT command.

Locking A system feature that prevents different users from simultaneously updating the same data in a table at the same time. Locking is automatically performed by dBASE IV when SQL commands are executed.

Operators A symbol used to indicate a logical or mathematical operation. For example, the plus sign (+) is an arithmetic operator for addition. AND is a logical operator that joins two different conditions to produce a single true or false result based on the state of each condition it joins.

Optimization The process by which the system finds an optimum strategy for obtaining the results of a query.

Precedence Natural order in which operators are applied in expressions. For example, with logical operators, the NOT operator has highest precedence and is applied first, followed by the AND operator and then the OR operator. The natural order of precedence can be overridden by the use of parentheses.

Predicate Conditions selecting rows in SQL statements. The inclusion of rows in a result table is *predicated* (dependent on) specified columns in rows meeting the conditions specified. Conditions or predicates include ALL, ANY, BETWEEN, EXISTS, IN, and LIKE, and the comparison operators, =, >, <, >=, <=, <>, !=, !<, or !>.

Privilege Permission to perform an operation on a database table or view by virtue of creating a table or being given authorization by another user. SQL provides authorization privileges for ALTER, DELETE, INDEX, INSERT, SELECT, and UPDATE operations.

PUBLIC A special keyword by which privileges can be assigned to all users without having to specify individual user IDs (user log-in names created using the PROTECT command).

Query A statement that retrieves selected rows of data from tables or views in an SQL database. All queries of data in SQL are made with the SELECT command.

Result table Rows (and columns) selected or operated on by an SQL statement, for example, the rows displayed by the SQL SELECT statement.

Rollback Restoration of the information in a table or view to its original state, reversing changes made by an operation. For example, you may want to roll back a table when an update of multiple rows cannot be completed.

Row A group of related columns in an SQL table treated as a unit (similar to a *record* in dBASE language terminology).

Search condition A criterion specified in a query that must be met for selected rows to be included in the result table. Search conditions can be specified with the WHERE clause in SQL DELETE, INSERT, SELECT, and UPDATE statements.

Self-join A special type of query in a single table in which rows are selected by comparing values of a column in each row with the column values in all other rows.

SQL Acronym for Structured Query Language.

SQLDBA A special user ID with full privileges (including the ability to drop any database, table, view, or index) provided for system administrators.

SQL mode Mode in dBASE IV in which SQL commands can be executed. Commands can be executed interactively at the SQL dot prompt (reached by typing SET SQL ON) or within a .prs SQL program file. SQL programs may be executed from either the dBASE or SQL dot prompt. dBASE switches modes depending on the extension of the program file (.prg or .prs).

Statement Expression of an SQL command with one or more clauses (for example, SELECT, FROM, and WHERE) and terminated by a semicolon (;).

Subquery Nesting of a query within a SELECT statement.

Subselect A query that may include SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. A subselect (versus a full-select) is specified as being allowed in the syntax of some commands. A subselect may also be joined with other subselects with UNION to form a full select.

Synonym An alternate (usually shorter) name that you can use to reference a table. Synonyms are created with the CREATE SYNONYM command.

Table The basic structure for storing data in SQL. SQL tables consist of a number of rows and columns (corresponding to *records* and *fields* in dBASE database file terminology). SQL catalogs maintain information on SQL table definitions and statistics. SQL maintains data from tables in .dbf files accessible in both dBASE and SQL modes.

Transaction A data update operation, such as DELETE, INSERT, or UPDATE, that may consist of a number of suboperations, all of which must be completed successfully for the entire operation to be considered successful. See the entry for *transaction processing* below.

Transaction processing Method by which data in tables is restored to its original state if an operation (transaction) is not successfully completed. The ROLLBACK command allows you to *roll back* the data in a table if a transaction fails. The BEGIN TRANSACTION and END TRANSACTION commands are used to define the commands that make up a transaction.

UNION Special keyword operator used with the SELECT command to combine two or more separate subselects (see entry for *subselect*).

UNIQUE Attribute of data in columns in which no values in a column are duplicated in different rows of the same table. SQL provides the UNIQUE keyword for the CREATE INDEX command to create indexes that prevent users from entering duplicate data in columns designated as UNIQUE.

Update A change in column data in one or more rows of a table. Data is updated with the DELETE, INSERT, and UPDATE commands.

Updatability Ability to update information in underlying tables of a view, determined by the design of the view. See the CREATE VIEW command in Chapter 7 for rules on updating data from views.

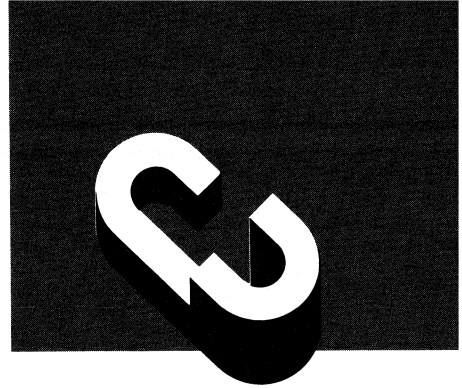
USER Special SQL keyword that is always equal to the user ID of the current user (on systems using PROTECT log-in security). USER can be referenced as part of any SQL statement where entry of an expression value is allowed, for example, in the WHERE clause of a SELECT statement.

User ID Identification of users by which authorization to perform SQL operations is granted, revoked, or checked. User IDs correspond to log-in names created using the PROTECT command. A special user ID, SQLDBA, can also be created with PROTECT and has full privileges (including dropping) for all SQL databases, tables, views, and indexes.

Wildcard Special characters, percent (%) and underscore (_), used in specifying a character search string with the LIKE predicate. The percent character is used to match any group of characters in a character column and the underscore character to match a single character in a character column.

