**SQL: Structured Query Language**

SQL statements are divided into two major categories: **data definition language (DDL)** and **data manipulation language (DML)**.

**Data definition language**

DDL statements are used to build and modify the structure of your tables and other objects in the database. When you execute a DDL statement, it takes effect immediately.

• The create table statement does exactly that:

CREATE TABLE <table name> (

<attribute name 1> <data type 1>,

...

<attribute name n> <data type n>);

The **data types** that you will use most frequently are character strings, which might be called VARCHAR or CHAR for variable or fixed length strings; numeric types such as NUMBER or INTEGER, which will usually specify a precision; and DATE or related types.

• The **alter table** statement may be used as you have seen to specify primary and foreign key constraints, as well as to make other modifications to the table structure. Key constraints may also be specified in the CREATE TABLE statement.

ALTER TABLE <table name> ADD CONSTRAINT <constraint name> PRIMARY KEY (<attribute list>);

• The **foreign key** constraint is a bit more complicated, since we have to specify both the FK attributes in this (child) table, and the PK attributes that they link to in the parent table.

ALTER TABLE <table name> ADD CONSTRAINT <constraint name> FOREIGN KEY (<attribute list>)REFERENCES <parent table name> (<attribute list>);

Name the constraint in the form childtable\_parenttable\_fk (for example, Orders\_Customers\_fk). If there is more than one attribute in the FK, all of them must be included (with commas between) in both the FK attribute list and the REFERENCES (parent table) attribute list.

The syntax is different for tables and constraints.

**DROP TABLE** <table name>;

ALTER TABLE <table name>

**DROP CONSTRAINT** <constraint name>;

• All of the information about objects in your schema is contained, not surprisingly, in a set of tables that is called the **data dictionary**.

## Data manipulation language

DML statements are used to work with the data in tables.

• The insert statement is used, obviously, to add new rows to a table.

INSERT INTO <table name> VALUES (<value 1>, ... <value n>);

• The update statement is used to change values that are already in a table.

UPDATE <table name> SET <attribute> = <expression> WHERE <condition>;

• The **delete** statement does just that, for rows in a table.

DELETE FROM <table name>

WHERE <condition>;

If the WHERE clause is omitted, then every row of the table is deleted

• If you are using a large multi-user system, you may need to make your DML changes visible to the rest of the users of the database. Although this might be done automatically when you log out, you could also just type:

COMMIT;

• If you’ve messed up your changes in this type of system, and want to restore your private copy of the database to the way it was before you started (this only works if you haven’t already typed COMMIT), just type:

ROLLBACK;

Although single-user systems don’t support **commit** and **rollback** statements, they are used in large systems to control**transactions**, which are sequences of changes to the database. Transactions are frequently covered in more advanced courses.

## Privileges

If you want anyone else to be able to view or manipulate the data in your tables, and if your system permits this, you will have to explicitly **grant** the appropriate privilege or privileges (select, insert, update, or delete) to them. This has to be done for each table. The most common case where you would use grants is for tables that you want to make available to scripts running on a Web server, for example:

GRANT select, insert ON customers TO webuser;

**CREATE TABLE** :

The SQL **syntax** for **CREATE TABLE** is

**CREATE TABLE "table\_name"  
("column 1" "data\_type\_for\_column\_1",  
"column 2" "data\_type\_for\_column\_2",  
... )**

So, if we are to create the customer table specified as above, we would type in

**CREATE TABLE customer  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth\_Date date)**

Sometimes, we want to provide a default value for each column. A default value is used when you do not specify a column's value when inserting data into the table. To specify a default value, add "Default [value]" after the data type declaration. In the above example, if we want to default column "Address" to "Unknown" and City to "Mumbai", we would type in

**CREATE TABLE customer  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50) default 'Unknown',  
City char(50) default 'Mumbai',  
Country char(25),  
Birth\_Date date)**

.

**Constraints**

Common types of constraints include the following:

* [**NOT NULL Constraint**](http://www.1keydata.com/sql/sql-not-null.html): Ensures that a column cannot have NULL value.
* [**DEFAULT Constraint**](http://www.1keydata.com/sql/sql-default.html): Provides a default value for a column when none is specified.
* [**UNIQUE Constraint**](http://www.1keydata.com/sql/sql-unique.html): Ensures that all values in a column are different.
* [**CHECK Constraint**](http://www.1keydata.com/sql/sql-check.html): Makes sure that all values in a column satisfy certain criteria.
* [**Primary Key Constraint**](http://www.1keydata.com/sql/sql-primary-key.html): Used to uniquely identify a row in the table.
* [**Foreign Key Constraint**](http://www.1keydata.com/sql/sql-foreign-key.html): Used to ensure referential integrity of the data.

**NOT NULL Constraint:**

By default, a column can hold NULL. If you not want to allow NULL value in a column, you will want to place a constraint on this column specifying that NULL is now not an allowable value.

For example, in the following statement,

**CREATE TABLE Customer   
(SID integer NOT NULL,   
Last\_Name varchar (30) NOT NULL,   
First\_Name varchar(30));**

Columns "SID" and "Last\_Name" cannot include NULL, while "First\_Name" can include NULL.

An attempt to execute the following SQL statement,

**INSERT INTO Customer (Last\_Name, First\_Name) values ('Wong','Ken');**

will result in an error because this will lead to column "SID" being NULL, which violates the NOT NULL constraint on that column.

[**SQL UNION ALL**](http://www.1keydata.com/sql/sqlunionall.html)  
[**SQL INTERSECT**](http://www.1keydata.com/sql/sql-intersect.html)  
[**SQL MINUS**](http://www.1keydata.com/sql/sql-minus.html)  
[**SQL LIMIT**](http://www.1keydata.com/sql/sql-limit.html)  
[**SQL TOP**](http://www.1keydata.com/sql/sql-top.html)  
[**SQL Subquery**](http://www.1keydata.com/sql/sql-subquery.html)  
[**SQL EXISTS**](http://www.1keydata.com/sql/sql-exists.html)  
[**SQL CASE**](http://www.1keydata.com/sql/sql-case.html)  
[**SQL NULL**](http://www.1keydata.com/sql/sql-null.html)  
[**SQL ISNULL**](http://www.1keydata.com/sql/sql-isnull.html)  
[**SQL IFNULL**](http://www.1keydata.com/sql/sql-ifnull.html)  
[**SQL NVL**](http://www.1keydata.com/sql/sql-nvl.html)  
[**SQL COALESCE**](http://www.1keydata.com/sql/sql-coalesce.html)  
[**SQL NULLIF**](http://www.1keydata.com/sql/sql-nullif.html)  
[**SQL Rank**](http://www.1keydata.com/sql/sql-rank.html)  
[**SQL Median**](http://www.1keydata.com/sql/sql-median.html)  
[**SQL Running Totals**](http://www.1keydata.com/sql/sql-running-totals.html)  
[**SQL Percent to Total**](http://www.1keydata.com/sql/sql-percent-to-total.html)  
[**SQL Cumulative**](http://www.1keydata.com/sql/sql-cumulative-percent-to-total.html)

**Union** :

The purpose of the SQL **UNION** query is to combine the results of two queries together. In this respect,**UNION** is somewhat similar to **[JOIN](http://www.1keydata.com/sql/sqljoins.html)**in that they are both used to related information from multiple tables. One restriction of **UNION** is that all corresponding columns need to be of the same data type. Also, when using **UNION**, only distinct values are selected (similar to **SELECT**[**DISTINCT**](http://www.1keydata.com/sql/sqldistinct.html)).

The syntax is as follows:

**[SQL Statement 1]  
UNION  
[SQL Statement 2]**

Say we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| Date | Sales |
| Jan-07-1999 | $250 |
| Jan-10-1999 | $535 |
| Jan-11-1999 | $320 |
| Jan-12-1999 | $750 |

and we want to find out all the dates where there is a sales transaction. To do so, we use the following SQL statement:

**SELECT Date FROM Store\_Information  
UNION  
SELECT Date FROM Internet\_Sales**

*Result:*

|  |
| --- |
| **Date** |
| **Jan-05-1999** |
| **Jan-07-1999** |
| **Jan-08-1999** |
| **Jan-10-1999** |
| **Jan-11-1999** |
| **Jan-12-1999** |

Please note that if we type "**SELECT DISTINCT Date**" for either or both of the SQL statement, we will get the same result set.

**Union All** :

The purpose of the SQL **UNION ALL** command is also to combine the results of two queries together. The difference between **UNION ALL** and **UNION** is that, while **UNION** only selects distinct values, **UNION ALL** selects all values.

The syntax for **UNION ALL** is as follows:

**[SQL Statement 1]  
UNION ALL  
[SQL Statement 2]**

Let's use the same example as the previous section to illustrate the difference. Assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| Date | Sales |
| Jan-07-1999 | $250 |
| Jan-10-1999 | $535 |
| Jan-11-1999 | $320 |
| Jan-12-1999 | $750 |

and we want to find out all the dates where there is a sales transaction at a store as well as all the dates where there is a sale over the internet. To do so, we use the following SQL statement:

**SELECT Date FROM Store\_Information  
UNION ALL  
SELECT Date FROM Internet\_Sales**

*Result:*

|  |
| --- |
| **Date** |
| **Jan-05-1999** |
| **Jan-07-1999** |
| **Jan-08-1999** |
| **Jan-08-1999** |
| **Jan-07-1999** |
| **Jan-10-1999** |
| **Jan-11-1999** |
| **Jan-12-1999** |

**Intersect** :

Similar to the **UNION** command, **INTERSECT** also operates on two SQL statements. The difference is that, while **UNION** essentially acts as an **OR** operator (value is selected if it appears in either the first or the second statement), the **INTERSECT** command acts as an **AND** operator (value is selected only if it appears in both statements).

The syntax is as follows:

**[SQL Statement 1]  
INTERSECT  
[SQL Statement 2]**

Let's assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| Date | Sales |
| Jan-07-1999 | $250 |
| Jan-10-1999 | $535 |
| Jan-11-1999 | $320 |
| Jan-12-1999 | $750 |

and we want to find out all the dates where there are both store sales and internet sales. To do so, we use the following SQL statement:

**SELECT Date FROM Store\_Information  
INTERSECT  
SELECT Date FROM Internet\_Sales**

*Result:*

|  |
| --- |
| **Date** |
| **Jan-07-1999** |

Please note that the **INTERSECT** command will only return distinct values.

**Minus** :

The **MINUS** operates on two SQL statements. It takes all the results from the first SQL statement, and then subtract out the ones that are present in the second SQL statement to get the final answer. If the second SQL statement includes results not present in the first SQL statement, such results are ignored.

The syntax is as follows:

**[SQL Statement 1]  
MINUS  
[SQL Statement 2]**

Let's continue with the same example:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| Date | Sales |
| Jan-07-1999 | $250 |
| Jan-10-1999 | $535 |
| Jan-11-1999 | $320 |
| Jan-12-1999 | $750 |

and we want to find out all the dates where there are store sales, but no internet sales. To do so, we use the following SQL statement:

**SELECT Date FROM Store\_Information  
MINUS  
SELECT Date FROM Internet\_Sales**

*Result:*

|  |
| --- |
| **Date** |
| **Jan-05-1999** |
| **Jan-08-1999** |

"Jan-05-1999", "Jan-07-1999", and "Jan-08-1999" are the distinct values returned from "**SELECT Date FROM Store\_Information**." "Jan-07-1999" is also returned from the second SQL statement, "**SELECT Date FROM Internet\_Sales**," so it is excluded from the final result set.

Please note that the **MINUS** command will only return distinct values.

Some databases may use **EXCEPT** instead of **MINUS**. Please check the documentation for your specific database for the correct usage.

**Limit** :

Sometimes we may not want to retrieve all the records that satisfy the critera specified in **WHERE** or **HAVING** clauses.

In MySQL, this is accomplished using the **LIMIT** keyword. The syntax for **LIMIT** is as follows:

**[SQL Statement 1]  
LIMIT [N]**

where [N] is the number of records to be returned. Please note that the[**ORDER BY**](http://www.1keydata.com/sql/sqlorderby.html) clause is usually included in the SQL statement. Without the **ORDER BY** clause, the results we get would be dependent on what the database default is.

