dBASE ODBC Driver

Note This driver is licensed for use only with Lotus applications.

Introduction

The Lotus ODBC dBASE driver supports

- Lotus applications only
- dBASE III and IV files
- Clipper files
- FoxPro and FoxBASE files

The dBASE driver executes the SQL statements directly on dBASE, Clipper, FoxPro, and FoxBASE files. You do not need to own the dBASE product to access these files.

The dBASE driver provides the following features:

- Transaction support for dBASE-compatible files.
- Read committed isolation level.
- File Locking.

Configuring dBASE Data Sources

Note This driver is licensed for use only with Lotus applications.

A data source definition identifies a dBASE database. All data source definitions include:

- A data source name that uniquely identifies the connection. For example, "My Accounting Database" or "dBASE files in directory C:\DBF."
- The database directory and path. For example, C:\DBF.

Lotus applications automatically create ODBC data sources when you specify a directory and path. You can manually configure a dBASE data source by using the ODBC administrator tool and by defining options in the dBASE ODBC Setup dialog box. The Setup dialog box allows you to include additional information about the data connection.

To configure a dBASE data source:

1. Double-click the ODBC icon in the Windows Control Panel.

   A list of data sources appears.

2. If you are configuring a new data source, click Add.

   A list of installed drivers appears.

3. Select Lotus Q+E dBASE, and click OK.
4. If you are configuring an existing data source, select the data source name and click Setup.
5. The Setup dialog box appears. In the dBASE ODBC Setup dialog box, enter information to set up the data source.
6. Click OK to exit the Setup dialog box. The driver writes these values to the ODBC.INI file. These values are now the defaults whenever you connect to the data source. You can change the default values stored in the ODBC.INI file by configuring your data source again.

dBASE IV Driver Communication and Security

Keep in mind the following points concerning security and communication as you work with the dBASE IV driver:

- The dBASE IV driver works directly with dBASE IV table. Users do not have to use dBASE IV to access tables.
- No standard protection scheme exists for databases. The dBASE IV driver does not support passwords or user IDs. To use a dBASE IV table, you must be able to access the table without using a password or user ID. The dBASE IV driver does not support encrypted files.
- When a user deletes a table, the dBASE IV driver deletes .MDX files named the same as the table being deleted, but does not delete associated forms, reports, or index files.
- When sharing files on a network, with the default setting of Locking=File, you can write to a table only if no one else is modifying the table.
- If a dBASE IV table has read-only access, users can read the data in the table but cannot write to the table, delete records in the table, or delete the table.
- You can use standard operating system commands or network security mechanisms to control user access to dBASE IV tables.

dBASE IV Supported Character Sets

The dBASE IV driver supports the following character sets: United States (code page 437), Multilingual (code page 850), Portuguese (code page 860), French-Canadian (code page 863), and Nordic (code page 865). The default character set is United States.

dBASE IV Data Types

The following list defines the dBASE IV data types and shows how they are mapped to the standard ODBC data types. These dBASE IV data types can be used in a CREATE TABLE statement.

dBASE IV Data Type: Logical

ODBC Data Type: SQL_BIT

Used for true/false or yes/no information. The possible values in the dBASE file are the letters T, F, Y, or N. At the application and ODBC API level the possible values are 1 or 0.

dBASE IV Data Type: Char

ODBC Data Type: SQL_CHAR

Values may contain letters, numbers, or any of the punctuation keys on your keyboard. A length parameter is required, giving the maximum length of a character value that can be stored. The length limit is 254 characters. For example, CHAR(12).

dBASE IV Data Type: Memo

ODBC Data Type: SQL_LONGVARCHAR

Values contain long, multi-line textual data.

dBASE IV Data Type: Float

ODBC Data Type: SQL_DECIMAL

Values are stored the same as numeric. Float has the same parameters as numeric. For example, FLOAT(10,2) is the same as NUMERIC(10,2).

dBASE IV Data Type: Numeric

ODBC Data Type: SQL_DECIMAL

Values may contain only numbers, and may include a decimal point and a leading minus sign. Two parameters give the maximum number of digits in the number and, optionally, the number of digits right of the decimal point. There is a limit of 19 total digits. Example: NUMERIC(10,2) declares a 10-digit number, 8 digits to the left of the decimal point and 2 digits to the right. NUMERIC(8) is equivalent to NUMERIC(8,0) and declares an 8-digit number with no digits to the right of the decimal point.

dBASE IV Data Type: Date

ODBC Data Type: SQL_DATE

Values contain date values. The time of day is not included. For example, DATE.

dBASE III and Clipper Data Types

The following list defines the dBASE III and Clipper data types and shows how they are mapped to the standard ODBC data types. These data types can be used in a CREATE TABLE statement.

dBASE III or Clipper Data Type: Logical

ODBC Data Type: SQL_BIT

Used for true/false or yes/no information. The possible values in the dBASE file are the letters T, F, Y, or N. At the application and ODBC API level the possible values are 1 or 0.