View A special type of SQL table (different from a dBASE view) that displays a subset of the rows and columns of one or more base tables (sometimes referred to as a *virtual* table). A view does not actually contain any data. Rather, the view reflects the data contained in the underlying tables on which it is based. Depending on how the view is constructed, data can also be changed in the view. In those cases, data in the underlying tables is changed.

dBASE Commands and Functions



The SQL language consists of a small set of commands and functions limited exclusively to the definition and access of data. dBASE IV supplements SQL commands with a set of dBASE commands and functions you can use with SQL. In interactive mode, for example, you can use dBASE IV commands to create and print reports from data selected with SQL queries.

The complete set of dBASE commands and functions is shown in Tables C-1 and C-2, respectively. Commands and functions that you can use in SQL interactive and embedded modes are shown in bold type.

NOTE
Certain functions may be allowed in SQL mode but not within an SQL statement, for example, in a SELECT or WHERE clause. These exceptions are noted with an asterisk next to the function.

Table C-1 Allowed dBASE commands in SQL mode

dBASE Command	Description
! or **RUN**	Runs a DOS-level system command
?, ??, or ???	Displays an expression list
&&	Embeds a program comment (allowed in SQL mode, but not as part of an SQL statement)
*	Comment indicator in programs
@...CLEAR...	Erases boxes and lines
@...TO...DOUBLE/ PANEL/NONE...	Draws boxes
@...FILL TO...	Fills an area with a color
@...SAY...GET	Displays/gets user data
ACCEPT	Enters a character string into a memory variable

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
ACTIVATE MENU	Activates a defined menu
ACTIVATE POPUP	Activates a defined pop-up menu
ACTIVATE WINDOW	Activates one or all defined windows
APPEND [BLANK]	Appends records
APPEND FROM	Appends/imports records from other files
APPEND MEMO	Appends a memo field from an existing field
ASSIST	Invokes the Control Center menu system
AVERAGE	Computes the arithmetic average of expressions and optionally saves to a memory variable or an array
BEGIN TRANSACTION	Opens a log file for transaction processing
BROWSE	Invokes menu-assisted screen display to browse a database file
CALCULATE	Calculates financial and statistical functions
CALL	Executes a binary program module
CANCEL	Stops program execution
CASE	Defines conditionally executed blocks for DO CASE command
CHANGE	Changes specified fields and records
CLEAR	Clears specified parameters from memory (all allowed in SQL mode except ALL, FIELDS, and MEMORY)
CLOSE	Closes specified files (all allowed in SQL mode except ALL, DATABASES, and INDEX)
COMPILE	Generates dBASE object code for dBASE programs
CONTINUE	Continues to next located record
CONVERT TO	Adds field (_dbaselock) to database file for multi-user lock detection
COPY FILE	Duplicates a file
COPY MEMO	Copies specified memo to field
COPY STRUCTURE	Duplicates a structure in a new file
COPY INDEXES/TAG	Creates .mdx index file tag from existing .mdx file tags or from .ndx files.
COPY TO ARRAY	Copies records to an array

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
COPY TO	Copies/exports records to a new database file
COUNT	Counts records in a database file
CREATE FROM	Creates a new database file from a database file created with COPY STRUCTURE EXTENDED
CREATE/MODIFY	Creates or modifies a database file
CREATE/MODIFY APPLICATION	Creates or modifies an application program using the Applications Generator
CREATE/MODIFY LABEL	Creates or modifies a label file
CREATE/MODIFY QUERY/VIEW	Creates or modifies a view file
CREATE/MODIFY REPORT	Creates or modifies a report file
CREATE/MODIFY SCREEN	Creates or modifies a screen form file
CREATE VIEW... FROM ENVIRONMENT	Creates a view file from current environment settings
DEACTIVATE MENU	Deactivates a defined menu, erasing it from the screen
DEACTIVATE POPUP	Deactivates a defined popup, erasing it from the screen
DEACTIVATE WINDOW	Deactivates a defined window, erasing it from the screen
DEBUG	Calls the debugger program
DECLARE	Defines an array list
DEFINE BAR	Defines a menu bar
DEFINE BOX	Defines a box
DEFINE MENU	Defines a menu
DEFINE PAD	Defines a pad in a defined menu
DEFINE POPUP	Defines a popup
DEFINE WINDOW	Defines a window
DELETE	Marks records in a file for deletion
DELETE FILE	Deletes a file
DELETE TAG	Deletes an index tag
DIR/DIRECTORY	Displays files in a specified directory

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
DISPLAY	Displays/prints selected fields and records
DISPLAY FILES	Displays/prints files
DISPLAY HISTORY	Displays/prints previously executed commands from history
DISPLAY MEMORY	Displays/prints current memory variables
DISPLAY STATUS	Displays/prints system status
DISPLAY STRUCTURE	Displays/prints current file structure
DISPLAY USERS	Displays user name list on a network
DO	Executes a program
DO CASE...ENDCASE	Starts block of CASE statements
DO WHILE...ENDDO	Executes conditional loop in a program
EDIT	Displays records one at a time
EJECT	Sends a form feed to a printer
ENDPRINTJOB	Defines the end of a print job
END TRANSACTION	Closes a transaction processsing log file
ERASE	Deletes a specified file
EXIT	Exits program execution
EXPORT TO	Converts/exports a database file
FIND	Finds a record
FUNCTION	Identifies the beginning of a user-defined function
GO/GOTO	Positions the record pointer in a database file
HELP	Displays context-sensitive Help
IF...ELSE...ENDIF	Provides conditional branching in program execution
IMPORT FROM	Imports a database file
INDEX	Creates an index file (.ndx or .mdx)
INPUT	Enters an expression into a memory variable
INSERT	Inserts a record into a file
JOIN	Combines records from two database files
LABEL	Prints labels