For example, we may wish to show the two highest sales amounts in Table ***Store\_Information***

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT store\_name, Sales, Date  
FROM Store\_Information  
ORDER BY Sales DESC  
LIMIT 2;**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **Los Angeles** | **$1500** | **Jan-05-1999** |
| **Boston** | **$700** | **Jan-08-1999** |

**Top** :

 the previous section, we saw how [**LIMIT**](http://www.1keydata.com/sql/sql-limit.html) can be used to retrieve a subset of records in MySQL. In Microsoft SQL Server, this is accomplished using the **TOP** keyword.

The syntax for **TOP** is as follows:

**SELECT TOP [TOP argument] "column\_name"  
FROM "table\_name"**

where [TOP argument] can be one of two possible types:

1. **[N]**: The first **N** records are returned.

2. **[N'] PERCENT**: The number of records corresponding to **N'%** of all qualifying records are returned.

For example, we may wish to show the two highest sales amounts in Table ***Store\_Information***

,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT TOP 2 store\_name, Sales, Date  
FROM Store\_Information  
ORDER BY Sales DESC;**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **Los Angeles** | **$1500** | **Jan-05-1999** |
| **Boston** | **$700** | **Jan-08-1999** |

Alternatively, if we want to show the top 25% of sales amounts from Table ***Store\_Information***, we key in,

**SELECT TOP 25 PERCENT store\_name, Sales, Date  
FROM Store\_Information  
ORDER BY Sales DESC;**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **Los Angeles** | **$1500** | **Jan-05-1999** |

**Subquery** :

is possible to embed a SQL statement within another. When this is done on the **WHERE** or the**HAVING** statements, we have a subquery construct.

The syntax is as follows:

**SELECT "column\_name1"  
FROM "table\_name1"  
WHERE "column\_name2" [Comparison Operator]  
(SELECT "column\_name3"  
FROM "table\_name2"  
WHERE [Condition])**

[Comparison Operator] could be equality operators such as =, >, <, >=, <=. It can also be a text operator such as "LIKE". The portion in **red** is considered as the "inner query", while the portion in **green** is considered as the "outer query".

Let's use the same example as we did to illustrate SQL joins:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

and we want to use a subquery to find the sales of all stores in the West region. To do so, we use the following SQL statement:

**SELECT SUM(Sales) FROM Store\_Information  
WHERE Store\_name IN  
(SELECT store\_name FROM Geography  
WHERE region\_name = 'West')**

*Result:*

|  |
| --- |
| **SUM(Sales)** |
| **2050** |

In this example, instead of joining the two tables directly and then adding up only the sales amount for stores in the West region, we first use the subquery to find out which stores are in the West region, and then we sum up the sales amount for these stores.

In the above example, the inner query is first executed, and the result is then fed into the outer query. This type of subquery is called a **simple subquery**. If the inner query is dependent on the outer query, we will have a **correlated subquery**. An example of a **correlated subquery** is shown below:

**SELECT SUM(a1.Sales) FROM Store\_Information a1  
WHERE a1.Store\_name IN  
(SELECT store\_name FROM Geography a2  
WHERE a2.store\_name = a1.store\_name)**

Notice the **WHERE** clause in the inner query, where the condition involves a table from the outer query.

**Exists** :

 the previous section, we used **IN** to link the inner query and the outer query in a subquery statement. **IN** is not the only way to do so -- one can use many operators such as >, <, or =. **EXISTS** is a special operator that we will discuss in this section.

**EXISTS** simply tests whether the inner query returns any row. If it does, then the outer query proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing.

The syntax for **EXISTS** is:

**SELECT "column\_name1"  
FROM "table\_name1"  
WHERE EXISTS   
(SELECT \*   
FROM "table\_name2"  
WHERE [Condition])**

Please note that instead of \*, you can select one or more columns in the inner query. The effect will be identical.

Let's use the same example tables:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

and we issue the following SQL query:

**SELECT SUM(Sales) FROM Store\_Information  
WHERE EXISTS  
(SELECT \* FROM Geography  
WHERE region\_name = 'West')**

We'll get the following result:

|  |
| --- |
| **SUM(Sales)** |
| **2750** |

At first, this may appear confusing, because the subquery includes the [region\_name = 'West'] condition, yet the query summed up stores for all regions. Upon closer inspection, we find that since the subquery returns more than 0 row, the **EXISTS** condition is true, and the condition placed inside the inner query does not influence how the outer query is run.

**Case** :

**CASE** is used to provide if-then-else type of logic to SQL. Its syntax is:

**SELECT CASE ("column\_name")  
  WHEN "condition1" THEN "result1"  
  WHEN "condition2" THEN "result2"  
  ...  
  [ELSE "resultN"]  
  END  
FROM "table\_name"**

"condition" can be a static value or an expression. The **ELSE** clause is optional.

In our Table ***Store\_Information***example,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

if we want to multiply the sales amount from 'Los Angeles' by 2 and the sales amount from 'San Diego' by 1.5, we key in,

**SELECT store\_name, CASE store\_name  
  WHEN 'Los Angeles' THEN Sales \* 2  
  WHEN 'San Diego' THEN Sales \* 1.5  
  ELSE Sales  
  END  
"New Sales",  
Date  
FROM Store\_Information**

"New Sales" is the name given to the column with the CASE statement.

*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **New Sales** | **Date** |
| **Los Angeles** | **$3000** | **Jan-05-1999** |
| **San Diego** | **$375** | **Jan-07-1999** |
| **San Francisco** | **$300** | **Jan-08-1999** |
| **Boston** | **$700** | **Jan-08-1999** |

**NULL** :

 SQL, **NULL** means that data does not exist. NULL does not equal to 0 or an empty string. Both 0 and empty string represent a value, while **NULL**has no value.

Any mathematical operations performed on **NULL** will result in**NULL**. For example,

10 + NULL = NULL

Aggregate functions such as SUM, COUNT, AVG, MAX, and MIN exclude NULL values. This is not likely to cause any issues for SUM, MAX, and MIN. However, this can lead to confusion with AVG and COUNT.

Let's take a look at the following example:

Table ***Sales\_Data***

|  |  |
| --- | --- |
| store\_name | Sales |
| Store A | 300 |
| Store B | 200 |
| Store C | 100 |
| Store D | NULL |

Below are the results for each aggregate function:

SUM (Sales) = 600

AVG (Sales) = 200

MAX (Sales) = 300

MIN (Sales) = 100

COUNT (Sales) = 3

Note that the AVG function counts only 3 rows (the NULL row is excluded), so the average is 600 / 3 = 200, not 600 / 4 = 150. The COUNT function also ignores the NULL rolw, which is why COUNT (Sales) = 3.

**ISNULL Function** :

he ISNULL function is available in both SQL Server and MySQL. However, their uses are different:

SQL Server

In SQL Server, the ISNULL() function is used to replace NULL value with another value.

For example, if we have the following table,

Table ***Sales\_Data***

|  |  |
| --- | --- |
| store\_name | Sales |
| Store A | 300 |
| Store B | NULL |

The following SQL,

SELECT SUM(ISNULL(Sales,100)) FROM Sales\_Data;

returns 400. This is because NULL has been replaced by 100 via the ISNULL function.

MySQL

In MySQL, the ISNULL() function is used to test whether an expression is NULL. If the expression is NULL, this function returns 1. Otherwise, this function returns 0.

For example,

ISNULL(3\*3) returns 0

ISNULL(3/0) returns 1

**IFNULL Function:**

he IFNULL() function is available in MySQL, and not in SQL Server or Oracle. This function takes two arguments. If the first argument is not NULL, the function returns the first argument. Otherwise, the second argument is returned. This function is commonly used to replace NULL value with another value. It is similar to the [**NVL function**](http://www.1keydata.com/sql/sql-nvl.html) in Oracle and the [**ISNULL Function**](http://www.1keydata.com/sql/sql-isnull.html) in SQL Server.

For example, if we have the following table,

Table ***Sales\_Data***

|  |  |
| --- | --- |
| store\_name | Sales |
| Store A | 300 |
| Store B | NULL |

The following SQL,

**SELECT SUM(IFNULL(Sales,100)) FROM Sales\_Data;**

returns 400. This is because NULL has been replaced by 100 via the ISNULL function.

**NVL Function:**

he NVL() function is available in Oracle, and not in MySQL or SQL Server. This function is used to replace NULL value with another value. It is similar to the [**IFNULL Function**](http://www.1keydata.com/sql/sql-ifnull.html) in MySQL and the [**ISNULL Function**](http://www.1keydata.com/sql/sql-isnull.html) in SQL Server.

For example, if we have the following table,

Table ***Sales\_Data***

|  |  |
| --- | --- |
| store\_name | Sales |
| Store A | 300 |
| Store B | NULL |
| Store C | 150 |

The following SQL,

SELECT SUM(NVL(Sales,100)) FROM Sales\_Data;

returns 550. This is because NULL has been replaced by 100 via the ISNULL function, hence the sum of the 3 rows is 300 + 100 + 150 = 550.

**Coalesce Function:**

he **COALESCE** function in SQL returns the first non-NULL expression among its arguments.

It is the same as the following **[CASE](http://www.1keydata.com/sql/sql-case.html)**statement:

**SELECT CASE ("column\_name")  
  WHEN "expression 1 is not NULL" THEN "expression 1"  
  WHEN "expression 2 is not NULL" THEN "expression 2"  
  ...  
  [ELSE "NULL"]  
  END  
FROM "table\_name"**

For examples, say we have the following table,

Table ***Contact\_Info***

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Business\_Phone | Cell\_Phone | Home\_Phone |
| Jeff | 531-2531 | 622-7813 | 565-9901 |
| Laura | NULL | 772-5588 | 312-4088 |
| Peter | NULL | NULL | 594-7477 |

and we want to find out the best way to contact each person according to the following rules:

1. If a person has a business phone, use the business phone number.

2. If a person does not have a business phone and has a cell phone, use the cell phone number.

3. If a person does not have a business phone, does not have a cell phone, and has a home phone, use the home phone number.

We can use the **COALESCE** function to achieve our goal:

**SELECT Name, COALESCE(Business\_Phone, Cell\_Phone, Home\_Phone) Contact\_Phone   
FROM Contact\_Info;**

*Result:*

|  |  |
| --- | --- |
| **Name** | **Contact\_Phone** |
| **Jeff** | **531-2531** |
| **Laura** | **772-5588** |
| **Peter** | **594-7477** |

**NULLIF Function** :

he **NULLIF** function takes two arguments. If the two arguments are equal, then NULL is returned. Otherwise, the first argument is returned.

It is the same as the following **[CASE](http://www.1keydata.com/sql/sql-case.html)**statement:

**SELECT CASE ("column\_name")  
  WHEN "expression 1 = expression 2 " THEN "NULL"  
  [ELSE "expression 1"]  
  END  
FROM "table\_name"**

For example, let's say we have a table that tracks actual sales and sales goal as below:

Table ***Sales\_Data***

|  |  |  |
| --- | --- | --- |
| Store\_name | Actual | Goal |
| Store A | 50 | 50 |
| Store B | 40 | 50 |
| Store C | 25 | 30 |

We want to show NULL if actual sales is equal to sales goal, and show actual sales if the two are different. To do this, we issue the following SQL statement:

**SELECT Store\_name, NULLIF(Actual,Goal) FROM Sales\_Data;**

The result is:

|  |  |
| --- | --- |
| **Store\_name** | **NULLIF(Actual,Goal)** |
| **Store A** | **NULL** |
| **Store B** | **40** |
| **Store C** | **25** |

**Rank** :

isplaying the rank associated with each row is a common request, and there is no straightforward way to do so in SQL. To display rank in SQL, the idea is to do a self-join, list out the results in order, and do a count on the number of records that's listed ahead of (and including) the record of interest. Let's use an example to illustrate. Say we have the following table,

Table ***Total\_Sales***

|  |  |
| --- | --- |
| Name | Sales |
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

we would type,

**SELECT a1.Name, a1.Sales, COUNT(a2.sales) Sales\_Rank   
FROM Total\_Sales a1, Total\_Sales a2   
WHERE a1.Sales <= a2.Sales or (a1.Sales=a2.Sales and a1.Name = a2.Name)   
GROUP BY a1.Name, a1.Sales   
ORDER BY a1.Sales DESC, a1.Name DESC;**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Name** | **Sales** | **Sales\_Rank** |
| **Greg** | **50** | **1** |
| **Sophia** | **40** | **2** |
| **Stella** | **20** | **3** |
| **Jeff** | **20** | **3** |
| **Jennifer** | **15** | **5** |
| **John** | **10** | **6** |

Let's focus on the **WHERE** clause. The first part of the clause, (a1.Sales <= a2.Sales), makes sure we are only counting the number of occurrences where the value in the Sales column is less than or equal to itself. If there are no duplicate values in the Sales column, this portion of the**WHERE** clause by itself would be sufficient to generate the correct ranking.

