Join

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as follows:

```
SQL> SELECT ID, NAME, AGE, AMOUNT
        FROM CUSTOMERS, ORDERS
        WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

SQL Join Types:

There are different types of joins available in SQL:

- INNER JOIN: returns rows when there is a match in both tables.
- LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN: returns rows when there is a match in one of the tables.

- SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

Displaying Data from Multiple Tables

The related tables of a large database are linked through the use of foreign and primary keys or what are often referred to as common columns. The ability to join tables will enable you to add more meaning to the result table that is produced. For 'n' number tables to be joined in a query, minimum (n-1) join conditions are necessary. Based on the join conditions, Oracle combines the matching pair of rows and displays the one which satisfies the join condition.

Joins are classified as below

- Natural join (also known as an equijoin or a simple join) - Creates a join by using a commonly named and defined column.

- Non-equality join - Joins tables when there are no equivalent rows in the tables to be joined-for example, to match values in one column of a table with a range of values in another table.

- Self-join - Joins a table to itself.

- Outer join - Includes records of a table in output when there's no matching record in the other table.

- Cartesian join (also known as a Cartesian product or cross join) - Replicates each row from the first table with every row from the second table.Creates a join between tables by displaying every possible record combination.

Natural Join

The NATURAL keyword can simplify the syntax of an equijoin.A NATURAL JOIN is possible whenever two (or more) tables have columns with the same name,and the columns are join compatible, i.e., the columns have a shared domain of values.The join operation joins rows from the tables that have equal column values for the same named columns.

Consider the one-to-many relationship between the DEPARTMENTS and EMPLOYEES tables.Each table has a column named DEPARTMENT_ID.This column is the primary key of the DEPARTMENTS table and a foreign key of the EMPLOYEES table.

```
SELECT E.first_name NAME,D.department_name DNAME
FROM employees E NATURAL JOIN departments D;

FIRST_NAME DNAME
---------- ------
MILLER     DEPT 1
JOHN       DEPT 1
MARTIN     DEPT 2
EDWIN      DEPT 2
```

The below SELECT query joins the two tables by explicitly specifying the join condition with the ON keyword.

```
SELECT E.first_name NAME,D.department_name DNAME
FROM employees E JOIN departments D
ON (E.department_id = D.department_id);
```

There are some limitations regarding the NATURAL JOIN.You cannot specify a LOB column with a NATURAL JOIN.Also, columns involved in the join cannot be qualified by a table name or alias.

USING Clause

Using Natural joins, Oracle implicitly identify columns to form the basis of join. Many situations require explicit declaration of join conditions. In such cases, we use USING clause to specify the joining criteria. Since, USING clause joins the tables based on equality of columns, it is also known as Equijoin. They are also known as Inner joins or simple joins.

Syntax:

```
SELECT <column list>
FROM   TABLE1   JOIN   TABLE2
USING (column name)
```

Consider the below SELECT query, EMPLOYEES table and DEPARTMENTS table are joined using the common column DEPARTMENT_ID.

```
SELECT E.first_name NAME,D.department_name DNAME
FROM employees E JOIN departments D
USING (department_id);
```

Self Join

A SELF-JOIN operation produces a result table when the relationship of interest exists among rows that are stored within a single table. In other words, when a table is joined to itself, the join is known as Self Join.

Consider EMPLOYEES table,which contains employee and their reporting managers.To find manager's name for an employee would require a join on the EMP table itself. This is a typical candidate for Self Join.

```
SELECT e1.FirstName Manager,e2.FirstName Employee
FROM employees e1 JOIN employees e2
ON (e1.employee_id = e2.manager_id)
ORDER BY e2.manager_id DESC;
```

Non Equijoins

A non-equality join is used when the related columns can't be joined with an equal sign-meaning there are no equivalent rows in the tables to be joined.A non-equality join enables you to store a range's minimum value in one column of a record and the maximum value in another column. So instead of finding a column-tocolumn match, you can use a non-equality join to determine whether the item being shipped falls between minimum and maximum ranges in the columns.If the join does find a matching range for the item, the corresponding shipping fee can be returned in the results. As with the traditional method of equality joins, a non-equality join can be performed in a WHERE clause. In addition, the JOIN keyword can be used with the ON clause to specify relevant columns for the join.