dBASE III or Clipper Data Type: Char

ODBC Data Type: SQL_CHAR

Values may contain letters, numbers, or any of the punctuation keys on your keyboard. A length parameter is required, giving the maximum length of a character value that can be stored. The length limit is 254 characters for dBASE III and 1024 characters for Clipper. For example, CHAR(12).

dBASE III or Clipper Data Type: Memo

ODBC Data Type: SQL_LONGVARCHAR

Values contain long, multi-line textual data.

dBASE III or Clipper Data Type: Numeric

ODBC Data Type: SQL_DECIMAL

Values may contain only numbers, and may include a decimal point and a leading minus sign. Two parameters give the maximum number of digits in the number and, optionally, the number of digits right of the decimal point. There is a limit of 19 total digits. Example: NUMERIC(10,2) declares a 10-digit number, 8 digits to the left of the decimal point and 2 digits to the right. NUMERIC(8) is equivalent to NUMERIC(8,0) and declares an 8-digit number with no digits to the right of the decimal point.

dBASE III or Clipper Data Type: Date

ODBC Data Type: SQL_DATE

Values contain date values. The time of day is not included. Example: DATE.

FoxPro and FoxBASE Data Types

The following list defines the FoxPro and FoxBASE data types and shows how they are mapped to the standard ODBC data types. These data types can be used in a CREATE TABLE statement.

FoxPro or FoxBASE Data Type: Logical

ODBC Data Type: SQL_BIT

Used for true/false or yes/no information. The possible values in the dBASE file are the letters T, F, Y, or N. At the application and ODBC API level the possible values are 1 or 0.

FoxPro or FoxBASE Data Type: Char

ODBC Data Type: SQL_CHAR

Values may contain letters, numbers, or any of the punctuation keys on your keyboard. A length parameter is required, giving the maximum length of a character value that can be stored. The length limit is 254 characters.

FoxPro or FoxBASE Data Type: Memo

ODBC Data Type: SQL_LONGVARCHAR

Values contain long, multi-line textual data.

FoxPro or FoxBASE Data Type: Numeric

ODBC Data Type: SQL_DECIMAL

Values may contain only numbers, and may include a decimal point and a leading minus sign. Two parameters give the maximum number of digits in the number and, optionally, the number of digits right of the decimal point. There is a limit of 19 total digits. Example: NUMERIC(10,2) declares a 10-digit number, 8 digits to the left of the decimal point and 2 digits to the right. NUMERIC (8) is equivalent to NUMERIC(8,0) and declares an 8-digit number with no digits to the right of the decimal point. .

FoxPro or FoxBASE Data Type: Date

ODBC Data Type: SQL_DATE

Values contain date values. The time of day is not included. Example: DATE.

FoxPro or FoxBASE Data Type: Picture

ODBC Data Type: SQL_LONGVARBINARY

Values contain graphical pictures.

FoxPro 2.5 Data Type: General

ODBC Data Type: SQL_LONGVARBINARY

Values contain binary data, such as OLE objects. Note that graphical picture data is not stored under this data type. This data type is valid only for FoxPro 2.5.

ROWID Pseudo-Column

Each record contains a special field named ROWID. This field contains a unique number that indicates the record's sequence in the database. For example, a table that contains 50 records has ROWID values from 1 to 50 (if no records are marked deleted). You can use ROWID in WHERE and SELECT clauses.

The fastest way of updating a single row is to use a WHERE clause with the ROWID. However, you cannot update the ROWID column.

Options in the FROM Clause

The FROM clause provides a number of options that you can use with table names. These options provide compatibility with earlier versions of Q+E products and enable you to override the default values set when you connected to the driver. However, if you use these options, your application will not be portable because this is not ANSI syntax.

The format of the FROM clause is

FROM tablename [options] [table_alias]

Options control which File Open options are used, whether deleted records are to be returned, and which index files are to be opened and used for sorting. The options specification is:

([COMPATIBILITY= {ANSI | DBASE},]

[CHARSET= {ANSI | IBMPC},]

[RECORDS= {DELETED | UNDELETED},]
[LOCKING= {NONE | RECORD | FILE},]

[index_spec])

COMPATIBILITY Provided only for compatibility with earlier Q+E Software products.

CHARSET Tells the driver whether the data is stored in the ANSI or IBM PC character set. You can set the default character set in the Setup dialog box.

RECORDS Determines whether to return undeleted records or deleted records. The default is to return undeleted records.

LOCKING Determines the level of record locking for the database file. You can set the default locking level in the Setup dialog box.

Index_spec controls the use of index files.

For .NTX, .NDX, and .IDX files, the form is:

index_file[/USE], ...

For .CDX and .MDX files, index_spec is:

[index_file] [/tag], ...

