SQL DML. INSERT, UPDATE, DELETE, SELECT Statements Overview.

Summary

SQL DML.

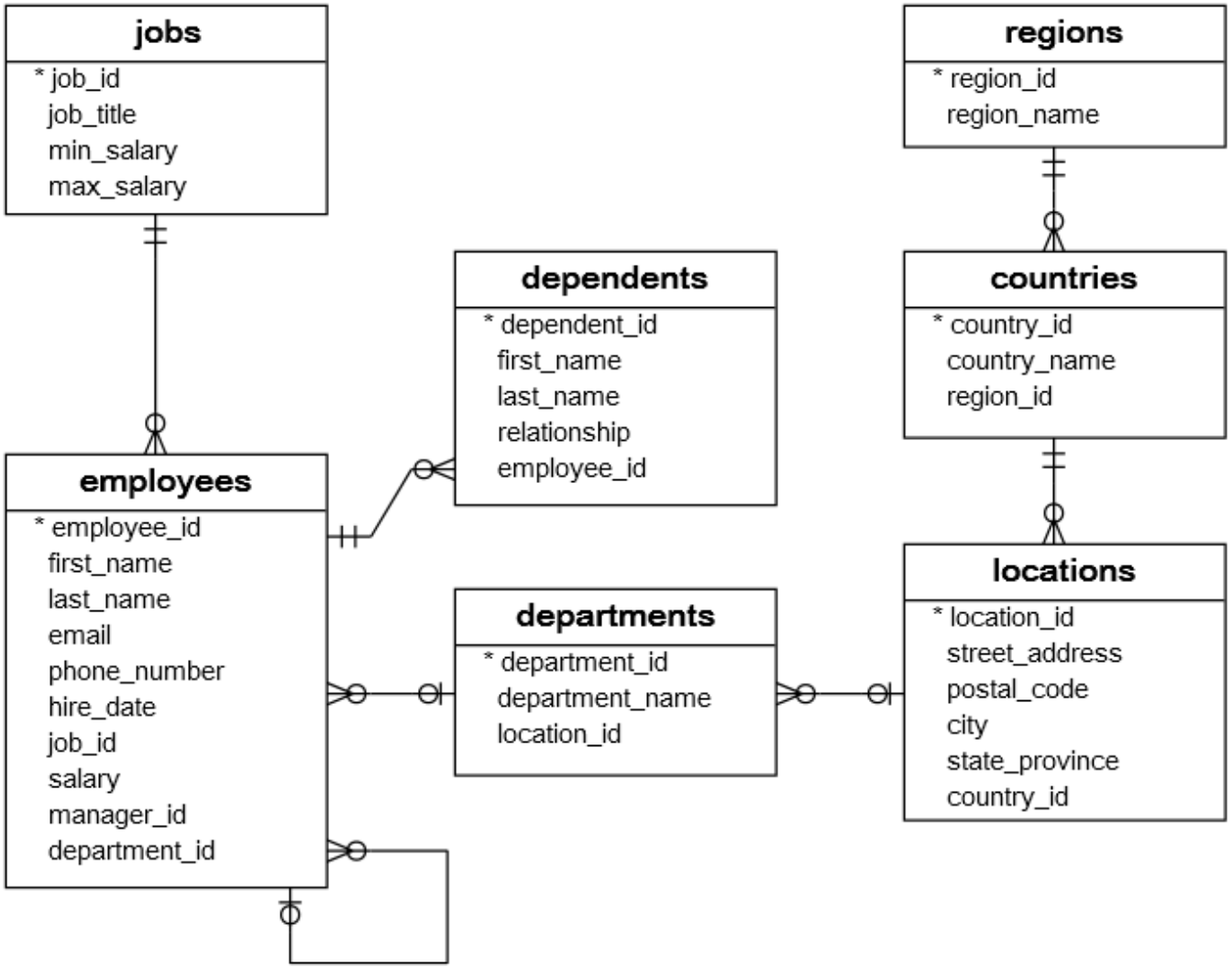
SQL Data Manipulation Language modifies the database instance by inserting, updating, deleting and selecting, its data. DML is responsible for all forms data modification in a database. SQL contains the following set of commands in its DML section:

- INSERT INTO/VALUES
- UPDATE/SET/WHERE
- DELETE FROM/WHERE
- SELECT/FROM/WHERE

These basic constructs allow database programmers and users to enter data and information into the database and retrieve efficiently using a number of sorting, filtering and grouping options with using different function and mathematical calculation.

HR sample database.

You can use SQL Tutorial site https://www.sqltutorial.org/seeit/ for online testing examples and exercises on real DB.



INSERT Statement Overview.

SQL provides the INSERT statement that allows you to insert data into a table. The INSERT statement allows you to:

1. Insert a single row into a table
2. Insert multiple rows into a table
3. Copy rows from a table to another table.

Insert one row into a table.

Syntax.

```
INSERT INTO table1 (column1, column3,...)
VALUES (value1, value3,...);
```

In this syntax:

1. The number of values must be the same as the number of columns. The columns and values must be the correspondent.
2. Before adding a new row, the database system checks for all integrity constraints e.g., FK constraint, PK constraint, CHECK constraint and NOT NULL constraint. If one of these constraints is violated, the database system will issue an error and terminate the statement without inserting any new row into the table.
3. If you don't specify a column and its value in the INSERT statement when that column will take a default value specified in the table structure (for example column2). The default value could be 'string', or 0, or a next integer value in a sequence, the current date/time, a NULL value, etc.
4. If the row is inserted successfully, the database system returned the number of the affected rows: "Affected rows: 1"

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

```
INSERT INTO dependent (depFname, depLname, relationship, empId)
VALUES ('Ivan', 'Green', 'Son', 25); -- depId adding as default sequence value
```

You can check whether the row has been inserted successfully or not by using the following SELECT statement.

```
SELECT * FROM dependent
WHERE empId = 25;
```

Insert multiple rows into a table.

Syntax.

```
INSERT INTO table1
  (column1, column2,...)
VALUES
  (value1, value2,...),
  (value1, value2,...),
  ...;
```

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

```
INSERT INTO dependent
  (depFname, depLname, relationship, empId)
VALUES
  ('Max', 'Black', 'Son', 27),
  ('Alexa', 'Black', 'Doter', 27);
```

Copy rows from other tables.

Syntax. You can use the INSERT statement to query data from one or more tables and insert it into another table as follows:

```
INSERT INTO table2
  (column3, column4)
SELECT
  column1, column2
FROM table1
WHERE condition1;
```

In this syntax, you use a SELECT which is called a **subselect** instead of the VALUES clause.

Example.

```
INSERT INTO dependent_archive
SELECT
  *
FROM
  dependent;
```

UPDATE Statement Overview.

To change existing data in a table, you use the UPDATE statement.

Syntax.

```
UPDATE table_name
SET
  column1 = value1,
  column2 = value3
[WHERE
  Condition];
```

In this syntax:

1. Indicate the table that you want to update in the UPDATE clause.
2. Specify the columns to modify in the SET clause. The columns that are not listed in the SET clause will save original values.
3. Specify which rows to update in the WHERE clause, any row that causes the condition in the WHERE to evaluate to true will be modified.
4. Because the WHERE clause is optional, therefore, if you omit it, the **all the rows** in the table will be affected.

One row UPDATE example.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

To update 25th empId name from Green to Blue, you can use the following UPDATE statement:

```
UPDATE employees
SET
  empLname = 'Blue'
WHERE
  empId = 25;
```

You can verify it by using the following SELECT statement.

```
SELECT empId, empLname FROM employees
WHERE empId = 25;
```

Multiple rows UPDATE example.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

Now, 27th empId wants to change all her children's last names from Black to White.

```
UPDATE dependent
SET
  depLname = 'White'
WHERE
  empId = 27;
```

In this case, you need to update all dependents in the dependent table.

