

# **Introduction to SQL**

Prepared for the RMAIR Conference SQL Workshop  
October 2011  
Albuquerque, New Mexico

Mike Ellison  
Manager, Data Systems and Application Development  
UNLV Office of Institutional Analysis & Planning

# Contents

---

<b>RETRIEVING RECORDS .....</b>	<b>3</b>
Specifying Columns .....	4
Selecting a Filtered Set of Records .....	5
The Special <i>NULL</i> Value.....	8
Applying Multiple Conditions when Filtering .....	10
Computing a Column by Calculating Values .....	12
Computing a Column based on Conditions .....	14
Specifying the Sort Order for a Result Set .....	16
Summarizing Values.....	18
Summarizing Values in Groups .....	19
<b>MODIFYING RECORDS .....</b>	<b>21</b>
Adding Records to a Table .....	22
Adding Records Selected From another Table .....	23
Modifying Data in Records.....	24
Removing Records from a Table .....	25
<b>WORKING WITH MULTIPLE TABLES .....</b>	<b>26</b>
Joining Multiple Tables in a Select Statement .....	27
Joining Multiple Tables using INNER JOIN .....	29
Joining Multiple Tables using Outer Joins.....	31
UNION queries .....	33
Sidebar – Can a Cartesian product be useful? .....	34
<b>DEFINING DATABASE OBJECTS.....</b>	<b>36</b>
Creating Tables .....	37
Creating Views .....	39
Deleting Objects.....	41
Creating and Executing Stored Procedures .....	42
<b>KEYWORD INDEX.....</b>	<b>44</b>

# Retrieving Records

---

In this section:

- Specifying Columns
- Selecting a Filtered Set of Records
- Applying Multiple Conditions when Filtering
- Computing a Column by Calculating Values
- Computing a Column based on Conditions
- Specifying the Sort Order for a Result Set
- Summarizing Values
- Summarizing Values in Groups

## Specifying Columns

Keywords: **SELECT, FROM**

Statement Syntax:

```
Select ListOfColumns
From TableName
```

Records are retrieved from a table using a Select statement with a From clause. *ListOfColumns* is one or more column names, separated with commas. *TableName* is the name of a table containing those fields.

Example:

```
Select FirstName, LastName, Standing
From Students
```

FirstName	LastName	Standing
Angie	VanArt	Soph
Bob	Schwartski	Soph
Jakie	Klunk	Sr
Jenny	Price	Soph
Jim	McInnes	Fresh

Notes:

- An asterisk (\*) may be used in place of a column list in the Select statement to indicate all fields. For example, the statement:

```
Select * From Spring_2001_FTE
```

selects all records and displays all fields from the Spring\_2001\_FTE table.

## Selecting a Filtered Set of Records

Keywords: **WHERE**

Statement Syntax:

```
Select ListOfColumns
  From TableName
  Where Criteria
```

*Criteria* is supplied in the Where clause of a select statement to filter the records returned. The criteria are specified as one or more *conditions* – if records match the given conditions, they are returned in the result set. If records do not match the conditions, they are excluded from the result set.

Conditions often follow this three-item pattern:

```
ColumnName Operator Value
```

*ColumnName* represents the column you wish to test in the condition. *Operator* represents the type of test to perform in the condition, such as equality, inequality, or greater than/less than. *Value* is the literal value, additional column, or other expression against which to test.

Here are some examples of conditions which follow this pattern:

Standing = 'Soph'	Matches records where the <i>Standing</i> column is equal to the literal value 'Soph' (without the quotes). Single quotes are used to surround literal text values in SQL. Without them, the SQL server would think you meant a column named <i>Soph</i> rather than the actual text value 'Soph'.
Age > 24	Matches records where the <i>Age</i> column is greater than the literal value 24.
Credits <> Fulltime + 2	Matches records where the <i>Credits</i> column does not equal the value of the expression <i>Fulltime + 2</i> .

Other operators like *IN* and *BETWEEN* follow a slightly different syntax:

Age BETWEEN 18 AND 21	Matches records where the <i>Age</i> column is between 18 and 21, inclusive.
Age IN (20, 25, 30)	Matches records where the <i>Age</i> column is any one of the following literal values: 20, 25, or 30.
Standing IN ('Fresh', 'Soph')	Matches records where the <i>Standing</i> column is any one of the following literal text values: 'Fresh' or 'Soph'.

