SQL References For Personal SQL Version 1.0

PSQL Statements

PSQL Clause

PSQL Aggregate Functions

PSQL Operators

PSQL Declaration, Operation, Predicates

Other PSQL References

ALTER TABLE Statement

Modifies the design of a table after it has been created with the CREATE TABLE statement.

Syntax

ALTER TABLE *table* {ADD {COLUMN *field type*[(*size*)] [NOT NULL] [CONSTRAINT *index*] |
CONSTRAINT *multifieldindex*} |
DROP {COLUMN *field* I CONSTRAINT *indexname*} }

Example

adds a Salary field with a data type of Currency to the Employees table

```
ALTER TABLE Employees
ADD COLUMN Salary CURRENCY;
```

removes the Salary field from the Employees table.

```
ALTER TABLE Employees
DROP COLUMN Salary;
```

The ALTER TABLE statement has these parts:

Part	Description
table	The name of the table to be altered.
field	The name of the field to be added to or deleted from *table*.
type	The data type of *field*.
size	The field size in characters (Text and Binary fields only).
index	The index for *field*. See the CONSTRAINT clause topic for more information on how to construct this index.
multifieldindex	The definition of a multiple-field index to be added to *table*. See the CONSTRAINT clause topic for more information on how to construct this clause.
indexname	The name of the multiple-field index to be removed.

Remarks

Using the **ALTER TABLE** statement, you can alter an existing table in several ways. You can:

· Use **ADD COLUMN** to add a new field to the table. You specify the field name, data type, and (for Text and Binary fields) an optional size. For example, the following statement adds a 25-character Text field called Notes to the Employees table:

  ALTER TABLE Employees **ADD COLUMN** Notes TEXT(25)

  You can also define an index on that field. For more information on single-field indexes, see the **CONSTRAINT** clause topic.

  If you specify **NOT NULL** for a field, then new records are required to have valid data in that field.

· Use **ADD CONSTRAINT** to add a multiple-field index. For more information on multiple-field indexes, see the **CONSTRAINT** clause topic.

· Use **DROP COLUMN** to delete a field. You specify only the name of the field.

· Use **DROP CONSTRAINT** to delete a multiple-field index. You specify only the index name following the **CONSTRAINT** reserved word.

Notes

· You can't add or delete more than one field or index at a time.

· You can use the **CREATE INDEX** statement to add a single- or multiple-field index to a table, and you can use **ALTER TABLE** or the **DROP** statement to delete an index created with **ALTER TABLE** or **CREATE INDEX**.

· You can use NOT NULL on a single field, or within a named CONSTRAINT clause that applies to either a single field or to a multiple-field named CONSTRAINT. However, you can apply the NOT NULL restriction only once to a field, or a run-time error occurs.

CONSTRAINT Clause

A constraint is similar to an index, although it can also be used to establish a relationship with another table.

You use the **CONSTRAINT** clause in **ALTER TABLE** and **CREATE TABLE** statements to create or delete constraints. There are two types of **CONSTRAINT** clauses: one for creating a constraint on a single field and one for creating a constraint on more than one field.

Syntax

Single-field constraint:

CONSTRAINT *name* {PRIMARY KEY | UNIQUE | NOT NULL |
REFERENCES *foreigntable* [(*foreignfield1, foreignfield2*)]}

Multiple-field constraint:

CONSTRAINT *name*
{PRIMARY KEY (*primary1*[, *primary2* [, ...]]) |
UNIQUE (*unique1*[, *unique2* [, ...]]) |
NOT NULL (*notnull1*[, *notnull2* [, ...]]) |
FOREIGN KEY (*ref1*[, *ref2* [, ...]]) REFERENCES *foreigntable* [(*foreignfield1* [, *foreignfield2* [, ...]])]}

Example

Creates a new table called MyTable with two Text fields, a Date/Time field, and a unique index made up of all three fields

```
CREATE TABLE MyTable
      (FirstName TEXT, LastName TEXT,
      DateOfBirth DATETIME,
         CONSTRAINT MyTableConstraint UNIQUE
      (FirstName, LastName, DateOfBirth));
```

The CONSTRAINT clause has these parts:

Part	Description
name	The name of the constraint to be created.
primary1, *primary2*	The name of the field or fields to be designated the primary key.
unique1, *unique2*	The name of the field or fields to be designated as a unique key.
notnull1, notnull2	The name of the field or fields that are restricted to non-**Null** values.
ref1, *ref2*	The name of a foreign key field or fields that refer to fields in another table.
foreigntable	The name of the foreign table containing the field or fields specified by *foreignfield*.
foreignfield1, *foreignfield2*	The name of the field or fields in *foreigntable* specified by *ref1*, *ref2*. You can omit this clause if the referenced field is the primary key of *foreigntable*.

Remarks

You use the syntax for a single-field constraint in the field-definition clause of an **ALTER TABLE** or **CREATE TABLE** statement immediately following the specification of the field's data type.

You use the syntax for a multiple-field constraint whenever you use the reserved word **CONSTRAINT** outside a field-definition clause in an **ALTER TABLE** or **CREATE TABLE** statement.

Using **CONSTRAINT**, you can designate a field as one of the following types of constraints:

- You can use the **UNIQUE** reserved word to designate a field as a unique key. This means that no two records in the table can have the same value in this field. You can constrain any field or list of fields as unique. If a multiple-field constraint is designated as a unique key, the combined values of all fields in the index must be unique, even if two or more records have the same value in just one of the fields.
- You can use the **PRIMARY KEY** reserved words to designate one field or set of fields in a table as a primary key. All values in the primary key must be unique and not **Null**, and there can be only one primary key for a table.

  Note

  Don't set a **PRIMARY KEY** constraint on a table that already has a primary key; if you do, an error occurs.
- You can use the **FOREIGN KEY** reserved words to designate a field as a foreign key. If the foreign table's primary key consists of more than one field, you must use a multiple-field constraint definition, listing all of the referencing fields, the name of the foreign table, and the names of the referenced fields in the foreign table in the same order that the referencing fields are listed. If the referenced field or fields are the foreign table's primary key, you don't have to specify the referenced fields ¾ by default, the database engine behaves as if the foreign table's primary key is the referenced field.

CREATE INDEX Statement

Creates a new index on an existing table.

Syntax

```
CREATE [ UNIQUE ] INDEX index
    ON table (field [ASC|DESC][, field [ASC|DESC], ...])
    [WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]
```

Example

Creates an index consisting of the fields Home Phone and Extension in the Employees table.

```
CREATE INDEX NewIndex ON Employees
(HomePhone, Extension);
```

The CREATE INDEX statement has these parts:

Part	Description
index	The name of the index to be created.
table	The name of the existing table that will contain the index.
field	The name of the field or fields to be indexed. To create a single-field index, list the field name in parentheses following the table name. To create a multiple-field index, list the name of each field to be included in the index. To create descending indexes, use the DESC reserved word; otherwise, indexes are assumed to be ascending.

Remarks

To prohibit duplicate values in the indexed field or fields of different records, use the **UNIQUE** reserved word.

In the optional **WITH** clause, you can enforce data validation rules. You can:

- Prohibit **Null** entries in the indexed field or fields of new records by using the **DISALLOW NULL** option.
- Prevent records with **Null** values in the indexed field or fields from being included in the index by using the **IGNORE NULL** option.
- Designate the indexed field or fields as the primary key by using the **PRIMARY** reserved word. This implies that the key is unique, so you can omit the **UNIQUE** reserved word.

You can use **CREATE INDEX** to create a pseudo index on a linked table in an ODBC data source, such as SQL Server, that does not already have an index. You don't need permission or access to the remote server to create a pseudo index, and the remote database is unaware of and unaffected by the pseudo index. You use the same syntax for both linked and native tables. This can be especially useful to create an index on a table that would ordinarily be read-only due to lack of an index.

You can also use the **ALTER TABLE** statement to add a single- or multiple-field index to a table, and you can use the **ALTER TABLE** statement or the DROP statement to remove an index created with **ALTER TABLE** or **CREATE INDEX**.

Note Don't use the **PRIMARY** reserved word when you create a new index on a table that already has a primary key; if you do, an error occurs.

CREATE TABLE Statement

Creates a new table.

Syntax

CREATE TABLE *table* (*field1 type* [(*size*)] [NOT NULL] [*index1*] [, *field2 type* [(*size*)] [NOT NULL] [*index2*] [, ...]] [, CONSTRAINT *multifieldindex* [, ...]])

Example

Creates a new table called ThisTable with two Text fields

```
CREATE TABLE ThisTable
(FirstName TEXT, LastName TEXT);
```

The CREATE TABLE statement has these parts:

Part	Description
table	The name of the table to be created.
field1, *field2*	The name of field or fields to be created in the new table. You must create at least one field.
type	The data type of *field* in the new table.
size	The field size in characters (Text and Binary fields only).
index1, *index2*	A CONSTRAINT clause defining a single-field index. See the CONSTRAINT clause topic for more information on how to create this index.
multifieldindex	A CONSTRAINT clause defining a multiple-field index. See the CONSTRAINT clause topic for more information on how to create this index.

Remarks

Use the **CREATE TABLE** statement to define a new table and its fields and field constraints. If NOT NULL is specified for a field, then new records are required to have valid data in that field.

A **CONSTRAINT** clause establishes various restrictions on a field, and can be used to establish the primary key. You can also use the **CREATE INDEX** statement to create a primary key or additional indexes on existing tables.

You can use **NOT NULL** on a single field, or within a named **CONSTRAINT** clause that applies to either a single field or to a multiple-field named **CONSTRAINT**. However, you can apply the **NOT NULL** restriction only once to a field, or a run-time error occurs.

