ARTICLE 2

Understanding SQL

Underlying every query in Microsoft Access 2010 is the Structured Query Language (SQL) database command language. Although you can design most queries using the simple Access 2010 design grid (or the view, function, or stored procedure designer in an Access project file), Access stores every query you design as an SQL command. When you use one of the designers, Access creates the SQL for you. However, for advanced types of queries that use the results of a second query as a comparison condition, you need to know SQL to define the second query (called a *subquery*). Also, you cannot use the design grid to construct all types of queries available in the product; you must use SQL for some of them. Understanding SQL is essential to building queries in an Access project file, because you're using Microsoft SQL Server.

Note

This article does not document all the syntax variants accepted by Access, but it does cover all the features of the SELECT statement and of action queries. Wherever possible, American National Standards Institute (ANSI) standard syntax is shown to provide portability across other databases that also support some form of SQL. You might notice that Access modifies the ANSI-standard syntax to a syntax that it prefers after you define and save a query. You can find some of the examples shown in the following pages in the ContactsDataCopy.accdb sample database. When an example is in the sample database, you'll find the name of the sample query in italics immediately preceding the query in the text. For a discussion of the syntax conventions used in this article, see the "Conventions and Features Used in This Book" section in the book's front matter.

How to Use This Article

This article contains two major sections: SQL select queries and SQL action queries. Within the first section, you can find keywords used in SQL in alphabetical order. You can also find entries for the basic building blocks you need to understand and use in various clauses: Column-Name, Expression, Search-Condition, and Subquery. If you're new to SQL, you might want to study these building block topics first. You can then study the major clauses of a SELECT statement in the order in which they appear in a SELECT statement: PARAMETERS, SELECT, FROM, WHERE, GROUP BY, HAVING, UNION, and ORDER BY.

In the second section, you can find a discussion of the syntax for the four types of queries that you can use to update your database, also in alphabetical order: DELETE, INSERT, SELECT INTO, and UPDATE. As you study these topics, you'll find references to some of the major clauses that you'll also use in a SELECT statement. You can find the details about those clauses in the first section.

SQL SELECT Queries

The SELECT statement forms the core of the SQL database language. You use the SELECT statement to select or retrieve rows and columns from database tables. The SELECT statement syntax contains six major clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY.

In an Access desktop application (.accdb), Access implements three significant extensions to the standard language: TRANSFORM, to allow you to build crosstab queries; IN, to allow you to specify a remote database connection or to specify column names in a crosstab query; and DISTINCTROW in a SELECT statement, to limit the rows returned from the <*table list*> to rows that have different primary key values in the tables that supply columns in the <*field list*>. In a previous version format database (.mdb), you can also use WITH OWNERACCESS OPTION in a SELECT statement to design queries in a secured database that can be run by users who are authorized to use the query, including those who have insufficient access rights to the tables referenced in the query.

> **Note**
>
> **When you save a query that you have written in SQL in your database, Access often examines your SQL command and adds brackets or extra parentheses to make the command easier to parse and compile. In some cases, Access restates complex predicates or changes the ANSI-standard syntax to one it prefers. For this reason, the examples shown in the book might not exactly match what you see in the sample queries when you open them in SQL view. If you enter the SQL exactly as shown in the book, it will return the same result as the sample query you find in the database.**

Aggregate Functions: AVG, CHECKSUM_AGG, COUNT, MAX, MIN, STDEV, STDEVP, SUM, VAR, and VARP

See Table 10-1 (on page 647 of the printed book).

BETWEEN Predicate

Compares a value with a range of values.

Syntax

<*expression*> [NOT] BETWEEN <*expression*> AND <*expression*>

Notes

The data types of all expressions must be compatible. Comparison of alphanumeric literals (strings) in Access or a default installation of SQL Server is case-insensitive.

Let *a*, *b*, and *c* be expressions. Then, in terms of other predicates, a BETWEEN b AND c is equivalent to the following:

```
(a >= b) AND (a <= c)
```

a NOT BETWEEN *b* AND *c* is equivalent to the following:

```
(a < b) OR (a > c)
```

The result is undefined if any of the expressions is Null.

Example

To determine whether the SoldPrice is greater than or equal to $100 and less than or equal to $500, enter the following:

```
SoldPrice BETWEEN 100 AND 500
```

See also Expression, SELECT Statement, Subquery, and WHERE Clause, in this article.

Column-Name

Specifies the name of a column in an expression.

Syntax

```
[[]{table-name | select-query-name |
  correlation-name}[]].][]field-name[]]
```

Notes

You must supply a qualifier to the field name only if the name is ambiguous within the context of the query or subquery (for example, if the same field name appears in more than one table or query listed in the FROM clause).

The *table-name*, *select-query-name*, or *correlation-name* that qualifies the field name must also appear in the FROM clause of the query or subquery. If a table or query has a correlation name, you must use the alias, not the actual name of the table or query. (A *correlation name* is an alias you assign to the table or query name in the FROM clause.)

You must supply the enclosing brackets in an Access desktop application (.accdb) only if the name contains an embedded blank or the name is also a reserved word (such as *select*, *table*, *name*, or *date*). Embedded blanks and enclosing brackets are not supported in the ANSI standard. You can use names that have embedded blanks in SQL Server by including a SET QUOTED_IDENTIFIER ON command and then enclosing each nonstandard name in double quotes ("). When you open a query from an Access project, Access automatically includes this command in the command stream that it sends to SQL Server.

If the *field-name* is a multi-value field, a query referencing the *field-name* returns the individual values separated by commas. A query datasheet provides a combo box that you can use to edit the multiple values. If you bind the column to a combo box control on a form, you can edit the field on the form. To edit the individual values in separate rows, use *field-name*.**Value** in your query. For records in the table that have multiple values in the field,

the query returns one row per value. The effect is identical to linking to a related many-to-many lookup table using a join. (See FROM Clause, on page 675, for details about defining a join in a query.) Note, however, that when you ask for *field-name*.**Value** from more than one multi-valued column in a table, the resulting query is not updatable because the query returns the Cartesian product of the multiple values in the two fields for each row in the source table.

If the *field-name* is an attachment data type, a query datasheet provides an attachment control to allow you to edit the data. You can also edit the data if you bind the field to an Attachment control in a form. You can individually reference one of the three properties of an attachment field: *field-name*.**FileData**, *field-name*.**FileName**, or *field-name*.**FileType**. All three properties return one row per separate attachment for each record in the source table, but you cannot update the values. The FileData property returns the binary attached file, the FileName property returns the original name of the file, and the FileType property returns the file extension.

Examples

To specify a field named Customer Last Name in a table named Customer List in an Access desktop application (.accdb), use the following:

```
[Customer List].[Customer Last Name]
```

To reference the same column in a view, stored procedure, or function for SQL Server, use the following:

```
"Customer List"."Customer Last Name"
```

To specify a field named StreetAddress that appears in only one table or query in the FROM clause, enter the following:

```
StreetAddress
```

To reference the individual values of a multi-valued field named ContactType, enter the following:

```
ContactType.Value
```

See also FROM Clause, SELECT Statement, and Subquery in this article.

Comparison Predicate

Compares the values of two expressions or the value of an expression and a single value returned by a subquery.

Syntax

```
<expression> {= | <> | > | < | >= | <=}
  {<expression> | <subquery>}
```

Notes

Comparison of strings in Access or a default installation of SQL Server is case-insensitive. The data type of the first expression must be compatible with the data type of the second expression or with the value returned by the subquery. If the subquery returns no rows or more than one row, an error is returned except when the select list of the subquery is COUNT(*), in which case the return of multiple rows yields one value. If the first expression, the second expression, or the subquery evaluates to Null, the result of the comparison is undefined.

Examples

To determine whether the sales date was in 2011, enter the following:

```
Year(DateSold) = 2011
```

To determine whether the invoice ID is not equal to 50, enter the following:

```
InvoiceID <> 50
```

To determine whether a product was sold in the first half of the year, enter the following:

```
Month(DateSold) < 7
```

To determine whether the date sold in the current row is less than the earliest order for ProductID 1, enter the following:

```
DateSold <
 (SELECT MIN(DateSold)
  FROM tblContactProducts
  WHERE ProductID = 1)
```

See also Expression, SELECT Statement, Subquery, and WHERE Clause in this article.

EXISTS Predicate

Tests the existence of at least one row that satisfies the selection criteria in a subquery.

Syntax

```
EXISTS (<subquery>)
```

Notes

The result cannot be undefined. If the subquery returns at least one row, the result is True; otherwise, the result is False. The subquery need not return values for this predicate; therefore, you can list any columns in the select list that exist in the underlying tables or queries or use an asterisk (*) to denote all columns.

Example

To find all contacts that own at least one product, enter the following (*qxmplContactsSomeProduct*):

```
SELECT tblContacts.FirstName, tblContacts.MiddleInit, tblContacts.LastName
 FROM tblContacts
 WHERE EXISTS
  (SELECT *
   FROM tblContactProducts
    INNER JOIN tblProducts
    ON tblContactProducts.ProductID = tblProducts.ProductID
   WHERE tblContactProducts.ContactID = tblContacts.ContactID
    AND tblProducts.TrialVersion = 0);
```

Note

In this example, the inner subquery makes a reference to the tblContacts table in the SELECT statement by referring to a column in the outer table (tblContacts.ContactID). This forces the subquery to be evaluated for every row in the SELECT statement, which might not be the most efficient way to achieve the desired result. (This type of subquery is also called a *correlated subquery*.) Whenever possible, the database query plan optimizer solves the query efficiently by reconstructing the query internally as a join between the source specified in the FROM clause and the subquery. In many cases, you can perform this reconstruction yourself, but the purpose of the query might not be as clear as when you state the problem using a subquery.

See also Expression, SELECT Statement, Subquery, and WHERE Clause in this article.

Expression

Specifies a value in a predicate or in the select list of a SELECT statement or subquery.

Syntax

```
[+ | -] {function | [(]<expression>[)] | literal |
  column-name} [{+ | - | * | / | \ | ^ | MOD | &}
  {function | [(]<expression>[)] | literal |
  column-name}]...
```

Notes

function—You can specify one of the SQL aggregate functions: AVG, COUNT, MAX, MIN, STDEV, STDEVP, SUM, VAR, or VARP; however, you cannot use an SQL aggregate function more than once in an expression. In a desktop application (.accdb), you can also use any of the functions built into Access or any function you define using Microsoft Visual Basic. In a project file (.adp), you can use any of the SQL Server built-in functions.

