Intermediate Microsoft Jet SQL for Access 2000

Acey James Bunch
Microsoft Corporation

February 2000

Applies To: Microsoft® Access 2000

Summary: Second in a series of three articles, Intermediate SQL builds on the concepts covered in the Fundamental SQL article and gives a much more detailed picture as to what can be accomplished with Microsoft Jet SQL in Access. For users and developers seeking to expand their understanding of Structured Query Language as it is used in the Microsoft Jet Database Engine, this article is designed to be the next logical step. (38 printed pages)

Download AcIntSQL.exe.

Contents

Introduction

This is the second in a series of articles that explain what Microsoft® Jet SQL is and how you can use it in your Access 2000 applications. There are three articles in all: a fundamental, an intermediate, and an advanced article. The articles are designed to progressively show the syntax and methods for using Jet SQL, and to demonstrate those features of Jet SQL that are new to Access 2000. For the remainder of this article, all references to SQL are meant to be to the dialect of SQL used in the Microsoft Jet 4.0 Database Engine.

Intermediate SQL Defined

By gaining an understanding of intermediate Structured Query Language (SQL) concepts, you can extend and enhance your ability to control the structures and objects of your database, and you can manipulate the data contained by those structures in many interesting and powerful ways. Used in conjunction with such data access methods as DAO and ADO, intermediate SQL can greatly improve the flexibility and performance of your application.

Differences Between Fundamental, Intermediate, and Advanced SQL Topics

It is difficult to draw the lines differentiating between fundamental, intermediate, and advanced SQL. In many cases, it is simply an arbitrary decision. But for this series of articles about using SQL in Access 2000, there are other aspects that were considered:

- First is the complexity level of the SQL statements themselves. In the previous article, every effort was made to include statements that are most often used, and to show them in their simplest form. This article introduces more complex statements that build upon what was covered in the fundamental SQL article.
- Second are the new SQL statements, clauses, and keywords that have been implemented in Access 2000. Although there will certainly be SQL statements in this article that have been included in previous versions of Access, there will also be some SQL statements that are making their appearance in Access for the first time, both in this article and in the Advanced SQL article.
- Finally, SQL statements about security and multiuser solutions are saved for the advanced article because they are often useful in more complex applications.

How Can Intermediate SQL Be Beneficial?

By using intermediate SQL, you can add greater power and flexibility to your Access applications. Although simple and straightforward SQL statements can accomplish many great things, using more complex statements will expand the range of ways to access and process the information in your database. Using intermediate SQL will also enable you to take greater control of how your database is used and maintained.

SQL Enhancements

In Access 2000, many enhancements were made to the SQL implementation in the Microsoft Jet 4.0 data engine in order to support new features of Access, to conform more closely to the ANSI-92 standard, and to allow for greater compatibility between Access and Microsoft® SQL ServerÂ™. The Jet database engine now has two modes of SQL syntax: one that supports previously used SQL syntax, and one that supports the new SQL syntax. It is very important to note that some of the new SQL syntax is available in code only when you use ActiveX® Data Objects (ADO) and the Jet OLE DB provider, and is not currently available through the Access SQL View user interface or DAO. This article points out when a certain SQL command is available only through the Jet OLE DB provider and ADO.

Using ADO vs. DAO

In previous versions of Access, Data Access Objects (DAO) was the primary data access method. That has now changed. Although DAO is still supported, the new way to access data is with ADO. ADO is part of Microsoft's Universal Data Access strategy, which has the basic premise of being able to access any kind of data wherever it may exist, whether in a database, a directory structure, or some custom type of data storage.

ADO is important to a discussion of Microsoft Jet SQL because, as mentioned previously, some of the new SQL statements are available only when using ADO and the Jet OLE DB provider. In this article, and in the sample database that accompanies it, all code is written with ADO. The SQL statements that are not specifically noted as being available only through ADO can be executed in either the Access SQL View user interface or DAO. Although a thorough discussion of ADO is beyond the scope of this article, you can find the latest information at http://www.microsoft.com/data/ado/.

SQL Coding Conventions

This article uses a consistent method of SQL coding conventions. As with all coding conventions, the idea is to display the code in such a way as to make it easy to read and understand. This is accomplished by using a mix of white space, new lines, and uppercase keywords. In general, use uppercase for all SQL keywords, and if you must break the line of SQL code, try to do so with a major section of the SQL statement. You'll get a better feel for it after seeing a few examples.

Poorly formatted SQL code

```
CREATE TABLE tblCustomers (CustomerID INTEGER NOT NULL, [Last Name]
TEXT(50) NOT NULL, [First Name] TEXT(50) NOT NULL, Phone TEXT(10), Email
TEXT(50))
```

Well-formatted SQL code

```
CREATE TABLE tblCustomers (
...CustomerID INTEGER NOT NULL,
...[Last Name] TEXT(5) NOT NULL,
...[First Name] TEXT(50) NOT NULL,
...Phone TEXT(10),
...Email TEXT(50))
```

Intermediate Data Definition Language

The article "Fundamental Microsoft Jet SQL for Access 2000" showed how to build the basic database objects that form a relational database. The following sections of this article discuss intermediate Data Definition Language (DDL) statements that will allow you to enhance and/or modify those basic structures.

Altering Tables

After you have created and populated a table, you may need to modify the table's design. To do so, use the ALTER TABLE statement. But be forewarned, altering an existing table's structure may cause you to loose some of the data. For example, changing a field's data type can result in data loss or rounding errors, depending on the data types you are using. It can also break other parts of your application that may refer to the changed field. You should always use extra caution before modifying an existing table's structure.

With the ALTER TABLE statement, you can add, remove, or change a column (or field), and you can add or remove a constraint. You can also declare a default value for a field; however, you can alter only one field at a time. Let's suppose that we have an invoicing database, and we want to add a field to the Customers table. To add a field with the ALTER TABLE statement, use the ADD COLUMN clause with the name of the field, its data type, and the size of the data type, if it is required.

```
ALTER TABLE tblCustomers
   ADD COLUMN Address TEXT(30)
```

To change the data type or size of a field, use the ALTER COLUMN clause with the name of the field, the desired data type, and the desired size of the data type, if it is required.

```
ALTER TABLE tblCustomers
   ALTER COLUMN Address TEXT(40)
```

If you want to change the name of a field, you will have to remove the field and then recreate it. To remove a field, use the DROP COLUMN clause with the field name only.

```
ALTER TABLE tblCustomers
   DROP COLUMN Address
```

Note that using this method will eliminate the existing data for the field. If you want to preserve the existing data, you should change the field's name with the table design mode of the Access user interface, or write code to preserve the current data in a temporary table and append it back to the renamed table.

A default value is the value that is entered in a field any time a new record is added to a table and no value is specified for that particular column. To set a default value for a field, use the DEFAULT keyword after declaring the field type in either an ADD COLUMN or ALTER COLUMN clause.

```
ALTER TABLE tblCustomers
   ALTER COLUMN Address TEXT(40) DEFAULT Unknown
```

Notice that the default value is not enclosed in single quotes. If it were, the quotes would also be inserted into the record. The DEFAULT keyword can also be used in a CREATE TABLE statement.

```
CREATE TABLE tblCustomers (
   CustomerID INTEGER CONSTRAINT PK_tblCustomers
      PRIMARY KEY,
   [Last Name] TEXT(50) NOT NULL,
   [First Name] TEXT(50) NOT NULL,
   Phone TEXT(10),
```

```
    Email TEXT(50),
    Address TEXT(40) DEFAULT Unknown)
```

> **Note** The DEFAULT statement can be executed only through the Jet OLE DB provider and ADO. It will return an error message if used through the Access SQL View user interface.

The next section discusses how to use constraints with the ALTER TABLE statement. For more information about using the ALTER TABLE statement, type **alter table** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

Constraints

The article "Fundamental Microsoft Jet SQL for Access 2000" discusses constraints as the way to establish relationships between tables. Constraints can also be used to establish primary keys and referential integrity, and to restrict values that can be inserted into a field. In general, constraints can be used to preserve the integrity and consistency of the data in your database.