DROP Statement

Deletes an existing table from a database or deletes an existing index from a table.

Syntax

DROP {TABLE *table* | INDEX *index* ON *table*}

Example

The following example assumes the existence of a hypothetical NewIndex
index on the Employees table in the Northwind database.
This example deletes the index MyIndex from the Employees table.
DROP INDEX NewIndex ON Employees;

The DROP statement has these parts:

Part	Description
table	The name of the table to be deleted or the table from which an index is to be deleted.
index	The name of the index to be deleted from *table.*

Remarks

You must close the table before you can delete it or remove an index from it.

You can also use **ALTER TABLE** to delete an index from a table.

You can use **CREATE TABLE** to create a table and **CREATE INDEX** or **ALTER TABLE** to create an index. To modify a table, use **ALTER TABLE**.

SELECT Statement

Instructs the PSQL Jet database engine to return information from the database as a set of records.

Syntax

SELECT [*predicate*] { * | *table*.* | [*table*.]*field1* [AS *alias1*] [, [*table*.]*field2* [AS *alias2*] [, ...]]}
FROM *tableexpression* [, ...] [IN *externaldatabase*]
[WHERE...]
[GROUP BY...]
[HAVING...]
[ORDER BY...]
[WITH OWNERACCESS OPTION]

Example

Counts the number of records that have an entry in the PostalCode field and names the returned field Tally.

```
SELECT Count (PostalCode) AS Tally FROM Customers;
```

The SELECT statement has these parts:

Part	Description
predicate	One of the following predicates: ALL, DISTINCT, DISTINCTROW, or TOP. You use the predicate to restrict the number of records returned. If none is specified, the default is ALL.
*	Specifies that all fields from the specified table or tables are selected.
table	The name of the table containing the fields from which records are selected.
field1, *field2*	The names of the fields containing the data you want to retrieve. If you include more than one field, they are retrieved in the order listed.
alias1, *alias2*	The names to use as column headers instead of the original column names in *table*.
tableexpression	The name of the table or tables containing the data you want to retrieve.
externaldatabase	The name of the database containing the tables in *tableexpression* if they are not in the current database.

Remarks

To perform this operation, the PSQL Jet database engine searches the specified table or tables, extracts the chosen columns, selects rows that meet the criterion, and sorts or groups the resulting rows into the order specified.

SELECT statements don't change data in the database.

SELECT is usually the first word in an SQL statement. Most SQL statements are either **SELECT** or **SELECT...INTO** statements.

The minimum syntax for a **SELECT** statement is:

SELECT *fields* FROM *table*

You can use an asterisk (*) to select all fields in a table. The following example selects all of the fields in the Employees table:

```
SELECT * FROM Employees;
```

If a field name is included in more than one table in the **FROM** clause, precede it with the table name and the **.** (dot) operator. In the following example, the Department field is in both the Employees table and the Supervisors table. The SQL statement selects departments from the Employees table and supervisor names from the Supervisors table:

```
SELECT Employees.Department, Supervisors.SupvName
FROM Employees INNER JOIN Supervisors
WHERE Employees.Department = Supervisors.Department;
```

When a **Recordset** object is created, the PSQL Jet database engine uses the table's field name as the **Field** object name in the **Recordset** object. If you want a different field name or a name isn't implied by the expression used to generate the field, use the AS reserved word. The following example uses the title Birth to name the returned **Field** object in the resulting **Recordset** object:

```
SELECT BirthDate
AS Birth FROM Employees;
```

Whenever you use aggregate functions or queries that return ambiguous or duplicate **Field** object names, you must use the **AS** clause to provide an alternate name for the **Field** object. The following example uses the title HeadCount to name the returned **Field** object in the resulting **Recordset** object:

```
SELECT COUNT(EmployeeID)
AS HeadCount FROM Employees;
```

You can use the other clauses in a **SELECT** statement to further restrict and organize your returned data. For more information, see the Help topic for the clause you're using.

FROM Clause

Specifies the tables or queries that contain the fields listed in the SELECT statement.

Syntax

```
SELECT fieldlist
   FROM tableexpression [IN externaldatabase]
```

Example

Shows the number of employees and the average and maximum salaries.

```
SELECT Count (*) AS TotalEmployees, Avg(Salary) AS AverageSalary,
Max(Salary) AS MaximumSalary
FROM Employees;
```

A SELECT statement containing a FROM clause has these parts:

Part	Description
fieldlist	The name of the field or fields to be retrieved along with any field-name aliases, SQL aggregate functions, selection predicates (ALL, DISTINCT, DISTINCTROW, or TOP), or other SELECT statement options.
tableexpression	An expression that identifies one or more tables from which data is retrieved. The expression can be a single table name, a saved query name, or a compound resulting from an INNER JOIN, LEFT JOIN, or RIGHT JOIN.
externaldatabase	The full path of an external database containing all the tables in *tableexpression.*

Remarks

FROM is required and follows any **SELECT** statement.

The order of the table names in *tableexpression* isn't important.

For improved performance and ease of use, it's recommended that you use a linked table instead of an IN clause to retrieve data from an external database.

The following example shows how you can retrieve data from the Employees table:

```
SELECT LastName, FirstName
FROM Employees;
```

IN Clause

Identifies tables in any external database to which the PSQL Jet database engine can connect, such as a dBASE or Paradox database or an external PSQL Jet database.

Syntax

To identify a destination table:

[SELECT | INSERT] INTO *destination* IN
{*path* | ["*path*" "*type*"] | ["" [*type*; DATABASE = *path*]]}

To identify a source table:

FROM *tableexpression* IN
{*path* | ["*path*" "*type*"] | ["" [*type*; DATABASE = *path*]]}

Example

The following table shows how you can use the IN clause to retrieve data from an external database.
In each example, assume the hypothetical Customers table is stored in an external database.

Microsoft Jet database
SELECT CustomerIDFROM CustomersIN OtherDB.mdb WHERE CustomerID Like "A*";
dBASE III or IV.To retrieve data from a dBASE III table, substitute "dBASE III;" for "dBASE IV;".
SELECT CustomerIDFROM CustomerIN "C:\DBASE\DATA\SALES" "dBASE IV;"WHERE CustomerID Like "A*";
dBASE III or IV using Database syntax.
SELECT CustomerIDFROM CustomerIN "" [dBASE IV; Database=C:\DBASE\DATA\SALES;] WHERE CustomerID Like "A*";
Paradox 3.x or 4.x. To retrieve data from a Paradox version 3.x table, substitute "Paradox 3.x;" for "Paradox 4.x;"
SELECT CustomerIDFROM CustomerIN "C:\PARADOX\DATA\SALES" "Paradox 4.x;"WHERE CustomerID Like "A*";
Paradox 3.x or 4.x using Database syntax.
SELECT CustomerIDFROM CustomerIN "" [Paradox 4.x;Database=C:\PARADOX\DATA\SALES;] WHERE CustomerID Like "A*";
A Microsoft Excel worksheet
SELECT CustomerID, CompanyNameFROM [Customers$] IN "c:\documents\xldata.xls" "EXCEL 5.0;"WHERE CustomerID Like "A*"ORDER BY CustomerID;
A named range in a worksheet
SELECT CustomerID, CompanyNameFROM CustomersRangeIN "c:\documents\xldata.xls" "EXCEL 5.0;"WHERE CustomerID Like "A*"ORDER BY CustomerID;

A SELECT statement containing an IN clause has these parts:

Part	Description
destination	The name of the external table into which data is inserted.
tableexpression	The name of the table or tables from which data is retrieved. This argument can be a single table name, a saved query, or a compound resulting from an INNER JOIN, LEFT JOIN, or RIGHT JOIN.
path	The full path for the directory or file containing *table.*
type	The name of the database type used to create *table* if a database isn't a PSQL Jet database (for example, dBASE III, dBASE IV, Paradox 3.x, or Paradox 4.x).

Remarks

You can use **IN** to connect to only one external database at a time.

In some cases, the *path* argument refers to the directory containing the database files. For example, when working with dBASE, FoxPro, or Paradox database tables, the *path* argument specifies the directory containing .dbf or .db files. The table file name is derived from the *destination* or *tableexpression* argument.

To specify a non-PSQL Jet database, append a semicolon (;) to the name, and enclose it in single (' ')

or double ("") quotation marks. For example, either 'dBASE IV;' or "dBASE IV;" is acceptable.

You can also use the **DATABASE reserved** word to specify the external database. For example, the following lines specify the same table:

```
... FROM Table IN "" [dBASE IV; DATABASE=C:\DBASE\DATA\SALES;];
... FROM Table IN "C:\DBASE\DATA\SALES" "dBASE IV;"
```

Notes

- For improved performance and ease of use, use a linked table instead of IN.
- You can also use the IN reserved word as a comparison operator in an expression. For more information, see the **In** operator.

WHERE Clause

Specifies which records from the tables listed in the **FROM** clause are affected by a **SELECT**, **UPDATE**, or **DELETE** statement.

Syntax

```
SELECT fieldlist
FROM tableexpression
WHERE criteria
```

Example

Example selects the LastName and FirstName fields of each record in which the last name is King

```
SELECT LastName, FirstName
FROM Employees
WHERE LastName = 'King';
```

A SELECT statement containing a WHERE clause has these parts:

Part	Description
fieldlist	The name of the field or fields to be retrieved along with any field-name aliases, selection predicates (ALL, DISTINCT, DISTINCTROW, or TOP), or other SELECT statement options.
tableexpression	The name of the table or tables from which data is retrieved.
criteria	An expression that records must satisfy to be included in the query results.