UPDATE with subquery example.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

To make sure that the names of children are always matched with the name of parents in the employees table, you use the following statement:

```
UPDATE dependent
SET depLname = (
  SELECT
    empLname
  FROM
    employees
  WHERE
    empId = dependent.empId
);
```

Because the WHERE clause is omitted, the UPDATE statement updated all rows in the dependents table.

In the SET clause, instead of using the literal values, we used a subquery to get the corresponding last name value from the employees table.

DELETE Statement Overview.

To remove one or more rows from a table, you can use the DELETE statement.

Syntax.

```
DELETE
FROM
  table_name
WHERE
  condition;
```

In this syntax:

1. Provide the name of the table where you want to remove rows.
2. Specify the condition in the WHERE clause to identify the rows that need to be deleted.
3. If you omit the WHERE clause all rows in the table will be deleted. Therefore, you should always use the DELETE statement with caution.
4. The DELETE statement does return the number of rows deleted.

DELETE one row in a table.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

Employee id 25, wants to remove Ivan Blue from his dependent list. We know that Ivan Blue has the dependent id 16, so we use the following DELETE statement to remove Ivan Blue from the dependent table.

```
DELETE FROM dependent
WHERE depId = 16;
```

Because the WHERE clause contains the primary key expression that identifies Fred, the DELETE statement removes **just one row**.

You can verify that the row with the dependent id 16 has been deleted by using the following statement:

```
SELECT COUNT(*)
FROM dependent
WHERE depId = 16;
```

DELETE multiple rows example.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

To delete multiple rows in a table, you use the condition in the WHERE clause to identify the rows that should be deleted. For example, the following statement uses the **IN operator** to include the dependents of the employees with the id is 25, 26, or 27.

```
DELETE FROM dependent
WHERE
  empId IN (25, 26, 27);
```

DELETE rows from related tables.

Example for DB: dependent(depId, depFname, depLname, relationship, empId) >0---have---|- employees(empId, empFname, empLname, …)

One employee may have zero or many dependents while one dependent belongs to only one employee.

Logically, a dependent cannot exist without referring to an employee. When you delete an employee, his dependents must be deleted as well.

To remove the employee id 27 and all the employee's dependents, you need to execute two DELETE statements as follows:

```
DELETE FROM employees
WHERE
  empId = 27;

DELETE FROM dependent
WHERE
  empId = 27;
```

Automatically DELETE rows from related tables (equivalent).

Most database systems support the foreign key constraint so that when one row from a table is deleted, the rows in the **foreign key tables** are also removed **automatically**. After DELETE empId=27 from employees table, all the rows with employee_id 27 from dependent table are also removed automatically:

```
DELETE FROM employees
WHERE
  empId = 27;
```

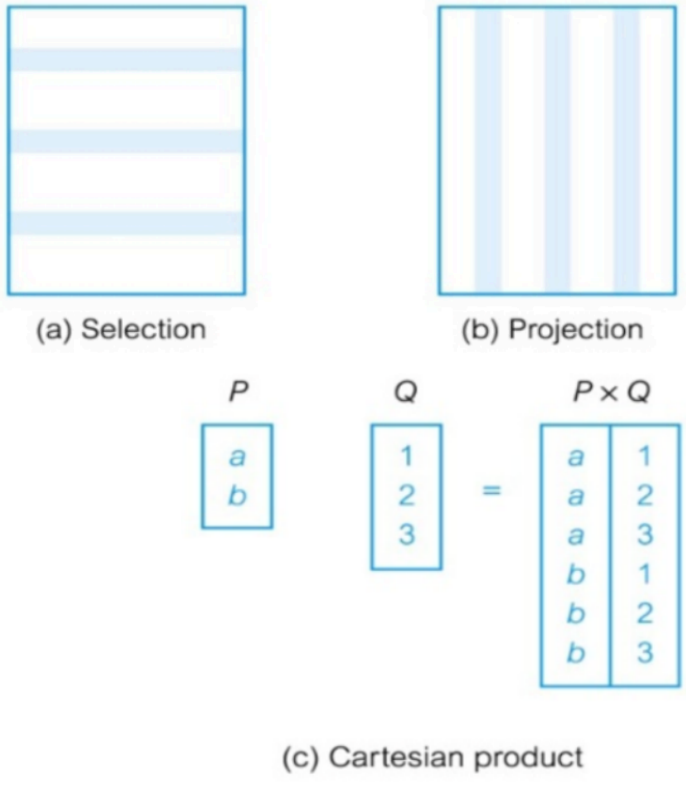
SELECT Statement Overview.