There are two types of constraints: a single-field or field-level constraint, and a multi-field or table-level constraint. Both kinds of constraints can be used in either the CREATE TABLE or the ALTER TABLE statement.

A single-field constraint, also known as a column-level constraint, is declared with the field itself, after the field and data type have been declared. Let's use the Customers table and create a single-field primary key on the CustomerID field. To add the constraint, use the CONSTRAINT keyword with the name of the field.

```
ALTER TABLE tblCustomers
   ALTER COLUMN CustomerID INTEGER
   CONSTRAINT PK_tblCustomers PRIMARY KEY
```

Notice that the name of the constraint is given. You could use a shortcut for declaring the primary key that leaves off the CONSTRAINT clause entirely.

```
ALTER TABLE tblCustomers
   ALTER COLUMN CustomerID INTEGER PRIMARY KEY
```

However, using the shortcut method will cause Access to randomly generate a name for the constraint, which will make it difficult to reference in code. It is a good idea always to name your constraints.

To drop a constraint, use the DROP CONSTRAINT clause with the ALTER TABLE statement, and supply the name of the constraint.

```
ALTER TABLE tblCustomers
   DROP CONSTRAINT PK_tblCustomers
```

Constraints can also be used to restrict the allowable values for a field. You can restrict values to NOT NULL or UNIQUE, or you can define a *check constraint*, which is a type of business rule that can be applied to a field. Let's say that you want to restrict (or constrain) the values of the first name and last name fields to be unique, meaning that there should never be a combination of first name and last name that is the same for any two records in the table. Because this is a multi-field constraint, it is declared at the table level, not the field level. Use the ADD CONSTRAINT clause and define a multi-field list.

```
ALTER TABLE tblCustomers
   ADD CONSTRAINT CustomerNames UNIQUE
   ([Last Name], [First Name])
```

> **Note** You probably would not want to limit proper names to unique values in a real application; we do it here only to demonstrate how you can use constraints.

A *check constraint* is a powerful new SQL feature that allows you to add data validation to a table by creating an expression that can refer to a single field, or multiple fields across one or more tables. Suppose that you want to make sure that the amounts entered in an invoice record are always greater than $0.00. To do so, use a check constraint by declaring the CHECK keyword and your validation expression in the ADD CONSTRAINT clause of an

ALTER TABLE statement.

```
ALTER TABLE tblInvoices
   ADD CONSTRAINT CheckAmount
   CHECK (Amount > 0)
```

> **Note** The check constraint statement can only be executed through the Jet OLE DB provider and ADO; it will return an error message if used though the Access SQL View user interface. Also note that to drop a check constraint, you must issue the DROP CONSTRAINT statement through the Jet OLE DB provider and ADO. Also, if you do define a check constraint: (1) it won't show as a validation rule in the Access user interface (UI), (2) you can't define the **ValidationText** property so that a generic error message will display in the Access UI, and (3) you won't be able to delete the table through the Access UI or from code until you drop the constraint by using a DROP CONSTRAINT statement from ADO.

The expression used to define a check constraint can also refer to more than one field in the same table, or to fields in other tables, and can use any operations that are valid in Microsoft Jet SQL, such as SELECT statements, mathematical operators, and aggregate functions. The expression that defines the check constraint can be no more than 64 characters long.

Let's suppose that you want to check each customer's credit limit before he or she is added to the Customers table. Using an ALTER TABLE statement with the ADD COLUMN and CONSTRAINT clauses, let's create a constraint that will look up the value in the CreditLimit table to verify the customer's credit limit. Use the following SQL statements to create the tblCreditLimit table, add the CustomerLimit field to the tblCustomers table, add the check constraint to the tblCustomers table, and test the check constraint.

```
CREATE TABLE tblCreditLimit (
   Limit DOUBLE)

INSERT INTO tblCreditLimit
   VALUES (100)

ALTER TABLE tblCustomers
   ADD COLUMN CustomerLimit DOUBLE

ALTER TABLE tblCustomers
   ADD CONSTRAINT LimitRule
   CHECK (CustomerLimit <= (SELECT Limit
      FROM tblCreditLimit))

UPDATE TABLE tblCustomers
   SET CustomerLimit = 200
   WHERE CustomerID = 1
```

Note that when you execute the UPDATE TABLE statement, you receive a message indicating that the update did not succeed because it violated the check constraint. If you update the CustomerLimit field to a value that is equal to or less than 100, the update will succeed.

Cascading updates and deletions

Constraints can also be used to establish *referential integrity* between database tables. Having referential integrity means that the data is consistent and uncorrupted. For example, if you deleted a customer record but that customer's shipping record remained in the database, the data would be inconsistent because you now have an orphaned record in the shipping table. Referential integrity is established when you build a relationship between tables. In addition to establishing referential integrity, you can also ensure that the records in the referenced tables stay in sync by using cascading updates and deletes. For example, when the cascading updates and deletes are declared, if you delete the customer record, the customer's shipping record is deleted automatically.

To enable cascading updates and deletions, use the ON UPDATE CASCADE and/or ON DELETE CASCADE keywords in the CONSTRAINT clause of an ALTER TABLE statement. Note that they must be applied to the foreign key.

```
ALTER TABLE tblShipping
   ADD CONSTRAINT FK_tblShipping
```

```
    FOREIGN KEY (CustomerID) REFERENCES
       tblCustomers (CustomerID)
    ON UPDATE CASCADE
    ON DELETE CASCADE
```

Foreign keys

When dealing with foreign keys, the concept of a *fast foreign key* may be useful. A fast foreign key is a foreign key that has no index. Although this may seem counter-intuitive, there is a valid explanation for it. By default, when a foreign key is defined, an index based on the column(s) in the foreign key is created automatically. In many instances this enhances performance when executing operations that maintain referential integrity. However, if there are many duplicated values in the foreign key field, the foreign key index will affect performance when records are added and deleted from the table. To prevent the automatic creation of indexes on foreign keys, use the NO INDEX keywords in the declaration of the foreign key.

```
 ALTER TABLE tblInvoices
    ADD CONSTRAINT FK_tblInvoices
    FOREIGN KEY NO INDEX (CustomerID) REFERENCES
       tblCustomers (CustomerID)
    ON UPDATE CASCADE
    ON DELETE CASCADE
```

> **Note** The fast foreign key statement can only be executed through the Jet OLE DB provider and ADO. It will return an error message if used through the Access SQL View user interface. Also note that to drop a fast foreign key, you must issue the DROP CONSTRAINT statement through the Jet OLE DB provider and ADO.

Another example of a situation where a fast foreign key would be useful is in an order entry database application. Assume that there is a table called CustomerTypes that identifies what type of customers are being tracked, a Customer table, and an Orders table. Assume that there are 10 rows in the CustomerTypes table, 100,000 rows in the Customer table, and 350,000 rows in the Orders table. A good choice for the Customers table would be a fast foreign key that references the primary key in the CustomerTypes table. This is because there is a maximum of 10 unique values out of 100,000 rows. An index here has little value for retrieving data and would be a drag on concurrency and inserts, deletions, and updates in the CustomerType column.

On the other hand, the fast foreign key would probably *not* be useful when applied to the CustomerID column in the Orders table, because those values are likely to be unique, since each represents a different customer. In this instance having the foreign key indexed in the regular manner is very advantageous because it is used in joins and other lookup criteria.

> **Note** Although most of the examples in this section used the ALTER TABLE statement, all of them could have been written with the CREATE TABLE statement.

For more information about the CONSTRAINT clause, type **constraints** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

Data Types

In an effort to ease the scaling and transition of Access applications based on the Jet database engine to those based on Microsoft SQL Server or MSDE, the Jet database engine has changed some of its implementation of data types, and added some new data type synonyms. The following sections discuss the primary Jet data types and how they are implemented.