The second part of the clause, (a1.Sales=a2.Sales and a1.Name = a2.Name), ensures that when there are duplicate values in the Sales column, each one would get the correct rank.

**Median:**

o get the median, we need to be able to accomplish the following:

* Sort the rows in order and find the rank for each row.
* Determine what is the "middle" rank. For example, if there are 9 rows, the middle rank would be 5.
* Obtain the value for the middle-ranked row.

Let's use an example to illustrate. Say we have the following table,

Table ***Total\_Sales***

|  |  |
| --- | --- |
| Name | Sales |
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

we would type,

**SELECT Sales Median FROM   
(SELECT a1.Name, a1.Sales, COUNT(a1.Sales) Rank   
FROM Total\_Sales a1, Total\_Sales a2   
WHERE a1.Sales < a2.Sales OR (a1.Sales=a2.Sales AND a1.Name <= a2.Name)   
group by a1.Name, a1.Sales   
order by a1.Sales desc) a3   
WHERE Rank = (SELECT (COUNT(\*)+1) DIV 2 FROM Total\_Sales);**

*Result:*

|  |
| --- |
| **Median** |
| **20** |

You will find that Lines 2-6 are the same as how we find the [rank](http://www.1keydata.com/sql/sql-rank.html) of each row. Line 7 finds the "middle" rank. DIV is the way to find the quotient in MySQL, the exact way to obtain the quotient may be different with other databases. Finally, Line 1 obtains the value for the middle-ranked row.

**Running Totals** :

isplaying running totals is a common request, and there is no straightforward way to do so in SQL. The idea for using SQL to display running totals similar to that for displaying rank: first do a self-join, then, list out the results in order. Where as finding the rank requires doing a count on the number of records that's listed ahead of (and including) the record of interest, finding the running total requires summing the values for the records that's listed ahead of (and including) the record of interest.

Let's use an example to illustrate. Say we have the following table,

Table ***Total\_Sales***

|  |  |
| --- | --- |
| Name | Sales |
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

we would type,

**SELECT a1.Name, a1.Sales, SUM(a2.Sales) Running\_Total   
FROM Total\_Sales a1, Total\_Sales a2   
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and a1.Name = a2.Name)   
GROUP BY a1.Name, a1.Sales   
ORDER BY a1.Sales DESC, a1.Name DESC;**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Name** | **Sales** | **Running\_Total** |
| **Greg** | **50** | **50** |
| **Sophia** | **40** | **90** |
| **Stella** | **20** | **110** |
| **Jeff** | **20** | **130** |
| **Jennifer** | **15** | **145** |
| **John** | **10** | **155** |

The combination of the **WHERE** clause and the **ORDER BY** clause ensure that the proper running totals are tabulated when there are duplicate values.

**Percent To Total** :

o display percent to total in SQL, we want to leverage the ideas we used for rank/running total plus subquery. Different from what we saw in the[**SQL Subquery**](http://www.1keydata.com/sql/sql-subquery.html) section, here we want to use the subquery as part of the **SELECT**. Let's use an example to illustrate. Say we have the following table,

Table ***Total\_Sales***

|  |  |
| --- | --- |
| Name | Sales |
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

we would type,

**SELECT a1.Name, a1.Sales, a1.Sales/(SELECT SUM(Sales) FROM Total\_Sales) Pct\_To\_Total   
FROM Total\_Sales a1, Total\_Sales a2   
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and a1.Name = a2.Name)   
GROUP BY a1.Name, a1.Sales   
ORDER BY a1.Sales DESC, a1.Name DESC;**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Name** | **Sales** | **Pct\_To\_Total** |
| **Greg** | **50** | **0.3226** |
| **Sophia** | **40** | **0.2581** |
| **Stella** | **20** | **0.1290** |
| **Jeff** | **20** | **0.1290** |
| **Jennifer** | **15** | **0.0968** |
| **John** | **10** | **0.0645** |

The subquery "SELECT SUM(Sales) FROM Total\_Sales" calculates the sum. We can then divide the individual values by this sum to obtain the percent to total for each row.

**Cumulative Percent To Total:**

o display cumulative percent to total in SQL, we use the same idea as we saw in the [**Percent To Total**](http://www.1keydata.com/sql/sql-percent-to-total.html) section. The difference is that we want the cumulative percent to total, not the percentage contribution of each individual row. Let's use the following example to illuatrate:

Table ***Total\_Sales***

|  |  |
| --- | --- |
| Name | Sales |
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

we would type,

**SELECT a1.Name, a1.Sales, SUM(a2.Sales)/(SELECT SUM(Sales) FROM Total\_Sales) Pct\_To\_Total   
FROM Total\_Sales a1, Total\_Sales a2   
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and a1.Name = a2.Name)   
GROUP BY a1.Name, a1.Sales   
ORDER BY a1.Sales DESC, a1.Name DESC;**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Name** | **Sales** | **Pct\_To\_Total** |
| **Greg** | **50** | **0.3226** |
| **Sophia** | **40** | **0.5806** |
| **Stella** | **20** | **0.7097** |
| **Jeff** | **20** | **0.8387** |
| **Jennifer** | **15** | **0.9355** |
| **John** | **10** | **1.0000** |

The subquery "SELECT SUM(Sales) FROM Total\_Sales" calculates the sum. We can then divide the running total, "SUM(a2.Sales)", by this sum to obtain the cumulative percent to total for each row.

**DEFAULT Constraint**:

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

For example: if we create a table as below:

**CREATE TABLE Student   
(Student\_ID integer Unique,   
Last\_Name varchar (30),   
First\_Name varchar (30),   
Score DEFAULT 80);**

and execute the following SQL statement,

**INSERT INTO Student (Student\_ID, Last\_Name, First\_Name) values ('10','Johnson','Rick');**

|  |  |  |  |
| --- | --- | --- | --- |
| Student\_ID | Last\_Name | First\_Name | Score |
| 10 | Johnson | Rick | 80 |

**UNIQUE Constraint** :

The UNIQUE constraint ensures that all values in a column are distinct.

For example: in the following CREATE TABLE statement,

**CREATE TABLE Customer   
(SID integer Unique,   
Last\_Name varchar (30),   
First\_Name varchar(30));**

column "SID" has a unique constraint, and hence cannot include duplicate values. Such constraint does not hold for columns "Last\_Name" and "First\_Name". So, if the table already contains the following rows:

|  |  |  |
| --- | --- | --- |
| SID | Last\_Name | First\_Name |
| 1 | Johnson | Stella |
| 2 | James | Gina |
| 3 | Aaron | Ralph |

Executing the following SQL statement,

**INSERT INTO Customer values ('3','Lee','Grace');**

will result in an error because '3' already exists in the SID column, thus trying to insert another row with that value violates the UNIQUE constraint.

Please note that a column that is specified as a primary key must also be unique. At the same time, a column that's unique may or may not be a primary key. In addition, multiple UNIQUE constraints can be defined on a table.

**CHECK constraint:**

The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the database will only insert a new row or update an existing row if the new value satisfies the CHECK constraint. The CHECK constraint is used to ensure data quality.

For example, in the following CREATE TABLE statement,

**CREATE TABLE Customer   
(SID integer CHECK (SID > 0),   
Last\_Name varchar (30),   
First\_Name varchar(30));**

Column "SID" has a constraint -- its value must only include integers greater than 0. So, attempting to execute the following statement,

**INSERT INTO Customer values ('-3','Gonzales','Lynn');**

will result in an error because the values for SID must be greater than 0.

Please note that the **CHECK**constraint does not get enforced by MySQL at this time.

**Primary Key** :

A primary key is used to uniquely identify each row in a table. It can either be part of the actual record itself , or it can be an artificial field (one that has nothing to do with the actual record). A primary key can consist of one or more fields on a table. When multiple fields are used as a primary key, they are called a composite key.

Primary keys can be specified either when the table is created (using[**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)) or by changing the existing table structure (using [**ALTER TABLE**](http://www.1keydata.com/sql/sql-alter-table.html)).

Below are examples for specifying a primary key when creating a table:

**MySQL**:   
**CREATE TABLE Customer   
(SID integer,   
Last\_Name varchar(30),   
First\_Name varchar(30),   
PRIMARY KEY (SID));**

**Oracle**:   
**CREATE TABLE Customer   
(SID integer PRIMARY KEY,   
Last\_Name varchar(30),   
First\_Name varchar(30));**

**SQL Server**:   
**CREATE TABLE Customer   
(SID integer PRIMARY KEY,   
Last\_Name varchar(30),   
First\_Name varchar(30));**

Below are examples for specifying a primary key by altering a table:

**MySQL**:   
**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

**Oracle**:   
**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

**SQL Server**:   
**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

Note: Before using the ALTER TABLE command to add a primary key, you'll need to make sure that the field is defined as 'NOT NULL' -- in other words, NULL cannot be an accepted value for that field.

**Foreign Key** :

A Foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

For example,

say we have two tables, a CUSTOMER table that includes all customer data, and an ORDERS table that includes all customer orders. The constraint here is that all orders must be associated with a customer that is already in the CUSTOMER table. In this case, we will place a foreign key on the ORDERS table and have it relate to the primary key of the CUSTOMER table. This way, we can ensure that all orders in the ORDERS table are related to a customer in the CUSTOMER table. In other words, the ORDERS table cannot contain information on a customer that is not in the CUSTOMER table.

The structure of these two tables will be as follows:

Table ***CUSTOMER***

|  |  |
| --- | --- |
| column name | characteristic |
| SID | Primary Key |
| Last\_Name |  |
| First\_Name |  |

Table ***ORDERS***

|  |  |
| --- | --- |
| column name | characteristic |
| Order\_ID | Primary Key |
| Order\_Date |  |
| Customer\_SID | Foreign Key |
| Amount |  |

In the above example, the Customer\_SID column in the ORDERS table is a foreign key pointing to the SID column in the CUSTOMER table.

Below we show examples of how to specify the foreign key when creating the ORDERS table:

**MySQL**:   
**CREATE TABLE ORDERS   
(Order\_ID integer,   
Order\_Date date,   
Customer\_SID integer,   
Amount double,   
Primary Key (Order\_ID),   
Foreign Key (Customer\_SID) references CUSTOMER(SID));**

**Oracle**:   
**CREATE TABLE ORDERS   
(Order\_ID integer primary key,   
Order\_Date date,   
Customer\_SID integer references CUSTOMER(SID),   
Amount double);**

**SQL Server**:   
**CREATE TABLE ORDERS   
(Order\_ID integer primary key,   
Order\_Date datetime,   
Customer\_SID integer references CUSTOMER(SID),   
Amount double);**

Below are examples for specifying a foreign key by altering a table. This assumes that the ORDERS table has been created, and the foreign key has not yet been put in:

**MySQL**:   
**ALTER TABLE ORDERS   
ADD FOREIGN KEY (customer\_sid) REFERENCES CUSTOMER(SID);**

**Oracle**:   
**ALTER TABLE ORDERS   
ADD (CONSTRAINT fk\_orders1) FOREIGN KEY (customer\_sid) REFERENCES CUSTOMER(SID);**

**SQL Server**:   
**ALTER TABLE ORDERS   
ADD FOREIGN KEY (customer\_sid) REFERENCES CUSTOMER(SID);**

**View** :

A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves.