Example:

```
Select FirstName, MI, LastName, Standing
From Students
Where Standing = 'Soph'
```

FirstName	MI	LastName	Standing
Angie	W	VanArt	Soph
Bob	I	Schwartski	Soph
Jenny	S	Price	Soph

Notes:

- The following summarizes conditional operators which may be used for character, date, and numeric comparisons when filtering:

Equal to	=
Not equal to	<> <i>(for some RDBMS's, != also represents inequality)</i>
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Among a list of choices	IN ( <i>choice1, choice2, choice3, ...</i> )
Within a range (inclusive)	BETWEEN <i>value1</i> AND <i>value2</i>
Negation	NOT <i>(NOT is usually used before IN or BETWEEN)</i>
Is a null (unknown) value	IS NULL
Is not a null value	IS NOT NULL

- For character comparisons, pattern matching may be performed with the LIKE operator and the wildcards % (percent sign) and \_ (underscore). The % wildcard matches zero, one, or more characters; the \_ wildcard matches one and only one character. For example, this SQL statement:

```
Select LastName  
From Students  
Where LastName Like 'J_N%'
```

matches records with the last names 'JAN', 'JEN', and 'JONSEN', but not 'JOHNSON'.

- Character text and date literal values should be surrounded by single quotes, such as 'JOHNSON' or '4-JUL-2010'. The single quotes here are referred to as *delimiters*. Numeric literal values, such as 14, -3, or 3.14 should not be surrounded by single quotes or any other delimiter.

## The Special *NULL* Value

Keywords: **NULL, WHERE**

SQL uses the special value *NULL* as a placeholder for missing or unknown information. If a record is missing information in a given column, *NULL* typically displays in the result set for that column instead of a value.

*NULL* is *not* the same as the number zero, or a blank string (two single quotes, with no characters in between). The value *zero* is a known value. A blank or empty text string is also a known value – a string known to be zero characters long. *NULL* implies the *absence of a known value*. We simply don't know what the value is.

Imagine a friend holds up her hand and asks, "How many fingers am I holding up?" With all fingers extended, you respond, "five fingers" – the known value is 5. Your friend then makes a fist and asks, "How many fingers am I holding up now?" You say "Zero fingers" – the known value is zero.

Then your friend holds her hand behind her back so you can't see it. She asks "Now how many fingers am I holding up?" You say "I have no idea!" That is what is meant by *NULL*. The value is simply *unknown*.

Any comparison that involves a *NULL* value results in a *NULL* value. Is *five* equal to *NULL*? Nope. Is 'WORLEY' *not* equal to *NULL*? Nope. Is 42.34 greater than *NULL*? Nope. Not even *NULL* equals *NULL*! Any attempt to make a *true-or-false* comparison involving an unknown value results in neither *true* nor *false*. The result itself is unknown – *NULL*.

In a *WHERE* clause, because the normal operators for testing equality and inequality simply don't apply to *NULL*, we don't use *field = NULL* or *field <> NULL* syntax. Instead, for cases where we want to query records that have or don't have *NULL* (unknown) values, we use the special syntax *IS NULL* or *IS NOT NULL*.

For example, to view records with no known value in the *Standing* field:

```
SELECT FirstName, MI, LastName, Standing
   FROM Students
  WHERE Standing IS NULL
```

And to see all records that do have some known value in the *Standing* field:

```
SELECT FirstName, MI, LastName, Standing
   FROM Students
  WHERE Standing IS NOT NULL
```

The following would *never* retrieve records and should be avoided, because *nothing ever equals NULL*:

```
SELECT FirstName, MI, LastName, Standing
   FROM Students
  WHERE Standing = NULL
```

For that matter, nothing ever *doesn't* equal *NULL*, and nothing is ever greater than, or less than *NULL* either. Any attempt to make comparisons specifically involving the *NULL* value should use the keywords *IS NULL* or *IS NOT NULL*.

## Applying Multiple Conditions when Filtering

Keywords: **AND, OR**

Statement Syntax:

```
Select ListOfColumns
      From TableName
      Where FirstCondition And SecondCondition
```

```
Select ListOfColumns
      From TableName
      Where FirstCondition Or SecondCondition
```

Multiple conditions in a Where clause are separated with keywords AND and OR. An AND between two conditions requires that both conditions be met for a record to be returned. An OR between two conditions allows for either condition to be met for a record to be returned.