[(]<expression>[)]—You can construct an expression from multiple expressions separated by operators. Use parentheses around expressions to clarify the evaluation order. (See the examples later in this section.)

literal—You can specify a numeric or an alphanumeric constant. You must enclose an alphanumeric constant in single quotation marks in a project file (.adp) or single or double quotation marks in a desktop database (.accdb). To include an apostrophe in an alphanumeric constant, enter the apostrophe character twice in the literal string; or, in a desktop database, you can also choose to enclose the literal string in double quotation marks. If the expression is numeric, you must use a numeric constant. In a desktop database (.accdb), enclose a date/time literal within pound (#) signs, and any date/time literal you enter in SQL view must follow the U.S. *mm/dd/yy* (or *mm/dd/yyyy*) format. This might be different from the format you use on the query design grid, which must follow the format defined for Short Date Style in your Regional And Language Options section of the Control Panel. In a project file (.adp), you must enclose date or time literals in single quotes, and you can use any specification inside the quotes that SQL Server can recognize as a date or time. For example, SQL Server recognizes any of the following as a valid date literal:

```
'April 15, 2011'
'15 April, 2011'
'110415'
'04/15/2011'
'2011-04-15'
```

column-name—You can specify the name of a column in a table or a query. You can use a column name only from a table or query that you've specified in the FROM clause of the statement. If the expression is arithmetic, you must use a column that contains numeric data. If the same column name appears in more than one of the tables or queries included

in the query, you must fully qualify the name with the query name, table name, or correlation name, as in *TableA.Column1*. When a table or column name contains a blank or is a reserved word (such as select, table, name, or date) in a desktop database (.accdb), you must enclose each name in brackets, as in [*Table A*].[*Column 1*]. When a table or column name contains a blank or is a reserved word in a project file (.adp), you must enclose each name in double quotes, as in "Table A"."Column 1". Note that when you open a query in an Access project, Access includes the required SET QUOTED_IDENTIFIER ON command in the command string. However, if you execute an SQL Server query from a desktop database with a pass-through query, you must include this command in the pass-through query. Although in ANSI SQL (and SQL Server) you can reference an *output-column-name* anywhere within an expression, Access supports this only within the <*field list*> of a SELECT statement. Access does not support references to named expression columns in GROUP BY, HAVING, ORDER BY, or WHERE clauses. You must repeat the expression rather than use the column name. See SELECT Statement, on page 675, for details about output-column-name.

+ | - | * | / | \ | ^ | *MOD*—You can combine multiple numeric expressions with arithmetic operators that specify a calculation. If you use arithmetic operators, all expressions within an expression must evaluate as numeric data types.

&—You can concatenate alphanumeric expressions by using the *&* operator in a desktop database (.accdb). In a project file (.adp), use + as the concatenation operator.

Examples

To specify the average of a column named COST, enter the following:

```
AVG(COST)
```

To specify one-half the value of a column named PRICE, enter the following:

```
(PRICE * .5)
```

To specify a literal for 3:00 P.M. on March 1, 2011, in a desktop database (.accdb), enter the following:

```
#3/1/2011 3:00PM#
```

To specify a literal for 3:00 P.M. on March 1, 2011, in a project file (.adp), enter the following:

```
'March 1, 2011 3:00PM'
```

To specify a character string that contains the name Acme Mail Order Company, enter the following:

```
'Acme Mail Order Company'
```

To specify a character string that contains a possessive noun (requiring an embedded apostrophe), enter the following:

```
'Andy''s Hardware Store'
```

or in a desktop database you can also enter:

```
"Andy's Hardware Store"
```

In a desktop database (.accdb), to specify a character string that is the concatenation of fields from a table named Customer List containing a person's first and last name with an intervening blank, enter the following:

```
[Customer List].[First Name] & " " & [Customer List].[Last Name]
```

In a project file (.adp), to specify a character string that is the concatenation of fields from a table named Customer List containing a person's first and last name with an intervening blank, enter the following:

```
"Customer List"."First Name" + ' ' + "Customer List"."Last Name"
```

See also Column-name, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), SELECT Statement, Subquery, and UPDATE Statement in this article.

FROM Clause

Specifies the tables or queries that provide the source data for your query.

Syntax

```
FROM {table-name [[AS] correlation-name] |
    select-query-name [[AS] correlation-name] |
    (<select-statement>) AS correlation-name |
    <joined table>},...
  [IN <"source database name"> <[source connect string]>]
```

where *<joined table>* is

```
({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  <joined table>}
ON <join-specification>)
```

where <*joined table*> is the result of another join operation, and where <*join-specification*> is a search condition made up of predicates that compare fields in the first table, query, or joined table with fields in the second table, query, or joined table.

Notes

You can supply a correlation name for each table name or query name and use this correlation name as an alias for the full table name when qualifying column names in the <*field-list*>, in the <*join-specification*>, or in the WHERE clause and subclauses. If you're joining a table or a query to itself, you must use correlation names to clarify which copy of the table or query you're referring to in the select list, join criteria, or selection criteria. If a table name or a query name is also an SQL reserved word (for example, *Order*), you must enclose the name in brackets. In SQL Server, you must enclose the name of a table or query that is also an SQL reserved word in double quotes. If you decide to use quotes, you must also ensure that the server has received the command SET QUOTED_IDENTIFIER ON. Note that when you open a query in an Access project, Access includes the required SET QUOTED_IDENTIFIER ON command in the command string to ensure that any names that you have enclosed in quotes are recognized correctly by SQL Server. However, if you execute an SQL Server query from a desktop database with a pass-through query, you must either use brackets or quotes and include this command in the pass-through query.

Use INNER JOIN to return all the rows that match the join specification in both tables. Use LEFT [OUTER] JOIN to return all the rows from the first logical table (where *logical table* is any table, query, or joined table expression) joined on the join specification with any matching rows from the second logical table. When no row matches in the second logical table, the database returns Null values for the columns from that table. Conversely, RIGHT [OUTER] JOIN returns all the rows from the second logical table joined with any matching rows from the first logical table. A FULL [OUTER] JOIN returns all rows from the tables or queries on both sides of the join, but only SQL Server supports this operation.

When you use only *equals* comparison predicates in the join specification, the result is called an *equi-join*. The joins that Access displays in the design grid are equi-joins. Access cannot display on the design grid any join specification that uses any comparison operator other than equals (=)—also called a *non-equijoin*. If you want to define a join on a non-equals comparison (<, >, <>, <=, or >=) in Access, you must define the query using the SQL view. The query designer in an Access project can display non-equijoins. When you join a table to itself using an equals comparison predicate, the result is called a *self-join*.

SQL Server also supports a CROSS JOIN (with no ON clause). A CROSS JOIN produces the same result as listing table or query names separated by commas with no JOIN specification (a Cartesian product).

If you include multiple tables in the FROM clause with no JOIN specification but do include a predicate that matches fields from the multiple tables in the WHERE clause, the database in most cases optimizes how it solves the query by treating the query as a JOIN. For example:

```
SELECT *
  FROM TableA, TableB
  WHERE TableA.ID = TableB.ID
```

is solved by the database as though you had specified

```
SELECT *
  FROM TableA
    INNER JOIN TableB
    ON TableA.ID = TableB.ID
```

You cannot update fields in a table by using a recordset opened on the query, the query datasheet, or a form bound to a multiple table query where the join is expressed using a table-list and a WHERE clause. In many cases you can update the fields in the underlying tables when you use the JOIN syntax.

When you list more than one table or query without join criteria, the source is the *Cartesian product* of all the tables. For example, FROM *TableA*, *TableB* instructs the database to fetch all the rows of TableA matched with all the rows of TableB. Unless you specify other restricting criteria, the number of logical rows that the database processes could equal the number of rows in TableA *times* the number of rows in TableB. When you include the WHERE or HAVING clause, the database returns the rows in which the selection criteria specified in those clauses evaluate to True.

Example

To select information about all companies and contacts and any products purchased, enter the following (*qxmplAllCompanyContactsAnyProducts*):

```
SELECT tblCompanies.CompanyName, tblContacts.FirstName,
   tblContacts.LastName, CP.ProductName, CP.DateSold, CP.SoldPrice
 FROM ((tblCompanies
   INNER JOIN tblCompanyContacts
   ON tblCompanies.CompanyID = tblCompanyContacts.CompanyID)
   INNER JOIN tblContacts
   ON tblContacts.ContactID = tblCompanyContacts.ContactID)
   LEFT JOIN
     (SELECT tblContactProducts.ContactID, tblProducts.ProductName,
       tblContactProducts.DateSold, tblContactProducts.SoldPrice
     FROM tblProducts
     INNER JOIN tblContactProducts
     ON tblProducts.ProductID = tblContactProducts.ProductID
     WHERE tblProducts.TrialVersion = 0) AS CP
   ON tblContacts.ContactID = CP.ContactID;
```

Note

If you save the previous query in a previous version of Access, when you open the query in Design view, you'll find that Access saves the inner <select-statement> with brackets:

```
[SELECT tblContactProducts.ContactID, tblProducts.ProductName,
tblContactProducts.DateSold, tblContactProducts.SoldPrice
FROM tblProducts
INNER JOIN tblContactProducts
ON tblProducts.ProductID = tblContactProducts.ProductID
WHERE tblProducts.TrialVersion = 0]. AS CP
```

This is the internal syntax supported by the Joint Engine Technology (JET) database engine installed with Access 2003 and earlier. The Access Database Engine (ACE) supplied with Access 2007 and Access 2010 no longer modifies the SQL—you'll find the sample query saved exactly as stated in the example without brackets.

See also HAVING Clause, IN Clause, SELECT Statement, Subquery, and WHERE Clause in this article.

GROUP BY Clause

In a SELECT statement, specifies the columns used to form groups from the rows selected. Each group contains identical values in the specified column(s). In Access, you use the GROUP BY clause to define a totals query. You must also include a GROUP BY clause in a crosstab query in Access. (See TRANSFORM Statement for details.)

Syntax

GROUP BY *column-name*,...