SQL Select Queries and Relation Algebra Operations

- SQL queries use the SELECT statement
- Very central part of SQL language - concepts appear in all DML commands
- General form is:

```
SELECT A1, A2, ...
  FROM r1, r2, ...
  WHERE P;
```

  - Ai are attributes (columns)
  - ri are the relations (tables)
  - P is the selection predicate (rows)



Operations

SELECT A1, A2, ... – corresponds to a relational algebra **Project operation (columns)**

$$\Pi(\ldots)$$

FROM r1, r2, ... – corresponds to a **Cartesian product (tables)** of relations r1, r2, …

$$r1 \times r2 \times \ldots$$

WHERE P – corresponds to a **Selection operation (rows)**. Can be omitted. When left off, P = true

$$\sigma_P(\ldots)$$

SELECT A1, A2, ... FROM r1, r2, ... WHERE P; Assembling it all – equivalent to:

$$\Pi(\sigma_P(r1 \times r2 \times \ldots))$$

SELECT Statement Basic Syntax.

The following illustrates the *basic* syntax of the SELECT statement that retrieves data from a single table.

```
SELECT
	column1, column2, column3, ...
FROM
	table_name;
```

In this syntax, you specify a list of comma-separated columns from which you want to query the data in the SELECT clause and specify the table name in the FROM clause. When evaluating the SELECT statement, the database system evaluates the FROM clause first and then the SELECT clause.

The semicolon (;) is not the part of a query. Typically, the database system uses the semicolon to separate two SQL queries.

In case you want to query data from all columns of a table, you can use the asterisk (*) operator instead of the column list as shown below.

```
SELECT
	*
FROM
	table_name;
```

Using the asterisk (*) operator is only convenient for querying data interactively through an SQL client application. However, if you use the asterisk (*) operator in embedded statements in your application, you may have some potential problems.

Best practice: specify exactly which columns you want to receive data in, in what order and with what column name.

For example

```
SELECT author_name as COVID_19_risk_author
FROM book_author
WHERE age > 60;
```

This command will yield the names of authors from the relation **book_author** whose age is greater than 60 (Author with COVID-19 Risk).

SELECT Statement Full Syntax

To query data from a table, you use the SQL SELECT statement, where contains the syntax for selecting columns, selecting rows, grouping data, joining tables, and performing simple calculations.

```
-- Complete SELECT query
SELECT DISTINCT column, AGG_FUNC(column_or_expression), …
FROM mytable
    JOIN another_table
      ON mytable.column = another_table.column
    WHERE constraint_expression
    GROUP BY column
    HAVING constraint_expression
    ORDER BY column ASC/DESC
    LIMIT count OFFSET COUNT;
```

Each query begins with finding the data that we need in a database, and then filtering that data down into something that can be processed and understood as quickly as possible. Because each part of the query is executed sequentially, it's important to understand the order of execution so that you know what results are accessible where.

