http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Create and Delete Tables and Indexes Using Access SQL

Creating and Deleting Tables

Tables are the primary building blocks of a relational database.
A table contains rows (or records) of data, and each row is organized into a finite number of columns (or fields).
To build a new table in Access by using Access SQL, you must name the table, name the fields, and define the type of data that the fields will contain.
Use the CREATE TABLE statement to define the table in SQL.

```
CREATE TABLE tblCustomers
    (CustomerID INTEGER,
    [Last Name] TEXT(50),
    [First Name] TEXT(50),
    Phone TEXT(10),
    Email TEXT(50))
```

Notes

If a field name includes a space or some other nonalphanumeric character, you must enclose that field name within square brackets ([ ]). If you do not declare a length for text fields, they will default to 255 characters. For consistency and code readability, you should always define your field lengths.

You can declare a field to be NOT NULL, which means that null values cannot be inserted into that particular field; a value is always required. A null value should not be confused with an empty string or a value of 0; it is simply the database representation of an unknown value.

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

To remove a table from the database, use the DROP TABLE statement.

```
DROP TABLE tblCustomers
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Working with Indexes

An index is an external data structure used to sort or arrange pointers to data in a table.
When you apply an index to a table, you are specifying a certain arrangement of the data so that it can be accessed more quickly. However, if you apply too many indexes to a table, you may slow down the performance because there is extra overhead involved in maintaining the index, and because an index can cause locking issues when used in a multiuser environment. Used in the correct context, an index can greatly improve the performance of an application.
To build an index on a table, you must name the index, name the table to build the index on, name the field or fields within the table to use, and name the options you want to use.
You use the CREATE INDEX statement to build the index.

```
CREATE INDEX idxCustomerID
    ON tblCustomers (CustomerID)
```

Indexed fields can be sorted in one of two ways: ascending (ASC) or descending (DESC). The default order is ascending, and it does not have to be declared. You should declare the sort order with each field in the index.

```
CREATE INDEX idxCustomerID
    ON tblCustomers (CustomerID DESC)
```

There are four main options that you can use with an index:

- **PRIMARY**
- **DISALLOW NULL**
- **IGNORE NULL**
- **UNIQUE.**

The PRIMARY option designates the index as the primary key for the table. You can have only one primary key index per table, although the primary key index can be declared with more than one field. Use the WITH keyword to declare the index options.

```
CREATE INDEX idxCustomerID
    ON tblCustomers (CustomerID)
    WITH PRIMARY
```

To create a primary key index on more than one field, include all of the field names in the field list.

```
CREATE INDEX idxCustomerName
    ON tblCustomers ([Last Name], [First Name])
    WITH PRIMARY
```

The DISALLOW NULL option prevents insertion of null data in the field. (This is similar to the NOT NULL declaration used in the CREATE TABLE statement.)

```
CREATE INDEX idxCustomerEmail
    ON tblCustomers (Email)
    WITH DISALLOW NULL
```

The IGNORE NULL option causes null data in the table to be ignored for the index. That means that any record that has a null value in the declared field will not be used (or counted) in the index.

```
CREATE INDEX idxCustomerLastName
    ON tblCustomers ([Last Name])
    WITH IGNORE NULL
```

In addition to the PRIMARY, DISALLOW NULL, and IGNORE NULL options, you can also declare the index as UNIQUE, which means that only unique, non-repeating values can be inserted in the indexed field.

```
CREATE UNIQUE INDEX idxCustomerPhone
    ON tblCustomers (Phone)
```

To remove an index from a table, use the DROP INDEX statement.

```
DROP INDEX idxName
    ON tblCustomers
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Define Relationships Between Tables Using Access SQL

Relationships are the established associations between two or more tables.
Relationships are based on common fields from more than one table, often involving primary and foreign keys.
A `primary key` is the field (or fields) that is used to uniquely identify each record in a table.
There are three requirements for a primary key:

- **It cannot be null**
- **it must be unique**
- **there can be only one defined per table.**

You can define a primary key either by creating a primary key index after the table is created, or by using the `CONSTRAINT` clause in the table declaration, as shown in the examples later in this section. A constraint limits (or constrains) the values that are entered in a field.