Examples:

```
Select FirstName, MI, LastName, Standing
      From Students
      Where Standing = 'Soph'
      And FirstName Like 'M%'
```

FirstName	MI	LastName	Standing
Mike	A	Diltz	Soph

```
Select FirstName, LastName, Standing
      From Students
      Where Standing = 'Soph'
      Or FirstName Like 'M%'
```

FirstName	MI	LastName	Standing
Angie	W	VanArt	Soph
Bob	I	Schwartski	Soph
Jenny	S	Price	Soph
Justin	E	Time	Soph
Karen		Hoy	Soph
Monica	E	McFarland	Sr

## Notes:

- More than one AND and/or OR operator may be used in a Where clause to refine filtering conditions.
- AND operations are given precedence over OR operations (*i.e.* conditions separated with AND are evaluated first, followed by conditions separated with OR).
- Parentheses are used to change the precedence in which AND or OR operators are performed. Conditional statements inside innermost parentheses are evaluated first.

For example, take the following statement:

```
Select FirstName, MI, LastName, Standing
   From Students
   Where Standing = 'Soph'
      And (FirstName Like 'A%' Or FirstName Like 'B%')
```

Because the two conditions separated by OR are enclosed in parentheses, they are evaluated first. The result of that comparison is then matched against the first condition with the AND operator. This statement therefore returns any sophomore student with a first name beginning with either an 'A' or a 'B'. Without the parentheses --

```
Select FirstName, MI, LastName, Standing
   From Students
   Where Standing = 'Soph'
      And FirstName Like 'A%' Or FirstName Like 'B%'
```

the two conditions separated by AND (Standing = 'Soph' And FirstName like 'A%') would be evaluated first. This would then return all sophomores with a first name starting with 'A' *plus* all students with a first name beginning with 'B', *regardless of their standing*.

Appropriate use of parentheses can make for more precise condition clauses as well as cleaner syntax.

## Computing a Column by Calculating Values

Keywords: **AS**

Statement Syntax:

```
Select Calculation AS columnName  
From TableName
```

The column list of a Select statement may include calculations, using literal values or other fields as operands. The AS keyword defines the column name for the calculation in the result set. The AS keyword may also be used to create an alias column name for an existing field.

Example:

```
Select LastName + ', ' + FirstName As FullName,  
Credits,  
Credits / 2 as SemAvg  
From Students
```

FullName	Credits	SemAvg
VanArt, Angie	48.0	24.0
Schwartski, Bob	45.5	22.75
Klunk, Jakie	109.0	54.5

Notes:

- The following operators are commonly used for calculations:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Character string Concatenation	<i>Varies depending on the RDBMS, such as:</i>    , + , concat(string1, string2)

Character string concatenation symbols vary depending on the RDBMS vendor. The above example using the + symbol works with Microsoft SQL Server. The same statement for Oracle would use || (two vertical bars) as the concatenation symbol, as so:

```
Select LastName || ', ' || FirstName As FullName,  
       Credits,  
       Credits / 2 as SemAvg  
From Students
```

## Computing a Column based on Conditions

Keywords: **CASE, WHEN, THEN, ELSE, END, AS**

Statement Syntax:

```
Select CASE
    WHEN condition1 THEN value1
    WHEN condition2 THEN value2
    WHEN condition3 THEN value3
    ...
    WHEN conditionN THEN valueN
    ELSE valueIfNoConditionsApply

    END AS columnTitle

From TableName
```

A *CASE...END* block may be used to assign values for a result set column based on one or more sets of conditions. Condition sets are expressed in the same way as they are for a WHERE clause and are evaluated in the order presented in the CASE statement.

Each condition set in a *CASE...END* block begins with the keyword WHEN, followed by the set of one or more conditions to attempt to match, followed by the keyword THEN and the value to return if that condition set matches. If none of the WHEN condition sets match, the server returns for the column the value associated with the optional ELSE keyword. If ELSE is not used and no WHEN condition sets apply, the *null* value is returned.

Example:

The following example uses a *CASE...END* block to derive a *StudentStatus* result column, showing whether a student is Full-time, Half-time, or Quarter-time, depending on the number of Credits taken.

```
Select StudentID, Credits,
    CASE WHEN Credits >=12 THEN 'Full-time'
         WHEN Credits >=6 THEN 'Half-time'
         WHEN Credits >=3 THEN 'Quarter-time'
         ELSE 'Less than 3 Credits'
    END AS StudentStatus
From Students
```

StudentID	Credits	StudentStatus
S00001	12	Full-time
S00002	12	Full-time
...	...	...
S00006	9	Half-time
...	...	...
S00026	3	Quarter-time
...	...	...