You specify the list of index files to be opened. The driver automatically opens the production or structured index file, if it exists.

Tag is the name of the index within the index file to be used to sort the records.

The following example specifies that the table 'emp' is stored in the IBM PC character set

SELECT * FROM emp (CHARSET=IBMPC, /emphire)

Index Files

An *index* is a database structure that you can use to improve the performance of database queries. A database file can have one or more indexes associated with it. To optimize queries and to maintain the indexes, the driver must be aware of all the indexes associated with a table. This association is done automatically for production (.CDX and .MDX) indexes and for any indexes listed in the QEDBF.INI file. Nonproduction indexes (.NDX, .NTX, and .IDX), which are not specified in the QEDBF.INI file, must be explicitly mentioned in the FROM clause to be recognized. The driver tries to choose the best index to use among the indexes available. You can force the driver to use a particular index by specifying the /USE switch.

.CDX and .MDX Files

.CDX and .MDX files may contain more than one index. Each index has a tag to identify the index. By default, the index file has the same name as the database file, with the .MDX or .CDX extension. If you know which is the better index to use, specify the tag after the database file name in the SELECT statement. For example using dBASE IV indexes,

```
SELECT * FROM emp (/emp_id) WHERE ...
```

If you specify the tag and do not specify an ORDER BY clause, the driver returns the records ordered by the index. In the above example, it returns the employees ordered by their employee ID values.

The driver automatically opens the dBASE IV index file having the same name as the database file. If the index filename is different from the database filename and you are not maintaining indexes with QEDBF.INI, include in the SQL statement the name of the index to use. For example,

```
SELECT * FROM emp (EMP2.MDX)
```

.NTX, .NDX, and .IDX Files

Each .NTX, .NDX, and .IDX index file contains only one index.

If you are not maintaining indexes with QEDBF.INI, you can include in the SQL statement the name of the index to use. You can have several index files open for the same database file. For example,

```
SELECT * FROM emp (EMPHIRE.NDX, EMPDEPT.NDX /USE)
```

Maintaining Index Files

There are two ways to maintain index files: automatically, through the QEDBF.INI file, or manually, by specifying them when necessary in the INSERT and UPDATE statements. Maintaining index files through QEDBF.INI is more efficient, because the driver automatically updates the indexes for you, which ensures that they match the records in the database file.

All production and structured indexes are maintained automatically.

Maintaining Indexes Automatically

When you maintain indexes automatically, the dBASE driver uses the QEDBF.INI file to maintain information about dBASE files on that directory. The driver maintains the following information for a dBASE file:

- Charset (CS), which tells the driver whether the data is stored in the IBM PC character set or the ANSI character set.
- Index files that need to be maintained when the dBASE file is modified
- ANSI unique indexes maintained for the index file.

To maintain indexes automatically, you must define the index attributes when you run Setup. The program prompts you for the names of the index files that you wish to maintain automatically and creates the QEDBF.INI file. QEDBF.INI is updated when you execute a CREATE or DROP INDEX statement, and you need not include an index name when you execute an INSERT or UPDATE statement.

Maintaining Indexes Manually

However, if you are maintaining the indexes manually, you must include an index name in the INSERT or UPDATE statements. Otherwise, the index files will not match the records in the database file.

INSERT Statement Format

INSERT INTO *file_name* [(*index_file*,...)] [(*col_name*,...)] VALUES (*expr*,...)

UPDATE Statement Format

UPDATE *file_name* [(*index_file*,...)] SET *col_name* = (*expr*,...) [WHERE *conditions*]

- For .NTX, .NDX, and .IDX files, list every index file for *file_name*.
- If .CDX and .MDX files have a different name than the database file (excluding the extension), specify the index file name. If the names are the same, you do not have to specify the index file, because the driver opens it automatically.

Defining Index Attributes

Since dBASE lets you create index files that have different names than their corresponding data files, the driver provides a method to define what indexes should be maintained when a data file is modified, and what indexes should be treated as unique indexes. When you maintain indexes automatically, the system updates the indexes for you, which ensures that they match the records in the database file. The system also makes indexes available for optimizing queries. It is not necessary to mark production .MDX files or structured .CDX files as maintained, as the driver maintains them for you. However, you may wish to use this method to mark a tag as unique.

To maintain indexes automatically:

1. Click Define in the ODBC dBASE Driver Setup dialog box.

   The Define File dialog box appears.

2. Select the .DBF file and click OK to define the special indexes.

3. The upper section of the dialog box displays the name and directory that contains the data file. If this file is stored in the IBM PC character set, select OEM to ANSI Translation.

4. The lower section of the dialog box displays the index information for the data file. The Index File combo box lets you select any index file in the database directory. If the index file is in a different directory, you must provide the full pathname.

   To specify that an index file is unique, select the index name in the combo box and then select Unique to the right of the name. Similarly, select Maintain if the index should be maintained. The topic Maintaining Index Files discusses index file maintenance.