Remarks

The PSQL Jet database engine selects the records that meet the conditions listed in the **WHERE** clause. If you don't specify a **WHERE** clause, your query returns all rows from the table. If you specify more than one table in your query and you haven't included a **WHERE** clause or a **JOIN** clause, your query generates a Cartesian product of the tables.

WHERE is optional, but when included, follows **FROM**. For example, you can select all employees in the sales department (WHERE Dept = 'Sales') or all customers between the ages of 18 and 30 (WHERE Age Between 18 And 30).

If you don't use a **JOIN** clause to perform SQL join operations on multiple tables, the resulting **Recordset** object won't be updatable.

WHERE is similar to **HAVING**. **WHERE** determines which records are selected. Similarly, once records are grouped with **GROUP BY**, **HAVING** determines which records are displayed.

Use the **WHERE** clause to eliminate records you don't want grouped by a **GROUP BY** clause.

Use various expressions to determine which records the SQL statement returns. For example, the following SQL statement selects all employees whose salaries are more than $21,000:

```
SELECT LastName, Salary
FROM Employees
WHERE Salary > 21000;
```

A **WHERE** clause can contain up to 40 expressions linked by logical operators, such as **And** and **Or**.

When you enter a field name that contains a space or punctuation, surround the name with brackets ([]).   For example, a customer information table might include information about specific customers :

```
SELECT [Customer’s Favorite Restarant]
```

When you specify the *criteria* argument, date literals must be in U.S. format, even if you're not using the U.S. version of the PSQL Jet database engine. For example, May 10, 1996, is written 10/5/96 in the United Kingdom and 5/10/96 in the United States. Be sure to enclose your date literals with the number sign (#) as shown in the following examples.

To find records dated May 10, 1996 in a United Kingdom database, you must use the following SQL statement:

```
SELECT *
FROM Orders
WHERE ShippedDate = #5/10/96#;
```

You can also use the **DateValue** function which is aware of the international settings established by Microsoft Windows. For example, use this code for the United States:

```
SELECT *
FROM Orders
WHERE ShippedDate = DateValue('5/10/96');
```

And use this code for the United Kingdom:

```
SELECT *
FROM Orders
WHERE ShippedDate = DateValue('10/5/96');
```

Note If the column referenced in the criteria string is of type GUID, the criteria expression uses a slightly different syntax:

```
WHERE ReplicaID = {GUID {12345678-90AB-CDEF-1234-567890ABCDEF}}
```

Be sure to include the nested braces and hyphens as shown.

GROUP BY Clause

Combines records with identical values in the specified field list into a single record. A summary value is created for each record if you include an SQL aggregate function, such as **Sum** or **Count**, in the **SELECT** statement.

Syntax

```
SELECT fieldlist
FROM table
WHERE criteria
[GROUP BY groupfieldlist]
```

Example

Calls the EnumFields procedure, which you can find in the SELECT statement example

```
SELECT Title, Count([Title]) AS Tally
FROM Employees GROUP BY Title;
```

A SELECT statement containing a GROUP BY clause has these parts:

Part	Description
fieldlist	The name of the field or fields to be retrieved along with any field-name aliases, SQL aggregate functions, selection predicates (ALL, DISTINCT, DISTINCTROW, or TOP), or other SELECT statement options.
table	The name of the table from which records are retrieved. For more information, see the FROM clause.
criteria	Selection criteria. If the statement includes a WHERE clause, the PSQL Jet database engine groups values after applying the WHERE conditions to the records.
groupfieldlist	The names of up to 10 fields used to group records. The order of the field names in *groupfieldlist* determines the grouping levels from the highest to the lowest level of grouping.

Remarks

GROUP BY is optional.

Summary values are omitted if there is no SQL aggregate function in the **SELECT** statement.

Null values in **GROUP BY** fields are grouped and aren't omitted. However, **Null** values aren't evaluated in any SQL aggregate function.

Use the **WHERE** clause to exclude rows you don't want grouped, and use the **HAVING** clause to filter records after they've been grouped.

Unless it contains Memo or OLE Object data, a field in the **GROUP BY** field list can refer to any field in any table listed in the FROM clause, even if the field isn't included in the **SELECT** statement, provided the **SELECT** statement includes at least one SQL aggregate function. The PSQL Jet database engine can't group on Memo or OLE Object fields.

All fields in the **SELECT** field list must either be included in the GROUP BY clause or be included as arguments to an SQL aggregate function.

HAVING Clause

Specifies which grouped records are displayed in a **SELECT** statement with a **GROUP BY** clause. After **GROUP BY** combines records, **HAVING** displays any records grouped by the **GROUP BY** clause that satisfy the conditions of the **HAVING** clause.

Syntax

```
SELECT fieldlist
   FROM table
   WHERE selectcriteria
   GROUP BY groupfieldlist
   [HAVING groupcriteria]
```

Example

Calls the EnumFields procedure, which you can find in the SELECT statement example

```
SELECT Title, Count(Title) as Total
FROM Employees
WHERE Region = 'WA'
GROUP BY Title HAVING Count(Title) > 1;
```

A SELECT statement containing a HAVING clause has these parts:

Part	Description
fieldlist	The name of the field or fields to be retrieved along with any field-name aliases, SQL aggregate functions, selection predicates (ALL, DISTINCT, DISTINCTROW, or TOP), or other SELECT statement options.
table	The name of the table from which records are retrieved. For more information, see the FROM clause.
selectcriteria	Selection criteria. If the statement includes a WHERE clause, the PSQL Jet database engine groups values after applying the WHERE conditions to the records.
groupfieldlist	The names of up to 10 fields used to group records. The order of the field names in *groupfieldlist* determines the grouping levels from the highest to the lowest level of grouping.
groupcriteria	An expression that determines which grouped records to display.

Remarks

HAVING is optional.

HAVING is similar to **WHERE**, which determines which records are selected. After records are grouped with **GROUP BY**, **HAVING** determines which records are displayed:

```
SELECT CategoryID,
Sum(UnitsInStock)
FROM Products
GROUP BY CategoryID
HAVING Sum(UnitsInStock) > 100 And Like "BOS*";
```

A **HAVING** clause can contain up to 40 expressions linked by logical operators, such as **And** and **Or**.

ORDER BY Clause

Sorts a query's resulting records on a specified field or fields in ascending or descending order.

Syntax

```
SELECT fieldlist
   FROM table
   WHERE selectcriteria
   [ORDER BY field1 [ASC | DESC ][, field2 [ASC | DESC ]][, ...]]]
```

Example

Calls the EnumFields procedure, which you can find in the SELECT statement example

```
SELECT LastName, FirstName
FROM Employees
ORDER BY LastName DESC;
```

A SELECT statement containing an ORDER BY clause has these parts:

Part	Description
fieldlist	The name of the field or fields to be retrieved along with any field-name aliases, SQL aggregate functions, selection predicates (ALL, DISTINCT, DISTINCTROW, or TOP), or other SELECT statement options.
table	The name of the table from which records are retrieved. For more information, see the FROM clause.
selectcriteria	Selection criteria. If the statement includes a WHERE clause, the PSQL Jet database engine orders values after applying the WHERE conditions to the records.
field1, *field2*	The names of the fields on which to sort records.

Remarks

ORDER BY is optional. However, if you want your data displayed in sorted order, then you must use **ORDER BY**.

The default sort order is ascending (A to Z, 0 to 9). Both of the following examples sort employee names in last name order:

```
SELECT LastName, FirstName
FROM Employees
ORDER BY LastName;
```

```
SELECT LastName, FirstName
FROM Employees
ORDER BY LastName ASC;
```

To sort in descending order (Z to A, 9 to 0), add the **DESC** reserved word to the end of each field you want to sort in descending order. The following example selects salaries and sorts them in descending order:

```
SELECT LastName, Salary
FROM Employees
ORDER BY Salary DESC, LastName;
```

If you specify a field containing Memo or OLE Object data in the **ORDER BY** clause, an error occurs. The PSQL Jet database engine doesn't sort on fields of these types.

ORDER BY is usually the last item in an SQL statement.

You can include additional fields in the **ORDER BY** clause. Records are sorted first by the first field listed after **ORDER BY**. Records that have equal values in that field are then sorted by the value in the second field listed, and so on.

ALL, DISTINCT, DISTINCTROW, TOP Predicates

Specifies records selected with SQL queries.

Syntax

```
SELECT [ALL | DISTINCT | DISTINCTROW | [TOP n [PERCENT]]]
   FROM table
```

Example

This example creates a query that joins the Customers and Orders tables on the CustomerID field. The Customers table contains no duplicate CustomerID fields, but the Orders table does because each customer can have many orders. Using DISTINCTROW produces a list of companies that have at least one order but without any details about those orders.

```
SELECT DISTINCTROW
CompanyName FROM Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
ORDER BY CompanyName;
```

A SELECT statement containing these predicates has the following parts:

Part	Description

ALL	Assumed if you don't include one of the predicates. The PSQL Jet database engine selects all of the records that meet the conditions in the SQL statement. The following two examples are equivalent and return all records from the Employees table:
	`SELECT ALL *`
	`FROM Employees`
	`ORDER BY EmployeeID;`
	`SELECT *`
	`FROM Employees`
	`ORDER BY EmployeeID;`

DISTINCT	Omits records that contain duplicate data in the selected fields. To be included in the results of the query, the values for each field listed in the SELECT statement must be unique. For example, several employees listed in an Employees table may have the same last name. If two records contain Smith in the LastName field, the following SQL statement returns only one record that contains Smith:
	`SELECT DISTINCT`
	`LastName`
	`FROM Employees;`
	If you omit DISTINCT, this query returns both Smith records.
	If the SELECT clause contains more than one field, the combination of values from all fields must be unique for a given record to be included in the results.
	The output of a query that uses DISTINCT isn't updatable and doesn't reflect subsequent changes made by other users.