Notes:

- Multiple conditions may be used in a WHEN clause, using AND and OR keywords in the same way as they're used in a WHERE clause.
- Use the AS keyword after a *CASE...END* block to provide a title for the derived column.
- WHEN clauses are evaluated in the order in which they appear in the *CASE...END* block. The THEN value is returned for the first WHEN condition set that matches.
- The ELSE clause is optional. If it is omitted, and none of the listed WHEN condition sets match, then the *null* value is returned for the *CASE...END* block.
- THEN values may be either literal values such as *42* or *'Full-time'*, or expressions such as *Credits + 2*.
- A common mistake when using a *CASE...END* block is to forget to include the closing END keyword.

## Specifying the Sort Order for a Result Set

Keywords: **ORDER BY, ASC, DESC**

Statement Syntax:

```
Select ListOfColumns
  From TableName
  [Where Criteria]
Order By SortingColumnList
```

An Order By clause in a Select statement may be used to specify the sort order for resulting records. *SortingColumnList* is either one column name, or multiple column names separated with commas. An Order By clause may contain the keywords ASC or DESC following a field name to specify an ascending or descending sort, with ascending the default if neither is specified.

Examples:

```
Select FirstName, MI, LastName, Standing
  From Students
Order By LastName
```

FirstName	MI	LastName	Standing
Tracey	I	Ackley	Jr
William	NULL	Altermatt	Sr
Sue	L	Ashleman	Sr
Kenny	O	Banya	Sr

```
Select LastName, MI, FirstName, Standing, GPA
  From Students
Order By GPA ASC, LastName DESC
```

LastName	MI	FirstName	Standing	GPA
Mensch	W	Matt	Sr	1.35
Kauffman	E	Kyle	Jr	1.36
Kramer	N	Cosmo	Jr	2.20
McFarland	E	Monica	Sr	2.30

Notes:

- The Order By clause appears after the Where clause in a Select statement that contains both.
- In some RDBMS systems, one may use numbers rather than column names in the Order By clause. The numbers refer to the indexes of columns from the SELECT clause, typically with 1 representing the first column, 2 the second, etc. For example, assuming *LastName* is the third column in the result set, one may specify *ORDER BY 3* to sort by the *LastName* column:

```
Select FirstName, MI, LastName, Standing
      From Students
Order By 3
```

FirstName	MI	LastName	Standing
Tracey	I	Ackley	Jr
William	NULL	Altermatt	Sr
Sue	L	Ashleman	Sr
Kenny	O	Banya	Sr

## Summarizing Values

Keywords: **SELECT, AS**

Statement Syntax:

```
Select function(fieldName) As columnTitle ...
      From TableName
      [Where Criteria]
```

A function can be applied against a field to summarize data for that field across all records in a table, or a subset of records filtered with a Where clause.

Example:

```
Select Count(*)      As StudentCount,
       Sum(Credits)  As TotalCreds,
       Avg(Credits)  As AvgCreds,
       Max(Credits)  As HighestCreds
From Students
```

StudentCount	TotalCreds	AvgCreds	HighestCreds
82	4815.60	59.45185	130.0

Notes:

- The following functions are commonly used for summarizing data:

Add values	Sum( <i>columnToAdd</i> )
Average values	Avg( <i>columnToAverage</i> )
Highest value	Max( <i>columnToAggregate</i> )
Lowest value	Min( <i>columnToAggregate</i> )
Count number of records	Count(*) <i>(A single asterisk between the parentheses. SQL coders might also refer to this as 'Count Star')</i>
Count records containing a non- <i>null</i> value in a given column	Count( <i>column</i> )
Count the number of distinct, non- <i>null</i> values in a given column	Count(DISTINCT <i>column</i> )

## Summarizing Values in Groups

Keywords: **GROUP BY**

Statement Syntax:

```
Select GroupFieldList, function(fieldName)...  
    From TableName  
    [Where Criteria]  
    Group By GroupFieldList  
    [Order By fieldList]
```

In addition to summarizing data throughout a table, functions may be used to summarize data in groups. *GroupFieldList* is one or more field names used to define the groupings.

Example:

```
Select Gender,  
       Count(*) As StudentCount  
    From Students  
    Group By Gender
```

Gender	StudentCount
F	41
M	40

```
Select Standing,  
       Gender,  
       Count(*) As StudentCount  
    From Students  
    Group By Standing, Gender
```

Standing	Gender	StudentCount
Fresh	F	11
Jr	F	6
Soph	F	15
Sr	F	9
Fresh	M	11
Jr	M	11
Soph	M	9
Sr	M	9

Notes:

- The Order By clause appears after the Group By clause in Select statements that use both.