Notes

A column name in the GROUP BY clause can refer to any column from any table in the FROM clause, even if the column is not named in the select list. If the GROUP BY clause is preceded by a WHERE clause, the database creates the groups from the rows selected after it applies the WHERE clause. When you include a GROUP BY clause in a SELECT statement, the select list must be made up of either SQL aggregate functions or column names specified in the GROUP BY clause.

Example

To find the average and maximum prices for products by category name, enter the following (*qxmplCategoryAvgMaxPrice*):

```
SELECT tblProducts.CategoryDescription,
   Avg(tblProducts.UnitPrice) AS AvgOfUnitPrice,
   Max(tblProducts.UnitPrice) AS MaxOfUnitPrice
 FROM tblProducts
 WHERE tblProducts.TrialVersion = 0
 GROUP BY tblProducts.CategoryDescription;
```

See also Aggregate Functions, HAVING Clause, Search-Condition, SELECT Statement, and WHERE Clause in this article.

HAVING Clause

Specifies groups of rows that appear in the logical table (a recordset) defined by a SELECT statement. The search condition applies to columns specified in a GROUP BY clause, to columns created by aggregate functions, or to expressions containing aggregate functions. If a group doesn't pass the search condition, the database does not include it in the logical table.

Syntax

HAVING *<search-condition>*

Notes

If you do not include a GROUP BY clause, the select list must be formed by using one or more of the SQL aggregate functions.

The difference between the HAVING clause and the WHERE clause is that WHERE *<search-condition>* applies to single rows before they are grouped, while HAVING *<search-condition>* applies to groups of rows.

If you include a GROUP BY clause preceding the HAVING clause, the *<search-condition>* applies to each of the groups formed by equal values in the specified columns. If you do not include a GROUP BY clause, the *<search-condition>* applies to the entire logical table defined by the SELECT statement.

Example

To find the invoice amounts for all invoices that total more than $150, enter the following (*qxmplTotalInvoices>150*):

```
SELECT tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate, Sum(tblContactProducts.SoldPrice) AS InvoiceTotal
 FROM (tblCompanies
   INNER JOIN tblInvoices
   ON tblCompanies.CompanyID = tblInvoices.CompanyID)
   INNER JOIN tblContactProducts
   ON tblInvoices.InvoiceID = tblContactProducts.InvoiceID
 GROUP BY tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate
HAVING Sum(tblContactProducts.SoldPrice) > 150;
```

See also Aggregate Functions, GROUP BY Clause, Search-Condition, SELECT Statement, and WHERE Clause in this article.

IN Clause

In a desktop database (.accdb), specifies the source for the tables in a query. The source can be another Access database; a dBASE, or any database for which you have an Open Database Connectivity (ODBC) driver. This is an Access extension to standard SQL.

Syntax

IN <"*source database name*"> <[*source connect string*]>

Enter *"source database name"* and [*source connect string*]. (Be sure to include the quotation marks and the brackets.) If your database source is Access, enter only *"source database name"*. Enter these parameters according to the type of database to which you are connecting, as shown in Table A2-1.

Table A2-1 IN Parameters for Various Database Types

Database Name	Source Database Name	Source Connect String
Access	"drive:\path\filename"	(none)
dBASE III	"drive:\path"	[dBASE III;]
dBASE IV	"drive:\path"	[dBASE IV;]
dBASE 5	"drive:\path"	[dBASE 5.0;]
ODBC	(none)	[ODBC; DATABASE= defaultdatabase; UID=user; PWD= password;DSN= datasourcename]

Notes

The IN clause applies to all tables referenced in the FROM clause and any subqueries in your query. You can refer to only one external database within a query, but if the IN clause points to a database that contains more than one table, you can use any of those tables in your query. If you need to refer to more than one external file or database, attach those files as tables in Access and use the logical attached table names instead.

For ODBC, if you omit the DSN= or DATABASE= parameter, Access prompts you with a dialog box showing available data sources so that you can select the one you want. If you omit the UID= or PWD= parameter and the server requires a user ID and password, Access prompts you with a login dialog box for each table accessed.

For dBASE, you can provide an empty string ("") for *source database name* and provide the path or dictionary filename using the DATABASE= parameter in *source connect string instead, as in*

```
"[dBase IV; DATABASE=C:\MyDB\dbase.dbf]"
```

Example

In a desktop database (.accdb), to retrieve the Company Name field in the Northwind Traders sample database without having to attach the Customers table, enter the following:

```
SELECT Customers.CompanyName
FROM Customers
IN "C:\My Documents\Shortcut to NORTHWIND.ACCDB";
```

See also SELECT Statement in this article.

IN Predicate

Determines whether a value is equal to any of the values or is unequal to all values in a set returned from a subquery or provided in a list of values.

Syntax

```
<expression> [NOT] IN {(<subquery>) |
  ({literal},...) |<expression>}
```

Notes

Comparison of strings in Access or a default installation of SQL Server is case-insensitive. The data types of all expressions, literals, and the column returned by the subquery must be compatible. If the expression is Null or any value returned by the subquery is Null, the result is undefined. In terms of other predicates, <*expression*> IN <*expression*> is equivalent to the following:

```
<expression> = <expression>
```

<*expression*> IN (<*subquery*>) is equivalent to the following:

```
<expression> = ANY (<subquery>)
```

<*expression*> IN (*a, b, c,...*), where *a*, *b*, and *c* are literals, is equivalent to the following:

```
(<expression> = a) OR (<expression> = b) OR
  (<expression> = c) ...
```

<*expression*> NOT IN ... is equivalent to the following:

```
NOT (<expression> IN ...)
```

Examples

To test whether StateOrProvince is on the west coast of the United States, enter the following:

```
[StateOrProvince] IN ('CA', 'OR', 'WA')
```

To list all contacts who have not purchased a multi-user product, enter the following (*qxmplContactsNotMultiUser*):

```
SELECT tblContacts.ContactID, tblContacts.FirstName,
   tblContacts.MiddleInit, tblContacts.LastName
 FROM tblContacts
 WHERE tblContacts.ContactID NOT IN
  (SELECT ContactID
    FROM tblContactProducts
     INNER JOIN tblProducts
     ON tblContactProducts.ProductID = tblProducts.ProductID
    WHERE tblProducts.CategoryDescription = 'Multi-User');
```

See also Expression, Quantified Predicate, SELECT Statement, Subquery, and WHERE Clause in this article.

LIKE Predicate

Searches for strings that match a pattern.

Syntax

```
column-name [NOT] LIKE match-string [ESCAPE escape-character]
```

Notes

String comparisons in Access or a default installation of SQL Server are case-insensitive. If the column specified by *column-name* contains a Null, the result is undefined. Comparison of two empty strings or an empty string with the special asterisk (*) character (% character in SQL Server) evaluates to True.

You provide a text string as a *match-string* value that defines what characters can exist in which positions for the comparison to be true. Access and SQL Server understand a number of wildcard characters (shown in Table A2-2) that you can use to define positions that can contain any single character, zero or more characters, or any single number.

Table A2-2 Wildcard Characters for String Comparisons

Desktop Database	Project File	Meaning
?	_	Any single character
*	%	Zero or more characters (used to define leading, trailing, or embedded strings that don't have to match any of the pattern characters)
#	[0-9]	Any single number

You can also specify in the match string that any particular position in the text or memo field can contain only characters from a list that you provide. To define a list of comparison characters for a particular position, enclose the list in brackets ([ ]). You can specify a range of characters within a list by entering the low-value character, a hyphen, and the high-value character, as in *[A-Z]* or *[3-7]*. If you want to test a position for any characters except those in a list, start the list with an exclamation point (!) in a desktop database or a caret symbol (^) in a project file.

If you want to test for one of the special characters *, ?, #, and [, (and _ or % in a project file), you must enclose the character in brackets. Alternatively, in a project file, you can

specify an ESCAPE clause. When you place the escape character in the match string, the database ignores the character and uses the following character as a literal comparison value. Therefore, you can include the escape character immediately preceding one of the special characters to use the special character as a literal comparison instead of a pattern character. Desktop databases do not support the ESCAPE clause.

Examples

In a desktop database, to determine whether a contact's LastName is at least four characters long and begins with Smi, enter the following:

```
tblContacts.LastName LIKE "Smi?*"
```

In a project file, write the previous test as follows:

```
tblContacts.LastName LIKE 'Smi_%'
```

In a desktop database, to test whether PostalCode is a valid Canadian postal code, enter the following:

```
PostalCode LIKE "[A-Z]#[A-Z] #[A-Z]#"
```

In a project file, to test whether a character column named Discount ends in 5%, enter the following:

```
Discount LIKE '%5$%' ESCAPE '$'
```

See also Expression, SELECT Statement, Subquery, and WHERE Clause in this article.

NULL Predicate

Determines whether the expression evaluates to Null or not Null. This predicate evaluates only to True or False and will not evaluate to undefined.

Syntax

<expression> **IS** [**NOT**] **NULL**

Example

To determine whether the contact work phone number column contains the Null value, enter the following:

```
tblContacts.WorkPhone IS NULL
```

See also Expression, SELECT Statement, Subquery, and WHERE Clause in this article.

ORDER BY Clause

Specifies the sequence of rows to be returned by a SELECT statement or a subquery.

Syntax

```
ORDER BY {column-name | column-number [ASC | DESC]},...
```

Notes

You use column names or relative output column numbers to specify the columns on whose values the rows returned are ordered. (If you use relative output column numbers, the first output column is 1.) You can specify multiple columns in the ORDER BY clause. When you specify multiple columns, the list is ordered primarily by the first column. If rows exist for which the values of that column are equal, they are ordered by the next column in the ORDER BY list, and so on. When multiple rows contain the matching values in all the columns in the ORDER BY clause, the database can return the matching rows in any order. You can specify ascending (ASC) or descending (DESC) order for each column. If you do not specify ASC or DESC, ASC is assumed. Using an ORDER BY clause in a SELECT statement is the only means of defining the sequence of the returned rows.

When you include the DISTINCT keyword or use the UNION query operator in the SELECT statement, the ORDER BY clause can include only columns specified in the SELECT clause. Otherwise, you can include any column in the logical table returned by the FROM clause.