DISTINCTROW	Omits data based on entire duplicate records, not just duplicate fields. For example, you could create a query that joins the Customers and Orders tables on the CustomerID field. The Customers table contains no duplicate CustomerID fields, but the Orders table does because each customer can have many orders. The following SQL statement shows how you can use DISTINCTROW to produce a list of companies that have at least one order but without any details about those orders: `SELECT DISTINCTROW CompanyName`

```
FROM Customers INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
ORDER BY CompanyName;
```

If you omit DISTINCTROW, this query produces multiple rows for each company that has more than one order.

DISTINCTROW has an effect only when you select fields from some, but not all, of the tables used in the query. DISTINCTROW is ignored if your query includes only one table, or if you output fields from all tables.

TOP *n* [PERCENT]

Returns a certain number of records that fall at the top or the bottom of a range specified by an ORDER BY clause. Suppose you want the names of the top 25 students from the class of 1994:

```
SELECT TOP 25
FirstName, LastName
FROM Students
WHERE GraduationYear = 1994
ORDER BY GradePointAverage DESC;
```

If you don't include the ORDER BY clause, the query will return an arbitrary set of 25 records from the Students table that satisfy the WHERE clause.

The TOP predicate doesn't choose between equal values. In the preceding example, if the twenty-fifth and twenty-sixth highest grade point averages are the same, the query will return 26 records.

You can also use the PERCENT reserved word to return a certain percentage of records that fall at the top or the bottom of a range specified by an ORDER BY clause. Suppose that, instead of the top 25 students, you want the bottom 10 percent of the class:

```
SELECT TOP 10 PERCENT
FirstName, LastName
FROM Students
WHERE GraduationYear = 1994
ORDER BY GradePointAverage ASC;
```

The ASC predicate specifies a return of bottom values. The value that follows TOP must be an unsigned **Integer**.

TOP doesn't affect whether or not the query is updatable.

table

The name of the table from which records are retrieved.

DELETE Statement

Creates a delete query that removes records from one or more of the tables listed in the **FROM** clause that satisfy the **WHERE** clause.

Syntax

```
DELETE [table.*]
    FROM table
    WHERE criteria
```

Example

Deletes all records for employees whose title is Trainee. When the FROM clause includes only one table, you don't have to list the table name in the DELETE statement.

```
DELETE * FROM Employees
    WHERE Title = 'Trainee';
```

The DELETE statement has these parts:

Part	Description
table	The optional name of the table from which records are deleted.
table	The name of the table from which records are deleted.
criteria	An expression that determines which records to delete.

Remarks

DELETE is especially useful when you want to delete many records.

To drop an entire table from the database, you can use the **Execute** method with a **DROP** statement. If you delete the table, however, the structure is lost. In contrast, when you use **DELETE**, only the data is deleted; the table structure and all of the table properties, such as field attributes and indexes, remain intact.

You can use **DELETE** to remove records from tables that are in a one-to-many relationship with other tables. Cascade delete operations cause the records in tables that are on the many side of the relationship to be deleted when the corresponding record in the one side of the relationship is deleted in the query. For example, in the relationship between the Customers and Orders tables, the Customers table is on the one side and the Orders table is on the many side of the relationship. Deleting a record from Customers results in the corresponding Orders records being deleted if the cascade delete option is specified.

A delete query deletes entire records, not just data in specific fields. If you want to delete values in a specific field, create an update query that changes the values to **Null**.

Important

- After you remove records using a delete query, you can't undo the operation. If you want to know which records were deleted, first examine the results of a select query that uses the same criteria, and then run the delete query.
- Maintain backup copies of your data at all times. If you delete the wrong records, you can retrieve them from your backup copies.

INNER JOIN Operation

Combines records from two tables whenever there are matching values in a common field.

Syntax

```
FROM table1 INNER JOIN table2 ON table1.field1 compopr table2.field2
```

Example

Calls the EnumFields procedure, which you can find in the SELECT statement example

```
SELECT DISTINCTROW Sum (UnitPrice * Quantity) AS Sales,
(FirstName & Chr(32) & LastName) AS Name
FROM Employees
INNER JOIN (Orders INNER JOIN [Order Details]
ON [Order Details].OrderID = Orders.OrderID )
ON Orders.EmployeeID = Employees.EmployeeID
GROUP BY (FirstName & Chr(32) & LastName);
```

The INNER JOIN operation has these parts:

Part	Description
table1, *table2*	The names of the tables from which records are combined.
field1, *field2*	The names of the fields that are joined. If they aren't numeric, the fields must be of the same data type and contain the same kind of data, but they don't have to have the same name.
compopr	Any relational comparison operator: "=," "<," ">," "<=," ">=," or "<>."

Remarks

You can use an **INNER JOIN** operation in any **FROM** clause. This is the most common type of join. Inner joins combine records from two tables whenever there are matching values in a field common to both tables.

You can use **INNER JOIN** with the Departments and Employees tables to select all the employees in each department. In contrast, to select all departments (even if some have no employees assigned to them) or all employees (even if some aren't assigned to a department), you can use a **LEFT JOIN** or **RIGHT JOIN** operation to create an outer join.

If you try to join fields containing Memo or OLE Object data, an error occurs.

You can join any two numeric fields of like types. For example, you can join on AutoNumber and Long fields because they are like types. However, you cannot join Single and Double types of fields.

The following example shows how you could join the Categories and Products tables on the CategoryID field:

```
SELECT CategoryName, ProductName
FROM Categories INNER JOIN Products
ON Categories.CategoryID = Products.CategoryID;
```

In the preceding example, CategoryID is the joined field, but it isn't included in the query output because it isn't included in the **SELECT** statement. To include the joined field, include the field name in the **SELECT** statement ¾ in this case, Categories.CategoryID.

You can also link several **ON** clauses in a **JOIN** statement, using the following syntax:

```
SELECT fields
   FROM table1 INNER JOIN table2
   ON table1.field1 compopr table2.field1 AND
   ON table1.field2 compopr table2.field2) OR
   ON table1.field3 compopr table2.field3)];
```

You can also nest **JOIN** statements using the following syntax:

```
SELECT fields
   FROM table1 INNER JOIN
   (table2 INNER JOIN [( ]table3
   [INNER JOIN [( ]tablex [INNER JOIN ...)]
   ON table3.field3 compopr tablex.fieldx)]
   ON table2.field2 compopr table3.field3)
   ON table1.field1 compopr table2.field2;
```

A **LEFT JOIN** or a **RIGHT JOIN** may be nested inside an **INNER JOIN**, but an **INNER JOIN** may not be nested inside a **LEFT JOIN** or a **RIGHT JOIN**.

INSERT INTO Statement

Adds a record or multiple records to a table. This is referred to as an append query.

Syntax

Multiple-record append query:
INSERT INTO *target* [IN *externaldatabase*] [(*field1*[, *field2*[, ...]])]
SELECT [*source*.]*field1*[, *field2*[, ...]
FROM *tableexpression*

Single-record append query:
INSERT INTO *target* [(*field1*[, *field2*[, ...]])]
VALUES (*value1*[, *value2*[, ...])

Example

Creates a new record in the Employees table
INSERT INTO Employees (FirstName,LastName, Title)
VALUES ('Harry', 'Washington', 'Trainee');

The INSERT INTO statement has these parts:

Part	Description
target	The name of the table or query to append records to.
externaldatabase	The path to an external database. For a description of the path, see the IN clause.
source	The name of the table or query to copy records from.
field1, *field2*	Names of the fields to append data to, if following a *target* argument, or the names of fields to obtain data from, if following a *source* argument.
tableexpression	The name of the table or tables from which records are inserted. This argument can be a single table name or a compound resulting from an INNER JOIN, LEFT JOIN, or RIGHT JOIN operation or a saved query.
value1, *value2*	The values to insert into the specific fields of the new record. Each value is inserted into the field that corresponds to the value's position in the list: *value1* is inserted into *field1* of the new record, *value2* into *field2*, and so on. You must separate values with a comma, and enclose text fields in quotation marks (' ').

Remarks

You can use the **INSERT INTO** statement to add a single record to a table using the single-record append query syntax as shown above. In this case, your code specifies the name and value for each field of the record. You must specify each of the fields of the record that a value is to be assigned to and a value for that field. When you don't specify each field, the default value or **Null** is inserted for missing columns. Records are added to the end of the table.

You can also use **INSERT INTO** to append a set of records from another table or query by using the **SELECT ... FROM** clause as shown above in the multiple-record append query syntax. In this case, the **SELECT** clause specifies the fields to append to the specified *target* table.

The *source* or *target* table may specify a table or a query. If a query is specified, the PSQL Jet database engine appends records to any and all tables specified by the query.

INSERT INTO is optional but when included, precedes the **SELECT** statement.

If your destination table contains a primary key, make sure you append unique, non-**Null** values to the primary key field or fields; if you don't, the PSQL Jet database engine won't append the records.

If you append records to a table with an AutoNumber field and you want to renumber the appended records, don't include the AutoNumber field in your query. Do include the AutoNumber field in the query if you want to retain the original values from the field.

Use the **IN** clause to append records to a table in another database.

To create a new table, use the **SELECT... INTO** statement instead to create a make-table query.