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
LIST	Lists/prints specified fields and records
LIST FILES	Lists/prints matching files
LIST HISTORY	Lists/prints commands in history buffer (in chronological order)
LIST MEMORY	Lists/prints memory variables
LIST STATUS	Lists/prints system status and parameters
LIST STRUCTURE	Lists/prints a database file structure
LIST USERS	Lists/prints network user names
LOAD	Loads a binary program module
LOCATE	Locates a record and positions the record pointer
LOGOUT	Logs a user out of dBASE IV on a LAN
LOOP	Returns execution to the beginning of a DO WHILE loop
MODIFY COMMAND/FILE	Starts text editor for editing program files
MODIFY STRUCTURE	Changes a database file structure
MOVE	Saves/prints a screen image in memory
MOVE WINDOW	Moves a predefined window
NOTE	Indicates comments in a program file
ON ERROR/ESCAPE/KEY	Executes specifed command on condition or key
ON PAD	Activates an associated menu pad
ON PAGE	Provides report processing control over page breaks
ON READERROR	Executes specified command on incorrect entry
ON SELECTION PAD	Specifies the execution of a command for selection of a pad within a menu
ON SELECTION POPUP	Specifies the execution of a command for selection of a popup
PACK	Removes records marked for deletion
PARAMETERS	Specifies memory variables used with DO command
PLAY MACRO	Plays back a macro

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
PRINTJOB	Define start of a print job
PRIVATE	Defines local memory variables
PROCEDURE	Identifies the beginning of a procedure file
PROTECT	Menu-driven command that assigns user log-in names and access privileges
PUBLIC	Defines global memory variables
QUIT	Closes all files and exits to DOS
READ [SAVE]	Reads current GET fields of a record
RECALL	Unmarks records marked for deletion
REINDEX	Rebuilds open index files
RELEASE	Erases memory variables from memory (all options supported in SQL except RELEASE ALL)
RELEASE MENUS	Erases menus from memory
RELEASE MODULE	Erases a program module from memory
RELEASE POPUPS	Erases popups from memory
RELEASE WINDOWS	Erases windows from memory
RENAME	Assigns a new name to an existing database file
REPLACE	Changes the contents of fields in a record
REPORT FORM	Displays/prints tabular reports
RESET	Resets tag of transaction in a log file
RESTORE	Restores named memory variables to memory
RESTORE MACROS	Restores macros to memory from a disk file
RESTORE WINDOW	Restores windows from a disk file
RESUME	Resumes execution of a suspended program
RETRY	Retries a command after its previous execution from a program
RETURN	Returns to a point from which a program call was made
ROLLBACK	Restores a file to a pre-transaction status
RUN	Runs a DOS-level system command

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
SAVE MACROS	Saves macros to a disk file
SAVE TO	Copies memory variables to a disk file
SAVE WINDOW	Saves windows to a disk file
SCAN...ENDSCAN	Finds the next record of a search
SEEK	Positions the record pointer to the first record with an index key matching an expression
SELECT	Activates the specified work area
SET	Displays a menu for setting parameters
SET ALTERNATE ON/OFF	Controls recording of screen output in a text file
SET ALTERNATE TO	Opens a text file to capture screen output
SET AUTOSAVE ON/OFF	Controls automatic storage of open files
SET BELL ON/OFF	Sets the audible prompt on or off
SET BELL TO	Sets the tone and duration of the audible prompt
SET BLOCKSIZE	Sets memo field size
SET BORDER	Sets menu, window, or pop-up borders
SET CARRY ON/OFF	Sets carryover of fields to next record
SET CARRY TO	Specifies fields to carry over from previous records
SET CATALOG ON/OFF	Controls addition of open files to catalog
SET CATALOG TO	Opens a catalog file
SET CENTURY ON/OFF	Controls display of century in a date field
SET CLOCK ON/OFF	Controls the display of the system clock
SET CLOCK TO	Controls position of the clock display
SET COLOR OF	Sets color of special screen areas
SET COLOR ON/OFF	Specifies monochrome or color display
SET COLOR TO	Sets overall color
SET CONFIRM ON/OFF	Controls advance of cursor from field to field
SET CONSOLE ON/OFF	Controls output of display to screen
SET CURRENCY LEFT/RIGHT	Controls display of currency symbols

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
SET CURRENCY TO	Specifies the currency symbol used
SET DATE TO	Sets date format
SET DEBUG ON/OFF	Controls the output of SET ECHO ON to the printer
SET DECIMALS TO	Specifies the number of decimal places displayed with SET FIXED OFF
SET DEFAULT TO	Specifies the default drive
SET DELETED ON/OFF	Controls use of records marked for deletion
SET DELIMITERS ON/OFF	Controls the use of entry form delimiters
SET DELIMITERS TO	Specifies delimiters for field and variable displays
SET DESIGN ON/OFF	Restricts transfer to design mode
SET DEVELOPMENT ON/OFF	Checks creation dates of edited dBASE program files to verify execution of latest version
SET DEVICE TO	Controls output of the @...SAY command to the screen, a printer, or a file
SET DISPLAY TO	Sets display mode of monochrome or color displays
SET DOHISTORY ON/OFF	Command that performs no operation (left in dBASE IV for compatibility with earlier dBASE versions)
SET ECHO ON/OFF	Controls output of executed commands to screen or printer
SET ENCRYPTION ON/OFF	Controls encryption of protected files
SET ESCAPE ON/OFF	Controls interruption of program with **Esc** key
SET EXACT ON/OFF	Controls exactness of matches in character string comparisons
SET EXCLUSIVE ON/OFF	Controls whether files are accessed exclusively on a network
SET FIELDS ON/OFF	Controls use of fields list
SET FIELDS TO	Specifies a fields list
SET FILTER TO	Specifies select conditions that fields in records must meet