The TEXT data types

A *TEXT data type* is a data type that establishes a field that can store text, or a combination of text and numbers (alphanumeric), but whose numbers are not used for calculations. Examples are phone numbers or e-mail addresses. When you create your tables through the Access user interface, you have two basic text types: TEXT and MEMO. But when you use SQL statements such as CREATE TABLE or ALTER TABLE to create your tables, you have many other synonyms of the TEXT and MEMO types to choose from.

In general, text fields can be up to 255 characters, while memo fields can be up to 65,535 characters, but if the memo field does not contain any binary data, then its only limit is the maximum size of the database

(approximately 2.14GB or 1,070,000,000 Unicode characters). In addition, unused portions of text fields are not reserved in memory.

Following is a table that lists the basic Jet text data types, various synonyms, and the number of bytes allocated for each.

Jet Data Type	Synonyms	Storage Size
TEXT	TEXT, TEXT(n), CHAR, CHAR(n), ALPHANUMERIC, ALPHANUMERIC(n), STRING, STRING(n), VARCHAR, VARCHAR(n), NTEXT(n), NCHAR, NCHAR(n), CHAR VARYING, CHAR VARYING(n), CHARACTER VARYING, CHARACTER VARYING(n), NATIONAL CHAR, NATIONAL CHAR(n), NATIONAL CHARACTER, NATIONAL CHARACTER(n), NATIONAL CHAR VARYING, NATIONAL CHAR VARYING(n), NATIONAL CHARACTER VARYING, NATIONAL CHARACTER VARYING(n)	Up to 255 characters, 2 bytes per character unless compressed
MEMO	LONGTEXT, LONGCHAR, NOTE, NTEXT	65,535 characters; 2.14 GB if not binary data

The following CREATE TABLE statement shows the variety of TEXT and MEMO data type synonyms that can be used to create a table through the Access SQL View user interface.

```
CREATE TABLE tblUITextDataTypes (
   Field1_TEXT TEXT,
   Field2_TEXT25 TEXT(25),
   Field3_MEMO MEMO,
   Field4_CHAR CHAR,
   Field5_CHAR25 CHAR(25),
   Field6_LONGTEXT LONGTEXT,
   Field7_LONGCHAR LONGCHAR,
   Field8_ALPHA ALPHANUMERIC,
   Field9_ALPHA25 ALPHANUMERIC(25),
   Field10_STRING STRING,
   Field11_STRING25 STRING(25),
   Field12_VARCHAR VARCHAR,
   Field13_VARCHAR25 VARCHAR(25),
   Field14_NOTE NOTE)
```

If you view the table design of tblUITextDataTypes through the Access user interface, you will see that the MEMO, LONGTEXT, LONGCHAR, and NOTE synonyms result in a MEMO data type. All of the other synonyms result in a TEXT data type. For those TEXT data types that do not have the length declared, the length defaults to 255 characters.

Although the SQL statement above can also be executed through the Jet OLE DB provider and ADO, there are other variations of the TEXT and MEMO data types that can *only* be executed through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblCodeTextDataTypes
   Field1_NTEXT NTEXT,
   Field2_NTEXT25 NTEXT(25),
   Field3_NCHAR NCHAR,
   Field4_NCHAR NCHAR(25),
   Field5_VARYING CHAR VARYING,
   Field6_VARYING CHAR VARYING(25),
   Field7_VARYING CHARACTER VARYING,
   Field8_VARYING CHARACTER VARYING(25),
   Field9_NATIONAL NATIONAL CHAR,
   Field10_NATIONAL NATIONAL CHAR(25),
   Field11_NATIONAL NATIONAL CHARACTER,
   Field12_NATIONAL NATIONAL CHARACTER(25),
   Field13_NATIONAL NATIONAL CHAR VARYING,
   Field14_NATIONAL NATIONAL CHAR VARYING(25),
   Field15_NATIONAL NATIONAL CHARACTER VARYING,
   Field16_NATIONAL NATIONAL CHARACTER VARYING(25))
```

If you view the table design of tblUITextDataTypes through the Access user interface, you will see that only the NCHAR data type results in a MEMO data type. All of the other data types result in a TEXT data type. For those TEXT data types that do not have the length declared, the length defaults to 255 characters.

> **Note** The data types listed in the previous SQL statement can be executed only through the Jet OLE DB provider and ADO. They will result in an error message if used through the Access SQL View user interface. Also note that if you create a field with the TEXT data type through the Jet OLE DB provider and ADO, it will result in a data type of MEMO when you view the table design through the Access user interface.

Unicode compression

With the Microsoft Jet 4.0 database engine, all data for the TEXT data types are now stored in the Unicode 2-byte character representation format. It replaces the Multi-byte Character Set (MBCS) format that was used in previous versions. Although Unicode representation requires more space to store each character, columns with TEXT data types can be defined to automatically compress the data if it is possible to do so.

When you create TEXT data types with SQL, the Unicode compression property defaults to No. To set the Unicode compression property to Yes, use the WITH COMPRESSION (or WITH COMP) keywords at the field-level declaration.

The following CREATE TABLE statement creates a new customers table, this time setting the Unicode compression properties to Yes.

```
CREATE TABLE tblCompressedCustomers (
   CustomerID INTEGER CONSTRAINT
   PK_tblCompCustomers PRIMARY KEY,
   [Last Name] TEXT(50) WITH COMP NOT NULL,
   [First Name] TEXT(50) WITH COMPRESSION NOT NULL,
   Phone TEXT(10),
   Email TEXT(50),
   Address TEXT(40) DEFAULT Unknown)
```

Note that the WITH COMPRESSION and WITH COMP keywords are declared before the NOT NULL keywords. You can also change an existing field's Unicode compression property with an ALTER TABLE statement, like this:

```
ALTER TABLE tblCustomers
   ALTER COLUMN [Last Name] TEXT(50) WITH COMPRESSION
```

> **Note** The WITH COMPRESSION and WITH COMP keywords listed in the previous SQL statements can be executed only through the Jet OLE DB provider and ADO. They will result in an error message if used through the Access SQL View user interface.

Which of the data types you choose when declaring your table design depends on the goals of your application. If you know that the application will always be based on the Jet database, use the data types that you are most comfortable with. But if your application may eventually be scaled to an ODBC-compliant database, such as SQL Server or MSDE, use the data types that will make the transition the easiest.

The NUMERIC data types

A NUMERIC data type establishes a field that can store numbers that can be used in calculations. Typically, what sets one NUMERIC type apart from another is the number of bytes used to store the data, which also affects the precision of the number stored in that field. Many of the Jet SQL data types have synonyms that you can use in declaring the data type. Which one you use depends on if the table will remain in a Jet database or if it will be scaled to a database server, such as Microsoft SQL Server. If it will be scaled, you should use the data type declaration that will make the transition the easiest.

Following is a table that lists the basic Jet NUMERIC data types, various synonyms, and the number of bytes allocated for each.

Jet Data Type	Synonyms	Storage Size
TINYINT	INTEGER1, BYTE	1 byte
SMALLINT	SHORT, INTEGER2	2 bytes
INTEGER	LONG, INT, INTEGER4	4 bytes
REAL	SINGLE, FLOAT4, IEEESINGLE	4 bytes
FLOAT	DOUBLE, FLOAT8, IEEEDOUBLE, NUMBER	8 bytes
DECIMAL	NUMERIC, DEC	17 bytes

The following CREATE TABLE statement shows the variety of NUMERIC data types that can be used to create a table through the Access SQL View user interface.