To find out which records will be appended before you run the append query, first execute and view the results of a select query that uses the same selection criteria.

An append query copies records from one or more tables to another. The tables that contain the records you append aren't affected by the append query.

Instead of appending existing records from another table, you can specify the value for each field in a single new record using the **VALUES** clause. If you omit the field list, the **VALUES** clause must include a value for every field in the table; otherwise, the **INSERT** operation will fail. Use an additional **INSERT INTO** statement with a **VALUES** clause for each additional record you want to create.

LEFT JOIN, RIGHT JOIN Operations

Combines source-table records when used in any **FROM** clause.

Syntax

```
FROM table1 [ LEFT | RIGHT ] JOIN table2
   ON table1.field1 compopr table2.field2
```

Example

Calls the EnumFields procedure, which you can find in the SELECT statement example.

```
SELECT [Department Name], FirstName & Chr(32) & LastName AS Name
FROM Departments LEFT JOIN Employees
ON Departments.[Department ID] = Employees.[Department ID]
ORDER BY [Department Name];
```

The LEFT JOIN and RIGHT JOIN operations have these parts:

Part	Description
table1, *table2*	The names of the tables from which records are combined.
field1, *field2*	The names of the fields that are joined. The fields must be of the same data type and contain the same kind of data, but they don't need to have the same name.
compopr	Any relational comparison operator: "=," "<," ">," "<=," ">=," or "<>."

Remarks

Use a **LEFT JOIN** operation to create a left outer join. Left outer joins include all of the records from the first (left) of two tables, even if there are no matching values for records in the second (right) table.

Use a **RIGHT JOIN** operation to create a right outer join. Right outer joins include all of the records from the second (right) of two tables, even if there are no matching values for records in the first (left) table.

For example, you could use **LEFT JOIN** with the Departments (left) and Employees (right) tables to select all departments, including those that have no employees assigned to them. To select all employees, including those who aren't assigned to a department, you would use **RIGHT JOIN**.

The following example shows how you could join the Categories and Products tables on the CategoryID field. The query produces a list of all categories, including those that contain no products:

```
SELECT CategoryName,
ProductName
FROM Categories LEFT JOIN Products
ON Categories.CategoryID = Products.CategoryID;
```

In this example, CategoryID is the joined field, but it isn't included in the query results because it isn't included in the **SELECT** statement. To include the joined field, enter the field name in the **SELECT** statement ¾ in this case, Categories.CategoryID.

Notes

- To create a query that includes only records in which the data in the joined fields is the same, use an **INNER JOIN** operation.
- A **LEFT JOIN** or a **RIGHT JOIN** can be nested inside an **INNER JOIN**, but an **INNER JOIN** cannot be nested inside a **LEFT JOIN** or a **RIGHT JOIN**. See the discussion of nesting in the INNER JOIN

topic to see how to nest joins within other joins.

- You can link multiple **ON** clauses. See the discussion of clause linking in the **INNER JOIN** topic to see how this is done.
- If you try to join fields containing Memo or OLE Object data, an error occurs.

PARAMETERS Declaration

Declares the name and data type of each parameter in a parameter query.

Syntax

PARAMETERS *name datatype* [, *name datatype* [, ...]]

Example

```
PARAMETERS [Employee Title] TEXT;
```

The PARAMETERS declaration has these parts:

Part	Description
name	The name of the parameter. Assigned to the **Name** property of the **Parameter** object and used to identify this parameter in the **Parameters** collection. You can use *name* as a string that is displayed in a dialog box while your application runs the query. Use brackets ([ ]) to enclose text that contains spaces or punctuation. For example, [Low price] and [Begin report with which month?] are valid *name* arguments.
datatype	One of the primary PSQL Jet SQL data types or their synonyms.

Remarks

For queries that you run regularly, you can use a PARAMETERS declaration to create a parameter query. A parameter query can help automate the process of changing query criteria. With a parameter query, your code will need to provide the parameters each time the query is run.

The **PARAMETERS** declaration is optional but when included precedes any other statement, including **SELECT**.

If the declaration includes more than one parameter, separate them with commas. The following example includes two parameters:

```
PARAMETERS [Low price] Currency, [Beginning date] DateTime;
```

You can use *name* but not *datatype* in a **WHERE** or **HAVING** clause. The following example expects two parameters to be provided and then applies the criteria to records in the Orders table:

```
PARAMETERS [Low price] Currency,
[Beginning date] DateTime;
SELECT OrderID, OrderAmount
FROM Orders
WHERE OrderAmount > [Low price]
AND OrderDate >= [Beginning date];
```

PROCEDURE Clause

Defines a name and optional parameters for a query.

Syntax

```
PROCEDURE name [param1 datatype[, param2 datatype[, ...]]
```

Example

```
PROCEDURE CategoryList
SELECT DISTINCTROW CategoryName,
CategoryID FROM Categories
ORDER BY CategoryName;
```

The PROCEDURE clause has these parts:

Part	Description
name	A name for the procedure. It must follow standard naming conventions.
param1, *param2*	One or more field names or parameters. For example: PROCEDURE Sales_By_Country [Beginning Date] DateTime, [Ending Date] DateTime; For more information on parameters, see PARAMETERS.
datatype	One of the primary PSQL Jet SQL data types or their synonyms.

Remarks

An SQL procedure consists of a **PROCEDURE** clause (which specifies the name of the procedure), an optional list of parameter definitions, and a single SQL statement. For example, the procedure Get_Part_Number might run a query that retrieves a specified part number.

Notes

· If the clause includes more than one field definition (that is, *param-datatype* pairs), separate them with commas.

· The **PROCEDURE** clause must be followed by an SQL statement (for example, a **SELECT** or **UPDATE** statement).

SELECT...INTO Statement

Creates a make-table query.

Syntax

SELECT *field1*[, *field2*[, ...]] INTO *newtable* [IN *externaldatabase*]
  FROM *source*

Example

Selects all records in the Employees table and copies them into a new table named Emp Backup

```
SELECT Employees.*
INTO [Emp Backup] FROM Employees;
```

The SELECT...INTO statement has these parts:

Part	Description
field1, *field2*	The name of the fields to be copied into the new table.
newtable	The name of the table to be created. It must conform to standard naming conventions. If *newtable* is the same as the name of an existing table, a trappable error occurs.
externaldatabase	The path to an external database. For a description of the path, see the IN clause.
source	The name of the existing table from which records are selected. This can be single or multiple tables or a query.

Remarks

You can use make-table queries to archive records, make backup copies of your tables, or make copies to export to another database or to use as a basis for reports that display data for a particular time period. For example, you could produce a Monthly Sales by Region report by running the same make-table query each month.

Notes

- You may want to define a primary key for the new table. When you create the table, the fields in the new table inherit the data type and field size of each field in the query's underlying tables, but no other field or table properties are transferred.
- To add data to an existing table, use the **INSERT INTO** statement instead to create an append query.
- To find out which records will be selected before you run the make-table query, first examine the results of a **SELECT** statement that uses the same selection criteria.

SQL Subqueries

A subquery is a **SELECT** statement nested inside a **SELECT**, **SELECT...INTO**, **INSERT...INTO**, **DELETE**, or **UPDATE** statement or inside another subquery.

Syntax

You can use three forms of syntax to create a subquery:
comparison [ANY | ALL | SOME] (*sqlstatement*)
expression [NOT] IN (*sqlstatement*)
[NOT] EXISTS (*sqlstatement*)

Example

List the name and contact of every customer who placed an order in the second quarter of 1995.

```
SELECT ContactName, CompanyName, ContactTitle, Phone
FROM Customers
WHERE CustomerID
IN (SELECT CustomerID FROM Orders
WHERE OrderDate Between #04/1/95# And #07/1/95#;
```

A subquery has these parts:

Part	Description
comparison	An expression and a comparison operator that compares the expression with the results of the subquery.
expression	An expression for which the result set of the subquery is searched.
sqlstatement	A SELECT statement, following the same format and rules as any other SELECT statement. It must be enclosed in parentheses.

Remarks

You can use a subquery instead of an expression in the field list of a **SELECT** statement or in a **WHERE** or **HAVING** clause. In a subquery, you use a **SELECT** statement to provide a set of one or more specific values to evaluate in the **WHERE** or **HAVING** clause expression.

Use the **ANY** or **SOME** predicate, which are synonymous, to retrieve records in the main query that satisfy the comparison with any records retrieved in the subquery. The following example returns all products whose unit price is greater than that of any product sold at a discount of 25 percent or more:

```
SELECT * FROM Products
WHERE UnitPrice > ANY
(SELECT UnitPrice FROM OrderDetails
WHERE Discount >= .25);
```

Use the **ALL** predicate to retrieve only those records in the main query that satisfy the comparison with all records retrieved in the subquery. If you changed **ANY** to **ALL** in the previous example, the query would return only those products whose unit price is greater than that of all products sold at a discount of 25 percent or more. This is much more restrictive.

Use the **IN** predicate to retrieve only those records in the main query for which some record in the subquery contains an equal value. The following example returns all products with a discount of 25 percent or more:

```
SELECT * FROM Products
WHERE ProductID IN
(SELECT ProductID FROM OrderDetails
WHERE Discount >= .25);
```

Conversely, you can use **NOT IN** to retrieve only those records in the main query for which no record in the subquery contains an equal value.

Use the **EXISTS** predicate (with the optional **NOT** reserved word) in true/false comparisons to determine whether the subquery returns any records.