.Views offer the following advantages:

**1. Ease of use**: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.

**2. Space savings**: Views takes very little space to store, since they do not store actual data.

**3. Additional data security**: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

**Create View Statement** :

Views can be considered as virtual tables. Generally speaking, a table has a set of definition, and it physically stores the data. A view also has a set of definitions, which is build on top of table(s) or other view(s), and it does not physically store the data.

The syntax for creating a view is as follows:

**CREATE VIEW "VIEW\_NAME" AS "SQL Statement"**

"SQL Statement" can be any of the SQL statements we have discussed in this tutorial.

Let's use a simple example to illustrate. Say we have the following table:

**TABLE *Customer*  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth\_Date date)**

and we want to create a view called***V\_Customer*** that contains only the First\_Name, Last\_Name, and Country columns from this table, we would type in,

**CREATE VIEW V\_Customer  
AS SELECT First\_Name, Last\_Name, Country  
FROM Customer**

Now we have a view called ***V\_Customer*** with the following structure:

**View *V\_Customer*  
(First\_Name char(50),  
Last\_Name char(50),  
Country char(25))**

We can also use a view to apply joins to two tables. In this case, users only see one view rather than two tables, and the SQL statement users need to issue becomes much simpler. Let's say we have the following two tables:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

and we want to build a view that has sales by region information. We would issue the following SQL statement:

**CREATE VIEW V\_REGION\_SALES  
AS SELECT A1.region\_name REGION, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.store\_name = A2.store\_name  
GROUP BY A1.region\_name**

This gives us a view, ***V\_REGION\_SALES***, that has been defined to store sales by region records. If we want to find out the content of this view, we type in,

**SELECT \* FROM V\_REGION\_SALES**

*Result:*

|  |  |
| --- | --- |
| **REGION** | **SALES** |
| **East** | **$700** |
| **West** | **$2050** |

**Index**:

Indexes help us retrieve data from tables quicker.

Let's use an example to illustrate this point: Say we are interested in reading about how to grow peppers in a gardening book. Instead of reading the book from the beginning until we find a section on peppers, it is much quicker for us to go to the index section at the end of the book, locate which pages contain information on peppers, and then go to these pages directly. Going to the index first saves us time and is by far a more efficient method for locating the information we need.

The same principle applies for retrieving data from a database table. Without an index, the database system reads through the entire table (this process is called a 'table scan') to locate the desired information. With the proper index in place, the database system can then first go through the index to find out where to retrieve the data, and then go to these locations directly to get the needed data. This is much faster.

Therefore, it is often desirable to create indexes on tables. An index can cover one or more columns. The syntax for creating a table index is shown in the [CREATE INDEX](http://www.1keydata.com/sql/sql-create-index.html)section. Below we discuss some general strategies when building and using an index:

**1. Build index on columns of integer type**

Integers take less space to store, which means the query will be faster. If the column you want to build an index for is not of type integer, consider creating a surrogate integer key (or simply a surrogate column of type integer) which maps one-to-one to the column you want to build the index for.

**2. Keep index as narrow as possible**

Narrower indexes take less space, require less time to process, which in turn means the query will run faster.

**3. Column order is important**

For indexes covering multiple columns, the order of the columns in the index is important. The best practice is to use the column with the lowest cardinality first, and the column with the highest cardinality last. Recall cardinality means the number of distinct values for that column. So, if "SELECT DISTINCT (COLUMN1) FROM TABLE\_NAME;" returns 5, that means the cardinality for COLUMN1 is 5.

**4. Make sure the column you are building an index for is declared NOT NULL**

This can decrease the size of the index, which in turn will speed up the query.

**Create Index Statement**:

As mentioned in the [Index](http://www.1keydata.com/sql/sql-index.html) overview page, a table index helps SQL statements run faster. The syntax for creating an index is:

**CREATE INDEX "INDEX\_NAME" ON "TABLE\_NAME" (COLUMN\_NAME)**

Let's assume that we have the following table,

**TABLE *Customer*  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth\_Date date)**

and we want to create an index on the column Last\_Name, we would type in,

**CREATE INDEX IDX\_CUSTOMER\_LAST\_NAME  
on CUSTOMER (Last\_Name)**

If we want to create an index on both City and Country, we would type in,

**CREATE INDEX IDX\_CUSTOMER\_LOCATION  
on CUSTOMER (City, Country)**

There is no strict rule on how to name an index. The generally accepted method is to place a prefix, such as "IDX\_", before an index name to avoid confusion with other database objects. It is also a good idea to provide information on which table and column(s) the index is used on.

Please note that the exact syntax for **CREATE INDEX** may be different for different databases. You should consult with your database reference manual for the precise syntax.

**Alter Table Syntax:**

Once a table is created in the database, there are many occasions where one may wish to change the structure of the table. In general, the SQL syntax for **ALTER TABLE** is

**ALTER TABLE "table\_name"  
[alter specification]**

[alter specification] is dependent on the type of alteration we wish to perform. We list a number of common changes below:

 [ALTER TABLE Add Column](http://www.1keydata.com/sql/alter-table-add-column.html)

 [ALTER TABLE Modify Column](http://www.1keydata.com/sql/alter-table-modify-column.html)

 [ALTER TABLE Rename Column](http://www.1keydata.com/sql/alter-table-rename-column.html)

 [ALTER TABLE Drop Column](http://www.1keydata.com/sql/alter-table-drop-column.html)

 [ALTER TABLE Add Index](http://www.1keydata.com/sql/alter-table-add-index.html)

 [ALTER TABLE Drop Index](http://www.1keydata.com/sql/alter-table-drop-index.html)

 [ALTER TABLE Add Constraint](http://www.1keydata.com/sql/alter-table-add-constraint.html)

 [ALTER TABLE Drop Constraint](http://www.1keydata.com/sql/alter-table-drop-constraint.html)

For all cases, examples are provided for MySQL, Oracle, and SQL Server.

**Alter Table Add Column Syntax**:

The SQL syntax for **ALTER TABLE Add Column** is

**ALTER TABLE "table\_name"  
ADD "column 1" "Data Type"**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Our goal is to add a column called "Gender". To do this, we key in:

**MySQL**:   
**ALTER TABLE customer ADD Gender char(1);**

**Oracle**:   
**ALTER TABLE customer ADD Gender char(1);**

**SQL Server**:   
**ALTER TABLE customer ADD Gender char(1);**

Resulting table structure:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |
| Gender | char(1) |

Note that the new column Gender becomes the last column in the customer table.

It is also possible to add multiple columns. For example, if we want to add a column called "Email" and another column called "Telephone", we will type the following:

**MySQL**:   
**ALTER TABLE customer ADD (Email char(30), Telephone char(20));**

**Oracle**:   
**ALTER TABLE customer ADD (Email char(30), Telephone char(20));**

**SQL Server**:   
**ALTER TABLE customer ADD (Email char(30), Telephone char(20));**

The table now becomes:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |
| Gender | char(1) |
| Email | char(30) |
| Telephone | char(20) |

**Alter Table Modify Column Syntax:**

Sometimes we need to change the data type of a column. To do this, we use the **ALTER TABLE Modify Column** command. For Oracle and MySQL, the SQL syntax for **ALTER TABLE Modify Column** is,

**ALTER TABLE "table\_name"  
MODIFY "column 1" "New Data Type"**

For SQL Server, the syntax is,

**ALTER TABLE "table\_name"  
ALTER COLUMN "column 1" "New Data Type"**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Our goal is to alter the data type of the "Address" column to char(100). To do this, we key in:

**MySQL**:   
**ALTER TABLE customer MODIFY Address char(100);**

**Oracle**:   
**ALTER TABLE customer MODIFY Address char(100);**

**SQL Server**:   
**ALTER TABLE customer ALTER COLUMN Address char(100);**

Resulting table structure:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(100) |
| City | char(50) |
| Country | char(25) |

**Alter Table Rename Column Syntax:**

 In MySQL, the SQL syntax for**ALTER TABLE Rename Column** is

**ALTER TABLE "table\_name"  
Change "column 1" "column 2" ["Data Type"]**

In Oracle, the syntax is,

**ALTER TABLE "table\_name"  
RENAME COLUMN "column 1" TO "column 2"**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

To rename "Address" to "Addr", we key in,

**MySQL**:   
**ALTER table customer CHANGE Address Addr char(50);**

**Oracle**:   
**ALTER table customer RENAME COLUMN Address TO Addr;**

**SQL Server**:   
It is not possible to rename a column using the ALTER TABLE statement in SQL Server. Use sp\_rename instead.

Resulting table structure:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Addr | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

**Alter Table Drop Column Syntax:**

In MySQL, the SQL syntax for**ALTER TABLE Drop Column** is

**ALTER TABLE "table\_name"  
DROP "column 1"**

In Oracle and SQL Server, the SQL syntax for **ALTER TABLE Drop Column** is

**ALTER TABLE "table\_name"  
DROP COLUMN "column 1"**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Our goal is to drop the "Birth\_Date" column. To do this, we key in:

**MySQL**:   
**ALTER table customer drop Birth\_Date;**

**SQL Server**:   
**ALTER table customer drop column Birth\_Date;**

**Oracle**:   
**ALTER table customer drop column Birth\_Date;**

Resulting table structure:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |

**Alter Table Add Index Syntax:**

he SQL syntax for **ALTER TABLE Add Index** is

**ALTER TABLE "table\_name"  
ADD INDEX " INDEX\_NAME " (COLUMN\_NAME)**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html) section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Assume we want to add an index on the "Country" column. To do this, we type in the following:

**ALTER table customer ADD INDEX IDX\_COUNTRY (Country);**

Please note that using ALTER TABLE to add an index is not supported in Oracle or SQL Server.

**Alter Table Drop Index Syntax:**

the SQL syntax for **ALTER TABLE Dop Index** is

**ALTER TABLE "table\_name"  
DROP INDEX " INDEX\_NAME "**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Assume we want to drop the index created in the [**ALTER TABLE ADD INDEX**](http://www.1keydata.com/sql/alter-table-add-index.html) section. To do this, we type in the following:

**ALTER table customer DOP INDEX IDX\_COUNTRY;**

Please note that using ALTER TABLE to drop an index is not supported in Oracle or SQL Server.

**Alter Table Add Constraint Syntax:**

The SQL syntax for **ALTER TABLE Add Constraint** is

**ALTER TABLE "table\_name"  
ADD [CONSTRAINT\_NAME] [CONSTRAINT\_TYPE] [CONSTRAINT\_CONDITION]**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | date |

Assume we want to add a UNIQUE constraint to the "Address" column. To do this, we type in the following:

**MySQL**:   
**ALTER TABLE customer ADD CONSTRAINT con\_first UNIQUE (first\_name);**

**Oracle**:   
**ALTER TABLE customer ADD CONSTRAINT con\_first UNIQUE (first\_name);**

**SQL Server**:   
**ALTER TABLE customer ADD CONSTRAINT con\_first UNIQUE (first\_name);**

where con\_first is the name of the constraint.

**Alter Table Drop Constraint Syntax:**

he SQL syntax for **ALTER TABLE Drop Constraint** is

**ALTER TABLE "table\_name"  
DROP [CONSTRAINT|INDEX] "CONSTRAINT\_NAME"**

Let's look at the example. Assuming our starting point is the "customer" table created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section:

Table ***customer***

|  |  |
| --- | --- |
| Column Name | Data Type |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | Date |

Assume we want to drop the UNIQUE constraint on the "Address" column. To do this, we type in the following:

**MySQL**:   
**ALTER TABLE customer DROP INDEX con\_first;**

Note that MySQL uses DROP INDEX for index-type constraints such as UNIQUE. con\_first is the name of the constraint.