The SELECT statement is one of the most complex commands in SQL include many clauses:

- **SELECT** − This is one of the fundamental query command of SQL. It is similar to the projection operation of relational algebra. It selects the attributes based on the condition described by WHERE clause.
- **FROM** − This clause takes a relation name as an argument from which attributes are to be selected/projected. In case more than one relation names are given, this clause corresponds to Cartesian product.
- **JOIN** – for querying data from multiple related tables
- **WHERE** – This clause defines predicate or conditions for filtering data based on a specified condition.
- **GROUP BY** – for grouping data based on one or more columns
- **HAVING** – for filtering groups
- **ORDER BY** – for sorting the result set
- **LIMIT** – for limiting rows returned

You will learn about these clauses in the subsequent tutorials on Practice Works PW-01 and PW-02.

SELECT - Order of Query Execution

1. FROM and JOINs

The **FROM** clause, and subsequent **JOIN**s are first executed to determine the total working set of data that is being queried. This includes subqueries in this clause, and can cause temporary tables to be created under the hood containing all the columns and rows of the tables being joined.

2. WHERE

Once we have the total working set of data, the first-pass **WHERE** constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. Each of the constraints can only access columns directly from the tables requested in the **FROM** clause. Aliases in the **SELECT** part of the query are not accessible in most databases since they may include expressions dependent on parts of the query that have not yet executed.

3. GROUP BY

The remaining rows after the **WHERE** constraints are applied are then grouped based on common values in the column specified in the **GROUP BY** clause. As a result of the grouping, there will only be as many rows as there are unique values in that column. Implicitly, this means that you should only need to use this when you have aggregate functions in your query.

4. HAVING

If the query has a **GROUP BY** clause, then the constraints in the **HAVING** clause are then applied to the grouped rows, discard the grouped rows that don't satisfy the constraint. Like the **WHERE** clause, aliases are also not accessible from this step in most databases.

5. SELECT

Any expressions in the **SELECT** part of the query are finally computed.

6. DISTINCT

Of the remaining rows, rows with duplicate values in the column marked as **DISTINCT** will be discarded.

7. ORDER BY

If an order is specified by the **ORDER BY** clause, the rows are then sorted by the specified data in either ascending or descending order. Since all the expressions in the **SELECT** part of the query have been computed, you can reference aliases in this clause.

8. LIMIT / OFFSET. Finally, the rows that fall outside the range specified by the **LIMIT** and **OFFSET** are discarded, leaving the final set of rows to be returned from the query.

CONCLUSION. Not every query needs to have all the parts we listed above, but a part of why SQL is so flexible is that it allows developers and data analysts to quickly manipulate data without having to write additional code, all just by using the above clauses.

SQL Functions

Read topic MySQL Functions https://www.w3schools.com/sql/sql_ref_mysql.asp

SQL functions help simplify different types of operations on the data. SQL supports fife types of functions:

- Numeric Aggregate Functions
- Numeric Arithmetic Functions
- Character Functions
- Date Functions
- Advanced Functions

The functions are used as part of a select-list of a query, or if they refer to a specific row, they may be used in a WHERE clause. They are used to modify the values or format of data being retrieved.

DB Scheme for Examples:

COMPANY

EMP>|---work---|-DEPT (1,m:1,1)

EMP(**empno**, ename, job, mgr, hiredate, sal, comm, deptno)

DEPT(**deptno**, dname, loc)

SELECT * FROM dept;

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SELECT * FROM emp;