```
SELECT E.first_name,
            J.job_hisal,
            J.job_losal,
            E.salary
     FROM employees E JOIN job_sal J
     ON (E.salary BETWEEN J.job_losal AND J.job_losal);
```

We can make use all comparison parameter discussed earlier like equality and inequality operators, BETWEEN, IS NULL, IS NOT NULL, and RELATIONAL.

Outer Joins

An Outer Join is used to identify situations where rows in one table do not match rows in a second table, even though the two tables are related.

There are three types of outer joins: the LEFT, RIGHT, and FULL OUTER JOIN. They all begin with an INNER JOIN, and then they add back some of the rows that have been dropped. A LEFT OUTER JOIN adds back all the rows that are dropped from the first (left) table in the join condition, and output columns from the second (right) table are set to NULL. A RIGHT OUTER JOIN adds back all the rows that are dropped from the second (right) table in the join condition, and output columns from the first (left) table are set to NULL. The FULL OUTER JOIN adds back all the rows that are dropped from both the tables.

Right Outer Join

A RIGHT OUTER JOIN adds back all the rows that are dropped from the second (right) table in the join condition, and output columns from the first (left) table are set to NULL. Note the below query lists the employees and their corresponding departments. Also no employee has been assigned to department 30.

```
SELECT E.first_name, E.salary, D.department_id
FROM employees E, departments D
WHERE E.DEPARTMENT_ID (+) = D.DEPARTMENT_ID;

FIRST_NAME SALARY     DEPARTMENT_ID
---------- ---------- ----------
JOHN       6000       10
EDWIN      2000       20
MILLER     2500       10
MARTIN     4000       20
                      30
```

Left Outer Join

A LEFT OUTER JOIN adds back all the rows that are dropped from the first (left) table in the join condition, and output columns from the second (right) table are set to NULL. The query demonstrated above can be used to demonstrate left outer join, by exchanging the position of (+) sign.

```
SELECT E.first_name, E.salary, D.department_id
FROM employees E, departments D
WHERE   D.DEPARTMENT_ID = E.DEPARTMENT_ID (+);

FIRST_NAME SALARY     DEPARTMENT_ID
---------- ---------- ----------
JOHN       6000       10
EDWIN      2000       20
MILLER     2500       10
MARTIN     4000       20
                      30
```

Full Outer Join

The FULL OUTER JOIN adds back all the rows that are dropped from both the tables. Below query shows lists the employees and their departments. Note that employee 'MAN' has not been assigned any department till now (its NULL) and department 30 is not assigned to any employee.

```
SELECT  nvl (e.first_name,'-') first_name, nvl (to_char (d.department_id),'-')
department_id
FROM employee e FULL OUTER JOIN department d
ON e. depARTMENT_ID = d. depARTMENT_ID;

FIRST_NAME DEPARTMENT_ID
---------- --------------------
MAN        -
JOHN       10
EDWIN      20
MILLER     10
MARTIN     20
-          30

6 rows selected.
```

Cartesian product or Cross join

For two entities A and B, A * B is known as Cartesian product. A Cartesian product consists of all possible combinations of the rows from each of the tables. Therefore, when a table with 10 rows is joined with a table with 20 rows, the Cartesian product is 200 rows (10 * 20 = 200).For example, joining the employee table with eight rows and the department table with three rows will produce a Cartesian product table of 24 rows (8 * 3 = 24).

Cross join refers to the Cartesian product of two tables. It produces cross product of two tables. The above query can be written using CROSS JOIN clause.

A Cartesian product result table is normally not very useful. In fact, such a result table can be terribly misleading. If you execute the below query for the EMPLOYEES and DEPARTMENTS tables, the result table implies that every employee has a relationship with every department, and we know that this is simply not the case!

```
SELECT E.first_name, D.DNAME
FROM employees E,departments D;
```

Cross join can be written as,

```
SELECT E.first_name, D.DNAME
FROM employees E CROSS JOIN departments D;
```

Constraints

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL. These constraints have already been discussed in SQL - RDBMS Concepts chapter but its worth to revise them at this point.

- NOT NULL Constraint: Ensures that a column cannot have NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all values in a column are different.
- PRIMARY Key: Uniquely identified each rows/records in a database table.
- FOREIGN Key: Uniquely identified a rows/records in any another database table.
- CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.
- INDEX: Use to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

Dropping Constraints:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

Integrity Constraints:

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in referential integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints mentioned above.

Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is as follows:

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command:

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

You can check INDEX Constraint chapter to see actual examples on Indexes.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in similar way as you query an actual table. Following is the example:

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result:

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following is an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION:

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View:

A view can be updated under certain conditions:

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above-mentioned rules then you can update a view. Following is an example to update the age of Ramesh:

```
SQL > UPDATE CUSTOMERS_VIEW
      SET AGE = 35
      WHERE name='Ramesh';
```

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View:

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here we can not insert rows in CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in similar way as you insert them in a table.

Deleting Rows into a View:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
      WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Dropping Views:

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple as given below:

```
DROP VIEW view_name;
```

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