A `foreign key` is a field (or fields) in one table that references the primary key in another table. The data in the fields from both tables is exactly the same, and the table with the primary key record (the primary table) must have existing records before the table with the foreign key record (the foreign table) has the matching or related records. Like primary keys, you can define foreign keys in the table declaration by using the `CONSTRAINT` clause.
There are essentially three types of relationships:

- **`One-to-one` For every record in the primary table, there is one and only one record in the foreign table.**
- **`One-to-many` For every record in the primary table, there are one or more related records in the foreign table.**
- **`Many-to-many` For every record in the primary table, there are many related records in the foreign table, and for every record in the foreign table, there are many related records in the primary table.**

When defining the relationships between tables, you must make the `CONSTRAINT` declarations at the field level. This means that the constraints are defined within a `CREATE TABLE` statement. To apply the constraints, use the `CONSTRAINT` keyword after a field declaration, name the constraint, name the table that it references, and name the field or fields within that table that will make up the matching foreign key.
The following statement assumes that the tblCustomers table has already been built, and that it has a primary key defined on the CustomerID field. The statement now builds the tblInvoices table, defining its primary key on the InvoiceID field. It also builds the one-to-many relationship between the tblCustomers and tblInvoices tables by defining another CustomerID field in the tblInvoices table. This field is defined as a foreign key that references the CustomerID field in the customers table. Note that the name of each constraint follows the `CONSTRAINT` keyword.

```
CREATE TABLE tblInvoices
    (InvoiceID INTEGER CONSTRAINT PK_InvoiceID PRIMARY KEY,
    CustomerID INTEGER NOT NULL CONSTRAINT FK_CustomerID
        REFERENCES tblCustomers (CustomerID),
    InvoiceDate DATETIME,
    Amount CURRENCY)
```

Note that the primary key index (PK_InvoiceID) for the invoices table is declared within the CREATE TABLE statement. To enhance the performance of the primary key, an index is automatically created for it, so there is no need to use a separate CREATE INDEX statement.
Now create a shipping table that will contain each customer's shipping address. Assume that there will be only one shipping record for each customer record, so you will be establishing a one-to-one relationship.

```
CREATE TABLE tblShipping
    (CustomerID INTEGER CONSTRAINT PK_CustomerID PRIMARY KEY
        REFERENCES tblCustomers (CustomerID),
    Address TEXT(50),
    City TEXT(50),
    State TEXT(2),
    Zip TEXT(10))
```

Note that the CustomerID field is both the primary key for the shipping table and the foreign key reference to the customers table.

Constraints

Constraints can be used to establish primary keys and referential integrity, and to restrict values that can be inserted into a field. In general, constraints can be used to preserve the integrity and consistency of the data in your database.
There are two types of constraints: a single-field or field-level constraint, and a multi-field or table-level constraint. Both kinds of constraints can be used in either the CREATE TABLE or the ALTER TABLE statement.
A single-field constraint, also known as a column-level constraint, is declared with the field itself, after the field and data type have been declared. Use the customers table and create a single-field primary key on the CustomerID field. To add the constraint, use the CONSTRAINT keyword with the name of the field.

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

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

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

However, using the shortcut method will cause Access to randomly generate a name for the constraint, which will make it difficult to reference in code. It is a good idea always to name your constraints.
To drop a constraint, use the DROP CONSTRAINT clause with the ALTER TABLE statement, and supply the name of the constraint.

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

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

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

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

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

The expression used to define a check constraint also can refer to more than one field in the same table, or to fields in other tables, and can use any operations that are valid in Microsoft Access SQL, such as SELECT statements, mathematical operators, and aggregate functions. The expression that defines the check constraint can be no more than 64 characters long.
Suppose that you want to check each customer's credit limit before he or she is added to the customers table. Using an ALTER TABLE statement with the ADD COLUMNand CONSTRAINT clauses, create a constraint that will look up the value in the CreditLimit table to verify the customer's credit limit. Use the following SQL statements to create the tblCreditLimit table, add the CustomerLimit field to the tblCustomers table, add the check constraint to the tblCustomers table, and test the check constraint.

```
CREATE TABLE tblCreditLimit (
   Limit DOUBLE)

INSERT INTO tblCreditLimit
   VALUES (100)

ALTER TABLE tblCustomers
   ADD COLUMN CustomerLimit DOUBLE

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

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

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

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Group Records in a Result Set Using Access SQL

Sometimes records in a table are logically related, as in the case of the invoices table. Because one customer can have many invoices, it could be useful to treat all the invoices for one customer as a group, in order to find statistical and summary information about the group.

The key to grouping records is that one or more fields in each record must contain the same value for every record in the group. In the case of the invoices table, the CustomerID field value is the same for every invoice a particular customer has.

To create a group of records, use the GROUP BY clause with the name of the field or fields you want to group with.