**Oracle**:   
**ALTER TABLE customer DROP CONSTRAINT con\_first;**

**SQL Server**:   
**ALTER TABLE customer DROP CONSTRAINT con\_first;**

**Drop Table Statement:**

The syntax for **DROP TABLE** is

**DROP TABLE "table\_name"**

So, if we wanted to drop the table called customer that we created in the [**CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html)section, we simply type

**DROP TABLE customer**.

**Truncate Table Statement:**

if we wish to simply get rid of the data but not the table itself? For this, we can use the **TRUNCATE TABLE** command. The syntax for **TRUNCATE TABLE** is

**TRUNCATE TABLE "table\_name"**

So, if we wanted to truncate the table called customer that we created in [**SQL CREATE TABLE**](http://www.1keydata.com/sql/sqlcreate.html), we simply type,

**TRUNCATE TABLE customer**

**USE** :

The **USE** keyword is used to select a database in MySQL. The syntax is as follows:

**USE "Database Name"**

For example, if you want to connect to a database called "Scores", you can type in the following:

**USE Scores;**

In MySQL, you can access tables in multiple databases by specifying [Database Name].[Table Name]. If the table you want to access is currently in the database you use, there is no need to specify the database name.

For example, if you want to access table "Course\_110" from database "Scores" and table "Students" from database "Personnel", you can type in the following:

**USE Scores;**

**SELECT ...   
FROM Course\_110, Personnel.Students   
WHERE ...   
;**

**Insert Into Statement:**

n the previous sections, we have seen how to query information from tables. But how do these rows of data get into these tables in the first place? This is what this section, covering the **INSERT INTO**statement, and next section, covering tbe **UPDATE** statement, are about.

In SQL, there are essentially basically two ways to **INSERT** data into a table: One is to insert it one row at a time, the other is to insert multiple rows at a time. Let's take a look at each of them individually:

##### INSERT INTO VALUES

The syntax for inserting data into a table one row at a time is as follows:

**INSERT INTO "table\_name" ("column1", "column2", ...)  
VALUES ("value1", "value2", ...)**

Assuming that we have a table that has the following structure,

Table ***Store\_Information***

|  |  |
| --- | --- |
| Column Name | Data Type |
| store\_name | char(50) |
| Sales | float |
| Date | datetime |

and now we wish to insert one additional row into the table representing the sales data for Los Angeles on January 10, 1999. On that day, this store had $900 in sales. We will hence use the following SQL script:

**INSERT INTO Store\_Information (store\_name, Sales, Date)  
VALUES ('Los Angeles', 900, 'Jan-10-1999')**

##### INSERT INTO SELECT

The second type of **INSERT INTO** allows us to insert multiple rows into a table. Unlike the previous example, where we insert a single row by specifying its values for all columns, we now use a **SELECT** statement to specify the data that we want to insert into the table. If you are thinking whether this means that you are using information from another table, you are correct. The syntax is as follows:

**INSERT INTO "table1" ("column1", "column2", ...)  
SELECT "column3", "column4", ...  
FROM "table2"**

Note that this is the simplest form. The entire statement can easily contain **WHERE**, **GROUP BY**, and **HAVING** clauses, as well as table joins and aliases.

So for example, if we wish to have a table, ***Store\_Information***, that collects the sales information for year 1998, and you already know that the source data resides in the ***Sales\_Information table***, we'll type in:

**INSERT INTO Store\_Information (store\_name, Sales, Date)  
SELECT store\_name, Sales, Date  
FROM Sales\_Information  
WHERE Year(Date) = 1998**

Here I have used the SQL Server syntax to extract the year information out of a date. Other relational databases will have different syntax. For example, in Oracle, you will use to\_char(date,'yyyy')=1998.

**Update Statement** :

nce there's data in the table, we might find that there is a need to modify the data. To do so, we can use the **UPDATE** command. The syntax for this is

**UPDATE "table\_name"  
SET "column\_1" = [new value]  
WHERE {condition}**

For example, say we currently have a table as below:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

and we notice that the sales for Los Angeles on 01/08/1999 is actually $500 instead of $300, and that particular entry needs to be updated. To do so, we use the following SQL query:

**UPDATE Store\_Information  
SET Sales = 500  
WHERE store\_name = "Los Angeles"  
AND Date = "Jan-08-1999"**

The resulting table would look like

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $500 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

In this case, there is only one row that satisfies the condition in the **WHERE** clause. If there are multiple rows that satisfy the condition, all of them will be modified. If no **WHERE** clause is specified, all rows will be modified.

It is also possible to **UPDATE** multiple columns at the same time. The syntax in this case would look like the following:

**UPDATE "table\_name"  
SET column\_1 = [value1], column\_2 = [value2]  
WHERE {condition}**

**Delete From Statement** :

ometimes we may wish to use a query to remove records from a table. To do so, we can use the**DELETE FROM** command. The syntax for this is

**DELETE FROM "table\_name"  
WHERE {condition}**

It is easiest to use an example. Say we currently have a table as below:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

and we decide not to keep any information on Los Angeles in this table. To accomplish this, we type the following SQL:

**DELETE FROM Store\_Information  
WHERE store\_name = "Los Angeles"**

Now the content of table would look like,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| San Diego | $250 | Jan-07-1999 |
| Boston | $700 | Jan-08-1999 |

**Select** :

hat do we use SQL commands for? A common use is to select data from the tables located in a database. Immediately, we see two keywords: we need to **SELECT** information**FROM** a table. (Note that a table is a container that resides in the database where the data is stored. For more information about how to manipulate tables, go to the [**Table Manipulation Section**](http://www.1keydata.com/sql/sqlcreate.html)). Hence we have the most basic SQL query structure:

**SELECT "column\_name" FROM "table\_name"**

To illustrate the above example, assume that we have the following table:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

We shall use this table as an example throughout the tutorial (this table will appear in all sections). To select all the stores in this table, we key in,

**SELECT** store\_name **FROM**Store\_Information

***Result:***

|  |
| --- |
| **store\_name** |
| **Los Angeles** |
| **San Diego** |
| **Los Angeles** |
| **Boston** |

Multiple column names can be selected, as well as multiple table names.

**Distinct** :

he **SELECT** keyword allows us to grab all information from a column (or columns) on a table. This, of course, necessarily mean that there will be redundancies. What if we only want to select each **DISTINCT** element? This is easy to accomplish in SQL. All we need to do is to add **DISTINCT**after **SELECT**. The syntax is as follows:

**SELECT DISTINCT "column\_name"  
FROM "table\_name"**

For example, to select all distinct stores in Table ***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT DISTINCT store\_name FROM Store\_Information**

*Result:*

|  |
| --- |
| **store\_name** |
| **Los Angeles** |
| **San Diego** |
| **Boston** |

**Where** :

ext, we might want to conditionally select the data from a table. For example, we may want to only retrieve stores with sales above $1,000. To do this, we use the**WHERE** keyword. The syntax is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
WHERE "condition"**

For example, to select all stores with sales above $1,000 in Table***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT store\_name  
FROM Store\_Information  
WHERE Sales > 1000**  
  
*Result:*

|  |
| --- |
| **store\_name** |
| **Los Angeles** |

**And Or** :

n the previous section, we have seen that the **WHERE** keyword can be used to conditionally select data from a table. This condition can be a simple condition (like the one presented in the previous section), or it can be a compound condition. Compound conditions are made up of multiple simple conditions connected by **AND** or **OR**. There is no limit to the number of simple conditions that can be present in a single SQL statement.

The syntax for a compound condition is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
WHERE "simple condition"  
{[AND|OR] "simple condition"}+**

The {}+ means that the expression inside the bracket will occur one or more times. Note that **AND** and **OR**can be used interchangeably. In addition, we may use the parenthesis sign () to indicate the order of the condition.

For example, we may wish to select all stores with sales greater than $1,000 or all stores with sales less than $500 but greater than $275 in Table ***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT store\_name  
FROM Store\_Information  
WHERE Sales > 1000  
OR (Sales < 500 AND Sales > 275)**  
  
*Result:*

|  |
| --- |
| **store name** |
| **Los Angeles** |
| **San Francisco** |

**In** :

n SQL, there are two uses of the **IN**keyword, and this section introduces the one that is related to the **WHERE**clause. When used in this context, we know exactly the value of the returned values we want to see for at least one of the columns. The syntax for using the **IN** keyword is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
WHERE "column\_name" IN ('value1', 'value2', ...)**

The number of values in the parenthesis can be one or more, with each values separated by comma. Values can be numerical or characters. If there is only one value inside the parenthesis, this commend is equivalent to

**WHERE "column\_name" = 'value1'**

For example, we may wish to select all records for the Los Angeles and the San Diego stores in Table***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT \*  
FROM Store\_Information  
WHERE store\_name IN ('Los Angeles', 'San Diego')**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **Los Angeles** | **$1500** | **Jan-05-1999** |
| **San Diego** | **$250** | **Jan-07-1999** |

**Between:**

hereas the **IN** keyword help people to limit the selection criteria to one or more discrete values, the **BETWEEN**keyword allows for selecting a range. The syntax for the **BETWEEN** clause is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
WHERE "column\_name" BETWEEN 'value1' AND 'value2'**

This will select all rows whose column has a value between 'value1' and 'value2'.

For example, we may wish to select view all sales information between January 6, 1999, and January 10, 1999, in Table ***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT \*  
FROM Store\_Information  
WHERE Date BETWEEN 'Jan-06-1999' AND 'Jan-10-1999'**

Note that date may be stored in different formats in different databases. This tutorial simply choose one of the formats.

*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **San Diego** | **$250** | **Jan-07-1999** |
| **San Francisco** | **$300** | **Jan-08-1999** |
| **Boston** | **$700** | **Jan-08-1999** |

**Wildcard:**

here are times when we want to match on a string pattern. To do that, we will need to employ the concept of wildcard. In SQL, there are two wildcards:

**%** (percent sign) represents zero, one, or more characters.

**\_** (underscore) represents exactly one character.

Wildcards are used with the **[LIKE](http://www.1keydata.com/sql/sqllike.html)**keyword in SQL.

Below are some wildcard examples:

• 'A\_Z': All string that starts with 'A', another character, and end with 'Z'. For example, 'ABZ' and 'A2Z' would both satisfy the condition, while 'AKKZ' would not (because there are two characters between A and Z instead of one).

• 'ABC%': All strings that start with 'ABC'. For example, 'ABCD' and 'ABCABC' would both satisfy the condition.

• '%XYZ': All strings that end with 'XYZ'. For example, 'WXYZ' and 'ZZXYZ' would both satisfy the condition.

• '%AN%': All strings that contain the pattern 'AN' anywhere. For example, 'LOS ANGELES' and 'SAN FRANCISCO' would both satisfy the condition.

• '\_AN%': All strings that contain a character, then 'AN', followed by anything else. For example, 'SAN FRANCISCO' would satisfy the condition, while 'LOS ANGELES' would not satisfy the condition.

**Like** :

**LIKE** is another keyword that is used in the **WHERE** clause. Basically,**LIKE** allows you to do a search based on a pattern rather than specifying exactly what is desired (as in **IN**) or spell out a range (as in**BETWEEN**). The syntax is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
WHERE "column\_name" LIKE {PATTERN}**

{PATTERN} often consists of wildcards. We saw several examples of [**wildcard**](http://www.1keydata.com/sql/sql-wildcard.html) matching in the previous section. Below we use an example to see how wildcard is used in conjunction with **LIKE**:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| LOS ANGELES | $1500 | Jan-05-1999 |
| SAN DIEGO | $250 | Jan-07-1999 |
| SAN FRANCISCO | $300 | Jan-08-1999 |
| BOSTON | $700 | Jan-08-1999 |

We want to find all stores whose name contains 'AN'. To do so, we key in,

**SELECT \*  
FROM Store\_Information  
WHERE store\_name LIKE '%AN%'**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **LOS ANGELES** | **$1500** | **Jan-05-1999** |
| **SAN DIEGO** | **$250** | **Jan-07-1999** |
| **SAN FRANCISCO** | **$300** | **Jan-08-1999** |

**Order By:**

o far, we have seen how to get data out of a table using **SELECT**and **WHERE** commands. Often, however, we need to list the output in a particular order. This could be in ascending order, in descending order, or could be based on either numerical value or text value. In such cases, we can use the**ORDER BY** keyword to achieve our goal.