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
SET FIXED ON/OFF	Controls whether a fixed number of decimal places is displayed in calculations
SET FORMAT TO	Opens a format file for data entry
SET FULLPATH	Specifies whether certain functions return a full DOS path or only return drive and filename
SET FUNCTION	Sets operation of function keys
SET HEADING ON/OFF	Controls display of headings over the display of fields in LIST or DISPLAY
SET HELP ON/OFF	Controls the display of Help prompts
SET HISTORY ON/OFF	Controls whether commands are saved in the history buffer
SET HISTORY TO	Specifies the number of commands saved in the history buffer
SET HOURS TO	Specifies a 12- or 24-hour clock
SET INDEX TO	Opens index files
SET INSTRUCT ON/OFF	Controls display of instruction boxes
SET INTENSITY ON/OFF	Controls whether screens include enhanced display
SET LOCK ON/OFF	Controls use of automatic record locking
SET MARGIN TO	Controls setting of left printer margin
SET MARK TO	Specifies the date separator character
SET MEMOWIDTH TO	Specifies setting of memo field columns
SET MENUS ON/OFF	Controls whether menus are displayed in full-screen displays
SET MESSAGE TO	Specifies messages displayed at the bottom of the screen
SET NEAR ON/OFF	Specifies whether a seek is satisfied by *near-match*
SET ODOMETER TO	Controls the update interval of the record count for commands that display a count
SET ORDER TO	Specifies a controlling index
SET PATH TO	Specifies a directory path for file searches
SET PAUSE ON/OFF	Controls whether the SELECT command pauses after each full-screen display of data

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
SET POINT TO	Specifies the character used for decimal point displays
SET PRECISION TO	Specifies the precision of fixed point arithmetic
SET PRINTER ON/OFF	Controls output sent to a printer
SET PRINTER TO	Redirects print output
SET PROCEDURE TO	Opens a procedure file
SET REFRESH TO	Specifies length of time between checks for updated file information
SET RELATION TO	Links specified database files
SET REPROCESS TO	Sets command retry count
SET SAFETY ON/OFF	Prompts for confirmation before overwriting file
SET SCOREBOARD ON/OFF	Enables message line display on line 0
SET SEPARATOR TO	Specifies a numeric value separator
SET SKIP TO	Specifies the way in which the record pointer advances in related files
SET SPACE ON/OFF	Specifies whether a space is used between expressions printed with ? or ?? commands
SET SQL ON/OFF	Starts/stops SQL mode in interactive mode
SET STATUS ON/OFF	Enables display of status line
SET STEP ON/OFF	Specifies whether commands are executed one at a time in a program
SET TALK ON/OFF	Enables display of command execution on the screen
SET TITLE ON/OFF	Specifies whether files are titled when added to a dBASE catalog
SET TRAP ON/OFF	Specifies whether the debugger is automatically activated when an error occurs
SET TYPEAHEAD TO	Specifies the size of the typeahead buffer
SET UNIQUE ON/OFF	Specifies whether only records with unique index key values are displayed
SET VIEW TO	Activates a dBASE view
SET WINDOW OF MEMO	Sets the default window for editing a memo field with BROWSE or EDIT commands

(*continued*)

Table C-1 Allowed dBASE commands in SQL mode (*continued*)

dBASE Command	Description
SHOW MENU	Displays a menu without activating it
SHOW POPUP	Displays a popup without activating it
SKIP	Moves the record pointer forward or backward
SORT	Creates a new copy of a database file, arranging the records in specified order
STORE	Stores an expression to a memory variable
SUM	Computes an arithmetic sum
SUSPEND	Interrupts program execution without terminating
TEXT...ENDTEXT	Displays a block of text from a program file
TOTAL ON	Creates a summary database of numeric totals
TYPE	Displays contents of a file to the screen or printer
UNLOCK	Unlocks files or records in a specified work area
UPDATE	Makes batch changes to a database file
USE	Activates a database file
WAIT	Pauses program execution and waits for user response
ZAP	Removes all records from an active database file

Table C-2 Allowed dBASE functions in SQL mode

dBASE Function	Description
&*	Performs macro substitution (not allowed in SQL statement)
ABS()	Returns absolute value
ACCESS()	Returns access level of current user
ACOS()	Returns angle in radians from cosine
ALIAS()	Returns alias name of unselected work area
ASC()	Does character-to-ASCII conversion

(*continued*)

Table C-2 Allowed dBASE functions in SQL mode (*continued*)

dBASE Function	Description
ASIN()	Returns an arcsine value (angle size in radians)
AT()	Does substring search in character or memo field
ATAN()	Returns angle in radians from tangent
ATN2()	Returns angle in radians from cosine and sine
BAR()*	Returns pop-up bar name of last selected prompt bar
BOF()	Indicates beginning of file
CALL()*	Returns result of binary file execution
CDOW()	Returns day of week
CEILING()	Returns smallest integer greater than or equal to value
CHANGE()	Indicates whether record has changed
CHR()	Does ASCII decimal-to-character conversion
CMONTH()	Returns name of month from a date
CNT()	Performs count function (with CALCULATE command)
COL()*	Indicates cursor column position on screen
COMPLETED()*	Indicates whether a transaction is completed
COS()	Returns cosine from angle in radians
CTOD()	Does character-to-date conversion
DATE()	Returns system date
DAY()	Returns number of the day in the month
DBF()	Returns name of database file in use
DELETED()	Determines whether record is marked for deletion
DIFFERENCE()	Indicates difference between two SOUNDEX() codes
DISKSPACE()*	Returns number of free bytes on default drive
DMY()	Does date format conversion to DD/Mon/YY
DOW()	Returns number of the day of the week
DTOC()	Converts date to character string
DTOR()	Converts degrees to radians
DTOS()	Converts a date variable to a character string suitable for indexing
EOF()	Indicates end of file

(*continued*)