```
CREATE TABLE tblUINumericDataTypes (
   Field1_INT INT,
   Field2_INTEGER INTEGER,
   Field3_LONG LONG,
   Field4_INTEGER1 INTEGER1,
   Field5_BYTE BYTE,
   Field6_NUMERIC NUMERIC,
   Field7_REAL REAL,
   Field8_SINGLE SINGLE,
   Field9_FLOAT FLOAT,
   Field10_FLOAT4 FLOAT4,
   Field11_FLOAT8 FLOAT8,
   Field12_DOUBLE DOUBLE,
   Field13_IEEESINGLE IEEESINGLE,
   Field14_IEEEDOUBLE IEEEDOUBLE,
   Field15_NUMBER NUMBER,
   Field16_SMALLINT SMALLINT,
   Field17_SHORT SHORT,
   Field18_INTEGER2 INTEGER2,
   Field19_INTEGER4 INTEGER4)
```

Although the SQL statement above can also be executed through the Jet OLE DB provider and ADO, there are other variations of the NUMERIC data type that can *only* be executed through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblCodeNumericDataTypes (
   Field1_TINYINT TINYINT,
   Field2_DECIMAL DECIMAL,
   Field3_DEC DECIMAL,
   Field4_DPRECISION DOUBLE PRECISION)
```

> **Note** The data types listed in the previous SQL statement can be executed only through the Jet OLE DB provider and ADO. They will result in an error message if used through the Access SQL View user interface. Also note that if you create a field with the NUMERIC data type through the Access SQL View, it will result in a DOUBLE data type when you view the table design through the Access user interface. But if you create the NUMERIC data type through the Jet OLE DB provider and ADO, it will result in a data type of DECIMAL when you view the table design through the Access user interface.

With the new DECIMAL data type, you can also set the precision and scale of the number. The *precision* is the total number of digits that the field can contain, while the *scale* determines how many of those digits can be to the right of the decimal point. For the precision, the default is 18 and the maximum allowed value is 28. For the scale, the default is 0 and the maximum allowed value is 28.

```
CREATE TABLE tblDecimalDataTypes (
   DefaultType DECIMAL,
   SpecificType DECIMAL(10,5))
```

The CURRENCY data type

The CURRENCY data type is used to store numeric data that contains up to 15 digits on the left side of the decimal point, and up to 4 digits on the right. It uses 8 bytes of memory for storage, and its only synonym is MONEY.

The following CREATE TABLE statement shows the CURRENCY data type that can be used to create a table through the Access SQL View user interface or through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblCurrencyDataTypes (
   Field1_CURRENCY CURRENCY,
   Field2_MONEY MONEY)
```

The BOOLEAN data type

The BOOLEAN data types are logical types that result in either **True** or **False** values. They use 1 byte of

memory for storage, and their synonyms are BIT, LOGICAL, LOGICAL1, and YESNO. A **True** value is equal to –1 while a **False** value is equal to 0.

The following CREATE TABLE statement shows the different synonyms that can be used to create the BOOLEAN data type through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblUIBooleanDataTypes (
   Field1_BIT BIT,
   Field2_LOGICAL LOGICAL,
   Field3_LOGICAL1 LOGICAL1,
   Field4_YESNO YESNO)
```

The BINARY data type

The BINARY data type is used to store a small amount of any type of data in its native, binary format. It uses 1 byte of memory for each character stored, and you can optionally specify the number of bytes to be allocated. If the number of bytes is not specified, it defaults to 510 bytes, which is the maximum number of bytes allowed. Its synonyms are BINARY, VARBINARY, and BINARY VARYING. The BINARY data type is not available in the Access user interface.

The following CREATE TABLE statement shows the variety of BINARY data types that can be used to create a table through the Access SQL View user interface.

```
CREATE TABLE tblUIBinaryDataTypes (
   Field1_BINARY BINARY,
   Field2_BINARY250 BINARY(250),
   Field3_VARBINARY VARBINARY,
   Field4_VARBINARY250 VARBINARY(250))
```

Although the SQL statement above can be executed through the Jet OLE DB provider and ADO, there are other synonyms of the binary data type that can *only* be executed through the Jet OLE DB provider and ADO, like this:

```
CREATE TABLE tblCodeBinaryDataTypes (
   Field1_BVARYING BINARY VARYING,
   Field2_BVARYING250 BINARY VARYING(250))
```

The OLEOBJECT data types

The OLEOBJECT data types are used to store large binary objects such as Word documents or Excel spreadsheets. The number of bytes is not specified, and the maximum size is 2.14 gigabytes. Its synonyms are IMAGE, LONGBINARY, GENERAL, and OLEOBJECT.

The following CREATE TABLE statement shows the OLEOBJECT data type that can be used to create a table through the Access SQL View user interface or through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblImageDataTypes (
   Field1_IMAGE IMAGE,
   Field2_LONGBINARY LONGBINARY,
   Field3_GENERAL GENERAL,
   Field4_OLEOBJECT OLEOBJECT)
```

The DATETIME data type

The DATETIME data type is used to store date, time, and combination date/time values for the years ranging from 100 to 9999. It uses 8 bytes of memory for storage, and its synonyms are DATE, TIME, DATETIME, and TIMESTAMP.

The following CREATE TABLE statement shows the synonyms of the DATETIME data type that can be used to create a table through the Access SQL View user interface or through the Jet OLE DB provider and ADO.

```
CREATE TABLE tblDateTimeDataTypes (
   Field1_DATE DATE,
   Field2_TIME TIME,
   Field3_DATETIME DATETIME,
```

```
   Field4_TIMESTAMP TIMESTAMP)
```

The COUNTER data type

The COUNTER data type is used to store long integer values that automatically increment whenever a new record is inserted into a table. With the COUNTER data type, you can optionally set a seed value and an increment value. The *seed value* is the starting value that will be entered in the field the first time a new record is inserted into the table. The *increment value* is the number that is added to the last counter value to establish the next counter value. If the seed and increment values are not specified, both the seed and increment values default to 1. You can have only one COUNTER field in a table, and the synonyms are COUNTER, AUTOINCREMENT, and IDENTITY.

The following CREATE TABLE statements show the synonyms of the COUNTER data type that can be used to create a table through the Access SQL View user interface.

```
CREATE TABLE tblUICounterDataTypes (
   Field1 COUNTER,
   Field2 TEXT(10))
```

Note that since the seed and increment values were not specified, they each defaulted to 1. Another way to declare a COUNTER data type is to use the AUTOINCREMENT keyword, like this:

```
CREATE TABLE tblUICounterDataTypes (
   Field1 AUTOINCREMENT(10,5),
   Field2 TEXT(10))
```

This time the seed and increment values were specified. The starting value will be 10, and it will increment by 5. Although the SQL statements above can also be executed through the Jet OLE DB provider and ADO, there is another variation of the counter data type that can *only* be executed through the Jet OLE DB provider and ADO. The IDENTITY keyword can also be used to declare a COUNTER data type, and it is compatible with the SQL Server IDENTITY data type.

```
CREATE TABLE tblCodeCounterDataTypes
   Field1_IDENTITY IDENTITY(10,5),
   Field2 TEXT(10))
```

The seed and increment values can be modified with an ALTER TABLE statement, and all new rows inserted after the new seed and increment values will be based on them. However, COUNTER data types are often used in primary keys, which must be unique for every row. If you change the seed and increment values, thus possibly generating duplicate values for the primary key field, an error will occur.

```
ALTER TABLE tblUICounterDataTypes
ALTER COLUMN Field1 COUNTER(10,10)
```

> **Note** You cannot use the ALTER TABLE statement to change an existing column's data type to a COUNTER data type if the existing column already contains data.

In previous versions of the Jet database, the seed value would be reset to the maximum available counter value after compacting the database. This is still true in Jet 4.0, as long as the seed and increment values are set to the default of 1. If you specify seed and increment values that are not equal to the default, compacting the database will not reset the seed value.

The @@IDENTITY variable

The @@IDENTITY variable is a global SQL variable that you can use to retrieve the last value used in a COUNTER data type column. You can't specify a table name when retrieving the @@IDENTITY variable. The value returned is always from the last table with a COUNTER field that had a new record added to it from code. Use the SELECT statement to retrieve the @@IDENTITY value.

```
SELECT @@IDENTITY
```

To add a value to the @@IDENTITY value, enclose the variable in square brackets.