The syntax for an **ORDER BY**statement is as follows:

**SELECT "column\_name"  
FROM "table\_name"  
[WHERE "condition"]  
ORDER BY "column\_name" [ASC, DESC]**

The [] means that the **WHERE**statement is optional. However, if a**WHERE** clause exists, it comes before the **ORDER BY** clause. **ASC**means that the results will be shown in ascending order, and**DESC** means that the results will be shown in descending order. If neither is specified, the default is**ASC**.

It is possible to order by more than one column. In this case, the**ORDER BY** clause above becomes

**ORDER BY "column\_name1" [ASC, DESC], "column\_name2" [ASC, DESC]**

Assuming that we choose ascending order for both columns, the output will be ordered in ascending order according to column 1. If there is a tie for the value of column 1, we then sort in ascending order by column 2.

For example, we may wish to list the contents of Table ***Store\_Information*** by dollar amount, in descending order:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT store\_name, Sales, Date  
FROM Store\_Information  
ORDER BY Sales DESC**  
  
*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** | **Sales** | **Date** |
| **Los Angeles** | **$1500** | **Jan-05-1999** |
| **Boston** | **$700** | **Jan-08-1999** |
| **San Francisco** | **$300** | **Jan-08-1999** |
| **San Diego** | **$250** | **Jan-07-1999** |

In addition to column name, we may also use column position (based on the SQL query) to indicate which column we want to apply the **ORDER BY** clause. The first column is 1, second column is 2, and so on. In the above example, we will achieve the same results by the following command:

**SELECT store\_name, Sales, Date  
FROM Store\_Information  
ORDER BY 2 DESC**

**Aggregate Functions:**

ince we have started dealing with numbers, the next natural question to ask is if it is possible to do math on those numbers, such as summing them up or taking their average. The answer is yes! SQL has several arithematic functions, and they are:

• [**AVG**](http://www.1keydata.com/sql/sql-average.html): Average of the column.

• [**COUNT**](http://www.1keydata.com/sql/sqlcount.html): Number of records.

• [**MAX**](http://www.1keydata.com/sql/sql-max.html): Maximum of the column.

• [**MIN**](http://www.1keydata.com/sql/sql-min.html): Minimum of the column.

• [**SUM**](http://www.1keydata.com/sql/sql-sum.html): Sum of the column.

The syntax for using functions is,

**SELECT "function type" ("column\_name")  
FROM "table\_name"**

Examples of how these functions are used are presented individually in the next few pages.

In addition to using functions, it is also possible to use SQL to perform simple tasks such as addition (+) and subtraction (-). For character-type data, there are also several string functions available, such as[**concatenation**](http://www.1keydata.com/sql/sql-concatenate.html), [**trim**](http://www.1keydata.com/sql/sql-trim.html), and **[substring](http://www.1keydata.com/sql/sql-substring.html)**functions. Different RDBMS vendors have different string functions implementations, and it is best to consult the references for your RDBMS to see how these functions are used.

**Average:**

QL uses the AVG() function to calculate the average of a column. The syntax for using this function is,

**SELECT AVG("column\_name")  
FROM "table\_name"**

For example, if we want to get the average of all sales from the following table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we would type in

**SELECT AVG(Sales) FROM Store\_Information**  
  
*Result:*

|  |
| --- |
| **AVG(Sales)** |
| **$687.5** |

$687.5 represents the average of all Sales entries: ($1500 + $250 + $300 + $700) / 4.

**Count** :

nother arithmetic function is **COUNT**. This allows us to **COUNT** up the number of row in a certain table. The syntax is,

**SELECT COUNT("column\_name")  
FROM "table\_name"**

For example, if we want to find the number of store entries in our table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we'd key in

**SELECT COUNT(store\_name)  
FROM Store\_Information**

*Result:*

|  |
| --- |
| **Count(store\_name)** |
| **4** |

**COUNT** and **DISTINCT** can be used together in a statement to fetch the number of distinct entries in a table. For example, if we want to find out the number of distinct stores, we'd type,

**SELECT COUNT(DISTINCT store\_name)  
FROM Store\_Information**

*Result:*

|  |
| --- |
| **Count(DISTINCT store\_name)** |
| **3** |

**MAX Function** :

QL uses the MAX function to find the maximum value in a column. The syntax for using the MAX function is,

**SELECT MAX("column\_name")  
FROM "table\_name"**

For example, if we want to get the highest sales from the following table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we would type in

**SELECT MAX(Sales) FROM Store\_Information**  
  
*Result:*

|  |
| --- |
| **MAX(Sales)** |
| **$1500** |

$1500 represents the maximum value of all Sales entries: $1500, $250, $300, and $700.

**MIN Function:**

QL uses the MIN function to find the maximum value in a column. The syntax for using the MIN function is,

**SELECT MIN("column\_name")  
FROM "table\_name"**

For example, if we want to get the lowest sales from the following table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we would type in

**SELECT MIN(Sales) FROM Store\_Information**  
  
*Result:*

|  |
| --- |
| **MIN(Sales)** |
| **$250** |

$250 represents the minimum value of all Sales entries: $1500, $250, $300, and $700.

**SUM Function :**

he SUM function is used to calculate the total for a column. The syntax is,

**SELECT SUM("column\_name")  
FROM "table\_name"**

For example, if we want to get the sum of all sales from the following table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we would type in

**SELECT SUM(Sales) FROM Store\_Information**  
  
*Result:*

|  |
| --- |
| **SUM(Sales)** |
| **$2750** |

$2750 represents the sum of all Sales entries: $1500 + $250 + $300 + $700.

**Group By** :

ow we return to the aggregate functions. Remember we used the **SUM** keyword to calculate the total sales for all stores? What if we want to calculate the total sales for *each* store? Well, we need to do two things: First, we need to make sure we **select** the store name as well as total sales. Second, we need to make sure that all the sales figures are **grouped by**stores. The corresponding SQL syntax is,

**SELECT "column\_name1", SUM("column\_name2")  
FROM "table\_name"  
GROUP BY "column\_name1"**

Let's illustrate using the following table,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

We want to find total sales for each store. To do so, we would key in,

**SELECT store\_name, SUM(Sales)  
FROM Store\_Information  
GROUP BY store\_name**

*Result:*

|  |  |
| --- | --- |
| **store\_name** | **SUM(Sales)** |
| **Los Angeles** | **$1800** |
| **San Diego** | **$250** |
| **Boston>** | **$700** |

The **GROUP BY** keyword is used when we are selecting multiple columns from a table (or tables) and at least one arithmetic operator appears in the **SELECT** statement. When that happens, we need to**GROUP BY** all the other selected columns, *i.e.,* all columns except the one(s) operated on by the arithmetic operator.

**Having:**

nother thing people may want to do is to limit the output based on the corresponding sum (or any other aggregate functions). For example, we might want to see only the stores with sales over $1,500. Instead of using the **WHERE** clause in the SQL statement, though, we need to use the **HAVING** clause, which is reserved for aggregate functions. The**HAVING** clause is typically placed near the end of the SQL statement, and a SQL statement with the**HAVING** clause may or may not include the **GROUP BY** clause. The syntax for **HAVING** is,

**SELECT "column\_name1", SUM("column\_name2")  
FROM "table\_name"  
GROUP BY "column\_name1"  
HAVING (arithmetic function condition)**

Note: the **GROUP BY** clause is optional.

In our example, table***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we would type,

**SELECT store\_name, SUM(sales)  
FROM Store\_Information  
GROUP BY store\_name  
HAVING SUM(sales) > 1500**

*Result:*

|  |  |  |
| --- | --- | --- |
| **store\_name** |  | **SUM(Sales)** |
| **Los Angeles** |  | **$1800** |

**Alias** :

e next focus on the use of aliases. There are two types of aliases that are used most frequently: column alias and table alias.

In short, column aliases exist to help organizing output. In the previous example, whenever we see total sales, it is listed as **SUM**(sales). While this is comprehensible, we can envision cases where the column heading can be complicated (especially if it involves several arithmetic operations). Using a column alias would greatly make the output much more readable.

The second type of alias is the table alias. This is accomplished by putting an alias directly after the table name in the **FROM** clause. This is convenient when you want to obtain information from two separate tables (the technical term is 'perform joins'). The advantage of using a table alias when doing joins is readily apparent when we talk about joins.

Before we get into joins, though, let's look at the syntax for both the column and table aliases:

**SELECT "table\_alias"."column\_name1" "column\_alias"  
FROM "table\_name" "table\_alias"**

Briefly, both types of aliases are placed directly after the item they alias for, separate by a white space. We again use our table, ***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

We use the same example as that in the [**SQL GROUP BY**](http://www.1keydata.com/sql/sqlgroupby.html) section, except that we have put in both the column alias and the table alias:

**SELECT A1.store\_name Store, SUM(A1.Sales) "Total Sales"  
FROM Store\_Information A1  
GROUP BY A1.store\_name**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Store** |  | **Total Sales** |
| **Los Angeles** |  | **$1800** |
| **San Diego** |  | **$250** |
| **Boston** |  | **$700** |

Notice that difference in the result: the column titles are now different. That is the result of using the column alias. Instead of the somewhat cryptic "Sum(Sales)", we now have "Total Sales", which is much more understandable, as the column header. The advantage of using a table alias is not apparent in this example. However, they will become evident in the [**SQL Joins**](http://www.1keydata.com/sql/sqljoins.html) section.

**AS** :

n the [SQL Alias](http://www.1keydata.com/sql/sqlalias.html) section, we saw that the syntax for using table and column aliases is as follows:

**SELECT "table\_alias"."column\_name1" "column\_alias"  
FROM "table\_name" "table\_alias"**

The keyword **AS** is used to assign an alias to the column or a table. It is insert between the column name and the column alias or between the table name and the table alias. The syntax for using AS is as follows:

**SELECT "table\_alias"."column\_name1" AS "column\_alias"  
FROM "table\_name" AS "table\_alias"**

Let's take a look at the same example as we used in [SQL Alias](http://www.1keydata.com/sql/sqlalias.html). Assume we have the following table,***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

To find total sales by store using **AS**as part of the table and column alias, we type in:

**SELECT A1.store\_name Store, SUM(A1.Sales) AS "Total Sales"  
FROM Store\_Information AS A1  
GROUP BY A1.store\_name**

*Result:*

|  |  |  |
| --- | --- | --- |
| **Store** |  | **Total Sales** |
| **Los Angeles** |  | **$1800** |
| **San Diego** |  | **$250** |
| **Boston** |  | **$700** |

**Join:**

ow we want to look at joins. To do joins correctly in SQL requires many of the elements we have introduced so far. Let's assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

and we want to find out sales by region. We see that table ***Geography***includes information on regions and stores, and table ***Store\_Information***contains sales information for each store. To get the sales information by region, we have to combine the information from the two tables. Examining the two tables, we find that they are linked via the common field, "store\_name". We will first present the SQL statement and explain the use of each segment later:

**SELECT A1.region\_name REGION, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.store\_name = A2.store\_name  
GROUP BY A1.region\_name**

*Result:*

|  |  |  |
| --- | --- | --- |
| **REGION** |  | **SALES** |
| **East** |  | **$700** |
| **West** |  | **$2050** |

The first two lines tell SQL to select two fields, the first one is the field "region\_name" from table***Geography*** (aliased as REGION), and the second one is the sum of the field "Sales" from table***Store\_Information*** (aliased as SALES). Notice how the table aliases are used here: ***Geography***is aliased as A1, and ***Store\_Information*** is aliased as A2. Without the aliasing, the first line would become

**SELECT Geography.region\_name REGION, SUM(Store\_Information.Sales) SALES**

which is much more cumbersome. In essence, table aliases make the entire SQL statement easier to understand, especially when multiple tables are included.