5. If the selected index has an MDX or CDX extension, you cannot mark it as unique. Instead, you may mark the tags within the index as unique. To do so, select the tag name within the Tag combo box and select Unique to the right of the name.

6. Click OK to save this information or click Cancel to abort the dialog box.

CREATE INDEX Statement

The type of index you create is determined by the value of the CreateType attribute, which you set in the Setup dialog box. The index can be a

- dBASE III (.NDX) index
- dBASE IV (.MDX) index
- Clipper (.NTX) index
- FoxBASE (.IDX) index
- FoxPro (.CDX) index

dBASE IV command:

USE file1

INDEX ON <expression> [TO <index file>] / [TAG <index tag>]

SQL Statement:

CREATE [UNIQUE] INDEX *index_name* ON *base_table_name* (*field_name* [ASC | DESC] [,*field_name* [ASC | DESC]] ...)

UNIQUE means that the driver creates an ANSI-style unique index over the column and ensures uniqueness of the keys. Use of unique indexes improves performance. ANSI-style unique indexes are different from dBASE-style unique indexes. With ANSI-style unique indexes, you receive an error message when you try to insert a duplicate value into an indexed field. With dBASE-style unique indexes, you do not see an error message when you insert a duplicate value into an indexed field. This is because only one key is inserted in the index file.

Index_name is the name of the index file. For FoxPro 2.5 and dBASE IV, this is a tag, which is required to identify the indexes in an index file. Each index for a table must have a unique name.

Base_table_name is the name of the database file whose index is to be created. The .DBF extension is not required, the driver automatically adds it if it is not present. By default, dBASE IV index files are named *base_table_name.*MDX and FoxPro 2.5 indexes are named *base_table_name.*CDX.

Field_name is a name of a column in the dBASE file.

ASC tells dBASE to create the index in ascending order. DESC tells dBASE to create the index in descending order. By default, indexes are created in ascending order.

You can substitute a valid dBASE-style index expression for the list of field names.

DROP INDEX Statement

The DROP INDEX statement deletes an index from the current database.

dBASE IV command:

ERASE [<index file>] or DELETE TAG <tag name>

SQL Statement:

DROP INDEX {table_name.index_name}

Table_name is the name of the .DBF file without the extension.

For FoxPro 2.5 and dBASE IV, *index_name* is the tag. Otherwise, *index_name* is the name of the index file without the extension.

To drop the index emphire.ndx, issue the following statement:

DROP INDEX emp.emphire

Pack Statement

When records are deleted from a dBASE file, they are not removed from the file. Instead, they are marked as having been deleted. Also, when memo fields are updated, space may be wasted in the files. To remove the deleted records and to free the unused space from updated memo fields, you must pack the database.

To reuse space in a dBASE file, use the Pack statement. It has the following form:

```
PACK file_name
```

File_name is the name of the dBASE file to be packed. The .DBF extension is not required; the driver automatically adds the extension if it is not present. For example,

```
PACK emp
```

You cannot pack a file that is open by another user and you cannot use the Pack statement in manual commit mode.

For the file listed, the Pack statement does the following:

- Removes all deleted records from the file.
- Removes all deleted records from .CDX and .MDX files having the same name as the file.
- Removes all deleted records from .NTX, .NDX, and .IDX files specified in QEDBF.INI.
- Compresses unused space in the memo (.DBT or .FPT) file.

File Locking

With the dBASE driver, you can build and run applications that share dBASE files on a network. Whenever more than one user is running an application that accesses a shared database file, it is important that the applications lock the records that are being changed. Locking a record prevents other users from locking, updating, or deleting the record.

Levels of Locking

The dBASE driver supports three levels of database locking: NONE, RECORD, and FILE. You can set these levels in

- The Setup dialog box
- At the statement level, in the FROM clause. For example:

  SELECT * FROM emp (locking=none). If you use this method, however, your application will not be portable.

No locking offers the best performance but is intended only for single-user environments.

With record or file locking, the system locks the database files during INSERT, UPDATE, DELETE, or SELECT...FOR UPDATE statements. The locks are released when the user commits the transaction. The locks prevent other users from modifying the locked objects, but they do not lock out readers.

With record locking, only records affected by the statement are locked. Record locking provides better concurrency with other users who also want to modify the table.

With file locking, the entire file is locked. File locking has lower overhead and may work better if records are modified infrequently, if records are modified primarily by one user, or if a large number of records are modified.

Using Locks on Local Files

If you use database locking and are accessing files locally (not on a network), run the DOS utility SHARE.EXE before running Windows. If you add SHARE.EXE to your AUTOEXEC.BAT file, it runs automatically each time you boot your computer.