```
SELECT [@@IDENTITY] + 1
```

> **Note** The @@IDENTITY variable listed in the previous SQL statements can be executed only through the Jet OLE DB provider and ADO; it will result in a value of 0 if used through the Access SQL View user interface. In addition, the variable is set only when records are inserted through programming code. If a record is inserted through the user interface, either with datasheets, forms, or SQL statements in the Access SQL View window, the @@IDENTITY variable will return 0. For this reason, the value of @@IDENTITY is only accurate immediately after adding a record from code.

For more information about data types, type **Jet SQL data types** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

Intermediate Data Manipulation Language

The article "Fundamental Microsoft Jet SQL for Access 2000" showed how to use SQL to retrieve and manage the information stored in a database. In the sections that follow in this article, we discuss intermediate Data Manipulation Language (DML) statements that will allow you to have even greater control over how information can be retrieved and manipulated.

Predicates

A predicate is an SQL clause that qualifies a SELECT statement, similar to a WHERE clause, except that the predicate is declared before the column list. Predicates can further restrict the set of records you are retrieving, and in some instances filter out any duplicate data that may exist.

The ALL keyword

The ALL keyword is the default keyword that is used when no predicate is declared in an SQL statement. It simply means that all records will be retrieved that match the qualifying criteria of the SQL statement. Returning to our invoices database example, let's select all records from the customers table:

```
SELECT *
   FROM tblCustomers
```

Notice that although the ALL keyword was not declared, it is the default predicate. We could have written the statement like this:

```
SELECT ALL *
   FROM tblCustomers
```

The DISTINCT keyword

The DISTINCT keyword is used to control how duplicate values in a result set are handled. Based on the column (s) specified in the field list, those rows that have duplicate values in the specified columns are filtered out. If more than one column is specified, it is the combination of all of the columns that is used as the filter. For example, if you query the Customers table for distinct last names, only the unique names will be returned; any duplicate names will result in only one instance of that name in the result set.

```
SELECT DISTINCT [Last Name]
   FROM tblCustomers
```

It is important to note that the result set returned by a query that uses the **DISTINCT** keyword cannot be updated; it is read-only.

The DISTINCTROW keyword

The DISTINCTROW keyword is similar to the DISTINCT keyword except that it is based on entire rows, not just individual fields. It is useful only when based on multiple tables, and only when you select fields from some, but not all, of the tables. If you base your query on one table, or select fields from every table, the DISTINCTROW keyword essentially acts as an ALL keyword.

For example, in our invoices database, every customer can have no invoices, or one or more invoices. Let's suppose that we want to find out how many customers have one or more invoices. We will use the DISTINCTROW keyword to determine our list of customers.

```
SELECT DISTINCTROW [Last Name], [First Name]
   FROM tblCustomers INNER JOIN tblInvoices
   ON tblCustomers.CustomerID = tblInvoices.CustomerID
```

If we had left off the DISTINCTROW keyword, we would have gotten a row returned for every invoice each customer has. (The INNER JOIN statement will be covered in a later section.)

The TOP keyword

The TOP keyword is used to return a certain number of rows that fall at the top or bottom of a range that is specified by an ORDER BY clause. The ORDER BY clause is used to sort the rows in either ascending or descending order. If there are equal values present, the TOP keyword will return all rows that have the equal value. Let's say that we want to determine the highest three invoice amounts in our invoices database. We'd write a statement like this:

```
SELECT TOP 3 InvoiceDate, Amount
   FROM tblInvoices
   ORDER BY Amount DESC
```

We can also use the optional PERCENT keyword with the TOP keyword to return a percentage of rows that fall at the top or bottom of a range that is specified by an ORDER BY clause. The code looks like this:

```
SELECT TOP 25 PERCENT InvoiceDate, Amount
   FROM tblInvoices
   ORDER BY Amount DESC
```

Note that if you do not specify an ORDER BY clause, the TOP keyword will not be helpful: it will return a random sampling of rows.

For more information about predicates, type **all, distinct predicates** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

SQL Expressions

An *SQL expression* is a string that is used as part of an SQL statement that resolves to a single value. You can use any combination of operators, constants, literal values, functions, field names, controls, or properties to build your SQL expressions. The article "Fundamental Microsoft Jet SQL for Access 2000" describes how you can use expressions in WHERE clauses to qualify SQL statements; and in the following sections of this article, we examine different SQL operators that can be used in expressions.

The IN operator

The IN operator is used to determine if the value of an expression is equal to any of several values in a specified list. If the expression matches a value in the list, the IN operator returns **True**. If it is not found, the IN operator returns **False**. Let's suppose that we want to find all shipping addresses that are in the states of Washington and Georgia. Although we could write an SQL statement with a long WHERE clause that uses the AND logical operator, using the IN operator will shorten our statement.

```
SELECT *
   FROM tblShipping
   WHERE State IN ('WA','GA')
```

By using the NOT logical operator, we can check the opposite of the IN operator. This statement will return all shipping addresses that are *not* in Washington State:

```
SELECT *
   FROM tblShipping
   WHERE State NOT IN ('WA')
```

The BETWEEN operator

The BETWEEN operator is used to determine if the value of an expression falls within a specified range of values. If the expression's value falls within the specified range, including both the beginning and ending range values, the BETWEEN operator returns **True**. If the expression's value does not fall within the range, the BETWEEN operator returns **False**. Let's suppose that we want to find all invoices that have an amount between $50 and $100 dollars. We'd use the BETWEEN operator in the WHERE clause with the AND keyword that specifies the range.

```
SELECT *
   FROM tblInvoices
   WHERE Amount BETWEEN 50 AND 100
```

By using the NOT logical operator, we can check the opposite of the BETWEEN operator to find invoice amounts that fall *outside* that range.

```
SELECT *
   FROM tblInvoices
   WHERE Amount NOT BETWEEN 50 AND 100
```

Note that the range can be in reverse order and still achieve the same results (BETWEEN 100 AND 50), but many ODBC-compliant databases require that the range follow the begin-value-to-end-value method. If you plan for your application to be scaled or upsized to an ODBC-compliant database, you should always use the begin-value-to-end-value method.

The LIKE operator

The LIKE operator is used to determine if the value of an expression compares to that of a pattern. A *pattern* is either a full string value, or a partial string value that also contains one or more wildcard characters. By using the LIKE operator, you can search a field within a result set and find all of the values that match the specified pattern.

```
SELECT *
   FROM tblCustomers
   WHERE [Last Name] LIKE 'Johnson'
```

To return all customers who have a last name that starts with the letter J, use the asterisk wildcard character.

```
SELECT *
   FROM tblCustomers
   WHERE [Last Name] LIKE 'J*'
```

By using the NOT logical operator, we can check the opposite of the LIKE operator and filter out all the Johnsons from the list.

```
SELECT *
   FROM tblCustomers
   WHERE [Last Name] NOT LIKE 'Johnson'
```

There are a variety of wildcard characters that you can use in the LIKE operator patterns, as shown in the following table.

Wildcard Character	Description
***** (asterisk)	Matches any number of characters and can be used anywhere in the pattern string.
% (percent sign)	Matches any number of characters and can be used anywhere in the pattern string. (ADO and the Jet OLE DB provider only)
? (question mark)	Matches any single character and can be used anywhere in the pattern string.
_ (underscore)	Matches any single character and can be used anywhere in the pattern string. (ADO and the Jet OLE DB provider only)
# (number sign)	Matches any single digit and can be used anywhere in the pattern string.
[] (square brackets)	Matches any single character within the list that is enclosed within brackets, and can be used anywhere in the pattern string.

! (exclamation point)	Matches any single character not in the list that is enclosed within the square brackets.
- (hyphen)	Matches any one of a range of characters that is enclosed within the square brackets.

> **Note** The "%" and "_" wildcard characters in the previous table can be executed only through the Jet OLE DB provider and ADO. They will yield an empty result set if used through the Access SQL View user interface.

For more information about wildcard characters, type **wildcard characters** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

The IS NULL operator

A null value is one that indicates missing or unknown data. The IS NULL operator is used to determine if the value of an expression is equal to the null value.

```
SELECT *
   FROM tblInvoices
   WHERE Amount IS NULL
```

By adding the NOT logical operator, we can check the opposite of the IS NULL operator. In this case, the statement will weed out invoices with missing or unknown amounts.