Table C-2 Allowed dBASE functions in SQL mode (*continued*)

dBASE Function	Description
ERROR()*	Returns error number causing ON ERROR condition
EXP()	Determines number from its natural log
FIELD()	Determines names of fields from their numbers
FILE()*	Verifies the existence of a file
FIXED()	Does floating point to fixed decimal number conversion
FKLABEL()*	Determines name of function key from its number
FKMAX()*	Indicates maximum number of programmable function keys
FLOAT()	Converts numeric data type to float
FLOCK()	Attempts to lock all records in a database file
FLOOR()	Returns largest integer less than or equal to value
FOUND()	Indicates logical result of database search
FV()	Returns future value of investment at fixed interest
GETENV()*	Returns DOS SET environment parameters
IIF()*	Performs immediate IF operation (not allowed in SQL statement)
INKEY()*	Returns decimal ASCII value of the last key pressed
INT()	Does conversion to integer by truncating decimals
ISALPHA()*	Determines if first character of character expression is alphabetic
ISCOLOR()*	Determines if color graphics board is installed
ISLOWER()*	Determines if first character of character expression is lower case
ISMARKED()	Indicates if changes exist in current database file
ISUPPER()*	Determines if first character of character expression is upper case
KEY()	Returns key expression of an index file or .mdx tag
LASTKEY()*	Returns ASCII code of key pressed to exit full-screen command
LEFT()	Returns a specified number of characters counting from the left of a character expression

(*continued*)

Table C-2 Allowed dBASE functions in SQL mode (*continued*)

dBASE Function	Description
LEN()	Returns number or characters in a character expression or memo field
LIKE()*	Query function that compares character expressions using wildcards
LINENO()*	Returns current line number in program about to be executed
LKSYS()	Returns information on lock owner
LOCK()	Attempts to lock specified records in a database file
LOG()	Returns natural logarithm to base e
LOG10()	Returns logarithm to base 10
LOOKUP()	Looks up record from another database file
LOWER()	Indicates whether letter is upper or lower case
LTRIM()	Removes leading blanks from field
LUPDATE()	Indicates last date of file update
MAX()*	Returns larger of two values
MDX()	Returns the currently active .mdx index name
MDY()	Converts date to format Mon DD YY
MEMLINES()	Returns the number of word-wrapped lines in memo field at current width
MEMORY()*	Indicates the amount of memory in 1,024-byte units
MENU()*	Returns the name of active menu
MESSAGE()*	Returns the error message string of last error causing ON ERROR condition
MIN()*	Returns smaller of two values
MLINE()	Indicates given line of memo field
MOD()	Does modulus arithmetic (remainder of a division)
MONTH()	Returns number of the month in a date
NDX()	Returns name of an index file or specific index tag name
NETWORK()*	Determines if system is running on a network
ORDER()	Indicates name of primary order index file or specific index tag name
OS()*	Indicates operating system in use

(*continued*)

Table C-2 Allowed dBASE functions in SQL mode (*continued*)

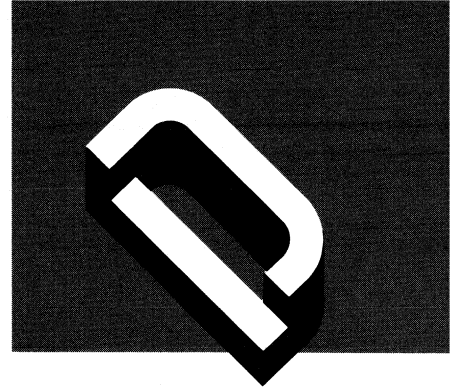
dBASE Function	Description
PAD()*	Returns selected prompt pad name of active menu
PAYMENT()	Indicates value of periodic payment on a loan with fixed interest
PCOL()*	Returns printer column position
PI()	Provides a constant for the ratio of circumference to diameter
POPUP()*	Returns the name of active popup
PRINTSTATUS()*	Returns printer status
PROGRAM()*	Returns name of current executing program
PROMPT()*	Returns prompt of the last selected option
PROW()*	Returns printer row position
PV()	Returns the present value of equal payments invested at fixed interest for a specified time
RAND()	Provides random number
READKEY()*	Returns integer value of key pressed to exit full-screen menu
RECCOUNT()	Returns the number of records in current database
RECNO()	Returns the current record number of selected database
RECSIZE()	Indicates size of a record in selected database
REPLICATE()	Repeats a character expression a given number of times
RIGHT()	Returns specified number of characters from the right of the character expression
RLOCK()	Attempts to lock specified records in a database file
ROLLBACK()*	Determines if ROLLBACK is successful
ROUND()	Rounds off numbers to specified number of decimals
ROW()*	Returns row number of screen cursor position
RTOD()	Does radians-to-degrees conversion
RTRIM()	Removes trailing blanks
SEEK()	Determines if index key is found
SELECT()	Returns number specifying the highest unused work area

(*continued*)

Table C-2 Allowed dBASE functions in SQL mode (*continued*)

dBASE Function	Description
SET()*	Returns parameters set with the SET...TO, SET...ON commands
SIGN()	Returns mathematical sign of number
SIN()	Returns sine value from an angle in radians
SOUNDEX()	Returns code indicating phonetic comparison with a specified string
SPACE()	Provides character string made of blank spaces
SQRT()	Calculates square root of a number
STR()	Does number-to-character string conversion
STUFF()	Replaces part of a string with another
SUBSTR()	Extracts specified number of characters from a string or a memo field
TAG()	Returns name of specified index tag
TAN()	Returns tangent value from an angle in radians
TIME()	Provides system clock value
TRANSFORM()	Performs picture formatting without @...SAY of character or number
TRIM()	Removes trailing blanks
TYPE()*	Evaluates an expression, returning a single upper-case letter to indicate the data type of a specified field
UPPER()	Does upper-case conversion
USER()	Returns the log-in name of the current user
VAL()	Does character string to number conversion
VARREAD()*	Returns name of field or memory variable being edited
VERSION()*	Returns dBASE IV version number
YEAR()	Returns year from date expression

The Sample Database



This section describes the sample SQL tables used in the examples in this book.