Limit on Number of Locks

There is a limit on the number of locks that can be placed on a file. If you are accessing a dBASE file from a server, the number of locks depends on the server (see your server documentation). If you are accessing a dBASE file locally, the limitation depends on the buffer space allocated when SHARE.EXE was loaded (for more information, see your DOS documentation). If you are exceeding the number of locks available, you may want to switch to file locking.

Locking Compatibility

The dBASE driver supports several different locking schemes.

All applications that share tables must share the same locking scheme. For example, if you intend to share your tables with Clipper applications, choose Clipper locking.

Q+EVIRTUAL works just like Q+E's original locking style; however, files are no longer locked exclusively. With the original Q+E style locking, although other Q+E applications could read and write to the database, all third party non-Q+E applications were locked out of the file completely (even to read other records).

Use DBASE, CLIPPER, and FOX lock settings if you are using a Q+E application with a dBASE, Clipper, or FoxPro application and you want the locking mechanisms to be compatible.

An advantage to using a Q+E style locking scheme over the DBASE style locking is the fact that Q+E only locks individual tags in an index, whereas DBASE requires locking the entire index on Inserts and

Updates.

How transactions affect record locks

When an UPDATE or DELETE statement is executed, the driver locks the records that are affected by that statement. The locks are released after the driver commits the changes to the database. Under manual commit mode, this means that the locks are held until the application commits the transaction. Under autocommit mode, this means that the locks are held until the statement is executed.

On a SELECT...FOR UPDATE statement, the driver locks the record only when it is fetched. If the record is updated, the driver holds the lock until the changes are committed to the database. Otherwise, the lock is released when the next record is fetched.

Supported SQL Statements

This help topic discusses the following SQL statements in the form supported by the dBASE database driver.

SELECT Statement

CREATE TABLE Statement

CREATE INDEX Statement

DROP INDEX Statement

DROP TABLE Statement

INSERT Statement

UPDATE Statement

DELETE Statement

SELECT Statement

The SELECT statement selects rows and columns from tables either for display or as input to other SQL statements.

dBASE IV command:

```
USE emp

DISPLAY ALL lastname, firstname for salary > 1000
```

SQL Statement:

```
SELECT [DISTINCT] {* | column_expression, ...}

    FROM filespec, ...

    [ WHERE conditions ]

    [ GROUP BY {column_expression, ...} ]
    [ HAVING {conditions} ]

    [ UNION [ALL] (SELECT...) ]

    [ ORDER BY {sort_expression [DESC | ASC]}, ... ]

    [ FOR UPDATE OF {column_expression, ...}]
```

Follow the SELECT keyword with a list of *column_expressions* that you want to retrieve or an asterisk (*) to retrieve all fields.

The most common *expression* is simply a field name (for example, LAST_NAME). More complex expressions may include mathematical operations or string manipulation (for example, SALARY * 1.05).

Separate more than one *column_expression* by commas (for example, LAST_NAME, FIRST_NAME, HIRE_DATE).

Field names may be prefixed with the table name or alias For example, EMP.LAST_NAME or E.LAST_NAME where E is the alias for table EMP.

Distinct

The Distinct operator can precede the first column expression. This operator eliminates duplicate rows from the result of a query. For example,

SELECT DISTINCT last_name FROM emp

FROM Clause

Follow FROM with a list of file specifications (filespec). File specifications have the form:

FROM *tablename* [*options*] [*table_alias*]

Tablename can be a simple table name in the current working directory or a complete pathname (C:\ ODBC\EMP).

Options are database dependent. For example, for dBASE, one option is the level of record locking used.

Table_alias is a name used to refer to this table in the rest of the SELECT statement. Database field names may be prefixed by the table alias. Given the table specification

FROM emp E

you may refer to the LAST_NAME field as E.LAST_NAME. Table aliases must be used if the SELECT statement joins a table to itself. For example,

```
SELECT * FROM emp E, emp F
WHERE E.mgr_id = F.emp_id
```

WHERE Clause

The WHERE clause specifies the conditions that records must meet to be retrieved.

dBASE IV command:

... FOR <expression>

SQL Statement:

WHERE *expr1 rel_operator expr2*

Expr1 and *expr2* may be field names, constant values, or expressions. *Rel_operator* is the relational operator that links the two expressions.

For example, the following SELECT statement retrieves the names of employees that make at least $20,000:

```
SELECT last_name,first_name FROM emp
WHERE salary >= 20000
```

GROUP BY Clause

The GROUP BY clause specifies the names of one or more fields by which the returned values should be grouped. This clause is used to return a set of aggregate values.

dBASE IV command:

REPORT ... Summary

SQL Statement:

GROUP BY *column_expressions*

Column_expressions can be one or more field names of the database table, separated by a comma (,) or one or more expressions, separated by a comma (,).