To use ORDER BY in a view, function, or stored procedure in SQL Server, you must also include the TOP keyword in the SELECT clause. To fetch and sort all rows, specify TOP 100 PERCENT. Note, however, that a view, function, or stored procedure returns the result ordered only when you directly execute the query from code. When Access runs a query in SQL Server that is identified as the record source of a form or report or the row source of a combo box or list box, it sends a SELECT * FROM *queryname* command to the server. The server returns the rows sorted only when you specify the ORDER BY clause again in the record source or row source as part of a SELECT statement on the query.

Examples

To calculate the total for all invoices and list the result for each customer and invoice in descending sequence by order total, enter the following (qxmplOrderTotalSorted):

```
SELECT TOP 100 PERCENT tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate, Sum(tblContactProducts.SoldPrice) AS InvoiceTotal
 FROM (tblCompanies
   INNER JOIN tblInvoices
   ON tblCompanies.CompanyID = tblInvoices.CompanyID)
   INNER JOIN tblContactProducts
   ON tblInvoices.InvoiceID = tblContactProducts.InvoiceID
 GROUP BY tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate
 ORDER BY Sum(tblContactProducts.SoldPrice) DESC;
```

Note

The TOP keyword is optional in a desktop database (.accdb). In SQL Server, you can also specify the calculated column alias name in the ORDER BY clause: ORDER BY InvoiceTotal DESC. In a desktop database, you must repeat the calculation expression as shown in the example.

In a desktop database (.accdb), to create a mailing list for all companies and all contacts, sorted in ascending order by postal code, enter the following (qxmplSortedMailingList):

```
SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
   tblCompanies.StateOrProvince, tblCompanies.PostalCode
 FROM tblCompanies
UNION
SELECT [FirstName] & " " & ([MiddleInit]+". ") & [LastName] AS Contact,
   tblContacts.HomeAddress, tblContacts.HomeCity,
   tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
 FROM tblContacts
ORDER BY 5;
```

Note

If you decide to use column names in the ORDER BY clause of a UNION query, the database derives the column names from the names returned by the first query. In this example, you could change the ORDER BY clause to read ORDER BY PostalCode.

To create the same mailing list in a view or in-line function in an SQL Server database, enter the following:

```
SELECT TOP 100 PERCENT CompanyName, Address, City,
   StateOrProvince, PostalCode
FROM
(SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
   tblCompanies.StateOrProvince, tblCompanies.PostalCode
  FROM tblCompanies
UNION
SELECT tblContacts.FirstName + ' ' +
   IsNull(tblContacts.MiddleInit + '. ', '') +
   tblContacts.LastName AS Contact,
   tblContacts.HomeAddress, tblContacts.HomeCity,
   tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
  FROM tblContacts) AS U
ORDER BY 5;
```

Notice that you must UNION the rows first and then select and sort them all.

See also INSERT Statement, SELECT Statement, and UNION Query Operator in this article.

PARAMETERS Declaration

In a desktop database (.accdb), precedes an SQL statement to define the data types of any parameters you include in the query. You can use parameters to prompt the user for data values or to match data values in controls on an open form. (In an SQL Server database, you declare the parameters for a function or procedure as part of the CREATE statement.)

Syntax

PARAMETERS {[*parameter-name*] *data-type*},... ;

Notes

If your query prompts the user for values, each parameter name should describe the value that the user needs to enter. For example, [Print invoices from orders on date:] is much more descriptive than [Enter date:]. If you want to refer to a control on an open form, use this format:

```
[Forms]![Myform]![Mycontrol]
```

To refer to a control on a subform, use this format:

```
[Forms]![Myform]![Mysubformcontrol].[Form]![ControlOnSubform]
```

Valid data type entries are shown in Table A2-3.

Table A2-3 SQL Parameter Data Types

SQL Parameter Data Types	Equivalent Access Data Type
Char, Text(n)[1], VarChar	Text
Text[1], LongText, LongChar, Memo	Memo
TinyInt, Byte, Integer1	Number, Byte
SmallInt, Short, Integer2	Number, Integer
Integer, Long, Integer4	Number, Long Integer
Real, Single, Float4, IEEESingle	Number, Single
Float, Double, Float8, IEEEDouble	Number, Double
Decimal, Numeric	Number, Decimal
UniqueIdentifier, GUID	Number, Replication ID
DateTime, Date, Time	Date/Time
Money, Currency	Currency
Bit, Boolean, Logical, YesNo	Yes/No
Image, LongBinary, OLEObject	OLE Object
Text, LongText, LongChar, Memo	Hyperlink[2]
Binary, VarBinary	Binary[3]

[1] Text with a length descriptor of 255 or less maps to the Access Text data type. Text with no length descriptor is a Memo field.

[2] Internally, Access stores a Hyperlink in a Memo field but sets a custom property to indicate a Hyperlink format.

[3] The ACE supports a Binary data type (raw hexadecimal), but the Access user interface does not. If you encounter a non-Access table that has a data type that maps to Binary, you will be able to see the data type in the table definition, but you won't be able to successfully edit this data in a datasheet or form. You can manipulate Binary data in Visual Basic.

Example

To create a parameter query that summarizes the sales and the cost of goods for all items sold in a given month, enter the following (qxmplMonthSalesParameter):

```
PARAMETERS [Year to summarize:] Short, [Month to summarize:] Short;
SELECT tblProducts.ProductName,
   Format([DateSold],"mmmm"", ""yyyy") AS OrderMonth,
   Sum(tblContactProducts.SoldPrice) AS TotalSales
 FROM tblProducts
   INNER JOIN tblContactProducts
   ON tblProducts.ProductID = tblContactProducts.ProductID
 WHERE (Year([DateSold]) = [Year to summarize:])
   AND (Month([DateSold]) = [Month to summarize:])
GROUP BY tblProducts.ProductName, Format([DateSold],"mmmm"", ""yyyy");
```

See also SELECT Statement in this article.

Quantified Predicate

Compares the value of an expression to some, any, or all values of a single column returned by a subquery.

Syntax

```
<expression> {= | <> | > | < | >= | <=}
  [SOME | ANY | ALL] (<subquery>)
```

Notes

String comparisons in Access or a default installation of SQL Server are case-insensitive. The data type of the expression must be compatible with the data type of the value returned by the subquery.

When you use ALL, the predicate is true if the comparison is True for all the values returned by the subquery. If the expression or any of the values returned by the subquery is Null, the result is undefined. When you use SOME or ANY, the predicate is True if the comparison is true for any of the values returned by the subquery. If the expression is a Null value, the result is undefined. If the subquery returns no values, the predicate is False.

Examples

To find the products whose price is greater than all the products in the Support category, enter the following (*qxmplProductPrice>AllSupport*):

```
SELECT tblProducts.ProductID, tblProducts.ProductName, tblProducts.UnitPrice
 FROM tblProducts
 WHERE tblProducts.UnitPrice >All
  (SELECT tblProducts.UnitPrice
   FROM tblProducts
   WHERE tblProducts.CategoryDescription = 'Support');
```

To find the products whose price is greater than any of the products in the Support category, enter the following (*qxmplProductPrice>AnySupport*):

```
SELECT tblProducts.ProductID, tblProducts.ProductName, tblProducts.UnitPrice
 FROM tblProducts
 WHERE tblProducts.UnitPrice >Any
  (SELECT tblProducts.UnitPrice
   FROM tblProducts
   WHERE tblProducts.CategoryDescription = 'Support');
```

See also Expression, SELECT Statement, Subquery, and WHERE Clause in this article.

Search Condition

Describes a simple or compound predicate that is True, False, or undefined for a given row or group. Use a search condition in the WHERE clause of a SELECT statement, a subquery, a DELETE statement, or an UPDATE statement. You can also use a search condition within the HAVING clause in a SELECT statement. The search condition defines the rows that should appear in the resulting logical table or the rows that should be acted upon by the change operation. If the search condition is True when applied to a row, that row is included in the result.

Syntax

```
[NOT] {predicate | (<search-condition>)}
  [{AND | OR | XOR | EQV | IMP}
  [NOT] {predicate | (<search-condition>)}]...
```

Notes

If you include a comparison predicate in the form of *<expression>* comparison-operator *<subquery>*, the database returns an error if the subquery returns no rows. The database effectively applies any subquery in a predicate within a search condition to each row of the table that is the result of the previous clauses. The database then evaluates the result of the subquery with regard to each candidate row.

The order of evaluation of the Boolean operators is NOT, AND, OR, XOR (exclusive OR), EQV (equivalence), and IMP (implication). You can include additional parentheses to influence the order in which the Boolean expressions are processed. SQL Server does not support the XOR, EQV, and IMP logical operators.

INSIDE OUT

Using XOR, EQV, and IMP in the Access Query Designer

You can express AND and OR Boolean operations directly by using the design grid. If you need to use XOR, EQV, or IMP, you must create an expression in the Field row, clear the Show check box, and set the Criteria row to <> False.

When you use the Boolean operator NOT, the following holds: NOT (True) is False, NOT (False) is True, and NOT (undefined) is undefined. The result is undefined whenever a predicate references a null value. If a search condition evaluates to False or undefined when applied to a row, the row is not selected. The database returns True, False, or undefined values as a result of applying Boolean operators (AND, OR, XOR, EQV, IMP) against two predicates or search conditions according to the tables shown in Figure A2-1.