You can also use table name aliases in a subquery to refer to tables listed in a **FROM** clause outside the subquery. The following example returns the names of employees whose salaries are equal to or greater than the average salary of all employees having the same job title. The Employees table is given the alias "T1":

```
SELECT LastName,
FirstName, Title, Salary
FROM Employees AS T1
WHERE Salary >=
(SELECT Avg(Salary)
FROM Employees
WHERE T1.Title = Employees.Title) Order by Title;
```

In the preceding example, the AS reserved word is optional.

Some subqueries are allowed in crosstab queries ¾ specifically, as predicates (those in the **WHERE** clause). Subqueries as output (those in the **SELECT** list) are not allowed in crosstab queries.

TRANSFORM Statement

Creates a crosstab query.

Syntax

```
TRANSFORM aggfunction
    selectstatement
    PIVOT pivotfield [IN (value1[, value2[, ...]])]
```

Example

```
TRANSFORM Count(OrderID)
SELECT FirstName AS FullName
FROM Employees INNER JOIN Orders
ON Employees.EmployeeID = Orders.EmployeeID
WHERE DatePart (""yyyy"", OrderDate) = [prmYear];
```

The TRANSFORM statement has these parts:

Part	Description
aggfunction	An SQL aggregate function that operates on the selected data.
selectstatement	A SELECT statement.
pivotfield	The field or expression you want to use to create column headings in the query's result set.
value1, *value2*	Fixed values used to create column headings.

Remarks

When you summarize data using a crosstab query, you select values from specified fields or expressions as column headings so you can view data in a more compact format than with a select query.

TRANSFORM is optional but when included is the first statement in an **SQL** string. It precedes a **SELECT** statement that specifies the fields used as row headings and a **GROUP BY** clause that specifies row grouping. Optionally, you can include other clauses, such as **WHERE**, that specify additional selection or sorting criteria. You can also use subqueries as predicates ¾ specifically, those in the **WHERE** clause ¾ in a crosstab query.

The values returned in *pivotfield* are used as column headings in the query's result set. For example, pivoting the sales figures on the month of the sale in a crosstab query would create 12 columns. You can restrict *pivotfield* to create headings from fixed values (*value1*, *value2*) listed in the optional **IN** clause. You can also include fixed values for which no data exists to create additional columns.

UNION Operation

Creates a union query, which combines the results of two or more independent queries or tables.

Syntax

[TABLE] *query1* UNION [ALL] [TABLE] *query2* [UNION [ALL] [TABLE] *queryn* [...]]

Example

Retrieves the names and cities of all suppliers and customers in Brazil

```
SELECT CompanyName, City FROM Suppliers
WHERE Country = 'Brazil'
UNION
SELECT CompanyName, City FROM Customers
WHERE Country = 'Brazil';
```

The UNION operation has these parts:

Part	Description
query1-n	A SELECT statement, the name of a stored query, or the name of a stored table preceded by the TABLE keyword.

Remarks

You can merge the results of two or more queries, tables, and **SELECT** statements, in any combination, in a single **UNION** operation. The following example merges an existing table named New Accounts and a **SELECT** statement:

```
TABLE [New Accounts] UNION ALL
SELECT *
FROM Customers
WHERE OrderAmount > 1000;
```

By default, no duplicate records are returned when you use a UNION operation; however, you can include the ALL predicate to ensure that all records are returned. This also makes the query run faster.

All queries in a **UNION** operation must request the same number of fields; however, the fields don't have to be of the same size or data type.

Use aliases only in the first **SELECT** statement because they are ignored in any others. In the **ORDER BY** clause, refer to fields by what they are called in the first **SELECT** statement.

Notes

- You can use a **GROUP BY** or **HAVING** clause in each *query* argument to group the returned data.
- You can use an **ORDER BY** clause at the end of the last *query* argument to display the returned data in a specified order.

UPDATE Statement

Creates an update query that changes values in fields in a specified table based on specified criteria.

Syntax

```
UPDATE table
   SET newvalue
   WHERE criteria;
```

Example

Changes values in the ReportsTo field to 5 for all employee records that currently have ReportsTo values of 2

```
UPDATE Employees
SET ReportsTo = 5
WHERE ReportsTo = 2;
```

The UPDATE statement has these parts:

Part	Description
table	The name of the table containing the data you want to modify.
newvalue	An expression that determines the value to be inserted into a particular field in the updated records.
criteria	An expression that determines which records will be updated. Only records that satisfy the expression are updated.

Remarks

UPDATE is especially useful when you want to change many records or when the records that you want to change are in multiple tables.

You can change several fields at the same time. The following example increases the Order Amount values by 10 percent and the Freight values by 3 percent for shippers in the United Kingdom:

```
UPDATE Orders
SET OrderAmount = OrderAmount * 1.1,
Freight = Freight * 1.03
WHERE ShipCountry = 'UK';
```

Important

- **UPDATE** doesn't generate a result set. Also, after you update records using an update query, you can't undo the operation. If you want to know which records were updated, first examine the results of a select query that uses the same criteria, and then run the update query.
- Maintain backup copies of your data at all times. If you update the wrong records, you can retrieve them from your backup copies.

WITH OWNERACCESS OPTION Declaration

In a multiuser environment with a secure workgroup, use this declaration with a query to give the user who runs the query the same permissions as the query's owner.

Syntax

```
sqlstatement
   WITH OWNERACCESS OPTION
```

Remarks

The **WITH OWNERACCESS OPTION** declaration is optional.

The following example enables the user to view salary information (even if the user doesn't otherwise have permission to view the Payroll table), provided that the query's owner does have that permission:

```
SELECT LastName,
FirstName, Salary
FROM Employees
ORDER BY LastName
WITH OWNERACCESS OPTION;
```

If a user is otherwise prevented from creating or adding to a table, you can use **WITH OWNERACCESS OPTION** to enable the user to run a make-table or append query.

If you want to enforce workgroup security settings and users' permissions, don't include the **WITH OWNERACCESS OPTION** declaration.

This option requires you to have access to the System.mdw file associated with the database. It's really useful only in secured multiuser implementations.

Avg Function

Calculates the arithmetic mean of a set of values contained in a specified field on a query.

Syntax

Avg(***expr***)

Example

Calculate the average freight charges for orders with freight charges over $100

```
SELECT Avg(Freight) AS [Average Freight]
FROM Orders WHERE Freight > 100;
```

The *expr* placeholder represents a string expression identifying the field that contains the numeric data you want to average or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

The average calculated by **Avg** is the arithmetic mean (the sum of the values divided by the number of values). You could use **Avg**, for example, to calculate average freight cost.

The **Avg** function doesn't include any **Null** fields in the calculation.

You can use **Avg** in a query expression and in the **SQL** property of a **QueryDef** object or when creating a **Recordset** object based on an SQL query.

Count Function

Calculates the number of records returned by a query.

Syntax

Count(***expr***)

Example

Calculate the number of orders shipped to the United Kingdom.

```
SELECT Count (ShipCountry) AS [UK Orders]
FROM Orders WHERE ShipCountry = 'UK';
```

The *expr* placeholder represents a string expression identifying the field that contains the data you want to count or an expression that performs a calculation using the data in the field. Operands in *expr* can include the name of a table field or function (which can be either intrinsic or user-defined but not other SQL aggregate functions). You can count any kind of data, including text.

Remarks

You can use **Count** to count the number of records in an underlying query. For example, you could use **Count** to count the number of orders shipped to a particular country.

Although *expr* can perform a calculation on a field, **Count** simply tallies the number of records. It doesn't matter what values are stored in the records.

The **Count** function doesn't count records that have **Null** fields unless *expr* is the asterisk (*) wildcard character. If you use an asterisk, **Count** calculates the total number of records, including those that contain **Null** fields. **Count(*)** is considerably faster than **Count(**[*Column Name*]**)**. Don't enclose the asterisk in quotation marks (' '). The following example calculates the number of records in the Orders table:

```
SELECT Count(*)
AS TotalOrders FROM Orders;
```

If *expr* identifies multiple fields, the **Count** function counts a record only if at least one of the fields is not **Null**. If all of the specified fields are **Null**, the record isn't counted. Separate the field names with an ampersand (&). The following example shows how you can limit the count to records in which either ShippedDate or Freight isn't **Null**:

```
SELECT
Count('ShippedDate & Freight')
AS [Not Null] FROM Orders;
```

You can use **Count** in a query expression. You can also use this expression in the **SQL** property of a **QueryDef** object or when creating a **Recordset** object based on an SQL query.

First, Last Functions

Return a field value from the first or last record in the result set returned by a query.

Syntax

First(*expr***)**
Last(*expr***)**

Example

Return the values from the LastName field of the first and last records returned from the table.

```
SELECT First(LastName) as First, Last(LastName) as Last
FROM Employees;
```

The *expr* placeholder represents a string expression identifying the field that contains the data you want to use or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

They simply return the value of a specified field in the first or last record, respectively, of the result set returned by a query. Because records are usually returned in no particular order (unless the query includes an **ORDER BY** clause), the records returned by these functions will be arbitrary.

Min, Max Functions

Return the minimum or maximum of a set of values contained in a specified field on a query.

Syntax

Min(*expr***)**
Max(*expr***)**

Example

Return the lowest and highest freight charges for orders shipped to the United Kingdom

```
SELECT Min(Freight) AS [Low Freight],
Max(Freight)AS [High Freight]
FROM Orders WHERE ShipCountry = 'UK';
```

The *expr* placeholder represents a string expression identifying the field that contains the data you want to evaluate or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

You can use **Min** and **Max** to determine the smallest and largest values in a field based on the specified aggregation, or grouping. For example, you could use these functions to return the lowest and highest freight cost. If there is no aggregation specified, then the entire table is used.