```
DROP VIEW CUSTOMERS_VIEW;
```

Date Functions

Following is a list of all important Date and Time related functions available through SQL. There are various other functions supported by your RDBMS. Given list is based on MySQL RDBMS.

Name	Description
ADDDATE()	Adds dates
ADDTIME()	Adds time
CONVERT_TZ()	Converts from one timezone to another
CURDATE()	Returns the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Returns the current time
DATE_ADD()	Adds two dates
DATE_FORMAT()	Formats date as specified
DATE_SUB()	Subtracts two dates
DATE()	Extracts the date part of a date or datetime expression
DATEDIFF()	Subtracts two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Returns the name of the weekday
DAYOFMONTH()	Returns the day of the month (1-31)
DAYOFWEEK()	Returns the weekday index of the argument
DAYOFYEAR()	Returns the day of the year (1-366)
EXTRACT	Extracts part of a date
FROM_DAYS()	Converts a day number to a date
FROM_UNIXTIME()	Formats date as a UNIX timestamp
HOUR()	Extracts the hour
LAST_DAY	Returns the last day of the month for the argument
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Creates a date from the year and day of year
MAKETIME	MAKETIME()
MICROSECOND()	Returns the microseconds from argument

MINUTE()	Returns the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Returns the name of the month
NOW()	Returns the current date and time
PERIOD_ADD()	Adds a period to a year-month
PERIOD_DIFF()	Returns the number of months between periods
QUARTER()	Returns the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format
SECOND()	Returns the second (0-59)
STR_TO_DATE()	Converts a string to a date
SUBDATE()	When invoked with three arguments a synonym for DATE_SUB()
SUBTIME()	Subtracts times
SYSDATE()	Returns the time at which the function executes
TIME_FORMAT()	Formats as time
TIME_TO_SEC()	Returns the argument converted to seconds
TIME()	Extracts the time portion of the expression passed
TIMEDIFF()	Subtracts time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression. With two arguments, the sum of the arguments
TIMESTAMPADD()	Adds an interval to a datetime expression
TIMESTAMPDIFF()	Subtracts an interval from a datetime expression
TO_DAYS()	Returns the date argument converted to days
UNIX_TIMESTAMP()	Returns a UNIX timestamp
UTC_DATE()	Returns the current UTC date
UTC_TIME()	Returns the current UTC time
UTC_TIMESTAMP()	Returns the current UTC date and time
WEEK()	Returns the week number
WEEKDAY()	Returns the weekday index
WEEKOFYEAR()	Returns the calendar week of the date (1-53)
YEAR()	Returns the year
YEARWEEK()	Returns the year and week

ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE_ADD(). The related function SUBDATE() is a synonym for DATE_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE_ADD().

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)                 |
+---------------------------------------------------------+
| 1998-02-02                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------------+
| ADDDATE('1998-01-02', INTERVAL 31 DAY)                  |
+---------------------------------------------------------+
| 1998-02-02                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

When invoked with the days form of the second argument, MySQL treats it as an integer number of days to be added to expr.

```
mysql> SELECT ADDDATE('1998-01-02', 31);
+---------------------------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)                 |
+---------------------------------------------------------+
| 1998-02-02                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

ADDTIME(expr1,expr2)

ADDTIME() adds expr2 to expr1 and returns the result. expr1 is a time or datetime expression, and expr2 is a time expression.

```
mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');
+---------------------------------------------------------+
| DATE_ADD('1997-12-31 23:59:59.999999','1 1:1:1.000002') |
+---------------------------------------------------------+
| 1998-01-02 01:01:01.000001                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

CONVERT_TZ(dt,from_tz,to_tz)

This converts a datetime value dt from the time zone given by from_tz to the time zone given by to_tz and returns the resulting value. This function returns NULL if the arguments are invalid.

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
+---------------------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','GMT','MET')           |
+---------------------------------------------------------+
| 2004-01-01 13:00:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00');
+---------------------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00')     |
+---------------------------------------------------------+
| 2004-01-01 22:00:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

CURDATE()

Returns the current date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT CURDATE();
+---------------------------------------------------------+
| CURDATE()                                               |
+---------------------------------------------------------+
| 1997-12-15                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CURDATE() + 0;
+---------------------------------------------------------+
| CURDATE() + 0                                           |
+---------------------------------------------------------+
| 19971215                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

CURRENT_DATE and CURRENT_DATE()

CURRENT_DATE and CURRENT_DATE() are synonyms for CURDATE()

CURTIME()

Returns the current time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

```
mysql> SELECT CURTIME();
+---------------------------------------------------------+
| CURTIME()                                               |
+---------------------------------------------------------+
| 23:50:26                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CURTIME() + 0;
+---------------------------------------------------------+
| CURTIME() + 0                                           |
+---------------------------------------------------------+
| 235026                                                  |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

CURRENT_TIME and CURRENT_TIME()

CURRENT_TIME and CURRENT_TIME() are synonyms for CURTIME().