AND	True	False	Undefined (Null)
True	True	False	Null
False	False	False	False
Undefined (Null)	Null	False	Null

OR	True	False	Undefined (Null)
True	True	True	True
False	True	False	Null
Undefined (Null)	True	Null	Null

XOR	True	False	Undefined (Null)
True	False	True	Null
False	True	False	Null
Undefined (Null)	Null	Null	Null

EQV	True	False	Undefined (Null)
True	True	False	Null
False	False	True	Null
Undefined (Null)	Null	Null	Null

IMP [(Not A) OR B]	True	False	Undefined (Null)
True	True	False	Null
False	True	True	True
Undefined (Null)	True	Null	Null

Figure A2-1 Truth tables for SQL Boolean operators

Example

In a desktop database, to find all products for which the unit price is greater than $100 and for which the category description number is equal to Multi-User or the product has a pre-requisite, but not both, enter the following (*qxmplXOR*):

```
SELECT tblProducts.ProductID, tblProducts.ProductName,
   tblProducts.CategoryDescription, tblProducts.UnitPrice,
   tblProducts.PreRequisite
 FROM tblProducts
 WHERE tblProducts.UnitPrice>100
   AND ((tblProducts.CategoryDescription = "Multi-User")
   XOR (tblProducts.PreRequisite Is Not Null));
```

In a project file, to find all products for which the unit price is greater than $100 and for which the category description number is equal to Multi-User or the product has a prerequisite, but not both, enter the following:

```
SELECT tblProducts.ProductID, tblProducts.ProductName,
   tblProducts.CategoryDescription, tblProducts.UnitPrice,
   tblProducts.PreRequisite
 FROM tblProducts
 WHERE tblProducts.UnitPrice>100
   AND ((tblProducts.CategoryDescription = "Multi-User")
   OR (tblProducts.PreRequisite Is Not Null))
   AND NOT ((tblProducts.CategoryDescription = "Multi-User")
   AND (tblProducts.PreRequisite Is Not Null));
```

See also DELETE Statement, Expression, HAVING Clause, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), SELECT Statement, Subquery, UPDATE Statement, and WHERE Clause in this article.

SELECT Statement

Fetches data from one or more tables or queries to create a logical table (recordset). The items in the select list identify the columns or calculated values to return from the source tables to the new recordset. You identify the tables to be joined in the FROM clause, and you identify the rows to be selected in the WHERE clause. Use GROUP BY to specify how to form groups for an aggregate query, and use HAVING to specify which resulting groups should be included in the result.

Syntax

```
SELECT [ALL | DISTINCT | DISTINCTROW | TOP number
       [PERCENT]] <select-list>
FROM {table-name [[AS] correlation-name] |
    select-query-name [[AS] correlation-name] |
    (<select-statement>) AS correlation-name |
    <joined table>},...
  [IN <"source database name"> <[source connect
      string]>]
  [WHERE <search-condition>]
  [GROUP BY column-name,...]
  [HAVING <search-condition>]
  [UNION [ALL] <select-statement>]
  [ORDER BY {column-name [ASC | DESC]},...]
  [WITH OWNERACCESS OPTION];
```

where <*select-list*> is

```
{* | {<expression> [AS output-column-name] |
  table-name.* | query-name.* |
  correlation-name.*},...}
```

and where <*joined table*> is

```
({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
ON <join-specification>)
```

Notes

You can supply a correlation name for each table name or query name and use this correlation name as an alias for the full table name when qualifying column names in the <*select-list*>, in the <*join-specification*>, or in the WHERE clause and subclauses. If you're joining a table or a query to itself, you must use correlation names to clarify which copy of the table or query you're referring to in the select list, join criteria, or selection criteria. If a table name or a query name is also an SQL reserved word (for example, *Order*), you must enclose the name in brackets. In SQL Server, you must enclose the name of a table or query that is also an SQL reserved word in brackets or double quotes. If you decide to use quotes, you must also ensure that the server has received the command SET QUOTED_IDENTIFIER ON. Note that when you open a query in an Access project, Access includes the required SET QUOTED_IDENTIFIER ON command in the command string to ensure that any names that you have enclosed in quotes are recognized correctly by SQL Server. However, if you execute an SQL Server query from a desktop database with a pass-through query, you must use brackets or quotes and include this command in the pass-through query.

When you list more than one table or query without join criteria, the source is the Cartesian product of all the tables. For example, FROM *TableA*, *TableB* instructs the database to search all the rows of TableA matched with all the rows of TableB. Unless you specify other restricting criteria, the number of logical rows that the database processes could equal the number of rows in TableA *times* the number of rows in TableB. The database then returns the rows in which the selection criteria specified in the WHERE and HAVING clauses are true. (See FROM Clause, on page 675, for further details about specifying joins.)

You can further define which rows the database includes in the output recordset by specifying ALL, DISTINCT, DISTINCTROW (in a desktop database only), TOP *n*, or TOP *n* PERCENT.

ALL includes all rows that match the search criteria from the source tables, including potential duplicate rows. DISTINCT requests that the database return only rows that are different from any other row. You cannot update any columns in a query that uses DISTINCT because the database can't identify which of several potentially duplicate rows you intend to update.

DISTINCTROW (the default in Access 7.0—Access 95—and earlier) requests that Access return only rows in which the concatenation of the primary keys from all tables supplying output columns is unique. Depending on the columns you select, you might see rows in the result that contain duplicate values, but each row in the result is derived from a distinct combination of rows in the underlying tables. DISTINCTROW is significant only when you include a join in a query and do not include output columns from all tables. For example, the statement

```
SELECT tblContacts.WorkStateOrProvince
FROM tblContacts
  INNER JOIN tblContactProducts
  ON tblContacts.ContactID = tblContactProducts.ContactID
WHERE tblContactProducts.DateSold > #7/1/2010#;
```

returns 92 rows in the ContactsDataCopy.accdb sample database—one row for each product owned by a contact. On the other hand, the statement

```
SELECT DISTINCTROW tblContacts.WorkStateOrProvince
FROM tblContacts
  INNER JOIN tblContactProducts
  ON tblContacts.ContactID = tblContactProducts.ContactID
WHERE tblContactProducts.DateSold > #7/1/2010#;
```

returns only 29 rows—one for each *distinct row* in the tblContacts table, the only table with output columns. The equivalent of the second example in ANSI-standard SQL is as follows:

```
SELECT tblContacts.WorkStateOrProvince
FROM tblContacts
WHERE tblContacts.ContactID
  IN (Select tblContactProducts.ContactID FROM tblContactProducts
  WHERE tblContactProducts.DateSold > '2010-07-01');
```

We suspect that Microsoft implemented DISTINCTROW in version 1 because the first release of Access did not support subqueries.

Specify TOP *n* or TOP *n* PERCENT to request that the recordset contain only the first n or first n percent of rows. In general, you should specify an ORDER BY clause when you use TOP to indicate the sequence that defines which rows are first, or top. The parameter *n* must be a positive integer and must be less than or equal to 100 if you include the PERCENT keyword. If you do not include an ORDER BY clause, the sequence of rows returned is undefined. In a TOP query, if the *n*th and any rows immediately following the *n*th row are duplicates, the database returns the duplicates; thus, the recordset might have more than

n rows. Note that if you specify an order, using TOP does not cause the query to execute any faster; the database must still solve the entire query, order the rows, and return the top rows.

When you include a GROUP BY clause, the select list must be made up of one or more of the SQL aggregate functions or one or more of the column names specified in the GROUP BY clause. A column name in a GROUP BY clause can refer to any column from any table in the FROM clause, even if the column is not named in the select list. If you want to refer to a calculated expression in the GROUP BY clause, you must assign an output column name to the expression in the select list and then refer to that name in the GROUP BY clause. If the GROUP BY clause is preceded by a WHERE clause, the database forms the groups from the rows selected after it applies the WHERE clause.

If you use a HAVING clause but do not include a GROUP BY clause, the select list must be formed using SQL aggregate functions. If you include a GROUP BY clause preceding the HAVING clause, the HAVING search condition applies to each of the groups formed by equal values in the specified columns. If you do not include a GROUP BY clause, the HAVING search condition applies to the entire logical table defined by the SELECT statement.

You use column names or relative output column numbers to specify the columns on whose values the rows returned are ordered. (If you use relative output column numbers, the first output column is 1.) You can specify multiple columns in the ORDER BY clause. When you specify multiple columns, the list is ordered primarily by the first column. If rows exist for which the values of that column are equal, they are ordered by the next column in the ORDER BY list, and so on. When multiple rows contain the matching values in all the columns in the ORDER BY clause, the database can return the matching rows in any order. You can specify ascending (ASC) or descending (DESC) order for each column. If you do not specify ASC or DESC, ASC is assumed. Using an ORDER BY clause in a SELECT statement is the only means of defining the sequence of the returned rows.

In an .mdb-format desktop database that has user-level security implemented, the person running the query not only must have rights to the query but also must have the appropriate rights to the tables used in the query. (These rights include reading data to select rows and updating, inserting, and deleting data using the query.) If your application has multiple users, you might want to secure the tables so that no user has direct access to any of the tables and all users can still run queries defined by you. Assuming you're the owner of both the queries and the tables, you can deny access to the tables but allow access to the queries. To make sure that the queries run properly, you must add the WITH OWNERACCESS OPTION clause to allow users the same access rights as the table owner when accessing the data via the query. Access 2010 does not support user-level security in .accdb-format databases.

If the *select-list* references a multi-value field, the query returns the individual values separated by commas. A query datasheet provides a combo box that you can use to edit the multiple values. If you bind the column to a combo box control on a form, you can edit the field on the form. To edit the individual values in separate rows, use *field-name*.**Value** in your query. For records in the table that have multiple values in the field, the query returns one row per value. The effect is identical to linking to a related many-to-many lookup table using a join. (See FROM Clause, on page 675, for details about defining a join in a query.) Note, however, that when you ask for *field-name*.**Value** from more than one multi-valued column in a table, the resulting query is not updatable because the query returns the Cartesian product of the multiple values in the two fields for each row in the source table.

If the select-list contains an attachment data type, the query datasheet provides an attachment control to allow you to edit the data. You can also edit the data if you bind the field to an Attachment control in a form. You can individually reference one of the three properties of an attachment field: *field-name*.**FileData**, *field-name*.**FileName**, or *field-name*.**FileType**. All three properties return one row per separate attachment for each record in the source table, but you cannot update the values. The FileData property returns the binary attached file, the FileName property returns the original name of the file, and the FileType property returns the file extension.