```
SELECT CustomerID, Count(*) AS [Number of Invoices],
    Avg(Amount) AS [Average Invoice Amount]
    FROM tblInvoices
    GROUP BY CustomerID
```

Note that the statement will return one record that shows the customer ID, the number of invoices the customer has, and the average invoice amount, for every customer who has an invoice record in the invoices table. Because each customer's invoices are treated as a group, you are able to count the number of invoices and then determine the average invoice amount.

You can specify a condition at the group level by using the HAVING clause, which is similar to the WHERE clause. For example, the following query returns only those records for each customer whose average invoice amount is less than 100:

```
SELECT CustomerID, Count(*) AS [Number of Invoices],
    Avg(Amount) AS [Average Invoice Amount]
    FROM tblInvoices
    GROUP BY CustomerID
    HAVING Avg(Amount) < 100
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Insert, Update, and Delete Records From a Table Using Access SQL

Inserting Records into a Table

There are essentially two methods for adding records to a table. The first is to add one record at a time; the second is to add many records at a time. In both cases, you use the SQL statement INSERT INTO to accomplish the task. INSERT INTO statements are commonly referred to as append queries.
To add one record to a table, you must use the field list to define which fields to put the data in, and then you must supply the data itself in a value list. To define the value list, use the VALUES clause. For example, the following statement will insert the values "1", "Kelly", and "Jill" into the CustomerID, Last Name, and First Name fields, respectively.

```
INSERT INTO tblCustomers (CustomerID, [Last Name], [First Name])
    VALUES (1, 'Kelly', 'Jill')
```

You can omit the field list, but only if you supply all the values that record can contain.

```
INSERT INTO tblCustomers
    VALUES (1, Kelly, 'Jill', '555-1040', 'someone@microsoft.com')
```

To add many records to a table at one time, use the INSERT INTO statement along with a SELECT statement. When you are inserting records from another table, each value being inserted must be compatible with the type of field that will be receiving the data.
The following INSERT INTO statement inserts all the values in the CustomerID, Last Name, and First Name fields from the tblOldCustomers table into the corresponding fields in the tblCustomers table.

```
INSERT INTO tblCustomers (CustomerID, [Last Name], [First Name])
    SELECT CustomerID, [Last Name], [First Name]
    FROM tblOldCustomers
```

If the tables are defined exactly alike, you can leave out the field lists.

```
INSERT INTO tblCustomers
    SELECT * FROM tblOldCustomers
```

Updating Records in a Table

To modify the data that is currently in a table, you use the UPDATE statement, which is commonly referred to as an update query. The UPDATE statement can modify one or more records and generally takes this form:

```
UPDATE table name
    SET field name = some value
```

To update all the records in a table, specify the table name, and then use the SET clause to specify the field or fields to be changed.

```
UPDATE tblCustomers
    SET Phone = 'None'
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

In most cases, you will want to qualify the UPDATE statement with a WHERE clause to limit the number of records changed.

```
UPDATE tblCustomers
    SET Email = 'None'
    WHERE [Last Name] = 'Smith'
```

Deleting Records from a Table

To delete the data that is currently in a table, you use the DELETE statement, which is commonly referred to as a delete query. This is also known as truncating a table. The DELETE statement can remove one or more records from a table and generally takes this form:

```
DELETE FROM table list
```

The DELETE statement does not remove the table structure, only the data that is currently being held by the table structure. To remove all the records from a table, use the DELETE statement and specify which table or tables from which you want to delete all the records.

```
DELETE FROM tblInvoices
```

In most cases, you will want to qualify the DELETE statement with a WHERE clause to limit the number of records to be removed.

```
DELETE FROM tblInvoices
    WHERE InvoiceID = 3
```

If you want to remove data only from certain fields in a table, use the UPDATE statement and set those fields equal to NULL, but only if they are nullable fields.

```
UPDATE tblCustomers
    SET Email = Null
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Modify a Table's Design Using Access SQL