The following example sums the salaries in each department.

```
SELECT dept_id, sum(salary)
FROM emp
GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

HAVING Clause

The HAVING clause enables you to specify conditions for groups of records (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a GROUP BY clause.

dBASE IV command:

REPORT... Summary

SQL Statement:

HAVING *expr1 rel_operator expr2*

Expr1 and *expr2* can be field names, constant values, or expressions. *Rel_operator* is the relational operator that links the two expressions.

The following example returns only the departments whose sums of salaries are greater than $200,000.

```
SELECT dept_id, sum(salary)
FROM emp
GROUP BY dept_id
HAVING sum(salary) > 200000
```

UNION Operator

The UNION operator combines the results of two SELECT statements into a single result. The single result is all of the returned records from both SELECT statements. By default, duplicate records are not returned. To return duplicate records, use the All keyword (UNION ALL). The form is

```
SELECT statement
UNION [ALL]
SELECT statement
```

When using the UNION operator, the select lists for the SELECT statements must have the same number of column expressions with the same data types and in the same order. For example,

```
SELECT last_name, salary, hire_date
FROM emp
UNION
SELECT name, pay, birth_date
FROM person
```

This example has the same number of column expressions, and each column expression, in order, has the same data type.

The following example is *not* valid because the data types of the column expressions are different (SALARY from EMP has a different data type than LAST_NAME from RAISES). This example does have the same number of column expressions in each SELECT statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary
FROM emp
UNION
SELECT salary, last_name
FROM raises
```

ORDER BY Clause

The ORDER BY clause indicates how the records are to be sorted.

dBASE IV command:

```
USE emp INDEX empid
DISPLAY empid,lastname
```

SQL Statement:

ORDER BY {*sort_expression* [DESC | ASC]}, ...

Sort_expression can be field names, expressions, or the positional number of the column expression to use. As an example, to sort by LAST_NAME, you could use either of the following SELECT statements:

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY last_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2
```

In the second example, LAST_NAME is the second column expression following SELECT, so ORDER BY 2 sorts by LAST_NAME.

FOR UPDATE Of Clause

The FOR UPDATE Of clause locks the records of the database file that are selected by the SELECT statement. The form is

FOR UPDATE OF *column_expressions*

Column_expressions is a list of field names in the database file that you intend to update, separated by a comma (,).

The following example returns all records of the employee database that have a SALARY field value of more than $20,000. When each record is fetched, it is locked. The lock is released when you do a positioned update or delete using a "current of cursor" statement and COMMIT the update or delete. If the record is not updated or deleted, the lock is released when you fetch the next record.

```
SELECT *
FROM emp
WHERE salary > 20000
FOR UPDATE OF last_name, first_name, salary
```

CREATE TABLE Statement

dBASE IV command:

CREATE tablename

SQL Statement:

CREATE TABLE *file_name* (*col_definition*, *col_definition* ...)

The *file_name* can be a simple file name (EMP) or a full pathname (C:\ODBC\EMP).

Col_definition is the column name followed by the data type. Valid values for column names are database specific.

Data type is the specification of a column's data type.

> **Note:** The first number in the definitions of NUMERIC and FLOAT fields is the number of integer digits. The second number is the number of decimal places. This is not an overall length with decimal specification as in dBASE. If no decimal places are involved, a length of 20 may be specified for NUMERIC and FLOAT fields. If there are decimal places, the combined number of integer and decimal places may not exceed 19.

A sample CREATE TABLE statement to create an employee database file is

CREATE TABLE emp (last_name CHAR(20), first_name CHAR(12), salary NUMERIC (10,2), hire_date DATE)

DROP TABLE Statement

The DROP TABLE statement deletes a table from the current database.

dBASE IV command:

    ERASE tablename

SQL Statement:

    DROP TABLE *file_name*

The *file_name* may be a simple file name (EMP) or a full pathname (C:\ODBC\EMP).

A sample DROP TABLE statement to delete the employee database file is

    DROP TABLE emp

INSERT Statement

The SQL INSERT statement is used to add new records to a database file.

dBASE IV command:

```
USE file1
APPEND
```

SQL Statement:

```
INSERT INTO file_name [options] [(col_name,...)] VALUES (expr, ...)
```

The list of indexes and the column names are optional.

The *file_name* may be a simple file name (EMP) or a full pathname (C:\ODBC\EMP).

Options are database specific.

The *col_name* list is an optional list of column names giving the name and order of the columns whose values are specified in the Values clause. If you omit col_name, the value expressions (*expr*) must values for all of the columns defined in the file and must be in the same order that the columns are defined for the file.

The *expr* list is the list of expressions giving the values for the columns of the new record. Usually, the expressions are constant values for the columns. Character string values must be enclosed with single or double quote characters, date values must be enclosed by braces {}, and logical values must be enclosed by periods (for example, .T. or .F.).

An example of an INSERT statement on the employee file is