CURRENT_TIMESTAMP and CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP and CURRENT_TIMESTAMP() are synonyms for NOW().

DATE(expr)

Extracts the date part of the date or datetime expression expr.

```
mysql> SELECT DATE('2003-12-31 01:02:03');
+---------------------------------------------------------+
| DATE('2003-12-31 01:02:03')                             |
+---------------------------------------------------------+
|  2003-12-31                                             |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DATEDIFF(expr1,expr2)

DATEDIFF() returns expr1 . expr2 expressed as a value in days from one date to the other. expr1 and expr2 are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

```
mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');
+---------------------------------------------------------+
| DATEDIFF('1997-12-31 23:59:59','1997-12-30')            |
+---------------------------------------------------------+
| 1                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_ADD(date,INTERVAL expr unit), DATE_SUB(date,INTERVAL expr unit)

These functions perform date arithmetic. date is a DATETIME or DATE value specifying the starting date. expr is an expression specifying the interval value to be added or subtracted from the starting date. expr is a string; it may start with a '-' for negative intervals. unit is a keyword indicating the units in which the expression should be interpreted.

The INTERVAL keyword and the unit specifier are not case sensitive.

The following table shows the expected form of the expr argument for each unit value;

unit **Value**	**Expected**expr**Format**
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'

HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

The values QUARTER and WEEK are available beginning with MySQL 5.0.0.

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
    -> INTERVAL '1:1' MINUTE_SECOND);
+---------------------------------------------------------+
| DATE_ADD('1997-12-31 23:59:59', INTERVAL...             |
+---------------------------------------------------------+
| 1998-01-01 00:01:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR);
+---------------------------------------------------------+
| DATE_ADD('1999-01-01', INTERVAL 1 HOUR)                 |
+---------------------------------------------------------+
| 1999-01-01 01:00:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_FORMAT(date,format)

Formats the date value according to the format string.

The following specifiers may be used in the format string. The '%' character is required before format specifier characters.

Specifier	Description
%a	Abbreviated weekday name (Sun..Sat)
%b	Abbreviated month name (Jan..Dec)
%c	Month, numeric (0..12)
%D	Day of the month with English suffix (0th, 1st, 2nd, 3rd, .)
%d	Day of the month, numeric (00..31)
%e	Day of the month, numeric (0..31)
%f	Microseconds (000000..999999)
%H	Hour (00..23)
%h	Hour (01..12)
%I	Hour (01..12)
%i	Minutes, numeric (00..59)
%j	Day of year (001..366)
%k	Hour (0..23)

%l	Hour (1..12)
%M	Month name (January..December)
%m	Month, numeric (00..12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00..59)
%s	Seconds (00..59)
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00..53), where Sunday is the first day of the week
%u	Week (00..53), where Monday is the first day of the week
%V	Week (01..53), where Sunday is the first day of the week; used with %X
%v	Week (01..53), where Monday is the first day of the week; used with %x
%W	Weekday name (Sunday..Saturday)
%w	Day of the week (0=Sunday..6=Saturday)
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v
%Y	Year, numeric, four digits
%y	Year, numeric (two digits)
%%	A literal .%. character
%x	x, for any.x. not listed above

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
+---------------------------------------------------------+
| DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y')          |
+---------------------------------------------------------+
| Saturday October 1997                                   |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00'
    -> '%H %k %I %r %T %S %w');
+---------------------------------------------------------+
| DATE_FORMAT('1997-10-04 22:23:00.......                 |
+---------------------------------------------------------+
|  22 22 10 10:23:00 PM 22:23:00 00 6                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_SUB(date,INTERVAL expr unit)

This is similar to DATE_ADD() function.

DAY(date)

DAY() is a synonym for DAYOFMONTH().

DAYNAME(date)

Returns the name of the weekday for date.

```
mysql> SELECT DAYNAME('1998-02-05');
+---------------------------------------------------------+
| DAYNAME('1998-02-05')                                   |
+---------------------------------------------------------+
| Thursday                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFMONTH(date)

Returns the day of the month for date, in the range 0 to 31.

```
mysql> SELECT DAYOFMONTH('1998-02-03');
+---------------------------------------------------------+
| DAYOFMONTH('1998-02-03')                                |
+---------------------------------------------------------+
| 3                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFWEEK(date)

Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday). These index values correspond to the ODBC standard.

```
mysql> SELECT DAYOFWEEK('1998-02-03');
+---------------------------------------------------------+
|DAYOFWEEK('1998-02-03')                                  |
+---------------------------------------------------------+
| 3                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFYEAR(date)

Returns the day of the year for date, in the range 1 to 366.