Examples

To select information about all companies and contacts and any products purchased, enter the following (*qxmplAllCompanyContactsAnyProducts*):

```
SELECT tblCompanies.CompanyName, tblContacts.FirstName,
   tblContacts.LastName, CP.ProductName, CP.DateSold, CP.SoldPrice
 FROM ((tblCompanies
   INNER JOIN tblCompanyContacts
   ON tblCompanies.CompanyID = tblCompanyContacts.CompanyID)
   INNER JOIN tblContacts
   ON tblContacts.ContactID = tblCompanyContacts.ContactID)
   LEFT JOIN
    (SELECT tblContactProducts.ContactID, tblProducts.ProductName,
      tblContactProducts.DateSold, tblContactProducts.SoldPrice
    FROM tblProducts
    INNER JOIN tblContactProducts
    ON tblProducts.ProductID = tblContactProducts.ProductID
    WHERE tblProducts.TrialVersion = 0) AS CP
   ON tblContacts.ContactID = CP.ContactID;
```

Note

If you save the previous query in a previous version of Access, when you open the query in Design view, you'll find that Access saves the inner <*select-statement*> with brackets, like this:

```
[SELECT tblContactProducts.ContactID, tblProducts.ProductName,
tblContactProducts.DateSold, tblContactProducts.SoldPrice
FROM tblProducts
INNER JOIN tblContactProducts
ON tblProducts.ProductID = tblContactProducts.ProductID
WHERE tblProducts.TrialVersion = 0]. AS CP
```

This is the internal syntax supported by the JET database engine installed with Access 2003 and earlier. The ACE supplied with Access 2010 no longer modifies the SQL—you'll find the sample query saved exactly as stated in the example without brackets.

To find the average and maximum prices for products by category name, enter the following (*qxmplCategoryAvgMaxPrice*):

```
SELECT tblProducts.CategoryDescription,
   Avg(tblProducts.UnitPrice) AS AvgOfUnitPrice,
   Max(tblProducts.UnitPrice) AS MaxOfUnitPrice
  FROM tblProducts
  WHERE tblProducts.TrialVersion = 0
  GROUP BY tblProducts.CategoryDescription;
```

To find the invoice amounts for all invoices that total more than $150, enter the following (*qxmplTotalInvoices>150*):

```
SELECT tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate, Sum(tblContactProducts.SoldPrice) AS InvoiceTotal
  FROM (tblCompanies
   INNER JOIN tblInvoices
   ON tblCompanies.CompanyID = tblInvoices.CompanyID)
   INNER JOIN tblContactProducts
   ON tblInvoices.InvoiceID = tblContactProducts.InvoiceID
  GROUP BY tblCompanies.CompanyName, tblInvoices.InvoiceID,
   tblInvoices.InvoiceDate
HAVING Sum(tblContactProducts.SoldPrice) > 150;
```

To calculate the total for all invoices and list the result for each customer and invoice in descending sequence by order total, enter the following (*qxmplOrderTotalSorted*):

```
SELECT TOP 100 PERCENT tblCompanies.CompanyName, tblInvoices.InvoiceID,
  tblInvoices.InvoiceDate, Sum(tblContactProducts.SoldPrice) AS InvoiceTotal
 FROM (tblCompanies
  INNER JOIN tblInvoices
  ON tblCompanies.CompanyID = tblInvoices.CompanyID)
  INNER JOIN tblContactProducts
  ON tblInvoices.InvoiceID = tblContactProducts.InvoiceID
 GROUP BY tblCompanies.CompanyName, tblInvoices.InvoiceID,
  tblInvoices.InvoiceDate
 ORDER BY Sum(tblContactProducts.SoldPrice) DESC;
```

Note

The TOP keyword is optional in a desktop database (.accdb). In SQL Server, you can also specify the calculated column alias name in the ORDER BY clause: ORDER BY InvoiceTotal DESC. In a desktop database, you must repeat the calculation expression as shown in the example.

In a desktop database (.accdb), to create a mailing list for all companies and all contacts, sorted in ascending order by postal code, enter the following (*qxmplSortedMailingList*):

```
SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
    tblCompanies.StateOrProvince, tblCompanies.PostalCode
  FROM tblCompanies
UNION
SELECT [FirstName] & " " & ([MiddleInit]+". ") & [LastName] AS Contact,
    tblContacts.HomeAddress, tblContacts.HomeCity,
    tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
  FROM tblContacts
ORDER BY 5;
```

Note

If you decide to use column names in the ORDER BY clause of a UNION query, the database derives the column names from the names returned by the first query. In this example, you could change the ORDER BY clause to read ORDER BY PostalCode.

To create the same mailing list in a view or in-line function in an SQL Server database, enter the following:

```
SELECT TOP 100 PERCENT CompanyName, Address, City,
   StateOrProvince, PostalCode
FROM
(SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
   tblCompanies.StateOrProvince, tblCompanies.PostalCode
 FROM tblCompanies
UNION
SELECT tblContacts.FirstName + ' ' +
   IsNull(tblContacts.MiddleInit + '. ', '') +
   tblContacts.LastName AS Contact,
   tblContacts.HomeAddress, tblContacts.HomeCity,
   tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
 FROM tblContacts) AS U
ORDER BY 5;
```

Notice that you must UNION the rows first and then select and sort them all.

See also FROM Clause, GROUP BY Clause, HAVING Clause, INSERT Statement, Search-Condition, and UNION Query Operator in this article.

Subquery

Selects from a single column any number of values, or no values at all, for comparison in a predicate. You can also use a subquery that returns a single value in the select list of a SELECT clause.

Syntax

```
(SELECT [ALL | DISTINCT | DISTINCTROW | TOP number
       [PERCENT]] <select-list>
  FROM {table-name [[AS] correlation-name] |
    select-query-name [[AS] correlation-name] |
    <joined table>},...
  [WHERE <search-condition>]
  [GROUP BY column-name,...]
  [HAVING <search-condition>]
  [ORDER BY {column-name [ASC | DESC]},...])
```

where *select-list* is

```
{* | {<expression> | table-name.* |
  query-name.* | correlation-name.*}}
```

and where *<joined table>* is

```
(({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
ON <join-specification>)
```

Notes

You can use the special asterisk (*) character in the *<select-list>* of a subquery only when the subquery is used in an EXISTS predicate or when the FROM clause within the subquery refers to a single table or query that contains only one column.

You can supply a correlation name for each table name or query name and use this correlation name as an alias for the full table name when qualifying column names in the *<select-list>*, in the *<join-specification>*, or in the WHERE clause and subclauses. If you're joining a table or a query to itself, you must use correlation names to clarify which copy of the table or query you're referring to in the select list, join criteria, or selection criteria. You must also use a correlation name if one of the tables in the FROM clause is the same as a table in the outer query. If a table name or a query name is also an SQL reserved word (for example, *Order*), you must enclose the name in brackets. In SQL Server, you must enclose the name of a table or query that is also an SQL reserved word in double quotes. Note that when you open a query in an Access project, Access includes the required SET QUOTED_IDENTIFIER ON command in the command string. However, if you execute an SQL Server query from a desktop database with a pass-through query, you must include this command in the pass-through query.

When you list more than one table or query without join criteria, the source is the *Cartesian product* of all the tables. For example, FROM *TableA*, *TableB* instructs the database to search all the rows of TableA matched with all the rows of TableB. Unless you specify other restricting criteria, the number of logical rows that the database processes could equal the number of rows in TableA *times* the number of rows in TableB. The database then returns the rows in which the selection criteria specified in the WHERE and HAVING clauses are true. (See also FROM Clause, on page 675, for further details about specifying joins.)

You can further define which rows the database includes in the output recordset by specifying ALL, DISTINCT, DISTINCTROW (in a desktop database only), TOP *n*, or TOP *n* PERCENT. ALL includes all rows that match the search criteria from the source tables, including potential duplicate rows. DISTINCT requests that the database return only rows that are different from any other row.

DISTINCTROW (the default in Access version 7.0 and earlier) requests that Access return only rows in which the concatenation of the primary keys from all tables supplying output columns is unique. Depending on the columns you select, you might see rows in the result that contain duplicate values, but each row in the result is derived from a distinct combination of rows in the underlying tables. DISTINCTROW is significant only when you include a join in a query and do not include output columns from all tables. (See SELECT Statement, on page 675, for more information about DISTINCTROW.)

Specify TOP *n* or TOP *n* PERCENT to request that the recordset contain only the first n or first n percent of rows. In general, you should specify an ORDER BY clause when you use TOP to indicate the sequence that defines which rows are first, or top. The parameter n must be an integer and must be less than or equal to 100 if you include the PERCENT keyword. If you do not include an ORDER BY clause, the sequence of rows returned is undefined. In a TOP query, if the nth and any rows immediately following the nth row are duplicates, the database returns the duplicates; thus, the recordset might have more than n rows. Note that if you specify an order, using TOP does not cause the query to execute any faster; the database must still solve the entire query, order the rows, and return the top rows.

In the search condition of the WHERE clause of a subquery, you can use an outer reference to refer to the columns of any table or query that is defined in the outer queries. You must qualify the column name if the table or query reference is ambiguous.

A column name in the GROUP BY clause can refer to any column from any table in the FROM clause, even if the column is not named in the <*select-list*>. If the GROUP BY clause is preceded by a WHERE clause, the database creates the groups from the rows selected after the application of the WHERE clause.

When you include a GROUP BY or HAVING clause in a SELECT statement, the select list must be made up of either SQL aggregate functions or column names specified in the GROUP BY clause. If a GROUP BY clause precedes a HAVING clause, the HAVING clause's search condition applies to each of the groups formed by equal values in the specified columns. If you do not include a GROUP BY clause, the HAVING clause's search condition applies to the entire logical table defined by the SELECT statement.

Examples

To find all contacts who own at least one product, enter the following (*qxmplContactSomeProduct*):

```
SELECT tblContacts.FirstName, tblContacts.MiddleInit, tblContacts.LastName
 FROM tblContacts
 WHERE EXISTS
  (SELECT *
   FROM tblContactProducts
   INNER JOIN tblProducts
   ON tblContactProducts.ProductID = tblProducts.ProductID
   WHERE tblContactProducts.ContactID = tblContacts.ContactID
   AND tblProducts.TrialVersion = 0);
```

Note

In this example, the inner subquery makes a reference to the tblContacts table in the SELECT statement by referring to a column in the outer table (tblContacts.ContactID). This forces the subquery to be evaluated for every row in the SELECT statement, which might not be the most efficient way to achieve the desired result. (This type of subquery is also called a *correlated subquery*.) Whenever possible, the database query plan optimizer solves the query efficiently by reconstructing the query internally as a join between the source specified in the FROM clause and the subquery. In many cases, you can perform this reconstruction yourself, but the purpose of the query might not be as clear as when you state the problem using a subquery.