# Modifying Records

---

In this section:

- Adding Records to a Table
- Adding Records Selected From Another Table
- Modifying Data in Records
- Removing Records from a Table

## Adding Records to a Table

Keywords: **INSERT INTO, VALUES**

Statement Syntax:

```
Insert Into TableName
           ( ColumnList )
  Values
           ( ValuesList )
```

An Insert statement is used to add new records to a table. *ColumnList* is a list of field names from *TableName*, separated by commas. *ValuesList* are the literal values to add, in the same order as *ColumnList*.

Example:

```
Insert Into Students
           (FirstName, LastName, Standing, GPA)
  Values
           ('Abraham', 'Lincoln', 'Sr', 3.95)
```

FirstName	LastName	Standing	GPA
Abraham	Lincoln	Sr	3.95

Notes:

- *ColumnList* may be omitted. If so, the items in *ValuesList* must match field for field all fields in *TableName*, in the appropriate order.
- In some RDBMS's, the keyword INTO may be omitted.

## Adding Records Selected From another Table

Keywords: **INSERT INTO, SELECT**

Statement Syntax:

```
Insert Into TableName
           ( ColumnList )
           SelectStatement
```

Instead of using a Values clause, an Insert statement may specify that a selected set of records from another table be added to *TableName*. *SelectStatement* is any valid Select statement that would normally produce a set of records.

Example:

```
Insert Into Students
           (FirstName, LastName, Standing, Major, ST)
           Select FName, Lname, Standing, Major, ST
           From Transfers
```

Notes:

- If *ColumnList* is omitted, the columns returned by *SelectStatement* must match, field for field, each column in *TableName*.

## Modifying Data in Records

Keywords: **UPDATE, SET**

Statement Syntax:

```
Update TableName
  Set Field1 = Value1,
     Field2 = Value2,
     ...
     FieldN = ValueN
[Where Criteria]
```

An Update statement is used to modify the data in one or more records of a table. If a Where clause is used, only those records which satisfy the supplied *Criteria* are modified. If a Where clause is not used, all records in the table are modified. One or more fields in *TableName* may be specified for modification following the SET keyword. Each *Value* supplied may be a literal or calculated value.

Example:

```
Update Students
  Set Gender = 'M',
     Credits = Credits + 12
  Where StudentID = 10081
```

*Before*

ID	FirstName	LastName	Standing	Gender	Credits
10081	Loyd	Bron	Soph	<null>	25

*After*

ID	FirstName	LastName	Standing	Gender	Credits
10081	Loyd	Bron	Soph	M	37

Notes:

- Be very careful when constructing and executing Update statements. Make sure the Where clause correctly filters only those records you wish to modify. You may wish to create a Select statement first to ensure the Where clause is correct.

## Removing Records from a Table

Keywords: **DELETE**

Statement Syntax:

```
Delete
  From TableName
  [Where Criteria]
```

A Delete statement removes records from a table. If a Where clause is used, only those records which satisfy the supplied *Criteria* are deleted. If a Where clause is not used, all records in the table are deleted.

Example:

```
Delete
  From Students
  Where StudentID = 10081
```

*Before*

ID	FirstName	LastName	Standing	Gender	Credits
10079	Tony	Hess	Sr	M	110
10080	Yancci	Kinkade	Fresh	M	15
10081	Loyd	Bron	Soph	M	25

*After*

ID	FirstName	LastName	Standing	Gender	Credits
10079	Tony	Hess	Sr	M	110
10080	Yancci	Kinkade	Fresh	M	15

Notes:

- As with an Update statement, be very careful when constructing and executing Delete statements. You may wish to create a Select statement first to ensure the Where clause is correct.

# Working with Multiple Tables

---

In this section:

- Joining Multiple Tables in a Select Statement
- Joining Multiple Tables using INNER JOIN
- Joining Multiple Tables using Outer Joins
- UNION queries

## Joining Multiple Tables in a Select Statement

Keywords: **SELECT**

Statement Syntax:

```
Select Table1.fieldList,  
       Table2.fieldList  
From Table1, Table2  
Where Table1.KeyField = Table2.KeyField  
[And Table1.KeyField2 = Table2.KeyField2  
And . . . ]
```

Two (or more) tables may be specified in the From clause of a Select statement. If so, the two tables should be *joined* by a field or fields that they both have in common (referred to as *key fields*). A join may be specified as a criteria condition in a Where clause, with the key field(s) from the first table set equal to the field(s) in common from the second.