An alternative way to specify a join between tables is to use the JOIN and ON keywords. In the current example, the SQL query would be,

**SELECT A1.region\_name REGION, SUM(A2.Sales) SALES   
FROM Geography A1, Store\_Information A2   
JOIN Store\_Information A2   
ON A1.store\_name = A2.store\_name   
GROUP BY A1.region\_name**

Several different types of JOINs can be performed in SQL. The key ones are as follows:

* [**Inner Join**](http://www.1keydata.com/sql/inner-join.html)
* [**Outer Join**](http://www.1keydata.com/sql/sqlouterjoin.html)
* [**Left Outer Join**](http://www.1keydata.com/sql/left-outer-join)
* [**Cross Join**](http://www.1keydata.com/sql/cross-join.html)

The following sections explain each JOIN type in detail.

**Inner Join** :

In inner join in SQL returns rows where there is at least one match on both tables. Let's assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

We want to find out sales by store, and we only want to see stores with sales listed in the report. To do this, we can use the following SQL statement using INNER JOIN:

**SELECT A1.store\_name STORE, SUM(A2.Sales) SALES   
FROM Geography A1   
INNER JOIN Store\_Information A2   
ON A1.store\_name = A2.store\_name   
GROUP BY A1.store\_name**

*Result:*

|  |  |  |
| --- | --- | --- |
| **STORE** |  | **SALES** |
| **Los Angeles** |  | **$1800** |
| **San Diego** |  | **$250** |
| **Boston** |  | **$700** |

By using INNER JOIN, the result shows 3 stores, even though we are selecting from the***Geography*** table, which has 4 rows. The row "New York" is not selected because it is not present in the ***Store\_Information*** table.

**Outer Join:**

reviously, we had looked at left join, or inner join, where we select rows common to the participating tables to a join. What about the cases where we are interested in selecting elements in a table regardless of whether they are present in the second table? We will now need to use the **SQL OUTER JOIN**command.

The syntax for performing an outer join in SQL is database-dependent. For example, in Oracle, we will place an "(+)" in the **WHERE** clause on the other side of the table for which we want to include all the rows.

Let's assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

and we want to find out the sales amount for all of the stores. If we do a regular join, we will not be able to get what we want because we will have missed "New York," since it does not appear in the ***Store\_Information***table. Therefore, we need to perform an outer join on the two tables above:

**SELECT A1.store\_name, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.store\_name = A2.store\_name (+)  
GROUP BY A1.store\_name**

Note that in this case, we are using the Oracle syntax for outer join.

*Result:*

|  |  |
| --- | --- |
| **store\_name** | **SALES** |
| **Boston** | **$700** |
| **New York** |  |
| **Los Angeles** | **$1800** |
| **San Diego** | **$250** |

Note: NULL is returned when there is no match on the second table. In this case, "New York" does not appear in the table ***Store\_Information***, thus its corresponding "SALES" column is NULL.

**Left Outer Join :**

n an left outer join, all rows from the first table mentioned in the SQL query is selected, regardless whether there is a matching row on the second table mentioned in the SQL query. Let's assume that we have the following two tables,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

We want to find out sales by store, and we want to see the results for all stores regardless whether there is a sale in the ***Store\_Information*** table. To do this, we can use the following SQL statement using LEFT OUTER JOIN:

**SELECT A1.store\_name STORE, SUM(A2.Sales) SALES   
FROM Geography A1   
LEFT OUTER JOIN Store\_Information A2   
ON A1.store\_name = A2.store\_name   
GROUP BY A1.store\_name**

*Result:*

|  |  |  |
| --- | --- | --- |
| **STORE** |  | **SALES** |
| **Los Angeles** |  | **$1800** |
| **San Diego** |  | **$250** |
| **New York** |  | **NULL** |
| **Boston** |  | **$700** |

By using LEFT OUTER JOIN, all four rows in the ***Geography*** table is listed. Since there is no match for "New York" in the ***Store\_Information*** table, the Sales total for "New York" is NULL. Note that it is NULL and not 0, as NULL indicates there is no match.

**Cross Join:**

 cross join (also called a Cartesian join) is a join of tables without specifying the join condition. In this scenario, the query would return all possible combination of the tables in the SQL query. To see this in action, let's use the following example:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

The following SQL statement is a Cartesian join between the***Store\_Information*** and the***Geography*** tables:

**SELECT A1.store\_name STORE1, A2.store\_name STORE2, A2.Sales SALES   
FROM Geography A1   
JOIN Store\_Information A2**

*Result:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STORE1** |  | **STORE2** |  | **SALES** |
| **Boston** |  | **Los Angeles** |  | **$1500** |
|  |  |  |  |  |
| **New York** |  | **Los Angeles** |  | **$1500** |
|  |  |  |  |  |
| **Los Angeles** |  | **Los Angeles** |  | **$1500** |
|  |  |  |  |  |
| **San Diego** |  | **Los Angeles** |  | **$1500** |
|  |  |  |  |  |
| **Boston** |  | **San Diego** |  | **$250** |
|  |  |  |  |  |
| **New York** |  | **San Diego** |  | **$250** |
|  |  |  |  |  |
| **Los Angeles** |  | **San Diego** |  | **$250** |
|  |  |  |  |  |
| **San Diego** |  | **San Diego** |  | **$250** |
|  |  |  |  |  |
| **Boston** |  | **Los Angeles** |  | **$300** |
|  |  |  |  |  |
| **New York** |  | **Los Angeles** |  | **$300** |
|  |  |  |  |  |
| **Los Angeles** |  | **Los Angeles** |  | **$300** |
|  |  |  |  |  |
| **San Diego** |  | **Los Angeles** |  | **$300** |
|  |  |  |  |  |
| **Boston** |  | **Boston** |  | **$700** |
|  |  |  |  |  |
| **New York** |  | **Boston** |  | **$700** |
|  |  |  |  |  |
| **Los Angeles** |  | **Boston** |  | **$700** |
|  |  |  |  |  |
| **San Diego** |  | **Boston** |  | **$700** |
|  |  |  |  |  |

An alternative way of specifying a cross join is,

**SELECT A1.store\_name STORE1, A2.store\_name STORE2, A2.Sales SALES   
FROM Geography A1, Store\_Information A2**

A cross join is seldom the desired result. Rather, it is an indication that some required join condition is missing in the SQL query.

**Select Unique** :

he **SELECT UNIQUE** term is an Oracle-only SQL statement. It is equivalent to [**SELECT DISTINCT**](http://www.1keydata.com/sql/sqldistinct.html). The syntax for **SELECT UNIQUE** is as follows:

**SELECT UNIQUE "column\_name"  
FROM "table\_name"**

For example, to select all distinct stores in Table ***Store\_Information***,

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| store\_name | Sales | Date |
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

we key in,

**SELECT UNIQUE Sales FROM Store\_Information**

*Result:*

|  |
| --- |
| **store\_name** |
| **$1500** |
| **$250** |
| **$300** |
| **$700** |

**Concatenate:**

ometimes it is necessary to combine together (concatenate) the results from several different fields. Each database provides a way to do this:

* MySQL: CONCAT()
* Oracle: CONCAT(), ||
* SQL Server: +

The syntax for CONCAT() is as follows:

**CONCAT(str1, str2, str3, ...)**: Concatenate str1, str2, str3, and any other strings together. Please note the Oracle CONCAT() function only allows two arguments -- only two strings can be put together at a time using this function. However, it is possible to concatenate more than two strings at a time in Oracle using '||'.

Let's look at some examples. Assume we have the following table:

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

Example 1:

**MySQL/Oracle**:   
**SELECT CONCAT(region\_name,store\_name) FROM Geography   
WHERE store\_name = 'Boston';**

*Result*:

**'EastBoston'**

Example 2:

**Oracle**:   
**SELECT region\_name || ' ' || store\_name FROM Geography   
WHERE store\_name = 'Boston';**

*Result*:

**'East Boston'**

Example 3:

**SQL Server**:   
**SELECT region\_name + ' ' + store\_name FROM Geography   
WHERE store\_name = 'Boston';**

*Result*:

**'East Boston'**

**Substring** :

he Substring function in SQL is used to grab a portion of the stored data. This function is called differently for the different databases:

* MySQL: SUBSTR(), SUBSTRING()
* Oracle: SUBSTR()
* SQL Server: SUBSTRING()

The most frequent uses are as follows (we will use SUBSTR() here):

**SUBSTR(str,pos)**: Select all characters from <str> starting with position <pos>. Note that this syntax is not supported in SQL Server.

**SUBSTR(str,pos,len)**: Starting with the <pos>th character in string <str> and select the next <len> characters.

Assume we have the following table:

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

Example 1:

**SELECT SUBSTR(store\_name, 3)   
FROM Geography   
WHERE store\_name = 'Los Angeles';**

*Result*:

**'s Angeles'**

Example 2:

**SELECT SUBSTR(store\_name,2,4)   
FROM Geography   
WHERE store\_name = 'San Diego';**

*Result*:

**'an D'**

**Trim** :

he TRIM function in SQL is used to remove specified prefix or suffix from a string. The most common pattern being removed is white spaces. This function is called differently in different databases:

* MySQL: TRIM(), RTRIM(), LTRIM()
* Oracle: RTRIM(), LTRIM()
* SQL Server: RTRIM(), LTRIM()

The syntax for these trim functions are:

**TRIM([[LOCATION] [remstr] FROM ] str)**: [LOCATION] can be either LEADING, TRAILING, or BOTH. This function gets rid of the [remstr] pattern from either the beginning of the string or the end of the string, or both. If no [remstr] is specified, white spaces are removed.

**LTRIM(str)**: Removes all white spaces from the beginning of the string.

**RTRIM(str)**: Removes all white spaces at the end of the string.

Example 1:

**SELECT TRIM('   Sample   ');**

*Result*:

**'Sample'**

Example 2:

**SELECT LTRIM('   Sample   ');**

*Result*:

**'Sample   '**

Example 3:

**SELECT RTRIM('   Sample   ');**

*Result*:

**'   Sample'**

**Length Function** :

he Length function in SQL is used to get the length of a string. This function is called differently for the different databases:

* MySQL: LENGTH()
* Oracle: LENGTH()
* SQL Server: LEN()

The syntax for the Length function is as follows:

**Length(str)**: Find the length of the string *str*.

Let's take a look at some examples. Assume we have the following table:

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

Example 1:

**SELECT Length(store\_name)   
FROM Geography   
WHERE store\_name = 'Los Angeles';**

*Result*:

**11**

Example 2:

**SELECT region\_name, Length(region\_name)   
FROM Geography;**

*Result*:

|  |  |
| --- | --- |
| **region\_name** | **Length(region\_name)** |
| **East** | **4** |
| **East** | **4** |
| **West** | **4** |
| **West** | **4** |

**Replace Function** :

he **Replace** function in SQL is used to update the content of a string. The function call is REPLACE() for MySQL, Oracle, and SQL Server. The syntax of the Replace function is:

**Replace(str1, str2, str3)**: In str1, find where str2 occurs, and replace it with str3.

Assume we have the following table:

Table ***Geography***

|  |  |
| --- | --- |
| region\_name | store\_name |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

If we apply the following Replace function:

**SELECT REPLACE(region\_name, 'ast', 'astern')  
FROM Geography;**

*Result*:

|  |
| --- |
| **region\_name** |
| **Eastern** |
| **Eastern** |
| **West** |
| **West** |

**Dateadd Function:**

he DATEADD function is used to add an interval to a date. This function is available in SQL Server.

The usage for the DATEADD function is

**DATEADD (datepart, number, expression)**

where the data type of <expression> is some type of date, time, or datetime. <number> is an integer (can be positive or negative). <datepart> can be one of the following:

|  |  |
| --- | --- |
| datepart | abbreviation |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |
| TZoffset | tz |
| ISO\_WEEK | isowk, isoww |

The result returned has the same data type as <expression>.

Example: The SQL statement

**SELECT DATEADD(day, 10,'2000-01-05 00:05:00.000');**

yields the following result:

**'2000-01-15 00:05:00.000'**

**Datediff Function** :

he DATEDIFF function is used to calculate the difference between two days, and is used in MySQL and SQL Server. The syntax for this date function is different between these two databases, so each one is discussed below:

**MySQL**:

The usage for the DATEDIFF function in MySQL is

**DATEDIFF (expression1, expression2)**

where the data type of <expression1> and <expression2> is either DATE or DATETIME. The result is <expression1> - <expression2>.