To select contacts who first purchased a product before 2011 and list them in ascending order by postal code, enter the following (*qxmplContactsPurchaseBefore2011*):

```
SELECT TOP 100 PERCENT tblContacts.FirstName, tblContacts.MiddleInit,
  tblContacts.LastName, tblContacts.HomeCity, tblContacts.HomePostalCode
 FROM tblContacts
 WHERE #01/01/2011# >
  (SELECT Min(tblContactProducts.DateSold)
   FROM tblContactProducts
   WHERE tblContactProducts.ContactID = tblContacts.ContactID)
 ORDER BY tblContacts.HomePostalCode;
```

Note

The previous query also uses a correlated subquery.

To find the products whose price is greater than any of the support products, enter the following (*qxmplProductsPrice>AnySupport*):

```
SELECT tblProducts.ProductID, tblProducts.ProductName, tblProducts.UnitPrice
 FROM tblProducts
 WHERE tblProducts.UnitPrice >Any
  (SELECT tblProducts.UnitPrice
   FROM tblProducts
   WHERE tblProducts.CategoryDescription = "Support");
```

See also Expression, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), and SELECT Statement in this article.

TRANSFORM Statement

In a desktop database, produces a crosstab query that lets you summarize a single value by using the values found in a specified column or in an expression as the column headers and using other columns or expressions to define the grouping criteria to form rows. The result looks similar to a spreadsheet and is most useful as input to a graph object. This is an Access extension to standard SQL.

Syntax

```
TRANSFORM <aggregate-function-expression>
  <select-statement>
PIVOT <expression>
[IN (<column-value-list>)]
```

where <*aggregate-function-expression*> is an expression created with one of the aggregate functions, <*select-statement*> contains a GROUP BY clause, and <*column-value-list*> is a list of required values expected to be returned by the PIVOT expression, enclosed in quotes and separated by commas. (You can use the IN clause to force the output sequence of the columns.)

Notes

The <*aggregate-function-expression*> parameter is the value that you want to appear in the "body" of the crosstab datasheet. PIVOT <*expression*> defines the column or expression that provides the column headings in the crosstab result. You might, for example, use this value to provide a list of months with aggregate rows defined by product categories in the <*select-statement*> GROUP BY clause. You can use more than one column or expression in the SELECT statement to define the grouping criteria for rows.

Example

To produce a total sales amount for each month in the year 2010, categorized by product, enter the following (*qxmpl2010SalesByProductXtab*):

```
TRANSFORM Sum(tblContactProducts.SoldPrice) AS SumOfSoldPrice
 SELECT tblProducts.ProductID, tblProducts.ProductName,
  Sum(tblContactProducts.SoldPrice) AS TotSales
 FROM tblProducts
  INNER JOIN tblContactProducts
  ON tblProducts.ProductID = tblContactProducts.ProductID
 GROUP BY tblProducts.ProductID, tblProducts.ProductName
PIVOT Format([DateSold],"mmm yyyy")
 IN ("Jan 2010","Feb 2010","Mar 2010","Apr 2010","May 2010",
  "Jun 2010","Jul 2010","Aug 2010","Sep 2010",
  "Oct 2010","Nov 2010","Dec 2010");
```

> **Note**
>
> This example shows a special use of the IN predicate to define not only which months should be selected but also the sequence in which Access displays the months in the resulting recordset.

See also GROUP BY Clause, HAVING Clause, SELECT Statement, and Total Functions in this article.

UNION Query Operator

Produces a result table that contains the rows returned by both the first SELECT statement and the second SELECT statement.

Syntax

```
<select-statement>
UNION [ALL]
  <select-statement>
[ORDER BY {column-name | column-number
[ASC | DESC]},...]
```

Notes

When you specify ALL, the database returns all rows in both logical tables. When you do not specify ALL, the database eliminates duplicate rows. The tables returned by each *<select-statement>* must contain an equal number of columns, and each column must have identical attributes.

You must not use the ORDER BY clause in the *<select-statements>* that are joined by query operators; however, you can include a single ORDER BY clause at the end of a statement that uses one or more query operators. This action will apply the specified order to the result of the entire statement. The database derives the column names of the output from the column names returned by the first *<select-statement>*. If you want to use column names in the ORDER BY clause, be sure to use names from the first query. You can also use the output column numbers to define ORDER BY criteria.

In a project file, you can include the ORDER BY clause at the end of the statement in a stored procedure, but you cannot include this clause in a view or in-line function. To sort a UNION in a view or in-line function, you must create a view on the query containing the UNION and then sort the view. You can also embed the UNION query in a FROM clause of a query and then sort the result.

You can combine multiple SELECT statements using UNION to obtain complex results. You can also use parentheses to influence the sequence in which the database applies the operators, as shown here:

```
SELECT...UNION (SELECT...UNION SELECT...)
```

Example

In a desktop database (.accdb), to create a mailing list for all companies and all contacts, sorted in ascending order by postal code, enter the following (*qxmplSortedMailingList*):

```
SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
   tblCompanies.StateOrProvince, tblCompanies.PostalCode
  FROM tblCompanies
UNION
SELECT [FirstName] & " " & ([MiddleInit]+". ") & [LastName] AS Contact,
   tblContacts.HomeAddress, tblContacts.HomeCity,
   tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
  FROM tblContacts
ORDER BY 5;
```

> **Note**
> **If you decide to use column names in the ORDER BY clause of a UNION query, the database derives the column names from the names returned by the first query. In this example, you could change the ORDER BY clause to read ORDER BY PostalCode.**

To create the same mailing list in a view or in-line function in an SQL Server database, enter the following:

```
SELECT TOP 100 PERCENT CompanyName, Address, City,
   StateOrProvince, PostalCode
FROM
(SELECT tblCompanies.CompanyName, tblCompanies.Address, tblCompanies.City,
   tblCompanies.StateOrProvince, tblCompanies.PostalCode
 FROM tblCompanies
UNION
SELECT tblContacts.FirstName + ' ' +
   IsNull(tblContacts.MiddleInit + '. ', '') +
   tblContacts.LastName AS Contact,
   tblContacts.HomeAddress, tblContacts.HomeCity,
   tblContacts.HomeStateOrProvince, tblContacts.HomePostalCode
 FROM tblContacts) AS U
ORDER BY 5;
```

Notice that you must UNION the rows first and then select and sort them all.

See also ORDER BY Clause and SELECT Statement in this article.

WHERE Clause

Specifies a search condition in an SQL statement or an SQL clause. The DELETE, SELECT, and UPDATE statements and the subquery containing the WHERE clause operate only on those rows that satisfy the condition.

Syntax

WHERE *<search-condition>*

Notes

The database applies the *<search-condition>* to each row of the logical table assembled as a result of executing the previous clauses, and it rejects those rows for which the *<search-condition>* does not evaluate to True. If you use a subquery within a predicate in the *<search-condition>* (often called an *inner query*), the database must execute the subquery before it evaluates the predicate.

In a subquery, if you refer to a table or a query that you also use in an outer FROM clause (often called a *correlated subquery*), the database must execute the subquery for each row being evaluated in the outer table. If you do not use a reference to an outer table in a subquery, the database must execute the subquery only once. A correlated subquery can also be expressed as a join, which generally executes more efficiently. If you include a predicate in the <*search-condition*> in the form

```
<expression> <comparison-operator> <subquery>
```

the database returns an error if the subquery returns no rows.

The order of evaluation of the logical operators used in the <*search-condition*> is NOT, AND, OR, XOR (exclusive OR), EQV (equivalence), and then IMP (implication). (SQL Server does not support the XOR, EQV, and IMP logical operators.) You can include additional parentheses to influence the order in which the database processes expressions.

Examples

In a desktop database, to find all products for which the unit price is greater than $100 and for which the category description number is equal to Multi-User or the product has a prerequisite, but not both, enter the following (qxmplXOR):

```
SELECT tblProducts.ProductID, tblProducts.ProductName,
   tblProducts.CategoryDescription, tblProducts.UnitPrice,
   tblProducts.PreRequisite
 FROM tblProducts
 WHERE tblProducts.UnitPrice>100
   AND ((tblProducts.CategoryDescription = "Multi-User")
   XOR (tblProducts.PreRequisite Is Not Null));
```

In a project file, to find all products for which the unit price is greater than $100 and for which the category description number is equal to Multi-User or the product has a prerequisite, but not both, enter the following:

```
SELECT tblProducts.ProductID, tblProducts.ProductName,
   tblProducts.CategoryDescription, tblProducts.UnitPrice,
   tblProducts.PreRequisite
 FROM tblProducts
 WHERE tblProducts.UnitPrice>100
   AND ((tblProducts.CategoryDescription = "Multi-User")
   OR (tblProducts.PreRequisite Is Not Null))
   AND NOT ((tblProducts.CategoryDescription = "Multi-User")
   AND (tblProducts.PreRequisite Is Not Null));
```

See also DELETE Statement, Expression, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), Search Condition, SELECT Statement, Subquery, and UPDATE Statement in this article.

SQL Action Queries

Use SQL action queries to delete, insert, or update data or to create a new table from existing data. Action queries are particularly powerful because they allow you to operate on sets of data, not single rows. For example, an UPDATE statement or a DELETE statement affects all rows in the underlying tables that meet the selection criteria you specify.

DELETE Statement

Deletes one or more rows from a table or a query. The WHERE clause is optional. If you do not specify a WHERE clause, all rows are deleted from the table or the query that you specify in the FROM clause. If you specify a WHERE clause, the database applies the search condition to each row in the table or the query, and only those rows that evaluate to True are deleted.

Syntax

```
DELETE [<select-list>]
  FROM {table-name [[AS] correlation-name] |
    select-query-name [[AS] correlation-name] |
    <joined table>},...
  [IN <source specification>]
  [WHERE <search-condition>];
```

where <*select-list*> is

```
[* | table-name.*]
```

and where <*joined table*> is

```
({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
ON <join-specification>)
```

Notes

When you specify a query name in a DELETE statement, the query must not be constructed using the UNION query operator. The query also must not contain an SQL aggregate function, the DISTINCT keyword, a GROUP BY or HAVING clause, or a subquery that references the same base table as the DELETE statement.