```
SELECT *
   FROM tblInvoices
   WHERE Amount IS NOT NULL
```

The SELECT INTO Statement

The SELECT INTO statement, also known as a *make-table query*, can be used to create a new table from one or more existing tables. The table it produces can be based on any valid SELECT statement. The SELECT INTO statement can be used to archive records, create backup tables, or create new tables in an external database.

When you use the SELECT INTO statement to create a new table, all of the new table's fields inherit the data types from the original tables. However, no other table properties, such as primary keys or indexes, are created in the new table. You can of course add these properties by using the ALTER TABLE statement once the new table has been created.

To create a new table, use the SELECT INTO statement with a field list for the columns you want to include in the table, a name for the new table, and then supply the source of data in the FROM clause.

```
SELECT *
   INTO tblNewCustomers
   FROM tblCustomers
```

To specify certain fields for the new table, replace the asterisk with the original table's field names in the field list, and use the AS keyword to name the column(s) in the new table.

```
SELECT [First Name] & ' ' & [Last Name] AS FullName
   INTO tblNewCustomerNames
   FROM tblCustomers
```

To create the new table in an existing external database, use the IN keyword. If the external database does not exist, the SELECT INTO statement will return an error.

```
SELECT *
   INTO tblNewCustomers
   IN 'C:\Customers.mdb'
   FROM tblCustomers
```

Subqueries

A subquery is a SELECT statement that is used inside another SELECT, SELECT INTO, INSERT INTO, DELETE, or UPDATE statement. It can help further qualify a result set based on the results of another result set. This is called *nesting*, and since a subquery is a SELECT statement, you can also nest a subquery inside another subquery. When you use a subquery in an SQL statement, it can be part of a field list, a WHERE clause, or a HAVING clause.

There are three basic forms of subqueries, and each uses a different kind of predicate.

The IN subquery

The IN subquery is used to check the value of a particular column against a list of values from a column in another table or query. It is limited in that it can return only a single column from the other table. If it returns more than one column, an error is returned. Using the invoices database example, let's write an SQL statement that returns a list of all customers who have invoices.

```
SELECT *
   FROM tblCustomers
   WHERE CustomerID
   IN (SELECT CustomerID FROM tblInvoices)
```

Using the NOT logical operator, we can check the opposite of the IN subquery and generate a list of customers who do *not* have invoices.

```
SELECT *
   FROM tblCustomers
   WHERE CustomerID
   NOT IN (SELECT CustomerID FROM tblInvoices)
```

The ANY/SOME/ALL subqueries

The ANY, SOME, and ALL subquery predicates are used to compare records from the main query with multiple rows from the subquery. The ANY and SOME predicates are synonymous and can be used interchangeably.

Use the ANY or SOME predicate when you need to retrieve from the main query the set of records that satisfy the comparison with any of the records in the subquery. Use the predicate just before the opening parenthesis of the subquery.

```
SELECT *
   FROM tblCustomers
   WHERE CustomerID = ANY
   (SELECT CustomerID FROM tblInvoices)
```

Notice that the result set returned by the SQL statement above is the same as the one returned by the example with the IN subquery. What makes the ANY and SOME predicates different is that they can also be used with relational operators other than Equals (=), such as Greater Than (>) or Less Than (<).

```
SELECT *
   FROM tblCustomers
   WHERE CustomerID > ANY
   (SELECT CustomerID FROM tblInvoices)
```

When you want to retrieve records from the main query that satisfy the comparison with all of the records in the subquery, use the ALL predicate.

```
SELECT *
   FROM tblCustomers
   WHERE CustomerID > ALL
   (SELECT CustomerID FROM tblInvoices)
```

The EXISTS subquery

The EXISTS predicate is used in subqueries to check for the existence of values in a result set. In other words, if the subquery does not return any rows, the comparison is **False**. If it does return one or more rows, the comparison is **True**.

```
SELECT *
   FROM tblCustomers AS A
   WHERE EXISTS
   (SELECT * FROM tblInvoices
   WHERE A.CustomerID = tblInvoices.CustomerID)
```

Note that in the previous SQL statement an alias is used on the tblCustomers table. This is so that we can later refer to it in the subquery. When a subquery is linked to the main query in this manner, it is called a *correlated query*.

By using the NOT logical operator, we can check the opposite of the EXISTS subquery to obtain a result set of customers who do not have any invoices.

```
SELECT *
   FROM tblCustomers AS A
   WHERE NOT EXISTS
   (SELECT * FROM tblInvoices
   WHERE A.CustomerID = tblInvoices.CustomerID)
```

For more information about subqueries, type **SQL subqueries** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

Joins

In a relational database system like Access, you will often need to extract information from more than one table at a time. This can be accomplished by using an SQL *JOIN statement*. A JOIN statement enables you to retrieve records from tables that have defined relationships, whether they are one-to-one, one-to-many, or many-to-many.

INNER JOINs

The INNER JOIN, also known as an *equi-join*, is the most commonly used type of join. This join is used to retrieve rows from two or more tables by matching a field value that is common between the tables. The fields you join on must have similar data types, and you cannot join on MEMO or OLEOBJECT data types. To build an INNER JOIN statement, use the INNER JOIN keywords in the FROM clause of a SELECT statement. Let's use the INNER JOIN to build a result set of all customers who have invoices, plus the dates and amounts of those invoices.

```
SELECT [Last Name], InvoiceDate, Amount
   FROM tblCustomers INNER JOIN tblInvoices
   ON tblCustomers.CustomerID=tblInvoices.CustomerID
   ORDER BY InvoiceDate
```

Notice that the table names are divided by the INNER JOIN keywords and that the relational comparison is after the ON keyword. For the relational comparisons, you can also use the <, >, <=, >=, or <> operators, and you can also use the BETWEEN keyword. Also note that the ID fields from both tables are used only in the relational comparison, they are not part of the final result set.

To further qualify the SELECT statement, we can use a WHERE clause after the join comparison in the ON clause. In the following example, we have narrowed the result set to include only invoices dated after January 1, 1998.

```
SELECT [Last Name], InvoiceDate, Amount
   FROM tblCustomers INNER JOIN tblInvoices
   ON tblCustomers.CustomerID=tblInvoices.CustomerID
   WHERE tblInvoices.InvoiceDate > #01/01/1998#
   ORDER BY InvoiceDate
```

In cases where you need to join more than one table, you can nest the INNER JOIN clauses. In this example, we will build on a previous SELECT statement to create our result set, but we will also include the city and state of each customer by adding the INNER JOIN for the tblShipping table.

```
SELECT [Last Name], InvoiceDate, Amount, City, State
```

```
FROM (tblCustomers INNER JOIN tblInvoices
ON tblCustomers.CustomerID=tblInvoices.CustomerID)
   INNER JOIN tblShipping
   ON tblCustomers.CustomerID=tblShipping.CustomerID
ORDER BY InvoiceDate
```

Note that the first JOIN clause is enclosed in parentheses to keep it logically separated from the second JOIN clause. It is also possible to join a table to itself by using an alias for the second table name in the FROM clause. Let's suppose that we want to find all customer records that have duplicate last names. We do this by creating the alias "A" for the second table and checking for first names that are different.

```
SELECT tblCustomers.[Last Name],
   tblCustomers.[First Name]
   FROM tblCustomers INNER JOIN tblCustomers AS A
   ON tblCustomers.[Last Name]=A.[Last Name]
   WHERE tblCustomers.[First Name]<>A.[First Name]
   ORDER BY tblCustomers.[Last Name]
```

OUTER JOINs

The OUTER JOIN is used to retrieve records from multiple tables while preserving records from one of the tables, even if there is no matching record in the other table. There are two types of OUTER JOINs that the Jet database engine supports: LEFT OUTER JOINs and RIGHT OUTER JOINs. Think of two tables that are beside each other, a table on the left and a table on the right. The LEFT OUTER JOIN will select all rows in the right table that match the relational comparison criteria, and it will also select all rows from the left table, *even if no match exists in the right table*. The RIGHT OUTER JOIN is simply the reverse of the LEFT OUTER JOIN; all rows in the right table are preserved instead.

As an example, let's suppose that we want to determine the total amount invoiced to each customer, but if a customer has no invoices, we want to show it by displaying the word "NONE."