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

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

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

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

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

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

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

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

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

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

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

```
CREATE TABLE tblCustomers (
   CustomerID INTEGER CONSTRAINT PK_tblCustomers
      PRIMARY KEY,
   [Last Name] TEXT(50) NOT NULL,
   [First Name] TEXT(50) NOT NULL,
   Phone TEXT(10),
   Email TEXT(50),
   Address TEXT(40) DEFAULT Unknown)
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Note

The DEFAULT statement can be executed only through the Access OLE DB provider and ADO. It will return an error n
interface.

Constraints

Constraints can be used to establish primary keys and referential integrity, and to restrict values that can be inserted into a field. In general, constraints can be used to preserve the integrity and consistency of the data in your database.
There are two types of constraints:

- **a single-field or field-level constraint,**
- **and a multi-field or table-level constraint.**

Both kinds of constraints can be used in either the CREATE TABLE or the ALTER TABLE statement.
A single-field constraint, also known as a column-level constraint, is declared with the field itself, after the field and data type have been declared. For this example, use the Customers table and create a single-field primary key on the CustomerID field. To add the constraint, use the CONSTRAINT keyword with the name of the field.

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

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

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

However, using the shortcut method will cause Access to randomly generate a name for the constraint, which will make it difficult to reference in code. It is a good idea always to name your constraints.
To drop a constraint, use the DROP CONSTRAINT clause with the ALTER TABLE statement, and supply the name of the constraint.

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

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

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

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

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

The expression used to define a check constraint also can refer to more than one field in the same table, or to fields in other tables, and can use any operations that are valid in Microsoft Access SQL, such as SELECT statements, mathematical operators, and aggregate functions. The expression that defines the check constraint can be no more than 64 characters long.
Suppose that you want to check each customer's credit limit before he or she is added to the Customers table. Using an ALTER TABLE statement with the ADD COLUMN and CONSTRAINT clauses, create a constraint that will look up the value in the CreditLimit table to verify the customer's credit limit. Use the following SQL statements to create the tblCreditLimit table, add the CustomerLimit field to the tblCustomers table, add the check constraint to the tblCustomers table, and test the check constraint.

```
CREATE TABLE tblCreditLimit (
   Limit DOUBLE)

INSERT INTO tblCreditLimit
   VALUES (100)

ALTER TABLE tblCustomers
   ADD COLUMN CustomerLimit DOUBLE

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

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

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

Cascading updates and deletions

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

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

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

```
ALTER TABLE tblShipping
   ADD CONSTRAINT FK_tblShipping
   FOREIGN KEY (CustomerID) REFERENCES
      tblCustomers (CustomerID)
   ON UPDATE CASCADE
   ON DELETE CASCADE
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Perform Joins Using Access SQL

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

INNER JOINs

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

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

Notice that the table names are divided by the INNER JOIN keywords and that the relational comparison is after the ON keyword. For the relational comparisons, you can also use the <, >, <=, >=, or <> operators, and you can also use the BETWEEN keyword. Also note that the ID fields from both tables are used only in the relational comparison; they are not part of the final result set.
To further qualify the SELECT statement, you can use a WHERE clause after the join comparison in the ON clause. The following example narrows the result set to include only invoices dated after January 1, 1998.

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

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

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

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

```
SELECT tblCustomers.[Last Name],
   tblCustomers.[First Name]
   FROM tblCustomers INNER JOIN tblCustomers AS A
```

```
    ON tblCustomers.[Last Name]=A.[Last Name]
    WHERE tblCustomers.[First Name]<>A.[First Name]
    ORDER BY tblCustomers.[Last Name]
```

OUTER JOINs

An OUTER JOIN is used to retrieve records from multiple tables while preserving records from one of the tables, even if there is no matching record in the other table. There are two types of OUTER JOINs that the Access database engine supports: LEFT OUTER JOINs and RIGHT OUTER JOINs. Think of two tables that are beside each other, a table on the left and a table on the right. The LEFT OUTER JOIN selects all rows in the right table that match the relational comparison criteria, and also selects all rows from the left table, even if no match exists in the right table. The RIGHT OUTER JOIN is simply the reverse of the LEFT OUTER JOIN; all rows in the right table are preserved instead.
As an example, suppose that you want to determine the total amount invoiced to each customer, but if a customer has no invoices, you want to show it by displaying the word "NONE."

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

Several things occur in the previous SQL statement. The first is the use of the string concatenation operator "&". This operator allows you to join two or more fields together as one string. The second is the immediate if (IIf) statement, which checks to see if the total is null. If it is, the statement returns the word "NONE." If the total is not null, the value is returned. The final thing is the OUTER JOIN clause. Using the LEFT OUTER JOIN preserves the rows in the left table so that you see all customers, even those who do not have invoices.
OUTER JOINs can be nested inside INNER JOINs in a multi-table join, but INNER JOINs cannot be nested inside OUTER JOINs.

The Cartesian product

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

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

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

The UNION operator

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

the same number of fields, but the fields do not have to be the same data type. Suppose that you have an Employees table that has the same structure as the Customers table, and you want to build a list of names and e-mail addresses by combining both tables.

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

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