Example:

*Students Table*

FirstName	LastName	Gender	Major
Tracey	Ackley	F	BIO
William	Altermatt	M	ENG
Sue	Ashleman	F	ENG
Kenny	Banya	M	HIS

*Majors Table*

MajorCode	MajorDesc
BIO	Biology
COM	Communications
EDU	Education
ENG	English

```
Select FirstName, LastName, Gender, Major,  
       MajorDesc  
From Students, Majors  
Where Students.Major = Majors.MajorCode  
Order By LastName
```

FirstName	LastName	Gender	Major	MajorDesc
Tracey	Ackley	F	BIO	Biology
William	Altermatt	M	ENG	English
Sue	Ashleman	F	ENG	English

Notes:

- This type of join is often referred to as an *equi-join*, or *inner join*. Only those records in which the join field(s) match in both tables are returned. In the above example, Student “Kenny Banya” has major code “HIS”. This major code does not appear in the Majors table, so the record is not returned in the results.
- If a Where clause is not used and multiple tables are listed in the From clause, the resulting set of selected records will form a *Cartesian product*. Since the query processor will have no way of knowing how the tables should be joined, each record in the first will be matched to *every* record in the second. In such a Select statement, if the first table has four records and the second three, 12 records (four times three) will be retrieved in the result set – one for every possible combination. Typically this is not desired!
- When specifying column names in Select statements with multiple tables, a table prefix is required if using just the field name would be ambiguous. For example, if *Students* and *Majors* each had a field called *Code*, the following Select statement would be ambiguous, as the query processor would not know which *Code* field to use:

```
Select Code
  From Students, Majors
 Where . . .
```

To remove the ambiguity, the field should be specified with the name of its table as a prefix:

```
Select Majors.Code
  From Students, Majors
 Where . . .
```

The keyword AS may also be used to create an alias name for a table. The following example is functionally equivalent to the preceding one:

```
Select m.Code
  From Students as s, Majors as m
 Where . . .
```

## Joining Multiple Tables using INNER JOIN

Keywords: **INNER JOIN**

Statement Syntax:

```
Select Table1.fieldList,  
       Table2.fieldList  
From Table1  
Inner Join Table2  
On Table1.KeyField = Table2.KeyField
```

The Inner Join keyword is another way to specify an *equi-join* or *inner join* between two tables.

Example:

*Students Table*

FirstName	LastName	Gender	Major
Tracey	Ackley	F	BIO
William	Altermatt	M	ENG
Sue	Ashleman	F	ENG
Kenny	Banya	M	HIS

*Majors Table*

MajorCode	MajorDesc
BIO	Biology
COM	Communications
EDU	Education
ENG	English

```
Select FirstName, LastName, Gender, Major,  
       MajorDesc  
From Students  
Inner Join Majors  
On Students.Major = Majors.MajorCode  
Order By LastName
```

FirstName	LastName	Gender	Major	MajorDesc
Tracey	Ackley	F	BIO	Biology
William	Altermatt	M	ENG	English
Sue	Ashleman	F	ENG	English

Notes:

- This method of defining an inner join produces the same result as the previous Where clause syntax. Only those records in which the join field(s) match in both tables are returned.
- Join syntax varies among different RDBMS vendors. Consult the documentation for your specific implementation.

## Joining Multiple Tables using Outer Joins

Keywords: **LEFT (OUTER) JOIN, RIGHT (OUTER) JOIN**

Statement Syntax:

```
Select Table1.fieldList,  
       Table2.fieldList  
From Table1  
Left Join Table2  
On Table1.KeyField = Table2.KeyField
```

Whereas an inner join returns only those records in both tables with matching fields, an *outer* join returns all matching records plus those from one side that do not match. *Null values* (empty values) are then returned in the selected fields for the other.

In a *left* outer join, all records from the left side of the key field match are returned with nulls for selected fields from the right table. In a *right* outer join, all records from the right side of the key field match are returned with nulls for selected fields from the left table.

Example:

*Students Table*

FirstName	LastName	Gender	Major
Tracey	Ackley	F	BIO
William	Altermatt	M	ENG
Sue	Ashleman	F	ENG
Kenny	Banya	M	HIS

*Majors Table*

MajorCode	MajorDesc
BIO	Biology
COM	Communications
EDU	Education
ENG	English

```
Select FirstName, LastName, Gender, Major,  
       MajorDesc  
From Students  
Left Join Majors  
On Students.Major = Majors.MajorCode  
Order By LastName
```

FirstName	LastName	Gender	Major	MajorDesc
Tracey	Ackley	F	BIO	Biology
William	Altermatt	M	ENG	English
Sue	Ashleman	F	ENG	English
Kenny	Banya	M	HIS	<null>

```

Select FirstName, LastName, Gender, Major,
      MajorDesc
From Students
Right Join Majors
      On Students.Major = Majors.MajorCode
Order By LastName

```

FirstName	LastName	Gender	Major	MajorDesc
<null>	<null>	<null>	<null>	Communcations
<null>	<null>	<null>	<null>	Education
Tracey	Ackley	F	BIO	Biology
William	Altermatt	M	ENG	English
Sue	Ashleman	F	ENG	English

Notes:

- Outer join syntax varies among different RDBMS vendors. Some use the keyword OUTER along with LEFT or RIGHT. Some support FULL outer joins, which act as a combination of left and right. Consult the documentation for your specific RDBMS.