SQL Sample Tables

When you install dBASE IV and the associated SQL sample files, a sample SQL database is created. The database is created with a default name of Samples using the directory named \DBASE\SAMPLES.

The tables in the sample database were designed for an order entry and invoicing system. They resemble tables you might create for an application of your own.

Table D-1 Sample tables description

Table	Description
Customer	Names and addresses of customers
Staff	Names and information on employees
Inventry	Description, quantities on hand, and pricing for inventoried items
Assembly	Simple subassembly parts listing for assemblies listed in the Inventry table
Sales	Office equipment orders
Items	Part number and quantity of items corresponding to order numbers in Sales table

Examples used throughout this book demonstrate how you might create and use similar tables. You may also want to make a copy of the table structures and the data in the tables to reference as you run examples in this book.

Customer Table

The Customer table provides the name and addresses of past and present customers. The structure of the table and the data it contains are shown in Table D-2.

Table D-2 Customer table

Column Name	Data Type	Width	Decimal
CUST_NO	Character	6	
COMPANY	Character	25	
LASTNAME	Character	15	
FIRSTNAME	Character	10	
ADDRESS	Character	20	
CITY	Character	15	
STATE	Character	2	
ZIP	Character	5	

Record#	CUST_NO	COMPANY	LASTNAME	FIRSTNAME	ADDRESS	CITY	STATE	ZIP
1	000001	Leonard Design Services	Leonard	Rick	1550 Keystone St.	Oceanside	CA	92054
2	000003	Ace Furniture	Martin	Lisa	1960 Lindley Ave.	Wasau	WI	54401
3	000009	Custom Furniture	Pollock	Daniel	5934 Pine Needles	Yonkers	NY	10709
4	000011	The Office	LeClerc	Dominique	101 Pierce St.	New York	NY	10013
5	000016	American Business Supply	Daniels	George	5601 Grand Ave.	Los Angeles	CA	90233
6	000017	Black's Furniture Store	Jackson	Dennis	7010 Balcom Ave.	San Francisco	CA	94119
7	000018	Interior Systems	Goetz	John	899 Kenwood St.	Milwaukee	WI	53201
8	000019	The Designer	Hobbs	Luke	6043 Whiteside Blvd.	New York	NY	10713
9	000022	Las Vegas Furniture	Hart	Paul	8301 Sale St.	Las Vegas	NV	89106
10	000024	Baker Furniture	Campbell	Linda	6700 Tyler St.	Phoenix	AZ	85012
11	000025	Modern Furniture Store	Hamilton	Robert	366 Shirley Ave.	Phoenix	AZ	85004
12	000027	Al Office Supply Store	McVeigh	John	1240 Embarcadero	San Francisco	CA	94102
13	000028	Accent Furniture Designs	Squire	Ann	20984 Horizon Hills	Las Vegas	NV	89108
14	000031	Al's Furniture & Supplies	Thompson	Kathy	40555 Brentwood	St. Louis	MO	63121
15	000032	Contemporary Designs	Trujillo	Michelle	5670 Colorado Blvd.	Milwaukee	WI	53220
16	000033	Interior Designs	Long	Chuck	40677 Misty Isle Dr.	White Plains	NY	10605
17	000034	La Cienega Furniture	Keegan	Marilyn	6045 Vineland Blvd.	Los Angeles	CA	90815
18	000035	Valley Furniture	Yanish	Diane	10111 Ventura Blvd.	Encino	CA	91316
19	000036	New Horizons	Brendon	Kelly	12508 Robin Hood Ln.	Chicago	IL	60619
20	000040	Design Center Interiors	Gilbert	Chuck	7619 Kraft Dr.	Las Vegas	NV	89106
21	000042	Cohen's Furniture	Cohen	Larry	908 Glen Oaks Ave.	San Francisco	CA	94119
22	000043	To Design Furniture	Tsuma	Tamio	4564 Benedict Canyon	Rochester	NY	14625
23	000045	Classic Interiors	Lawson	Eric	2015 Edmonton	St. Louis	MO	63106
24	000046	Commercial Interiors LTD	Young	Sandy	14097 Gilmore	Ventura	CA	93003

Staff Table

The Staff table provides the names and locations of staff members in three locations. The structure of the table and the data it contains are shown in Table D-3.

Table D-3 Staff table

Column Name	Data Type	Width	Decimal
STAFF_NO	Character	6	
LASTNAME	Character	15	
FIRSTNAME	Character	10	
HIREDATE	Date	8	
LOCATION	Character	15	
SUPERVISOR	Character	6	
SALARY	Numeric	6	
COMMISSION	Numeric	4	1

Record#	STAFF_NO	LASTNAME	FIRSTNAME	HIREDATE	LOCATION	SUPERVISOR	SALARY	COMMISSION
1	000001	Zambini	Rick	02/15/80	LOS ANGELES	000000	6000	5.0
2	000003	Vidoni	Cheryl	03/06/80	NEW YORK	000000	5780	5.0
3	000004	Coudray	Sandy	06/06/80	LOS ANGELES	000001	6237	5.0
4	000006	Thomas	Pat	01/08/81	NEW YORK	000003	5875	5.0
5	000008	McLester	Debbie	04/12/81	LOS ANGELES	000001	4792	5.0
6	000011	Michaels	Delores	05/05/82	CHICAGO	000012	4927	7.0
7	000012	Charles	Ted	02/02/83	CHICAGO	000000	5945	5.0
8	000013	Marin	Mark	06/05/83	LOS ANGELES	000001	4802	11.0
9	000015	Roddick	Mary	02/13/84	NEW YORK	000003	5493	8.0
10	000016	Long	Nicole	08/18/84	NEW YORK	000003	5190	7.0
11	000019	Rolfes	Chuck	09/09/84	LOS ANGELES	000001	4586	6.0
12	000020	Sanders	Kathy	03/23/85	CHICAGO	000012	3783	5.0

Inventry Table

The Inventry table provides part numbers, descriptions, unit costs, and quantities in stock at each of three locations. The structure of the table and the data it contains are shown in Table D-4.