```
INSERT INTO emp (last_name, first_name, emp_id, salary, hire_date)
VALUES ('Smith', 'John', 'E22345', 27500, {4/6/91})
```

Each INSERT statement adds one record to the database file. In this case a record has been added to the employee database file, EMP. Values are specified for five columns. The remaining columns in the file are assigned a blank value, meaning Null. Note that character values must be enclosed in quotation marks and dates must be enclosed in braces {}.

UPDATE Statement

The SQL UPDATE statement is used to change records in a database file.

dBASE IV command:

```
USE file1

REPLACE <field> with <expression> [,...] FOR <condition>
```

SQL Statement:

```
UPDATE file_name [options] SET col_name = [expr, ...    ]
[ WHERE { conditions | CURRENT OF cursor_name } ]
```

File_name may be a simple file name (EMP) or a full pathname (C:\ODBC\EMP) of the file to be updated.

Options are database specific.

Col_name is the name of a column whose value is to be changed. Several columns can be changed in one statement.

Expr is the new value for the column. The expression can be a constant value or a subquery. Character string values must be enclosed with single or double quotation marks, date values must be enclosed by braces {}, and logical values must be enclosed by periods (for example, .T. or .F.). Subqueries must be enclosed in parentheses.

The WHERE clause determines which records are to be updated.

The WHERE CURRENT OF *cursor_name* clause can be used only by developers coding directly to the ODBC API. It causes the row at which *cursor_name* is positioned to be updated. This is called a positioned update. You must first execute a SELECT...FOR UPDATE statement with a named cursor and fetch the row to be updated.

An example of an UPDATE statement on the employee file is

```
UPDATE emp SET salary=32000, exempt=.T. WHERE emp_id = 'E10001'
```

The UPDATE statement changes every record that meets the conditions in the WHERE clause. In this case the salary and exempt status is changed for all employees having the employee ID E10001. Since employee IDs are unique in the employee file, one record is updated.

An example using a subquery is

```
UPDATE emp SET salary = (select avg(salary) from emp) where emp_id = 'E10001'
```

In this case, the salary is changed to the average salary in the company for the employee having employee ID E10001.

DELETE Statement

The SQL DELETE statement is used to delete records from a database file.

dBASE IV command:

```
USE file1
DELETE ... FOR <condition>
```

SQL Statement:

```
DELETE FROM file_name [options]
[ WHERE {conditions | CURRENT OF cursor_name } ]
```

File_name is the name of the database file whose records are to be deleted. The filename may be a simple file name (EMP) or a full pathname (C:\ODBC\EMP).

Options are database specific.

The WHERE clause determines which records are to be deleted.

The WHERE CURRENT OF *cursor_name* clause can be used only by developers coding directly to the ODBC API. It causes the row at which *cursor_name* is positioned to be deleted. This is called a positioned delete. You must first execute a SELECT...FOR UPDATE statement with a named cursor and fetch the row to be deleted.

An example of a DELETE statement on the employee file is

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each DELETE statement removes every record that meets the conditions in the WHERE clause. In this case every record having the employee id E10001 is deleted. Since employee IDs are unique in the employee file, one record is deleted.

Performance

This section discusses the ways in which you can improve the performance of database queries. It includes the following topics:

Index Files

Improving Record Selection Performance

Indexing Multiple Fields

Improving Join Performance

Index Files

A database driver can use indexes to find records quickly. An index on the EMP_ID field, for example, greatly reduces the time that the driver spends searching for a particular employee ID value. Consider the following WHERE clause:

```
WHERE emp_id = 'E10001'
```

Without an index, the driver must search the entire database table to find those records having an employee ID of E10001. By using an index on the EMP_ID field, however, the driver can quickly find those records using the index references.

Index files may improve the performance of SELECT statements. You may not notice this improvement with small files but it can be significant for large files. However, there are disadvantages to having too many indexes. Indexes slow down the performance of record inserts, updates, and deletes since the driver has to maintain the indexes as well as the database files. Also, indexes take additional disk space.

Improving Record Selection Performance

For indexes to improve the performance of selections, the index expression must match the selection condition exactly. For example, if you have created an index whose expression is LAST_NAME, the following SELECT statement uses the index:

```
SELECT * FROM emp WHERE last_name = 'Smith'
```

This SELECT statement does not use the index:

```
SELECT * FROM emp WHERE UPPER(last_name) = 'SMITH'
```

The second statement does not use the index because the WHERE clause contains UPPER(last_name), which does not match the index expression LAST_NAME. If you plan to use the UPPER function in all your SELECT statements and your database supports indexes on expressions, then you should define an index whose expression is UPPER(last_name).

Indexing Multiple Fields

If you often use WHERE clauses that involve more than one field, you may want to build an index containing multiple fields. Consider the following WHERE clause:

```
WHERE last_name = 'Smith' and first_name =   'Thomas'
```

For this condition, the optimal index field expression is LAST_NAME,FIRST_NAME. This creates a concatenated index.

Concatenated indexes can also be used for WHERE clauses that contain only the first of two concatenated fields. The LAST_NAME, FIRST_NAME index also improves the performance of the following WHERE clause (even though no first name value is specified):