## UNION queries

Keywords: **UNION**

Statement Syntax:

```
SelectStatement1
  UNION
SelectStatement2
```

Separate result sets from different Select statements can be combined with the *UNION* keyword into a single result set.

Example:

*FTE\_Fall2001*

StudentID	FTE
10000	3.20
10001	3.03
10002	7.27
10003	3.00
...	...

*FTE\_Spring2001*

StudentID	FTE
10000	3.01
10001	2.81
10002	6.83
10003	2.77
...	...

```
Select 'Fall 2001' as Term, Sum(FTE) as TotalFTE
  From FTE_Fall2001
  UNION
Select 'Spring 2001' as Term, Sum(FTE) as TotalFTE
  From FTE_Spring2001
```

Term	TotalFTE
Fall 2001	322.70
Spring 2001	322.58

## Sidebar – Can a Cartesian product be useful?

When executing a query, if a Cartesian product results, we've usually made a mistake – we probably have an incorrect JOIN statement, or even more likely, we're missing a JOIN. Remember that if two tables appear in a query's FROM clause without the right JOIN, the query processor has no choice but to match each and every record from the first table to each and every record from the second. Typically, such Cartesian products are not desired. Is there ever an occasion however when one could be useful?

Take the following distribution report as an example:

GradeRange	StudentCount
1.0 – 2.0	2
2.0 – 2.49	3
2.5 – 2.99	16
3.0 – 3.49	25
3.5 – 3.99	24
4.0	12

This result can be accomplished through multiple SQL queries UNION'd together, or through the use of CASE statements in a single SQL query. Each of those alternatives require that we *hardcode*, or explicitly specify, the ranges in our query code.

As an alternative, a table may be created to contain the desired ranges:

RangeID	GradeRange	LowerBound	UpperBound
100	1.0 - 2.0	1.00	2.00
200	2.0 - 2.49	2.00	2.50
250	2.5 - 2.99	2.50	3.00
300	3.0 - 3.49	3.00	3.50
350	3.5 - 3.99	3.50	4.00
400	4.0	4.00	5.00
50	Less than 1.0	.00	1.00

This “range” table may be used in a query with the detail table, *without* implementing a join. Instead, other criteria may be used in the WHERE clause (taking *LowerBound* and *UpperBound* into account) to ensure that the proper range identifier is assigned to the detail record.

```
select s.ID, s.GPA, r.GradeRange
  from Students s, GPA_Ranges r
 where s.GPA >= r.LowerBound and s.GPA < r.UpperBound
```

This query may then be further modified to apply the GROUP BY keyword and Count() function to produce the final aggregated result set:

```
select r.GradeRange, Count(*) as StudentCount
  from Students s, GPA_Ranges r
 where s.GPA >= r.LowerBound and s.GPA < r.UpperBound
 group by r.GradeRange
```

By creatively leveraging the otherwise undesired Cartesian product, we can maintain our ranges in a table, rather than in the query code. We can adjust the ranges if desired by modifying the range records without having to change the query. This is often useful in automated applications, where adjusting data in a table is preferable to changing and recompiling programming code.

# Defining Database Objects

---

In this section:

- Creating Tables
- Creating Views
- Deleting Objects
- Creating and Executing Stored Procedures

## Creating Tables

Keywords: **CREATE TABLE**

Statement Syntax:

```
Create Table tableName
(
  column1    dataType1,
  column2    dataType2,
  column3    dataType3,
  ...
  column     dataTypeN
)
```

The CREATE TABLE statement defines an empty table structure by specifying columns and the types of data each can hold.