```
mysql> SELECT DAYOFYEAR('1998-02-03');
+---------------------------------------------------------+
| DAYOFYEAR('1998-02-03')                                 |
+---------------------------------------------------------+
| 34                                                      |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

EXTRACT(unit FROM date)

The EXTRACT() function uses the same kinds of unit specifiers as DATE_ADD() or DATE_SUB(), but extracts parts from the date rather than performing date arithmetic.

```
mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');
+---------------------------------------------------------+
| EXTRACT(YEAR FROM '1999-07-02')                         |
+---------------------------------------------------------+
| 1999                                                    |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03');
+---------------------------------------------------------+
| EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03')          |
+---------------------------------------------------------+
| 199907                                                  |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

FROM_DAYS(N)

Given a day number N, returns a DATE value.

```
mysql> SELECT FROM_DAYS(729669);
+---------------------------------------------------------+
| FROM_DAYS(729669)                                       |
+---------------------------------------------------------+
| 1997-10-07                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

Use FROM_DAYS() with caution on old dates. It is not intended for use with values that precede the advent of the Gregorian calendar (1582).

FROM_UNIXTIME(unix_timestamp) FROM_UNIXTIME(unix_timestamp,format)

Returns a representation of the unix_timestamp argument as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone. unix_timestamp is an internal timestamp value such as is produced by the UNIX_TIMESTAMP() function.

If format is given, the result is formatted according to the format string, which is used the same way as listed in the entry for the DATE_FORMAT() function.

```
mysql> SELECT FROM_UNIXTIME(875996580);
+---------------------------------------------------------+
| FROM_UNIXTIME(875996580)                                |
+---------------------------------------------------------+
| 1997-10-04 22:23:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

HOUR(time)

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values. However, the range of TIME values actually is much larger, so HOUR can return values greater than 23.

```
mysql> SELECT HOUR('10:05:03');
+---------------------------------------------------------+
| HOUR('10:05:03')                                        |
+---------------------------------------------------------+
| 10                                                      |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

LAST_DAY(date)

Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.

```
mysql> SELECT LAST_DAY('2003-02-05');
+---------------------------------------------------------+
| LAST_DAY('2003-02-05')                                  |
+---------------------------------------------------------+
| 2003-02-28                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

LOCALTIME and LOCALTIME()

LOCALTIME and LOCALTIME() are synonyms for NOW().

LOCALTIMESTAMP and LOCALTIMESTAMP()

LOCALTIMESTAMP and LOCALTIMESTAMP() are synonyms for NOW().

MAKEDATE(year,dayofyear)

Returns a date, given year and day-of-year values. dayofyear must be greater than 0 or the result is NULL.

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);
+---------------------------------------------------------+
| MAKEDATE(2001,31), MAKEDATE(2001,32)                    |
+---------------------------------------------------------+
| '2001-01-31', '2001-02-01'                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

MAKETIME(hour,minute,second)

Returns a time value calculated from the hour, minute, and second arguments.

```
mysql> SELECT MAKETIME(12,15,30);
+---------------------------------------------------------+
| MAKETIME(12,15,30)                                      |
+---------------------------------------------------------+
| '12:15:30'                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

MICROSECOND(expr)

Returns the microseconds from the time or datetime expression expr as a number in the range from 0 to 999999.

```
mysql> SELECT MICROSECOND('12:00:00.123456');
+---------------------------------------------------------+
| MICROSECOND('12:00:00.123456')                          |
+---------------------------------------------------------+
| 123456                                                  |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

MINUTE(time)

Returns the minute for time, in the range 0 to 59.

```
mysql> SELECT MINUTE('98-02-03 10:05:03');
+---------------------------------------------------------+
| MINUTE('98-02-03 10:05:03')                             |
+---------------------------------------------------------+
| 5                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTH(date)

Returns the month for date, in the range 0 to 12.

```
mysql> SELECT MONTH('1998-02-03')
+---------------------------------------------------------+
| MONTH('1998-02-03')                                     |
+---------------------------------------------------------+
| 2                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTHNAME(date)

Returns the full name of the month for date.