```
TABLE tblCustomers
UNION
TABLE tblEmployees
```

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

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

The TRANSFORM statement

Although the TRANSFORM statement, also known as a crosstab query, is also not technically considered a join, it is included here because it does involve combining data from multiple sources of data into one result set, which is similar to some types of joins.
A TRANSFORM statement is used to calculate a sum, average, count, or other type of aggregate total on records. It then displays the information in a grid or spreadsheet format with data grouped both vertically (rows) and horizontally (columns). The general form for a TRANSFORM statement is this:

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

Suppose that you want to build a datasheet that displays the invoice totals for each customer on a year-by-year basis. The vertical headings will be the customer names, and the horizontal headings will be the years. You can modify a previous SQL statement to fit the transform statement.

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

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

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Retrieve Records Using Access SQL

The most basic and most often used SQL statement is the SELECT statement. SELECT statements are the workhorses of all SQL statements, and they are commonly referred to as select queries. You use the SELECT statement to retrieve data from the database tables, and the results are usually returned in a set of records (or rows) made up of any number of fields (or columns). You must use the FROM clause to designate which table or tables to select from. The basic structure of a SELECT statement is:

```
SELECT field list
    FROM table list
```

To select all fields from a table, use an asterisk (*). For example, the following statement selects all the fields and all the records from the Customers table:

```
SELECT *
    FROM tblCustomers
```

To limit the fields retrieved by the query, simply use the field names instead. For example:

```
SELECT [Last Name], Phone
    FROM tblCustomers
```

To designate a different name for a field in the result set, use the AS keyword to establish an alias for that field.

```
SELECT CustomerID AS [Customer Number]
    FROM tblCustomers
```

Restricting the Result Set

More often than not, you will not want to retrieve all records from a table. You will want only a subset of those records based on some qualifying criteria. To qualify a SELECT statement, you must use a WHERE clause, which will allow you to specify exactly which records you want to retrieve.

```
SELECT *
    FROM tblInvoices
    WHERE CustomerID = 1
```

Note the `CustomerID = 1` portion of the WHERE clause. A WHERE clause can contain up to 40 such expressions, and they can be joined with the And or Or logical operators. Using more than one expression allows you to further filter out records in the result set.

```
SELECT *
    FROM tblInvoices
    WHERE CustomerID = 1 AND InvoiceDate > #01/01/98#
```

Note that the date string is enclosed in number signs (#). If you are using a regular string in an expression, you must enclose the string in single quotation marks ('). For example:

```
SELECT *
    FROM tblCustomers
    WHERE [Last Name] = 'White'
```

If you do not know the whole string value, you can use wildcard characters with the Like operator.

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

There are a number of wildcard characters to choose from, and the following table details what they are and what they can be used for.

Wildcard character	Description
*	Zero or more characters
?	Any single character
#	Any single digit (0-9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

Sorting the Result Set

To specify a particular sort order on one or more fields in the result set, use the optional ORDER BY clause. Records can be sorted in either ascending (ASC) or descending (DESC) order; ascending is the default.
Fields referenced in the ORDER BY clause do not have to be part of the SELECT statement's field list, and sorting can be applied to string, numeric, and date/time values. Always place the ORDER BY clause at the end of the SELECT statement.

```
SELECT *
    FROM tblCustomers
    ORDER BY [Last Name], [First Name] DESC
```

You can also use the field numbers (or positions) instead of field names in the ORDER BY clause.

```
SELECT *
    FROM tblCustomers
    ORDER BY 2, 3 DESC
```

http://msdn.microsoft.com/en-us/library/office/ee336040(v=office.12).aspx

Use Aggregate Functions to Work with Values in Access SQL

Aggregate functions are used to calculate statistical and summary information from data in tables. These functions are used in SELECT statements, and all of them take fields or expressions as arguments.
To count the number of records in a result set, use the Count function. Using an asterisk with the Count function causes Null values to be counted as well.

```
SELECT Count(*) AS [Number of Invoices]
    FROM tblInvoices
```

To count only non-Null values, use the Count function with a field name:

```
SELECT Count(Amount) AS
    [Number of Valid Invoice Amounts]
    FROM tblInvoices
```

To find the average value for a column or expression of numeric data, use the Avg function:

```
SELECT Avg(Amount) AS [Average Invoice Amount]
    FROM tblInvoices
```

To find the total of the values in a column or expression of numeric data, use the Sum function:

```
SELECT Sum(Amount) AS [Total Invoice Amount]
    FROM tblInvoices
```

To find the minimum value for a column or expression, use the Min function:

```
SELECT Min(Amount) AS [Minimum Invoice Amount]
    FROM tblInvoices
```

To find the maximum value for a column or expression, use the Max function:

```
SELECT Max(Amount) AS [Maximum Invoice Amount]
    FROM tblInvoices
```

To find the first value in a column or expression, use the First function:

```
SELECT First(Amount) AS [First Invoice Amount]
    FROM tblInvoices
```

To find the last value in a column or expression, use the Last function:

```
SELECT Last(Amount) AS [Last Invoice Amount]
    FROM tblInvoices
```