```
last_name = 'Smith'
```

A driver uses only one index when processing WHERE clauses. If you have complex WHERE clauses that involve a number of conditions for different fields and have indexes on more than one of the fields, the driver chooses an index to use. The driver will first use indexes on conditions that have the equal sign (=) as the relational operator. Assume you have an index on the emp_id field as well as the last_name field and the following WHERE clause:

```
WHERE emp_id >= 'E10001' AND last_name = 'Smith'
```

The driver selects the index on the last_name field.

If no conditions have the equal sign, the driver first attempts to use an index on a condition that has a lower *and* upper bound, and then attempts to use an index on a condition that has a lower *or* upper bound. The driver always attempts to use the most restrictive index that will satisfy the WHERE clause.

In most cases, the driver does not use an index if the WHERE clause contains an OR comparison operator. For example, the driver does not use an index for the following WHERE clause:

```
WHERE emp_id >= 'E10001' OR last_name = 'Smith'
```

Improving Join Performance

When joining database files, index files can greatly improve performance. Unless the proper indexes are available, queries that use joins can take a long time.

Assume you have the following SELECT statement:

```
SELECT * FROM dept, emp WHERE dept.dept_id = emp.dept
```

In this example, the dept and emp database files are being joined using the department ID field. When the driver executes a query that contains a join, it processes the tables from left to right and uses an index on the second table's join field (the DEPT field of the EMP file).

To improve join performance, you need an index on the join field of the second file in the FROM clause. If there is a third file in the FROM clause, the driver also uses an index on the field in the third file that joins it to any previous file. For example,

```
SELECT * FROM dept, emp, addr
WHERE dept.dept_id = emp.dept AND emp.loc = addr.loc
```

In this case, you should have an index on the EMP.DEPT field and the ADDR.LOC field.

dBASE Data Types

dBASE ODBC Driver Setup Dialog Box

Note This driver is licensed for use only with Lotus applications.

Data Source Name

Identifies a single connection to a dBASE database. This can be any string. Examples include "Accounting" or "dBASE Files."

Description

An optional long description of a data source name. For example, "My Accounting Database" or "dBASE files in directory C:\DBF."

Database Directory

The directory in which the dBASE files are stored. If none is specified, the current working directory is used.

Create Type

Identifies what dBASE format the driver should use to create new tables. Select dBASE II, dBASE III, dBASE IV, Clipper, FoxBase, FoxPro1, or FoxPro25.

Lock Compatibility

Determines the physical locking for the database file. Select DBASE, Q+E, Q+EVIRTUAL, CLIPPER, or FOX.

Locking

Determines the level of locking for the database files (NONE, RECORD, or FILE). LCK=NONE offers the best performance but is intended only for single-user environments. With LCK=RECORD, only the records affected by the statement are locked. LCK=FILE locks the entire file.

File Open Cache

Determines the maximum number of unused file opens to cache. For example when FOC=4, and a user opens and closes 4 files, the files are not actually closed. The driver keeps them open so that if another query uses one of these files, the driver does not have to perform another open, which is expensive. The advantage of FOC is increased performance. The disadvantage is that a user who tries to open the file exclusively may get a locking conflict even though no one appears to have the file open.

Cache Size

Determines the number of 64K blocks that the driver uses to cache database records. The higher the number of blocks, the better the performance. The default is 4 blocks. The maximum number of blocks you can set depends on the system memory available. If CacheSize is >0, when browsing backwards, you cannot see updates made by other users until you reexecute the SELECT statement.

International Sort

Determines the order that records are retrieved when you issue a SELECT statement. Select International Sort to use international sort order as defined by Windows. The sort is case-insensitive (*a* precedes *B*); the sorting of accented characters is also affected (For more information see your Windows documentation.).Do not select International Sort if you want to use ASCII sort order, where uppercase letters precede lowercase letters (*B* precedes *a*).

Perform OEM to ANSI Translation

Determines whether the database file uses the ANSI or IBM PC character set. The two character sets

are generally the same unless you are using international characters. The ANSI character set has better support for international characters than the IBM PC character set.

Define...:

Choose Define to associate index files with your dBASE files.

Defining Index Attributes

The top section of the dialog box displays the name and directory where the dBASE file is located. If this file uses the IBMPC character set, then select OEM to ANSI Translation.

The bottom section of the dialog box displays all the index files in this directory. Select each index file that you want to associate with the dBASE file and select Maintain. If the index file is unique, check the Unique check box to the right of the file name.

If the selected index has an .MDX or .CDX extension, you cannot mark the index file as unique. However, you can mark the tags within the index as unique. To mark a tag unique, select the tag name within the Tag combo box and check the Unique box to the right of the name.