```
mysql> SELECT MONTHNAME('1998-02-05');
+---------------------------------------------------------+
| MONTHNAME('1998-02-05')                                 |
+---------------------------------------------------------+
| February                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

NOW()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

```
mysql> SELECT NOW();
+---------------------------------------------------------+
| NOW()                                                   |
+---------------------------------------------------------+
| 1997-12-15 23:50:26                                     |
+---------------------------------------------------------+1 row in set (0.00 sec)
```

PERIOD_ADD(P,N)

Adds N months to period P (in the format YYMM or YYYYMM). Returns a value in the format YYYYMM. Note that the period argument P is not a date value.

```
mysql> SELECT PERIOD_ADD(9801,2);
+---------------------------------------------------------+
| PERIOD_ADD(9801,2)                                      |
+---------------------------------------------------------+
| 199803                                                  |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

PERIOD_DIFF(P1,P2)

Returns the number of months between periods P1 and P2. P1 and P2 should be in the format YYMM or YYYYMM. Note that the period arguments P1 and P2 are not date values.

```
mysql> SELECT PERIOD_DIFF(9802,199703);
+---------------------------------------------------------+
| PERIOD_DIFF(9802,199703)                                |
+---------------------------------------------------------+
| 11                                                      |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

QUARTER(date)

Returns the quarter of the year for date, in the range 1 to 4.

```
mysql> SELECT QUARTER('98-04-01');
+---------------------------------------------------------+
| QUARTER('98-04-01')                                     |
+---------------------------------------------------------+
| 2                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

SECOND(time)

Returns the second for time, in the range 0 to 59.

```
mysql> SELECT SECOND('10:05:03');
+---------------------------------------------------------+
| SECOND('10:05:03')                                      |
+---------------------------------------------------------+
| 3                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

SEC_TO_TIME(seconds)

Returns the seconds argument, converted to hours, minutes and seconds, as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT SEC_TO_TIME(2378);
+---------------------------------------------------------+
| SEC_TO_TIME(2378)                                       |
+---------------------------------------------------------+
| 00:39:38                                                |
```

```
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

STR_TO_DATE(str,format)

This is the inverse of the DATE_FORMAT() function. It takes a string str and a format string format. STR_TO_DATE() returns a DATETIME value if the format string contains both date and time parts or a DATE or TIME value if the string contains only date or time parts.

```
mysql> SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y');
+---------------------------------------------------------+
| STR_TO_DATE('04/31/2004', '%m/%d/%Y')                   |
+---------------------------------------------------------+
| 2004-04-31                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

SUBDATE(date,INTERVAL expr unit) and SUBDATE(expr,days)

When invoked with the INTERVAL form of the second argument, SUBDATE() is a synonym for DATE_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE_ADD().

```
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------------+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY)                 |
+---------------------------------------------------------+
| 1997-12-02                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT SUBDATE('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------------+
| SUBDATE('1998-01-02', INTERVAL 31 DAY)                  |
+---------------------------------------------------------+
| 1997-12-02                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

SUBTIME(expr1,expr2)

SUBTIME() returns expr1 . expr2 expressed as a value in the same format as expr1. expr1 is a time or datetime expression, and expr2 is a time.

```
mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999',
    -> '1 1:1:1.000002');
+---------------------------------------------------------+
| SUBTIME('1997-12-31 23:59:59.999999'...                 |
+---------------------------------------------------------+
| 1997-12-30 22:58:58.999997                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

SYSDATE()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT SYSDATE();
+---------------------------------------------------------+
| SYSDATE()                                               |
+---------------------------------------------------------+
| 2006-04-12 13:47:44                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME(expr)

Extracts the time part of the time or datetime expression expr and returns it as a string.

```
mysql> SELECT TIME('2003-12-31 01:02:03');
+---------------------------------------------------------+
| TIME('2003-12-31 01:02:03')                             |
+---------------------------------------------------------+
| 01:02:03                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMEDIFF(expr1,expr2)

TIMEDIFF() returns expr1 . expr2 expressed as a time value. expr1 and expr2 are time or date-and-time expressions, but both must be of the same type.

```
mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',
    -> '1997-12-30 01:01:01.000002');
+---------------------------------------------------------+
| TIMEDIFF('1997-12-31 23:59:59.000001'.....              |
+---------------------------------------------------------+
|  46:58:57.999999                                        |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMP(expr), TIMESTAMP(expr1,expr2)

With a single argument, this function returns the date or datetime expression expr as a datetime value. With two arguments, it adds the time expression expr2 to the date or datetime expression expr1 and returns the result as a datetime value.

```
mysql> SELECT TIMESTAMP('2003-12-31');
+---------------------------------------------------------+
| TIMESTAMP('2003-12-31')                                 |
+---------------------------------------------------------+
| 2003-12-31 00:00:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPADD(unit,interval,datetime_expr)

Adds the integer expression interval to the date or datetime expression datetime_expr. The unit for interval is given by the unit argument, which should be one of the following values: FRAC_SECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER or YEAR.