You can use **Min** and **Max** in a query expression and in the **SQL** property of a **QueryDef** object or when creating a **Recordset** object based on an SQL query.

StDev, StDevP Functions

Return estimates of the standard deviation for a population or a population sample represented as a set of values contained in a specified field on a query.

Syntax

StDev(*expr***)**
StDevP(*expr***)**

Example

Estimate the standard deviation of the freight charges for orders shipped to the United Kingdom.

```
SELECT StDev(Freight)
AS [Freight Deviation] FROM Orders
WHERE ShipCountry = 'UK';
```

The *expr* placeholder represents a string expression identifying the field that contains the numeric data you want to evaluate or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

The **StDevP** function evaluates a population, and the **StDev** function evaluates a population sample.

If the underlying query contains fewer than two records (or no records, for the **StDevP** function), these functions return a **Null** value (which indicates that a standard deviation can't be calculated).

You can use the **StDev** and **StDevP** functions in a query expression. You can also use this expression in the **SQL** property of a **QueryDef** object or when creating a **Recordset** object based on an SQL query.

Sum Function

Returns the sum of a set of values contained in a specified field on a query.

Syntax

Sum(*expr*)

Example

Calculate the total sales for orders shipped to the United Kingdom.

```
SELECT Sum(UnitPrice*Quantity)AS [Total UK Sales]
FROM Orders
INNER JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID
WHERE (ShipCountry = 'UK');
```

The *expr* placeholder represents a string expression identifying the field that contains the numeric data you want to add or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

The **Sum** function totals the values in a field. For example, you could use the **Sum** function to determine the total cost of freight charges.

The **Sum** function ignores records that contain **Null** fields. The following example shows how you can calculate the sum of the products of UnitPrice and Quantity fields:

```
SELECT
Sum(UnitPrice * Quantity)
AS [Total Revenue] FROM [Order Details];
```

You can use the **Sum** function in a query expression. You can also use this expression in the **SQL** property of a **QueryDef** object or when creating a **Recordset** based on an SQL query.

Var, VarP Functions

Return estimates of the variance for a population or a population sample represented as a set of values contained in a specified field on a query.

Syntax

Var(*expr***)**
VarP(*expr***)**

Example

Estimate the variance of freight costs for orders shipped to the United Kingdom.

```
SELECT Var(Freight) AS [UK Freight Variance]
FROM Orders WHERE ShipCountry = 'UK';
```

The *expr* placeholder represents a string expression identifying the field that contains the numeric data you want to evaluate or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate functions).

Remarks

The **VarP** function evaluates a population, and the **Var** function evaluates a population sample.

If the underlying query contains fewer than two records, the **Var** and **VarP** functions return a **Null** value, which indicates that a variance can't be calculated.

You can use the **Var** and **VarP** functions in a query expression or in an SQL statement.

Calculating Fields in SQL Functions

You can use the string expression argument in an SQL aggregate function to perform a calculation on values in a field. For example, you could calculate a percentage (such as a surcharge or sales tax) by multiplying a field value by a fraction.

The following table provides examples of calculations on fields from the Orders and Order Details tables in the Northwind.mdb database.

Calculation	**Example**
Add a number to a field	Freight + 5
Subtract a number from a field	Freight - 5
Multiply a field by a number	UnitPrice * 2
Divide a field by a number	Freight / 2
Add one field to another	UnitsInStock + UnitsOnOrder
Subtract one field from another	ReorderLevel - UnitsInStock

The following example calculates the average discount amount of all orders in the Northwind.mdb database. It multiplies the values in the UnitPrice and Discount fields to determine the discount amount of each order and then calculates the average.

```
SELECT Avg(UnitPrice * Discount) AS [Average Discount] FROM [Order Details];
```

Between...And Operator

Determines whether the value of an expression falls within a specified range of values. You can use this operator within SQL statements.

Syntax

expr [Not] Between *value1* And *value2*

Example

Lists the name and contact of every customer who placed an order
in the second quarter of 1995

```
SELECT ContactName, CompanyName, ContactTitle, Phone
FROM Customers
WHERE CustomerID
IN (SELECT CustomerID FROM Orders
WHERE OrderDate Between #04/1/95# And #07/1/95#);"
```

The Between...And operator syntax has these parts:

Part	Description
expr	Expression identifying the field that contains the data you want to evaluate.
value1, *value2*	Expressions against which you want to evaluate *expr*.

Remarks

If the value of *expr* is between *value1* and *value2* (inclusive), the **Between...And** operator returns **True**; otherwise, it returns **False**. You can include the **Not** logical operator to evaluate the opposite condition (that is, whether *expr* lies outside the range defined by *value1* and *value2*).

You might use **Between...And** to determine whether the value of a field falls within a specified numeric range. The following example determines whether an order was shipped to a location within a range of postal codes. If the postal code is between 98101 and 98199, the **IIf** function returns "Local". Otherwise, it returns "Nonlocal".

```
SELECT IIf(PostalCode Between 98101 And 98199, "Local", "Nonlocal")
FROM Publishers
```

If *expr*, *value1*, or *value2* is **Null**, **Between...And** returns a **Null** value.

Because wildcard characters, such as *, are treated as literals, you cannot use them with the **Between...And** operator. For example, you cannot use 980* and 989* to find all postal codes that start with 980 to 989. Instead, you have two alternatives for accomplishing this. You can add an expression to the query that takes the left three characters of the text field and use **Between...And** on those characters. Or you can pad the high and low values with extra characters ¾ in this case, 98000 to 98999, or 98000 to 98999 – 9999 if using extended postal codes. (You must omit the – 0000 from the low values because otherwise 98000 is dropped if some postal codes have extended sections and others do not.)

Comparison of PSQL Jet Database Engine SQL and ANSI SQL

PSQL Jet database engine SQL is generally ANSI-89 Level 1 compliant. However, certain ANSI SQL features aren't implemented in PSQL Jet SQL. Conversely, PSQL Jet SQL includes reserved words and features not supported in ANSI SQL.

Major Differences

- PSQL Jet SQL and ANSI SQL each have different reserved words and data types. For more information, see PSQL Jet Database Engine SQL Reserved Words and Equivalent ANSI SQL Data Types.
- Different rules apply to the **Between...And** construct, which has the following syntax:

  expr1 [NOT] **Between** *value1* **And** *value2*

  In PSQL Jet SQL, *value1* can be greater than *value2*; in ANSI SQL, *value1* must be equal to or less than *value2.*
- Different wildcard characters are used with the **Like** operator.

Matching character	**PSQL Jet SQL**	**ANSI SQL**
Any single character	?	_ (underscore)
Zero or more characters	*	%

- PSQL Jet SQL is generally less restrictive. For example, it permits grouping and ordering on expressions.
- PSQL Jet SQL supports more powerful expressions.

Enhanced Features of PSQL Jet SQL

PSQL Jet SQL provides the following enhanced features:

- The TRANSFORM statement, which provides support for crosstab queries
- Additional aggregate functions, such as **StDev** and **VarP**
- The PARAMETERS declaration for defining parameter queries

ANSI SQL Features Not Supported in PSQL Jet SQL

PSQL Jet SQL doesn't support the following ANSI SQL features:

- Security statements, such as COMMIT, GRANT, and LOCK.
- DISTINCT aggregate function references. For example, PSQL Jet SQL doesn't allow SUM(DISTINCT *columnname*).
- The LIMIT TO *nn* ROWS clause used to limit the number of rows returned by a query. You can use only the **WHERE** clause to limit the scope of a query.

Equivalent ANSI SQL Data Types

The following table lists ANSI SQL data types and the equivalent PSQL Jet database engine SQL data types and their valid synonyms.

ANSI SQL data type	PSQL Jet SQL data type	Synonym
BIT, BIT VARYING	BINARY (See Notes)	VARBINARY
Not supported	BIT (See Notes)	BOOLEAN, LOGICAL, LOGICAL1, YESNO
Not supported	BYTE	INTEGER1
Not supported	COUNTER	AUTOINCREMENT
Not supported	CURRENCY	MONEY
DATE, TIME, TIMESTAMP	DATETIME	DATE, TIME, TIMESTAMP
Not supported	GUID	
DECIMAL	Not supported	
REAL	SINGLE	FLOAT4, IEEESINGLE, REAL
DOUBLE PRECISION, FLOAT	DOUBLE	FLOAT, FLOAT8, IEEEDOUBLE, NUMBER, NUMERIC
SMALLINT	SHORT	INTEGER2, SMALLINT
INTEGER	LONG	INT, INTEGER, INTEGER4
INTERVAL	Not supported	
Not supported	LONGBINARY	GENERAL, OLEOBJECT
Not supported	LONGTEXT	LONGCHAR, MEMO, NOTE
CHARACTER, CHARACTER VARYING	TEXT	ALPHANUMERIC, CHAR, CHARACTER, STRING, VARCHAR
Not supported	VALUE (See Notes)	

Notes

- The ANSI SQL BIT data type doesn't correspond to the PSQL Jet SQL BIT data type, but it corresponds to the BINARY data type instead. There is no ANSI SQL equivalent for the PSQL Jet SQL BIT data type.
- The VALUE reserved word doesn't represent a data type defined by the PSQL Jet database engine.

In Operator

Determines whether the value of an expression is equal to any of several values in a specified list.

Syntax

expr [Not] In(value1, value2, . . .)

Example

Query that includes all orders shipped to Lancashire and Essex and the dates shipped

```
SELECT CustomerID, ShippedDate FROM Orders
WHERE ShipRegion In ('Lancashire','Essex');
```

The In operator syntax has these parts:

Part	Description
expr	Expression identifying the field that contains the data you want to evaluate.
value1, *value2*	Expression or list of expressions against which you want to evaluate *expr*.