Example:

```
Create Table Departments
(
  DeptCode      Char(3),
  ShortName     Varchar(10),
  FullName      Varchar(50),
  PhoneExt      Numeric(4,0)
)
```

DeptCode	ShortName	FullName	PhoneExt

Notes:

- The following table lists common datatypes:

Char( <i>x</i> )	Fixed text data of exactly <i>x</i> characters. Unused characters are stored as spaces.
Varchar( <i>x</i> ) Varchar2( <i>x</i> )	Variable text data of no more than <i>x</i> characters. Unused characters are not stored.
Decimal ( <i>p</i> , <i>s</i> ) Number ( <i>p</i> , <i>s</i> ) Numeric ( <i>p</i> , <i>s</i> )	Numeric data with a precision (total number of digits) of <i>p</i> and a scale (total number of digits to the right of the decimal point) of <i>s</i>
Date DateTime	Date and/or time data

- Datatypes vary among RDBMSs. Consult the documentation for your specific implementation.
- Other column attributes or data constraints may be defined in a Create statement. A common constraint is to specify that a column may not contain null values:

```
Create Table Departments (  
    ...  
    ShortName    Varchar(10) NOT NULL,  
    ...  
)
```

Column definitions may also include primary keys and other data-specific constraints. Consult your database documentation for proper syntax.

## Creating Views

Keywords: **CREATE VIEW**

Statement Syntax:

```
Create View viewName As  
  selectStatement
```

The CREATE VIEW statement defines a query to be stored in the database. The view can then be accessed by name as though it were a table.

Examples:

```
Create View EnglishMajors As  
  Select * From Students  
    Where Major = 'ENG'  
  
  Select * From EnglishMajors
```

ID	FirstName	LastName	MI	Major
10030	Elaine	Benice	V	ENG
10031	James	McCoy	NULL	ENG
10032	Jamie	Cluck	NULL	ENG

```
Create View AvgGpaByMajor As  
  Select MajorDesc, Avg(GPA) as AvgGPA  
    From Students, Majors  
    Where Students.Major = Majors.MajorCode  
  Group By MajorDesc  
  
  Select * From AvgGpaByMajor
```

MajorDesc	AvgGPA
Biology	3.30
Communications	3.28
Education	2.84

Notes:

- Creating views from commonly used queries is a good way to reduce keystrokes.
- Creating views also allows for complex query logic to be encapsulated, simplifying data access.

## Deleting Objects

Keywords: **DROP**

Statement Syntax:

```
Drop Table tableName
```

```
Drop View viewName
```

```
Drop Procedure procedureName
```

The DROP statement permanently deletes an object from the database.

Example:

```
Drop View EnglishMajors
```

Notes:

- If used on a table, the DROP statement will delete the object and all its data. Use with caution!

## Creating and Executing Stored Procedures

Keywords: **CREATE PROCEDURE, EXECUTE**

Statement Syntax:

```
Create Procedure procName (optionalParameters)
As
Begin
    Procedure statements
End
```

The CREATE PROCEDURE statement stores a series of lines of executable code in the database. Standard SQL statements (SELECT, INSERT, UPDATE, DELETE) may be employed in a stored procedure, and can use parameters defined in the *optionalParameters* list.

Parameters in Microsoft SQL Server are defined with this syntax:

```
@parameterName dataType
```

Multiple parameters are separated with commas in the procedure definition.

A stored procedure is run with the EXECUTE statement.

Examples (specific to Microsoft SQL Server):

```
Create Procedure StudentListing (@deptCode varchar(4))
As
Begin
    Select * From Students
        Where Department = @deptCode
        Order By LastName, FirstName
End

Execute StudentListing 'ENG'
```

```
Create Procedure AddDepartment
(@deptCode varchar(4), @deptName varchar(128))
As
Begin
    Insert Into Departments
        (DepartmentCode, DepartmentDesc)
    Values
        (@deptCode, @deptName)
End

Execute AddDepartment 'MUS', 'Music'
```

Notes:

- Parameterized queries are one common application of stored procedures.
- Stored procedures play a role in security. Users may be denied direct access to tables or views, but provided more limited access to data through a stored procedure. This gives the database administrator a great deal of flexibility for what data individuals may inspect or manipulate.

# Keyword Index

---

**AND**, 10

**AS**, 12, 14, 18, 28

**ASC**, 16

**CASE**, 14

**CREATE**

    PROCEDURE, 42

    TABLE, 37

    VIEW, 39

**DELETE**, 25

**DESC**, 16

**DROP**, 41

**ELSE**, 14

**END**, 14

**EXECUTE**, 42

**FROM**, 4

**GROUP BY**, 19

**INNER JOIN**, 29

**INSERT INTO**, 22, 23

**LEFT JOIN**, 31

**NULL**, 8

**OR**, 10

**ORDER BY**, 16

**OUTER JOIN**, 31

**RIGHT JOIN**, 31

**SELECT**, 4, 18, 23, 27

**SET**, 24

**THEN**, 14

**UNION**, 33

**UPDATE**, 24

**VALUES**, 22

**WHEN**, 14

**WHERE**, 5, 8