The unit value may be specified using one of keywords as shown or with a prefix of SQL_TSI_. For example, DAY and SQL_TSI_DAY both are legal.

```
mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');
+---------------------------------------------------------+
| TIMESTAMPADD(MINUTE,1,'2003-01-02')                     |
+---------------------------------------------------------+
| 2003-01-02 00:01:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPDIFF(unit,datetime_expr1,datetime_expr2)

Returns the integer difference between the date or datetime expressions datetime_expr1 and datetime_expr2. The unit for the result is given by the unit argument. The legal values for unit are the same as those listed in the description of the TIMESTAMPADD() function.

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
+---------------------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01')          |
+---------------------------------------------------------+
| 3                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME_FORMAT(time,format)

This is used like the DATE_FORMAT() function, but the format string may contain format specifiers only for hours, minutes and seconds.

If the time value contains an hour part that is greater than 23, the %H and %k hour format specifiers produce a value larger than the usual range of 0..23. The other hour format specifiers produce the hour value modulo 12.

```
mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
+---------------------------------------------------------+
| TIME_FORMAT('100:00:00', '%H %k %h %I %l')              |
+---------------------------------------------------------+
| 100 100 04 04 4                                         |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME_TO_SEC(time)

Returns the time argument converted to seconds.

```
mysql> SELECT TIME_TO_SEC('22:23:00');
+---------------------------------------------------------+
| TIME_TO_SEC('22:23:00')                                 |
+---------------------------------------------------------+
| 80580                                                   |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TO_DAYS(date)

Given a date, returns a day number (the number of days since year 0).

```
mysql> SELECT TO_DAYS(950501);
+---------------------------------------------------------+
| TO_DAYS(950501)                                         |
+---------------------------------------------------------+
| 728779                                                  |
```

```
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(date)

If called with no argument, returns a Unix timestamp (seconds since '1970-01-01 00:00:00' UTC) as an unsigned integer. If UNIX_TIMESTAMP() is called with a date argument, it returns the value of the argument as seconds since '1970-01-01 00:00:00' UTC. date may be a DATE string, a DATETIME string, a TIMESTAMP, or a number in the format YYMMDD or YYYYMMDD.

```
mysql> SELECT UNIX_TIMESTAMP();
+---------------------------------------------------------+
| UNIX_TIMESTAMP()                                        |
+---------------------------------------------------------+
| 882226357                                               |
+---------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP('1997-10-04 22:23:00');
+---------------------------------------------------------+
| UNIX_TIMESTAMP('1997-10-04 22:23:00')                   |
+---------------------------------------------------------+
| 875996580                                               |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

UTC_DATE, UTC_DATE()

Returns the current UTC date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
+---------------------------------------------------------+
| UTC_DATE(), UTC_DATE() + 0                              |
+---------------------------------------------------------+
| 2003-08-14, 20030814                                    |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

UTC_TIME, UTC_TIME()

Returns the current UTC time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT UTC_TIME(), UTC_TIME() + 0;
+---------------------------------------------------------+
| UTC_TIME(), UTC_TIME() + 0                              |
+---------------------------------------------------------+
| 18:07:53, 180753                                        |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

UTC_TIMESTAMP, UTC_TIMESTAMP()

Returns the current UTC date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;
+---------------------------------------------------------+
| UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0                    |
+---------------------------------------------------------+
| 2003-08-14 18:08:04, 20030814180804                     |
```

```
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEK(date[,mode])

This function returns the week number for date. The two-argument form of WEEK() allows you to specify whether the week starts on Sunday or Monday and whether the return value should be in the range from 0 to 53 or from 1 to 53. If the mode argument is omitted, the value of the default_week_format system variable is used

Mode	**First Day of week**	**Range**	**Week 1 is the first week.**
0	Sunday	0-53	with a Sunday in this year
1	Monday	0-53	with more than 3 days this year
2	Sunday	1-53	with a Sunday in this year
3	Monday	1-53	with more than 3 days this year
4	Sunday	0-53	with more than 3 days this year
5	Monday	0-53	with a Monday in this year
6	Sunday	1-53	with more than 3 days this year
7	Monday	1-53	with a Monday in this year

```
mysql> SELECT WEEK('1998-02-20');
+---------------------------------------------------------+
| WEEK('1998-02-20')                                      |
+---------------------------------------------------------+
| 7                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEKDAY(date)

Returns the weekday index for date (0 = Monday, 1 = Tuesday, . 6 = Sunday).

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');
+---------------------------------------------------------+
| WEEKDAY('1998-02-03 22:23:00')                          |
+---------------------------------------------------------+
| 1                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEKOFYEAR(date)

Returns the calendar week of the date as a number in the range from 1 to 53. WEEKOFYEAR() is a compatibility function that is equivalent to WEEK(date,3).

```
mysql> SELECT WEEKOFYEAR('1998-02-20');
+---------------------------------------------------------+
| WEEKOFYEAR('1998-02-20')                                |
+---------------------------------------------------------+
| 8                                                       |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

YEAR(date)

Returns the year for date, in the range 1000 to 9999, or 0 for the .zero. date.