When you join two or more tables in the FROM clause, you can delete rows only from the *many* side of the relationship if the tables are related one-to-many; if the tables are related one-to-one, you can delete rows from either side. When you include more than one table in the FROM clause, you must also specify from which table the rows are to be deleted by using *table name*.* in the <*select-list*>. When you specify only one table in the FROM clause, you do not need to provide a <*select-list*>.

You can supply a correlation name for each table or query name. You can use this correlation name as an alias for the full table name when qualifying column names in the WHERE clause and in subclauses. You must use a correlation name when referring to a column name that occurs in more than one table in the FROM clause.

If you use a subquery in the <*search-condition*>, you must not reference the target table or the query or any underlying table of the query in the subquery.

Examples

To delete all rows in the tblContactProducts table, enter the following:

```
DELETE FROM tblContactProducts;
```

To delete all rows in the tblContactEventsHistory table for events that occurred before January 1, 2010, enter the following (*qxmplDeleteOldEventHistory*):

```
DELETE tblContactEventsHistory.*
 FROM tblContactEventsHistory
 WHERE tblContactEventsHistory.ContactDateTime < #01/01/2010#;
```

See also IN Clause, INSERT Statement, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), Search-Condition, and Subquery in this article.

INSERT Statement (Append Query)

Inserts one or more new rows into the specified table or query. When you use the VALUES clause, the database inserts only a single row. If you use a SELECT statement, the number of rows inserted equals the number of rows returned by the SELECT statement.

Syntax

```
INSERT INTO table-name [({column-name},...)]
  [IN <source specification>]
  {VALUES({literal},...) | select-statement}
  [WHERE <search-condition>];
```

Notes

If you do not include a column name list, you must supply values for all columns defined in the table in the order in which they were declared in the table definition. If you include a column name list, you must supply values for all columns in the list, and the values must be compatible with the receiving column attributes. You must include in the list all columns in the underlying table whose Required attribute is Yes and that do not have a default value.

If you include an IN clause in both the INSERT and the FROM clause of the SELECT statement, both must refer to the same source database.

If you supply values by using a SELECT statement, the statement's FROM clause cannot have the target table of the insert as its table name or as an underlying table. The target table also cannot be used in any subquery.

You cannot include an attachment field in the list of column names for an INSERT into a table. If the target table contains an attachment field, you must include the *column-name* list and specify any other fields into which you want to insert data. It is not possible to insert data into an attachment field using SQL.

You cannot include a calculated data type field in the list of column names for an INSERT into a table. If the target table contains a calculated data type field, you must include the *column-name* list and specify any other fields into which you want to insert data. It is not possible to insert data into a calculated data type field using SQL.

You cannot include a multi-valued field in the list of column names for an INSERT into a table unless the multi-value field is the only field in the *column-name* list and you include the Value property of the field. You can include a WHERE clause only when the target of the insert is the Value property of a single multi-valued field. In this case, you use the WHERE clause to specify which rows in the parent table should be affected by the INSERT. If you use a select-statement as the source of the inserted values when the target is the hidden recordset represented by the Value property of a multi-value field, the WHERE clause applies to the target table, not the *select-statement*, unless you can qualify the column names in the predicate to make it clear that the WHERE clause applies to the *select-statement*. You cannot include a WHERE clause filtering the *select-statement* and a second WHERE clause filtering the target table.

Because Access allows you to define column-value constraints (validation rules in a desktop database), table constraints (validation rule in a desktop database), and referential integrity checks, any values that you insert must pass these validations before Access will allow you to run the query.

Examples

To insert a new row in the tblProducts table, enter the following:

```
INSERT INTO tblProducts (ProductName,
     CategoryDescription, UnitPrice)
VALUES ('Support Renewal', 'Multi-User', 99);
```

To insert old event records into a history table and avoid duplicates, enter the following (*qxmplArchiveContactEventsByDate*):

```
PARAMETERS LastDateToKeep DateTime;
INSERT INTO tblContactEventsHistory
   (ContactID, ContactDateTime, ContactEventType, ContactNotes )
 SELECT tblContactEvents.ContactID, tblContactEvents.ContactDateTime,
    tlkpContactEventTypes.ContactEventTypeDescription,
    tblContactEvents.ContactNotes
   FROM tlkpContactEventTypes
   INNER JOIN (tblContactEvents
    LEFT JOIN tblContactEventsHistory
    ON (tblContactEvents.ContactID = tblContactEventsHistory.ContactID)
     AND (tblContactEvents.ContactDateTime =
        tblContactEventsHistory.ContactDateTime))
   ON tlkpContactEventTypes.ContactEventTypeID =
    tblContactEvents.ContactEventTypeID
 WHERE (tblContactEvents.ContactDateTime<[LastDateToKeep])
  AND (tblContactEventsHistory.ContactID Is Null);
```

Although Access accepts the ANSI-standard VALUES clause, you will discover in a desktop database that Access 2003 and earlier convert a statement such as

```
INSERT INTO MyTable (ColumnA, ColumnB)
VALUES (123, "Jane Doe");
```

to

```
INSERT INTO MyTable (ColumnA, ColumnB)
SELECT 123 As Expr1, "Jane Doe" as Expr2;
```

Access 2010 does not convert a VALUES clause.

To add the Sales Prospect value to the ContactType multi-valued field of the contact in the tblContacts table whose last name is Smith, enter the following:

```
INSERT INTO tblContacts (ContactType.Value)
VALUES ("Sales Prospect")
WHERE tblContacts.LastName = "Smith";
```

See also DELETE Statement, IN Clause, SELECT Statement, and Subquery in this article.

SELECT . . . INTO Statement (Make-Table Query)

Creates a new table from values selected from one or more other tables. Make-table queries are most useful for providing backup snapshots or for creating tables with rolled-up totals at the end of an accounting period.

Syntax

```
SELECT [ALL | DISTINCT | DISTINCTROW |
          TOP number PERCENT]] <select-list>
INTO new-table-name
  [IN <source specification>]
  FROM {table-name [[AS] correlation-name] |
    select-query-name [[AS] correlation-name] |
    <joined table>},...
  [IN <source specification>]
  [WHERE <search-condition>]
  [GROUP BY column-name,...]
  [HAVING <search-condition>]
[UNION [ALL] <select-statement>]
  [[ORDER BY {column-name [ASC | DESC]},...] |
  IN <"source database name">
     <[source connect string]>
  [WITH OWNERACCESS OPTION];
```

where <*select-list*> is

```
{* | {<expression> [AS output-column-name] |
  table-name.* | query-name.* |
  correlation-name.*},...}
```

and where <*joined table*> is

```
({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
ON <join-specification>)
```

Notes

A SELECT...INTO query creates a new table with the name specified in *new-table-name*. If a table with that name already exists, the database displays a dialog box that asks you to confirm the deletion of the existing table before it creates a new one in its place. The columns in the new table inherit the data type attributes of the columns produced by the <*select-list*>. However, you cannot include a multi-valued field, calculated field, or an attachment field in the <*select-list*>.

If you include an IN clause for both the INTO and the FROM clauses, both must refer to the same source database.

Example

To create a new table that summarizes all sales by product and by month, enter the following (*qxmplProductSalesMakeTable*):

```
SELECT tblProducts.ProductName, Format([DateSold],"yyyy mm") AS MonthSold,
   Sum(tblContactProducts.SoldPrice) AS TotalSales
INTO tblMonthSalesSummary
FROM tblProducts
  INNER JOIN tblContactProducts
  ON tblProducts.ProductID = tblContactProducts.ProductID
GROUP BY tblProducts.ProductName, Format([DateSold],"yyyy mm");
```

See also IN Clause, Search-Condition, and SELECT Statement in this article.

UPDATE Statement

In the specified table or query, updates the selected columns (either to the value of the given expression or to Null) in all rows that satisfy the search condition. If you do not enter a WHERE clause, all rows in the specified table or query are affected.

Syntax

```
UPDATE {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  <joined table>},...
[IN <source specification>]
SET {column-name = {<expression> | NULL}},...
[WHERE <search-condition>]
where <joined table> is
({table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
{INNER | {{LEFT | RIGHT | FULL} [OUTER]} JOIN
```

```
  {table-name [[AS] correlation-name] |
  select-query-name [[AS] correlation-name] |
  (<select-statement>) AS correlation-name |
  <joined table>}
ON <join-specification>)
```

Notes

If you provide more than one table name, you can update columns only in the table on the *many* side of a one-to-many relationship. If the tables are related one-to-one, you can update columns in either table. You can also update columns in the table on the *one* side of a relationship so long as the query returns unique rows for that table. The database must be able to determine the relationship between tables or queries to update columns in a query. In general, if a table is joined by its primary key to a query, you can update columns in the query (because the primary key indicates that the table is on the one side of the join). If you want to update a table with the results of a query, you must insert the query results into a temporary table that can be defined with a one-to-many or one-to-one relationship with the target table and then use the temporary table to update the target.

If you specify a <*search-condition*>, you can reference only columns found in the target table or query. If you use a subquery in the <*search-condition*>, you must not reference the target table, the query, or any underlying table of the query in the subquery.

In the SET clause, you cannot specify a column name more than once. You also cannot specify the name of a multi-valued field, a calculated field, or an attachment field. Values assigned to columns must be compatible with the column attributes. If you assign the Null value, the column cannot have the Required property set to Yes.

Both Access and SQL Server let you define column-value constraints (field validation rules in a desktop database), table constraints (table validation rules in a desktop database), and referential integrity checks, so any values that you update must pass these validations or the database will not let you run the query.

Example

To mark contacts who haven't had a contact event since January 1, 2010, enter the following (*qxmplSetInactive*):

```
UPDATE tblContacts
  LEFT JOIN
    (SELECT tblContactEvents.ContactID, tblContactEvents.ContactDateTime
      FROM tblContactEvents
      WHERE tblContactEvents.ContactDateTime>=#1/1/2010#) AS Active
  ON tblContacts.ContactID = Active.ContactID
SET tblContacts.Inactive = True
WHERE Active.ContactID IS NULL;
```

Note

Although the previous query updates rows on the one side of a relationship, the query is valid because the IS NULL test in conjunction with the LEFT JOIN returns exactly one unique row per contact.

See also Expression, IN Clause, Predicates (BETWEEN, Comparison, EXISTS, IN, LIKE NULL, and Quantified), Search-Condition, and WHERE Clause in this article.