Table D-4 Inventry table

Column Name	Data Type	Width	Decimal
PART_NO	Character	6	
DESCRIPT	Character	30	
ON_HAND	Numeric	4	
LOCATION	Character	15	
UNITCOST	Numeric	8	2
DISCNTINUE	Logical	1	

Record#	PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
1	001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
2	001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
3	001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
4	001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
5	001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
6	001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.38	.T.
7	001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
8	001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
9	001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
10	001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
11	001024	LAMP-BRASS TABLE	140	CHICAGO	230.79	.F.
12	001025	DESK-EXECUTIVE-5 FOOT	63	LOS ANGELES	985.00	.F.
13	001029	FILE CABINET-2 DRAWER	200	NEW YORK	89.95	.T.
14	001031	CHAIR-EXECUTIVE SWIVEL/TILT	79	LOS ANGELES	420.00	.F.
15	001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
16	001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
17	001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
18	001038	LAMP-DRAFTING SWING ARM	47	NEW YORK	149.59	.F.
19	001007	WOOD DESK-SINGLE PEDESTAL	35	CHICAGO	736.21	.F.
20	001007	WOOD DESK-SINGLE PEDESTAL	62	LOS ANGELES	736.21	.F.
21	001013	CHAIR-MODERN PNEUMATIC	115	CHICAGO	275.80	.F.
22	001013	CHAIR-MODERN PNEUMATIC	35	LOS ANGELES	275.80	.F.
23	001032	FILE CABINET-4 DRAWER	71	LOS ANGELES	134.69	.F.
24	001038	LAMP-DRAFTING SWING ARM	89	CHICAGO	149.59	.F.
25	001027	DESK-EXECUTIVE-6 FOOT	20	CHICAGO	1475.00	.F.
26	001027	DESK-EXECUTIVE-6 FOOT	56	NEW YORK	1475.00	.F.
27	001031	CHAIR-EXECUTIVE SWIVEL/TILT	44	CHICAGO	420.00	.F.
28	001031	CHAIR-EXECUTIVE SWIVEL/TILT	76	NEW YORK	420.00	.F.
29	001024	LAMP-BRASS TABLE	56	NEW YORK	230.79	.F.
30	001025	DESK-EXECUTIVE-5 FOOT	47	NEW YORK	985.00	.F.
31	001001	WORKSTATION-ELECTRONIC OFFICE	3	LOS ANGELES	1296.29	.F.
32	001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
33	001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Assembly Table

The Assembly table provides the part number of four different assemblies and the individual parts (subassemblies) and their quantities required to build each assembly. The structure of the table and the data it contains is shown in Table D-5.

Table D-5 Assembly table

Column Name	Data Type	Width	Decimal
ASSEMBLY	Character	6	
SUBASSY	Character	6	
QTY	Numeric	3	

Record#	ASSEMBLY	SUBASSY	QTY
1	001001	001007	1
2	001001	001013	1
3	001001	001032	1
4	001001	001038	1
5	001005	001027	1
6	001005	001031	1
7	001005	001024	1
8	001021	001025	1
9	001021	001031	1
10	001021	001024	1
11	001021	001015	1
12	001002	001025	1
13	001002	001032	1
14	001002	001013	1

Sales Table

The Sales table provides order information for customer orders over the period of a week. It records the customer and staff numbers, and the order number (by which items are entered in a related Items table). The structure of the table and the data it contains are shown in Table D-6.

Table D-6 Sales table

Column Name	Data Type	Width	Decimal
ORDER_NO	Character	6	
SALE_DATE	Date	8	
STAFF_NO	Character	6	
CUST_NO	Character	6	
INVOICED	Logical	1	

Record#	ORDER_NO	SALE_DATE	STAFF_NO	CUST_NO	INVOICED
1	020002	09/21/87	000008	000025	.F.
2	020003	09/21/87	000006	000043	.F.
3	020004	09/21/87	000019	000034	.F.
4	020005	09/21/87	000001	000016	.F.
5	020006	09/22/87	000012	000036	.F.
6	020007	09/22/87	000015	000019	.F.
7	020008	09/22/87	000003	000011	.F.
8	020009	09/22/87	000012	000018	.F.
9	020010	09/22/87	000011	000031	.F.
10	020011	09/22/87	000015	000040	.F.
11	020012	09/22/87	000008	000027	.F.
12	020013	09/23/87	000012	000036	.F.
13	020014	09/23/87	000001	000001	.F.
14	020015	09/23/87	000015	000019	.F.
15	020016	09/23/87	000015	000011	.F.
16	020017	09/24/87	000006	000032	.F.
17	020018	09/24/87	000011	000038	.F.
18	020019	09/24/87	000013	000016	.F.
19	020020	09/24/87	000013	000031	.F.
20	020021	09/25/87	000008	000046	.F.
21	020022	09/25/87	000004	000027	.F.
22	020023	09/25/87	000003	000040	.F.
23	020024	09/25/87	000012	000045	.F.
24	020025	09/25/87	000003	000019	.F.
25	020026	09/25/87	000004	000017	.F.