```
mysql> SELECT YEAR('98-02-03');
+---------------------------------------------------------+
| YEAR('98-02-03')                                        |
+---------------------------------------------------------+
| 1998                                                    |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

YEARWEEK(date), YEARWEEK(date,mode)

Returns year and week for a date. The mode argument works exactly like the mode argument to WEEK(). The year in the result may be different from the year in the date argument for the first and the last week of the year.

```
mysql> SELECT YEARWEEK('1987-01-01');
+---------------------------------------------------------+
| YEAR('98-02-03')YEARWEEK('1987-01-01')                  |
+---------------------------------------------------------+
| 198653                                                  |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

Note that the week number is different from what the WEEK() function would return (0) for optional arguments 0 or 1, as WEEK() then returns the week in the context of the given year.

Sub query

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE])
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, let us check following subquery with SELECT statement:

```
SQL> SELECT *
     FROM CUSTOMERS
     WHERE ID IN (SELECT ID
                  FROM CUSTOMERS
                  WHERE SALARY > 4500) ;
```

This would produce the following result:

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement:

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
           SELECT [ *|column1 [, column2 ]
           FROM table1 [, table2 ]
           [ WHERE VALUE OPERATOR ]
```

Example:

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy complete CUSTOMERS table into CUSTOMERS_BKP, following is the syntax:

```
SQL> INSERT INTO CUSTOMERS_BKP
     SELECT * FROM CUSTOMERS
     WHERE ID IN (SELECT ID
                  FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> UPDATE CUSTOMERS
     SET SALARY = SALARY * 0.25
     WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
                   WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |   125.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  2125.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Subqueries with the DELETE Statement:

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example deletes records from CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> DELETE FROM CUSTOMERS
     WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
                   WHERE AGE > 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records:

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Sequence

A sequence is a set of integers 1, 2, 3, ... that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them.

This chapter describes how to use sequences in MySQL.

Using AUTO_INCREMENT column:

The simplest way in MySQL to use sequences is to define a column as AUTO_INCREMENT and leave rest of the things to MySQL to take care.

Example:

Try out the following example. This will create table and after that it will insert few rows in this table where it is not required to give record ID because its auto-incremented by MySQL.

```
mysql> CREATE TABLE INSECT
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO INSECT (id,name,date,origin) VALUES
    -> (NULL,'housefly','2001-09-10','kitchen'),
    -> (NULL,'millipede','2001-09-10','driveway'),
    -> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT * FROM INSECT ORDER BY id;
+----+-------------+------------+------------+
| id | name        | date       | origin     |
+----+-------------+------------+------------+
|  1 | housefly    | 2001-09-10 | kitchen    |
|  2 | millipede   | 2001-09-10 | driveway   |
|  3 | grasshopper | 2001-09-10 | front yard |
+----+-------------+------------+------------+
3 rows in set (0.00 sec)
```

Obtain AUTO_INCREMENT Values:

LAST_INSERT_ID( ) is a SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

PERL Example:

Use the mysql_insertid attribute to obtain the AUTO_INCREMENT value generated by a query. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the query. The following example references it through the database handle:

```
$dbh->do ("INSERT INTO INSECT (name,date,origin)

VALUES('moth','2001-09-14','windowsill')");

my $seq = $dbh->{mysql_insertid};
```

PHP Example:

After issuing a query that generates an AUTO_INCREMENT value, retrieve the value by calling mysql_insert_id( ):

```
mysql_query ("INSERT INTO INSECT (name,date,origin)

VALUES('moth','2001-09-14','windowsill')", $conn_id);

$seq = mysql_insert_id ($conn_id);
```

Renumbering an Existing Sequence:

There may be a case when you have deleted many records from a table and you want to resequence all the records. This can be done by using a simple trick but you should be very careful to do so if your table is having join, with other table.

If you determine that resequencing an AUTO_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again. The following example shows how to renumber the id values in the insect table using this technique:

```
mysql> ALTER TABLE INSECT DROP id;

mysql> ALTER TABLE insect

    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,

    -> ADD PRIMARY KEY (id);
```

Starting a Sequence at a Particular Value:

By default, MySQL will start sequence from 1 but you can specify any other number as well at the time of table creation. Following is the example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE INSECT

    -> (

    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,

    -> PRIMARY KEY (id),

    -> name VARCHAR(30) NOT NULL, # type of insect
```

```
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

Operators

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Show Examples

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Show Examples

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).