Remarks

If *expr* is found in the list of values, the **In** operator returns **True**; otherwise, it returns **False**. You can include the **Not** logical operator to evaluate the opposite condition (that is, whether *expr* is not in the list of values).

For example, you can use **In** to determine which orders are shipped to a set of specified regions:

```
SELECT *
FROM Orders
WHERE ShipRegion In ('Avon','Glos','Som')
```

Like Operator

Compares a string expression to a pattern in an SQL expression.

Syntax

expression Like "pattern"

Example

Returns a list of employees whose names begin with the letters A through D.

```
SELECT LastName, FirstName
FROM Employees
WHERE LastName Like '[A-D]*';
```

The Like operator syntax has these parts:

Part	Description
expression	SQL expression used in a WHERE clause.
pattern	String or character string literal against which *expression* is compared.

Remarks

You can use the **Like** operator to find values in a field that match the pattern you specify. For *pattern*, you can specify the complete value (for example, Like "Smith"), or you can use wildcard characters to find a range of values (for example, Like "Sm*").

In an expression, you can use the **Like** operator to compare a field value to a string expression. For example, if you enter Like "C*" in an SQL query, the query returns all field values beginning with the letter C. In a parameter query, you can prompt the user for a pattern to search for.

The following example returns data that begins with the letter P followed by any letter between A and F and three digits:

```
Like "P[A-F]###"
```

The following table shows how you can use **Like** to test expressions for different patterns.

Kind of match	Pattern	Match (returns True)	No match (returns False)
Multiple characters	a*a	aa, aBa, aBBBa	aBC
	ab	abc, AABB, Xab	aZb, bac
Special character	a[*]a	a*a	aaa
Multiple characters	ab*	abcdefg, abc	cab, aab
Single character	a?a	aaa, a3a, aBa	aBBBa
Single digit	a#a	a0a, a1a, a2a	aaa, a10a
Range of characters	[a-z]	f, p, j	2, &
Outside a range	[!a-z]	9, &, %	b, a
Not a digit	[!0-9]	A, a, &, ~	0, 1, 9
Combined	a[!b-m]#	An9, az0, a99	abc, aj0

PSQL Jet Database Engine SQL Data Types

The PSQL Jet database engine SQL data types consist of 13 primary data types defined by the PSQL Jet database engine and several valid synonyms recognized for these data types.

The following table lists the primary data types. The synonyms are identified in PSQL Jet Database Engine SQL Reserved Words.

Data type	Storage size	Description
BINARY	1 byte per character	Any type of data may be stored in a field of this type. No translation of the data (for example, to text) is made. How the data is input in a binary field dictates how it will appear as output.
BIT	1 byte	Yes and No values and fields that contain only one of two values.
BYTE	1 byte	An integer value between 0 and 255.
COUNTER	4 bytes	A number automatically incremented by the PSQL Jet database engine whenever a new record is added to a table. In the PSQL Jet database engine, the data type for this value is **Long**.
CURRENCY	8 bytes	A scaled integer between – 922,337,203,685,477.5808 and 922,337,203,685,477.5807.
DATETIME (See DOUBLE)	8 bytes	A date or time value between the years 100 and 9999.
GUID	128 bits	A unique identification number used with remote procedure calls.
SINGLE	4 bytes	A single-precision floating-point value with a range of – 3.402823E38 to – 1.401298E-45 for negative values, 1.401298E-45 to 3.402823E38 for positive values, and 0.
DOUBLE	8 bytes	A double-precision floating-point value with a range of – 1.79769313486232E308 to – 4.94065645841247E-324 for negative values, 4.94065645841247E-324 to 1.79769313486232E308 for positive values, and 0.
SHORT	2 bytes	A short integer between – 32,768 and 32,767.
LONG	4 bytes	A long integer between – 2,147,483,648 and 2,147,483,647.
LONGTEXT	1 byte per character	Zero to a maximum of 1.2 gigabytes.
LONGBINARY	As required	Zero to a maximum of 1.2 gigabytes. Used for OLE objects.
TEXT	1 byte per character	Zero to 255 characters.

Note You can also use the VALUE reserved word in SQL statements.

PSQL Jet Database Engine SQL Reserved Words

The following list includes all words reserved by the PSQL Jet database engine for use in SQL statements. The words in the list that aren't in all uppercase letters are also reserved by other applications. Consequently, the individual Help topics for these words provide general descriptions that don't focus on SQL usage.

Note Words followed by an asterisk (*) are reserved but currently have no meaning in the context of a PSQL Jet SQL statement (for example, **Level** and **TableID**).

A

ADD	ANY
ALL	AS
Alphanumeric ¾ See TEXT	ASC
ALTER	AUTOINCREMENT ¾ See COUNTER
And	**Avg**

B-C

Between	COLUMN
BINARY	CONSTRAINT
BIT	**Count**
BOOLEAN ¾ See BIT	COUNTER
BY	CREATE
BYTE	CURRENCY
CHAR, CHARACTER ¾ See TEXT	

D

DATABASE	DISALLOW
DATE ¾ See DATETIME	DISTINCT
DATETIME	DISTINCTROW
DELETE	DOUBLE
DESC	DROP

E-H

Eqv	FROM
EXISTS	GENERAL ¾ See LONGBINARY
FLOAT, FLOAT8 ¾ See DOUBLE	GROUP
FLOAT4 ¾ See SINGLE	GUID
FOREIGN	HAVING

I

IEEEDOUBLE ¾ See DOUBLE	INNER
IEEESINGLE ¾ See SINGLE	INSERT
IGNORE	INT, INTEGER, INTEGER4 ¾ See LONG

Imp
In
IN
INDEX
INTEGER1 ¾ See BYTE
INTEGER2 ¾ See SHORT
INTO
Is

J-M

JOIN
KEY
LEFT
Level*
Like
LOGICAL, LOGICAL1 ¾ See BIT
LONG
LONGBINARY
LONGTEXT
Max
MEMO ¾ See LONGTEXT
Min
Mod
MONEY ¾ See CURRENCY

N-P

Not
NULL
NUMBER ¾ See DOUBLE
NUMERIC ¾ See DOUBLE
OLEOBJECT ¾ See LONGBINARY
ON
OPTION
Or
ORDER
Outer*
OWNERACCESS
PARAMETERS
PERCENT
PIVOT
PRIMARY
PROCEDURE

Q-S

REAL ¾ See SINGLE
REFERENCES
RIGHT
SELECT
SET
SHORT
SINGLE
SMALLINT ¾ See SHORT
SOME
StDev
StDevP
STRING ¾ See TEXT
Sum

T-Z

TABLE
TableID*
TEXT
TIME ¾ See DATETIME
TIMESTAMP ¾ See DATETIME
TOP
TRANSFORM
UNION
VALUE
VALUES
Var
VARBINARY ¾ See BINARY
VARCHAR ¾ See TEXT
VarP
WHERE
WITH

SQL Expressions

An SQL expression is a string that makes up all or part of an SQL statement. For example, the **FindFirst** method on a **Recordset** object uses an SQL expression consisting of the selection criteria found in an **SQL WHERE** clause.

The PSQL Jet database engine uses the Visual Basic for Applications (or VBA) expression service to perform simple arithmetic and function evaluation. All of the operators used in PSQL Jet SQL expressions (except **Between**, **In**, and **Like**) are defined by the VBA expression service. In addition, the VBA expression service offers over 100 VBA functions that you can use in SQL expressions. For example, you can use these VBA functions to compose an SQL query in the Microsoft Access query Design view, and you can also use these functions in an SQL query in the DAO **OpenRecordset** method in Microsoft Visual C++, Microsoft Visual Basic, and Microsoft Excel code.

Using Wildcard Characters in String Comparisons

Built-in pattern matching provides a versatile tool for making string comparisons. The following table shows the wildcard characters you can use with the **Like** operator and the number of digits or strings they match.

Character(s) in *pattern*	**Matches in** *expression*
?	Any single character
*	Zero or more characters
#	Any single digit (0 – 9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

You can use a group of one or more characters (*charlist*) enclosed in brackets ([ ]) to match any single character in *expression,* and *charlist* can include almost any characters in the ANSI character set, including digits. In fact, you can use the special characters opening bracket ([ ), question mark (?), number sign (#), and asterisk (*) to match themselves directly only if enclosed in brackets. You can't use the closing bracket ( ]) within a group to match itself, but you can use it outside a group as an individual character.

In addition to a simple list of characters enclosed in brackets, *charlist* can specify a range of characters by using a hyphen (-) to separate the upper and lower bounds of the range. For example, using [A-Z] in *pattern* results in a match if the corresponding character position in *expression* contains any of the uppercase letters in the range A through Z. You can include multiple ranges within the brackets without delimiting the ranges. For example, [a-zA-Z0-9] matches any alphanumeric character.

Other important rules for pattern matching include the following:

- An exclamation mark (!) at the beginning of *charlist* means that a match is made if any character except those in *charlist* are found in *expression*. When used outside brackets, the exclamation mark matches itself.
- You can use the hyphen (-) either at the beginning (after an exclamation mark if one is used) or at the end of *charlist* to match itself. In any other location, the hyphen identifies a range of ANSI characters.
- When you specify a range of characters, the characters must appear in ascending sort order (A-Z or 0-100). [A-Z] is a valid pattern, but [Z-A] isn't.
- The character sequence [ ] is ignored; it's considered to be a zero-length string ("").