```
SELECT [Last Name] & ', ' &  [First Name] AS Name,
   IIF(Sum(Amount) IS NULL,'NONE',Sum(Amount)) AS Total
   FROM tblCustomers LEFT OUTER JOIN tblInvoices
   ON tblCustomers.CustomerID=tblInvoices.CustomerID
   GROUP BY [Last Name] & ', ' &  [First Name]
```

There are a few things going on in the previous SQL statement. The first is the use of the string concatenation operator "&". This operator allows you to join two or more fields together as one string. The second is the *immediate if* (IIF) statement, which checks to see if the total is null. If it is, the statement returns the word "NONE." If the total is not null, the value is returned. The final thing is the OUTER JOIN clause. Using the LEFT OUTER JOIN preserves the rows in the left table so that we see all customers, even those who do not have invoices.

OUTER JOINs can be nested inside INNER JOINs in a multi-table join, but INNER JOINs cannot be nested inside OUTER JOINs.

The Cartesian product

A term that often comes up when discussing joins is the *Cartesian product*. A Cartesian product is defined as "all possible combinations of all rows in all tables." For example, if you were to join two tables without any kind of qualification or join type, you would get a Cartesian product.

```
SELECT *
   FROM tblCustomers, tblInvoices
```

This is not a good thing, especially with tables that contain hundreds or thousands of rows. You should avoid creating Cartesian products by always qualifying your joins.

The UNION operator

Although the UNION operator, also known as a *union query*, is not technically a join, it is included here because it does involve combining data from multiple sources of data into one result set, which is similar to some types of joins. The UNION operator is used to splice together data from tables, SELECT statements, or queries, while

leaving out any duplicate rows. Both data sources must have the same number of fields, but the fields do not have to be the same data type. Let's suppose that we have an Employees table that has the same structure as the Customers table, and we want to build a list of names and e-mail address by combining both tables.

```
SELECT [Last Name], [First Name], Email
   FROM tblCustomers
UNION
SELECT [Last Name], [First Name], Email
   FROM tblEmployees
```

If we wanted to retrieve all fields from both tables, we could use the TABLE keyword, like this:

```
TABLE tblCustomers
UNION
TABLE tblEmployees
```

The UNION operator will not display any records that are exact duplicates in both tables, but this can be overridden by using the ALL predicate after the UNION keyword, like this:

```
SELECT [Last Name], [First Name], Email
   FROM tblCustomers
UNION ALL
SELECT [Last Name], [First Name], Email
   FROM tblEmployees
```

The TRANSFORM statement

Although the TRANSFORM statement, also known as a *crosstab query*, is also not technically considered a join, it is included here because it does involve combining data from multiple sources of data into one result set, which is similar to some types of joins.

A TRANSFORM statement is used to calculate a sum, average, count, or other type of aggregate total on records. It then displays the information in a grid or spreadsheet format with data grouped both vertically (rows) and horizontally (columns). The general form for a TRANSFORM statement is this:

```
   TRANSFORM aggregating function
   SELECT statement
   PIVOT column heading field
```

Let's suppose that we want to build a datasheet that displays the invoice totals for each customer on a year-by-year basis. The vertical headings will be the customer names, and the horizontal headings will be the years. Let's modify a previous SQL statement to fit the transform statement.

```
TRANSFORM
IIF(Sum([Amount]) IS NULL,'NONE',Sum([Amount]))
   AS Total
SELECT [Last Name] & ', ' & [First Name] AS Name
      FROM tblCustomers LEFT JOIN tblInvoices
      ON tblCustomers.CustomerID=tblInvoices.CustomerID
      GROUP BY [Last Name] & ', ' & [First Name]
PIVOT Format(InvoiceDate, 'yyyy')
   IN ('1996','1997','1998','1999','2000')
```

Note that the aggregating function is the SUM function, the vertical headings are in the GROUP BY clause of the SELECT statement, and the horizontal headings are determined by the field listed after the PIVOT keyword.

For more information about joins, type **SQL joins** in the Office Assistant or on the **Answer Wizard** tab in the Microsoft Access Help window, and then click **Search**.

Using Intermediate SQL in Access

Now that we've had a discussion of the intermediate SQL syntax, let's look at some of the ways we can use it in an Access application.

Sample Database

Along with this article, there is a sample database called acIntSQL.mdb. Everything in acIntSQL is based on all the previously covered topics, and it demonstrates the different SQL statements discussed through queries and sample code.

Many of the sample queries used in acIntSQL depend on certain tables existing and containing data, or on certain other preexisting database objects. If you are having difficulty running one of the queries due to missing data, open the Reset Tables form and click the Reset Tables button. This will re-create the tables and their original default data. To step through the reset table process manually, execute the following queries in this order:

1. Drop Table Shipping
2. Drop Table Invoices
3. Drop Table Customers
4. Drop Table CreditLimit
5. Create Table Customers
6. Create Table Invoices
7. Create Table Shipping
8. Create Table CreditLimit
9. Populate Customers
10. Populate Invoices
11. Populate Shipping
12. Populate CreditLimit

Queries

Queries are SQL statements that are saved in an Access database and therefore can be used at any time, either directly from the Access user interface or from the Visual Basic® for Applications (VBA) programming language. Queries can be built with the Access Query Designer, which is a rich visual tool that greatly simplifies the building of SQL statements. Or you can build queries by entering SQL statements directly in the SQL View window.

As mentioned in the article "Fundamental Microsoft Jet SQL for Access 2000", Access converts all data-oriented tasks in the database into SQL statements. To demonstrate this, let's build a query by using the Access Query Designer.

1. Open the acIntSQL database.
1. Make sure that the tblCustomers and tblInvoices tables have been created and that they contain some data.
2. Select **Queries** from the **Objects** bar in the Database window.
3. Click the **New** button on the Database window toolbar.
4. In the **New Query** dialog box, select **Design View** and click **OK**.
5. In the **Show Table** dialog box, select **tblCustomers** and click **Add**; next select tblInvoices and click Add; and then click **Close**.
6. In the tblCustomers fields list, select the Last Name field and drag it to the first field in the design grid.
7. In the tblInvoices fields list, select the InvoiceDate and Amount fields and drag them to the design grid.
8. In the Sort property for the InvoiceDate field in the design grid, select **Ascending**.
9. Select **View** from the Access menu bar, and then click **SQL View**. This opens the SQL View window and displays the SQL syntax that Access is using for this query.

Note This query is similar to the "Join - Inner" query already saved in the acIntSQL database.

Inline Code

Using SQL statements inline means that you use the SQL statements within the Visual Basic for Applications (VBA) programming language. Although a deep discussion of how to use VBA is outside the scope of this article,

it is a simple task to execute SQL statements in code.

In the acIntSQL database, there are two forms that use inline SQL statements that are executed through the Jet OLE DB provider and ADO: The Intermediate DDL Statements form demonstrates the data definition statements, while the Intermediate DML Statements form demonstrates the data manipulation statements.

Intermediate DDL statements

The acIntSQL database has many examples of SQL statements that you can use to manage your database structure. Some of the Data Definition Language (DDL) statements are saved as data definition queries, while others are used as inline SQL within programming code. Depending on the DDL example you are trying to use, you may have to delete some database objects before executing it. For example, if you are trying to run the Create Currency Data Types query, you will first need to make sure that the Currency Data Types table does not already exist. If it does, the query will return a message stating that the table already exists. Before running any of the DDL examples, check the database object that it is creating or altering to make sure that it is configured so that the statement will execute properly.

In the case of the inline DDL statements, the same advice holds true: Check the database objects that are affected and configure them so that the DDL statements will execute properly.

In general, the inline DDL statements are executed by simply setting an ADO **Connection** object, and then passing the SQL statement to the **Execute** method of the **Connection** object. Following is the code from the **Binary Data Types** command button on the Intermediate DDL Statements form.