Example: The SQL statement

**SELECT DATEDIFF('2000-01-10','2000-01-05');**

yields the following result:

**5**

This is because 2000-01-10 is 5 days after 2000-01-05.

**SQL Server**:

The usage for the DATEDIFF function in SQL Server is

**DATEDIFF (datepart, expression1, expression2)**

where the data type of <expression1> and <expression2> is some type of date, time, or datetime. The result is <expression2> - <expression1>. datepart can be one of the following:

|  |  |
| --- | --- |
| datepart | abbreviation |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |
| TZoffset | tz |
| ISO\_WEEK | isowk, isoww |

Example: The SQL statement

**SELECT DATEDIFF(day, '2000-01-10','2000-01-05');**

yields the following result:

**-5**

This is because 2000-01-05 is 5 days before 2000-01-10.

**Datepart Function:**

**DATEPART** is a SQL Server function that extracts a specific part of the date/time value. Its syntax is as follows:

**DATEPART (part\_of\_day, expression)**

where part\_of\_day can have the following:

|  |  |
| --- | --- |
| datepart | abbreviation |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |
| TZoffset | tz |
| ISO\_WEEK | isowk, isoww |

Example 1:

**SELECT DATEPART (yyyy,'2000-01-20');**

*Result*:

**2001**

Example 2:

**SELECT DATEPART(dy, '2000-02-10');**

*Result*:

**41**

2000-02-01 is the 41st day in the year 2000.

**Getdate Function:**

he GETDATE function is used to retrieve the current database system time in SQL Server. Its syntax is

**GETDATE()**

GETDATE does not require any argument.

Example: The SQL statement

**SELECT GETDATE();**

yields the following result:

**'2000-03-15 00:05:02.123'**

GETDATE function is most useful when we need to record the time a particular transaction happens. In SQL Server, we simply insert the value of the GETDATE() function into the table to achieve this. We can also set the default value of a column to be GETDATE() to achieve the same purpose.

The Oracle and MySQL equivalent of GETDATE is [**SYSDATE**](http://www.1keydata.com/sql/sql-sysdate.html).

**Sysdate Function** :

he SYSDATE function is used to retrieve the current database system time in Oracle and MySQL.

**Oracle**:

The syntax of SYSDATE in Oracle is simply

**SYSDATE**

It does not require any argument.

Example: The SQL statement

**SELECT SYSDATE FROM DUAL;**

yields the following result:

**'16-JAN-2000'**

**MySQL**:

The syntax of SYSDATE in MySQL is simply

**SYSDATE()**

It does not require any argument.

Example: The SQL statement

**SELECT SYSDATE();**

yields the following result:

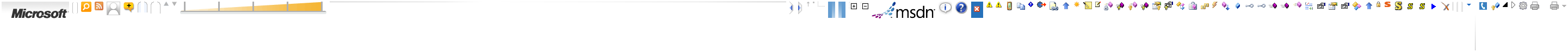
**'2000-01-16 09:06:22'**

The SQL Server equivalent of SYSDATE is [**GETDATE**](http://www.1keydata.com/sql/sql-getdate.html).

# Cascading Referential Integrity Constraints

**SQL Server 2008 R2**

[Other Versions](javascript:;)



1 out of 2 rated this helpful [Rate this topic](http://msdn.microsoft.com/en-us/library/ms186973.aspx#feedback)

By using cascading referential integrity constraints, you can define the actions that the SQL Server takes when a user tries to delete or update a key to which existing foreign keys point.

The REFERENCES clauses of the [CREATE TABLE](http://msdn.microsoft.com/en-us/library/ms174979.aspx) and [ALTER TABLE](http://msdn.microsoft.com/en-us/library/ms190273.aspx) statements support the ON DELETE and ON UPDATE clauses. Cascading actions can also be defined by using the[Foreign Key Relationships dialog box](http://msdn.microsoft.com/en-us/library/ms177288.aspx):

* [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
* [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]

NO ACTION is the default if ON DELETE or ON UPDATE is not specified.

ON DELETE NO ACTION

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE statement is rolled back.

ON UPDATE NO ACTION

Specifies that if an attempt is made to update a key value in a row whose key is referenced by foreign keys in existing rows in other tables, an error is raised and the UPDATE statement is rolled back.

CASCADE, SET NULL and SET DEFAULT allow for deletions or updates of key values to affect the tables defined to have foreign key relationships that can be traced back to the table on which the modification is performed. If cascading referential actions have also been defined on the target tables, the specified cascading actions also apply for those rows deleted or updated. CASCADE cannot be specified for any foreign keys or primary keys that have a **timestamp** column.

ON DELETE CASCADE

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, all rows that contain those foreign keys are also deleted.

ON UPDATE CASCADE

Specifies that if an attempt is made to update a key value in a row, where the key value is referenced by foreign keys in existing rows in other tables, all the values that make up the foreign key are also updated to the new value specified for the key.

|  |
| --- |
| **NoteNote** |
| CASCADE cannot be specified if a **timestamp** column is part of either the foreign key or the referenced key. |

ON DELETE SET NULL

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, all the values that make up the foreign key in the rows that are referenced are set to NULL. All foreign key columns of the target table must be nullable for this constraint to execute.

ON UPDATE SET NULL

Specifies that if an attempt is made to update a row with a key referenced by foreign keys in existing rows in other tables, all the values that make up the foreign key in the rows that are referenced are set to NULL. All foreign key columns of the target table must be nullable for this constraint to execute.

ON DELETE SET DEFAULT

Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, all the values that make up the foreign key in the rows that are referenced are set to their default value. All foreign key columns of the target table must have a default definition for this constraint to execute. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column. Any nonnull values that are set because of ON DELETE SET DEFAULT must have corresponding values in the primary table to maintain the validity of the foreign key constraint.

ON UPDATE SET DEFAULT

Specifies that if an attempt is made to update a row with a key referenced by foreign keys in existing rows in other tables, all the values that make up the foreign key in the rows that are referenced are set to their default value. All foreign key columns of the target table must have a default definition for this constraint to execute. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column. Any non-null values that are set because of ON UPDATE SET DEFAULT must have corresponding values in the primary table to maintain the validity of the foreign key constraint.

Consider the **FK\_ProductVendor\_Vendor\_VendorID** constraint on the **Purchasing.ProductVendor** table in AdventureWorks2008R2. This constraint establishes a foreign key relationship from the **VendorID** column in the **ProductVendor** table to the **VendorID** primary key column in the **Purchasing.Vendor** table. If ON DELETE CASCADE is specified for the constraint, deleting the row in **Vendor** where **VendorID** equals 100 also deletes the three rows in **ProductVendor** where **VendorID** equals 100. If ON UPDATE CASCADE is specified for the constraint, updating the **VendorID** value in the **Vendor** table from 100 to 155 also updates the **VendorID** values in the three rows in **ProductVendor** whose**VendorID** values currently equal 100.

ON DELETE CASCADE cannot be specified for a table that has an INSTEAD OF DELETE trigger. For tables that have INSTEAD OF UPDATE triggers, the following cannot be specified: ON DELETE SET NULL, ON DELETE SET DEFAULT, ON UPDATE CASCADE, ON UPDATE SET NULL, and ON UDATE SET DEFAULT.

[[E:\ALL TYPE OF NOTES\ORACLE PDF\Sql examples\Cascading Referential Integrity Constraints_files\030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))**Multiple Cascading Actions**](javascript:void(0))

Individual DELETE or UPDATE statements can start a series of cascading referential actions. For example, a database contains three tables: **TableA**, **TableB**, and **TableC**. A foreign key in **TableB** is defined with ON DELETE CASCADE against the primary key in **TableA**. A foreign key in **TableC** is defined with ON DELETE CASCADE against the primary key in **TableB**. If a DELETE statement deletes rows in **TableA**, the operation also deletes any rows in **TableB** that have foreign keys matching the deleted primary keys in**TableA**, and then deletes any rows in **TableC** that have foreign keys that match the deleted primary keys in **TableB**.

The series of cascading referential actions triggered by a single DELETE or UPDATE must form a tree that contains no circular references. No table can appear more than one time in the list of all cascading referential actions that result from the DELETE or UPDATE. Also, the tree of cascading referential actions must not have more than one path to any specified table. Any branch of the tree is ended when it encounters a table for which NO ACTION has been specified or is the default.

[[E:\ALL TYPE OF NOTES\ORACLE PDF\Sql examples\Cascading Referential Integrity Constraints_files\030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))**Triggers and Cascading Referential Actions**](javascript:void(0))

Cascading referential actions fire the AFTER UPDATE or AFTER DELETE triggers in the following manner:

All the cascading referential actions directly caused by the original DELETE or UPDATE are performed first.

If there are any AFTER triggers defined on the affected tables, these triggers fire after all cascading actions are performed. These triggers fire in opposite order of the cascading action. If there are multiple triggers on a single table, they fire in random order, unless there is a dedicated first or last trigger for the table. This order is as specified by using [sp\_settriggerorder](http://msdn.microsoft.com/en-us/library/ms186762.aspx).

If multiple cascading chains originate from the table that was the direct target of an UPDATE or DELETE action, the order in which these chains fire their respective triggers is unspecified. However, one chain always fires all its triggers before another chain starts firing.

An AFTER trigger on the table that is the direct target of an UPDATE or DELETE action fires regardless of whether any rows are affected. There are no other tables affected by cascading in this case.

If any one of the previous triggers perform UPDATE or DELETE operations on other tables, these actions can start secondary cascading chains. These secondary chains are processed for each UPDATE or DELETE operation at a time after all triggers on all primary chains fire. This process may be recursively repeated for subsequent UPDATE or DELETE operations.

Performing CREATE, ALTER, DELETE, or other data definition language (DDL) operations inside the triggers may cause DDL triggers to fire. This may subsequently perform DELETE or UPDATE operations that start additional cascading chains and triggers.

If an error is generated inside any particular cascading referential action chain, an error is raised, no AFTER triggers are fired in that chain, and the DELETE or UPDATE operation that created the chain is rolled back.

A table that has an INSTEAD OF trigger cannot also have a REFERENCES clause that specifies a cascading action. However, an AFTER trigger on a table targeted by a cascading action can execute an INSERT, UPDATE, or DELETE statement on another table or view that fires an INSTEAD OF trigger defined on that object.

[[E:\ALL TYPE OF NOTES\ORACLE PDF\Sql examples\Cascading Referential Integrity Constraints_files\030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))**Cascading Referential Constraints Catalog Information**](javascript:void(0))

Querying the **sys.foreign\_keys** catalog view returns the following values that indicate the cascading referential constraint specified for a foreign key.

|  |  |
| --- | --- |
| Value | Description |
| 0 | NO ACTION |
| 1 | CASCADE |
| 2 | SET NULL |
| 3 | SET DEFAULT |

The **UPDATE\_RULE** and **DELETE\_RULE** columns returned by **sp\_fkeys** and **sp\_foreignkeys** return 0 when CASCADE, SET NULL, or SET DEFAULT is specified; and return 1 when NO ACTION is specified or is the default.

When a foreign key is specified as the object of **sp\_help**, the output result set contains the following columns.

|  |  |  |
| --- | --- | --- |
| Column name | Data type | Description |
| **delete\_action** | **nvarchar(** 9**)** | Indicates whether the delete action is CASCADE, SET NULL, SET DEFAULT, NO ACTION, or N/A (not applicable). |
| **update\_action** | **nvarchar(** 9**)** | Indicates whether the update action is CASCADE, SET NULL, SET DEFAULT, NO ACTION, or N/A (not applicable). |

[[E:\ALL TYPE OF NOTES\ORACLE PDF\Sql examples\Cascading Referential Integrity Constraints_files\030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))**See Also**](javascript:void(0))

#### Reference

[CREATE TABLE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms174979.aspx)

[ALTER TABLE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms190273.aspx)

[DROP TABLE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms173790.aspx)

[sp\_settriggerorder (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186762.aspx)

[sys.foreign\_keys (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189807.aspx)

[sp\_fkeys (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms175090.aspx)

[sp\_foreignkeys (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187337.aspx)