Items Table

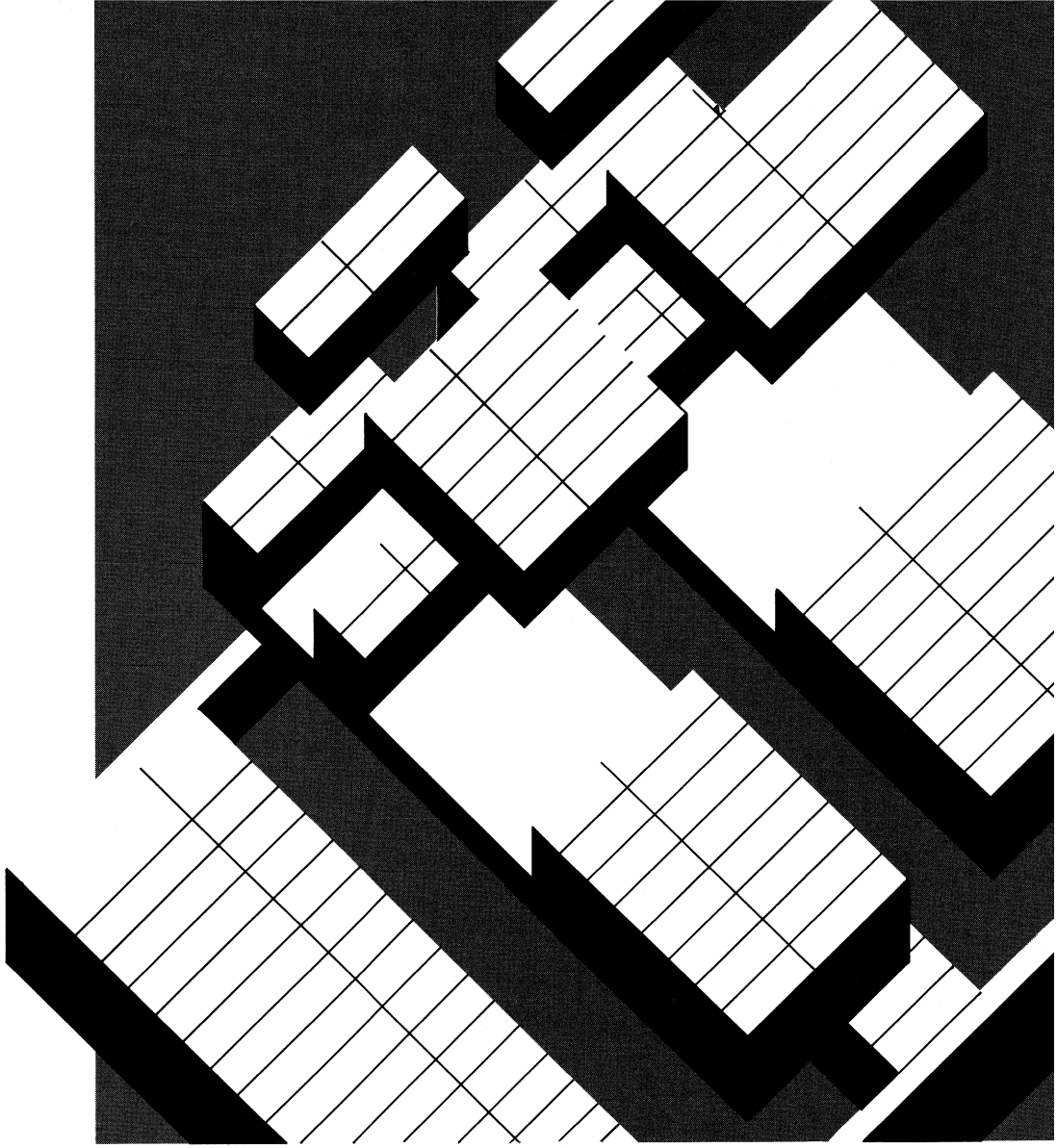
The Items table lists the individual quantities of parts ordered for each order number in the Sales table. The structure of the table and the data it contains are shown in Table D-7.

Table D-7 Items table

Column Name	Data Type	Width	Decimal
ORDER_NO	Character	6	
PART_NO	Character	6	
QTY	Numeric	3	
SHIPPED	Logical	1	

Record#	ORDER_NO	PART_NO	QTY	SHIPPED
1	020002	001032	2	.F.
2	020002	001025	3	.F.
3	020002	001013	3	.F.
4	020003	001021	4	.F.
5	020003	001005	2	.F.
6	020004	001027	5	.F.
7	020004	001038	5	.F.
8	020004	001013	5	.F.
9	020005	001019	2	.F.
10	020006	001007	25	.F.
11	020006	001031	25	.F.
12	020007	001022	3	.F.
13	020007	001033	3	.F.
14	020008	001007	3	.F.
15	020009	001029	31	.F.
16	020010	001005	5	.F.
17	020010	001021	2	.F.
18	020011	001029	7	.F.
19	020011	001025	4	.F.
20	020011	001031	7	.F.
21	020012	001015	5	.F.
22	020013	001022	2	.F.
23	020013	001019	1	.F.
24	020014	001021	2	.F.
25	020015	001025	15	.F.
26	020016	001031	4	.F.
27	020016	001025	2	.F.
28	020017	001029	6	.F.
29	020018	001038	4	.F.
30	020019	001027	3	.F.
31	020020	001024	7	.F.
32	020020	001032	4	.F.
33	020021	001013	8	.F.
34	020021	001025	8	.F.
35	020021	001024	6	.F.
36	020022	001015	1	.F.
37	020023	001024	12	.F.
38	020024	001009	3	.F.
39	020024	001027	3	.F.
40	020025	001019	1	.F.
41	020026	001007	9	.F.
42	020026	001013	9	.F.
43	020026	001024	5	.F.

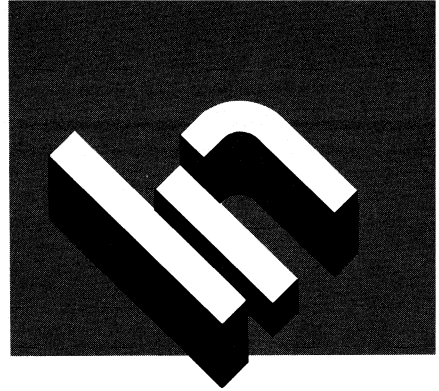
Index



i 1 2 3 4 5 6 7 8 A

B C D In

Index



D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

NOTES