```
Private Sub cmdBinary_Click()
   Dim conDatabase As ADODB.Connection
   Dim SQL As String

   On Error GoTo Error_Handler
   Set conDatabase = Application.CurrentProject.Connection
   'NOTE:  Fields 1 through 4 can be created through both
   'SQL View and the Jet OLEDB Provider.
   'Fields 5 and 6 can only be created through the
   'Jet OLE DB provider.

   SQL = "CREATE TABLE tblCodeBinaryDataTypes (" & _
      "Field1_BINARY BINARY, " & _
      "Field2_BINARY250 BINARY(250), " & _
      "Field3_VARBINARY VARBINARY, " & _
      "Field4_VARBINARY250 VARBINARY(250), " & _
      "Field5_BVARYING BINARY VARYING, " & _
      "Field6_BVARYING250 BINARY VARYING(250))"

   conDatabase.Execute SQL
      MsgBox "The BINARY data types table has been created!"
   conDatabase.Close
   Set conDatabase = Nothing
Exit Sub
Error_Handler:
   MsgBox Err.Description, vbInformation
End Sub
```

After executing one of the DDL statements, open the affected database object in design view to see what changes were made. If the DDL statement is affecting relationships between tables, open the Edit Relationships window to view the changes. For example, let's examine the **Alter Table w/ Fast Foreign Key** command button on the Intermediate DDL Statements form.

1. Open the acIntSQL database.
2. Make sure that the tblCustomers and tblInvoices tables have been created.
3. Select **Forms** from the **Objects** bar in the Database window.
4. Highlight the Intermediate DDL Statements form and then click the **Design** button on the Database window toolbar.
5. In the Intermediate DDL Statements form, right-click the **Alter Table w/ Fast Foreign Key** command button and then select **Build EventÂ...** from the pop-up menu. This will open the VBA development environment and the code window should contain the cmdFastKey_Click sub-procedure.
6. Examine the SQL statement that is assigned to the SQL variable.

```
SQL = "ALTER TABLE tblInvoices " & _
   "ADD CONSTRAINT FK_tblInvoices " & _
   "FOREIGN KEY NO INDEX (CustomerID) REFERENCES " & _
   "tblCustomers (CustomerID) " & _
   "ON UPDATE CASCADE " & _
   "ON DELETE CASCADE"
```

Note that the DDL statement is altering the tblInvoices table and adding a fast foreign key constraint. It is also establishing data consistency between tblInvoices and tblCustomers with the cascade clauses.

7. Close the VBA development environment.
8. Close the Intermediate DDL Statements form.
9. From the **Tools** menu, select the **RelationshipsÂ...** menu item to open the Relationships window.
10. Double-click the relationship link between tblCustomers and tblInvoices to open the **Edit Relationships** dialog box.
11. Note that the cascading update and delete options are not set.
12. Close the dialog box.
13. While the relationships link is still highlighted, press the Delete key to delete the link.
14. Close the Relationships window.
15. While the Intermediate DDL Statements is still highlighted in the Database window, click the **Open** button on the Database window toolbar.
16. Click the **Alter Table w/ Fast Foreign Key** command button to recreate the foreign key relationship.
17. Close the Intermediate DDL Statements form.
18. Using the steps outlined previously, open the **Edit Relationships** dialog box on the newly created relationship link.
19. Note that the cascading update and delete options are now set.

Intermediate DML statements

The acIntSQL database has many examples of Data Manipulation Language (DML) statements that you can use to retrieve data, and most of them are implemented as queries. The only DML statements that are implemented as inline SQL are found in the Intermediate DML Statements form. The three DML examples in the form deal with using the "_" and "%" wildcard characters with the LIKE clause, and with creating a table in an external database by using a SELECT INTO statement.

Two of the saved queries in the acIntSQL database are DML statements that perform DDL-like actions. These are the SELECT INTO statements that retrieve data from existing tables and then create new tables from that data. With these examples, you will be prompted to delete the target tables if they already exist.

The **Create Customers Database** command button on the Intermediate DML Statements form presents an interesting use of the SELECT INTO statement. It is a good example of the kinds of things you can do with intermediate SQL statements. Following is the code from the subprocedure of the command button.

```
Private Sub cmdCreateDB_Click()

   Dim conCatalog As ADOX.Catalog
   Dim conDatabase As ADODB.Connection
   Dim SQL As String
   Dim sDBName As String

   On Error GoTo Error_Handler

   'Initialize objects & variables.
   Set conDatabase = Application.CurrentProject.Connection
   Set conCatalog = New ADOX.Catalog
   sDBName = "C:\Customers.mdb"

   'Create the Customers database.
   conCatalog.Create "Provider=Microsoft.Jet.OLEDB.4.0;" & _
                     "Data Source=" & sDBName

   'Run the DML statement to build the Customers table.
```

```
    SQL = "SELECT * INTO tblCustomers IN '" & sDBName & _
        "'" & "FROM tblCustomers"
    conDatabase.Execute SQL

    MsgBox "The new CUSTOMERS database has been created " & _
          "as " & sDBName & ".", vbInformation

    conDatabase.Close
    Set conDatabase = Nothing
    Set conCatalog = Nothing

 Exit Sub

 Error_Handler:
    MsgBox Err.Description, vbInformation
 End Sub
```

If the Customers database already exists, the code will return a message stating that the database could not be created. Let's walk through running this code to see what it does.

1. Check your C: drive to make sure that Customers.mdb does not already exist. If it does, delete it.
2. Open the acIntSQL database.
3. Make sure that the tblCustomers table has been created and that it contains data.
4. Select **Forms** from the **Objects** bar in the Database window.
5. Highlight the Intermediate DM Statements form and then click the **Open** button on the Database window toolbar.
6. Click the **Create Customers Database** button to create the new database.
7. Switch to Windows Explorer and view the contents of your C: drive. The Customers.mdb database should have been created.
8. Double-click the Customers.mdb database to launch another instance of Access.
9. Open the tblCustomers table. Note that it contains the same data as tblCustomers in the acIntSQL database.

The code sample for creating the new database uses the ADOX object library to create the Access database through the Jet OLE DB provider. Coverage of the ADOX object library is beyond the scope of this article. For more information about it, search the Access 2000 online Help **Contents** tab for "Microsoft ActiveX Data Objects (ADO)" and expand the contents until you see "Microsoft ADO Extensions for DDL and Security (ADOX) Programmer's Reference."

One Last Comment

Although we have shown you how to do many new tasks, we've also shown you many alternative ways of performing the same task. Which method or SQL technique you use should be determined by the needs of your application and your own level of comfort with the SQL syntax.

Additional Resources

Resource	Description
Fundamental Microsoft Jet SQL for Access 2000	This article discusses the basic mechanics of using Jet SQL to work with data in an Access 2000 database. It also delves into using SQL to create and alter a database's structure. If you're new to manipulating data with SQL in an Access database, this article is a great place to start.
Advanced Microsoft Jet SQL for Access 2000	Third in the series of SQL articles, "Advanced Microsoft Jet SQL for Access 2000" builds on the concepts covered in the first two articles, this time focusing on the SQL syntax that is most often used in a multiuser environment.
Microsoft Jet SQL Reference Help	This is the definitive source for the SQL language as it applies to Access 2000. It can be found in the Contents section of Microsoft Access 2000 Help.
Microsoft Jet Database Engine Programmer's Guide	This book is everything you wanted to know about programming with the Microsoft Jet Database Engine.
Building Applications with Forms and Reports	Here you can find solid development techniques for

	developing an Access 2000 application.
Microsoft Access 2000 Help	An irreplaceable source of Access 2000 programming topics.
Microsoft Office 2000/Visual Basic Programmer's Guide	This comprehensive book covers how to use VBA to program Office 2000 applications.
MSDN, http://msdn.microsoft.com/	This Web site always has the latest information for developing solutions with Microsoft platforms and languages.
MSDN Office Developer Center, http://msdn.microsoft.com/office/default.asp	This Web site contains the latest information about developing applications with Microsoft Office.
MSDN Training, Career, and Events	Look to Microsoft's MSDN Training courses to provide the soundest techniques for developing Access 2000 applications.