EMPNO	ENAME	JOB	HIREDATE	MGR	SAL	COMM	DEPTNO
7369	SMITH	CLERK	17-DEC-80	7902	800		20
7499	ALLEN	SALESMAN	20-FEB-81	7698	1600	300	30
7521	WARD	SALESMAN	22-FEB-81	7698	1250	500	30
7566	JONES	MANAGER	02-APR-81	7839	2975		20
7654	MARTIN	SALESMAN	28-SEP-81	7698	1250	1400	30
7698	BLAKE	MANAGER	01-MAY-81	7839	2850		30
7782	CLARK	MANAGER	09-JUN-81	7839	2450		10
7788	SCOTT	ANALYST	19-APR-87	7566	3000		20
7839	KING	PRESIDENT	17-NOV-81		5000		10
7844	TURNER	SALESMAN	08-SEP-81	7698	1500	0	30
7876	ADAMS	CLERK	23-MAY-87	7788	1100		20
7900	JAMES	CLERK	03-DEC-81	7698	950		30
7902	FORD	ANALYST	03-DEC-81	7566	3000		20
7934	MILLER	CLERK	23-JAN-82	7782	1300		10

Numeric Aggregate Functions

All aggregate functions (exception of count) operate on numerical columns. All of the aggregate functions operate on a number of rows:

1. **avg**(column) - computes the average value and ignores null values
   SELECT avg(sal) FROM emp; - gives the average salary in the employees table
2. **sum**(column) - computes the total of all the values in the specified column and ignores null values
   sum(comm) - calculates the total commission paid to all employees
3. **min**(column) - finds the minimum value in a column
   min(sal) - returns the lowest salary
4. **max**(column) - finds the maximum value in a column
   max(comm) - returns the highest commission
5. **count**(column) - counts the number of values and ignores nulls
   count(empno) - counts the number of employees

Because an aggregate function operates on a set of values, it is often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause divides the result set into groups of values and the aggregate function returns a single value for each group.

Numeric Arithmetic Functions

The most commonly used arithmetic functions are as follows:

1. **greatest**(object-list) - returns the greatest of a list of values (items over coma)
   greatest(sal,comm) - returns whichever of the SALaries or COMMission attributes has the highest value
2. **least**(object-list) - returns the smallest of a list of values (items over coma)
   least(sal,comm) - returns whichever of the SALaries or COMMission attributes has the lowest value
3. **round**(number[,d]) - rounds the number to d digits right of the decimal point (d can be negative)

   round(sal,2) - rounds values of the SAL attribute to two decimal places
4. **trunc**(number,d) - truncates number to d decimal places (d can be negative). Note: The difference between the round and truncate functions is that round will round up digits of five or higher, whilst trunc always rounds down.
   trunc(sal,2) - truncates values of the SAL attribute to two decimal places
5. **abs**(number) - returns the absolute value of the number
   abs(comm-sal) - returns the absolute value of COMM - SAL; that is, if the number returned would be negative, the minus sign is discarded
6. **sign**(number) - returns 1 if number greater than zero, 0 if number = zero, -1 if number less than zero
   sign(comm-sal) - returns 1 if COMM - SAL > 0, 0 if COMM - SAL = 0, and - 1 if COMM - SAL < 0
7. **mod**(number1,number2) - returns the remainder when number1 is divided by number2
   mod(sal,comm) - returns the remainder when SAL is divided by COMM
8. **sqrt**(number) - returns the square root of the number. If the number is less than zero then sqrt returns null
   sqrt(sal) - returns the square root of salaries
9. **to_char**(number[picture]) - converts a number to a character string in the format specified
   to_char(sal,9999.99) - represents salary values with four digits before the decimal point, and two afterwards
10. **decode**(column,starting-value,substituted-value..) - substitutes alternative values for a specified column
    decode(comm,100,200,200,300,100) - returns values of commission increased by 100 for values of 100 and 200, and displays any other comm values as if they were 100
11. **ceil**(number) - rounds up a number to the nearest integer
    ceil(sal) - rounds up salaries to the nearest integer
12. **floor**(number) - truncates the number to the nearest integer
    floor(sal) - rounds down salary values to the nearest integer

Character Functions

The most commonly used character string functions are as follows:

1. **string1 || string2** - concatenates (links) string1 with string2
   deptno || empno - concatenates the employee number with the department number into one column in the query result
2. **distinct** <column> - lists the distinct values of the specific column
   distinct job - lists all the distinct values of job in the JOB attribute
3. **length**(string) - finds number of characters in the string
   length(ename) - returns the number of characters in values of the ENAME attribute
4. **substr**(column,start-position[,length]) - extracts a specified number of characters from a string
   substr(ename,1,3) - extracts three characters from the ENAME column, starting from the first character
5. **upper**(string) - converts all characters in the string to upper case
   upper(ename) - converts values of the ENAME attribute to upper case
6. **lower**(string) - converts all characters in the string to lower case
   lower(ename) - converts values of the ENAME attribute to lower case
7. **lpad**(string,len[,char]) - left pads the string with filler characters
   lpad(ename,10) - left pads values of the ENAME attribute with filler characters (spaces)
8. **rpad**(string,len[,char]) - right pads the string with filler characters
   rpad(ename,10) - right pads values of the ENAME attribute with filler characters (spaces)
9. **initcap**(string) - capitalises the initial letter of every word in a string
   initcap(job) - starts all values of the JOB attribute with a capital letter
10. **translate**(string,from,to) - translates the occurrences of the 'from' string to the 'to' characters
    translate(ename,'ABC','XYZ') - replaces all occurrences of the string 'ABC' in values of the ENAME attribute with the string 'XYZ'
11. **ltrim**(string,set) - trims all characters in a set from the left of the string
    ltrim(ename,' ') - removes all spaces from the start of values of the ENAME attribute
12. **rtrim**(string,set) - trims all characters in the set from the right of the string
    rtrim(job,'.') - removes any full-stop characters from the right-hand side of values of the JOB attribute

Date Functions

The date functions in most commercially available database systems are quite rich, reflecting the fact that many commercial applications are very date driven. The most commonly used date functions in SQL are as follows:

1. **sysdate, curdate, now** - returns the current date & time

   SELECT now(); return date-time

2. **year, month, day, hour, minute, second**(date) - returns the part for a given date

   SELECT year(hiredate), ename FROM emp;

3. **quarter, week, weekday**(date) - returns the quarter or week or weekday number for a given date

   SELECT quarter(now());

4. **add-months**(date, number) - adds a number of months from/to a date (number can be negative).

   add-months(hiredate, 3) - this adds three months to each value of the HIREDATE attribute

5. **months-between**(date1, date2) - subtracts date2 from date1 to yield the difference in months.

   months-between(sysdate, hiredate) - returns the number of months between the current date & the dates employees were hired

6. **last-day**(date) - moves a date forward to last day in the month.

   last-day(hiredate) - this moves hiredate forward to the last day of the month in which they occurred

7. **next-day**(date,day) - moves a date forward to the given day of week.

   next-day(hiredate,'monday') - this returns all hiredates moved forward to the Monday following the occurrence of the hiredate

8. **round**(date[,precision]) - rounds a date to a specified precision.

   round(hiredate,'month') - this displays hiredates rounded to the nearest month

9. **trunc**(date[,precision]) - truncates a date to a specified precision.

   trunc(hiredate,'month') - this displays hiredates truncated to the nearest month

10. **decode**(column,starting-value,substituted-value) - substitutes alternative values for a specified column.

    decode(hiredate,'25-dec-99','christmas day',hiredate) - this displays any hiredates of the 25th of December, 1999, as Christmas Day, and any default values of hiredate as hiredate

Advanced Functions

Function	Description
BIN	Returns a binary representation of a number
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list
CONNECTION_ID	Returns the unique connection ID for the current connection
CONV	Converts a number from one numeric base system to another
CONVERT	Converts a value into the specified datatype or character set
CURRENT_USER	Returns the user and host name for the MySQL account that the server used to authenticate the current client
DATABASE	Returns the name of the current database
IF	Returns a value if a condition is TRUE, or another value if a condition is FALSE
IFNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNULL	Returns 1 or 0 depending on whether an expression is NULL
NULLIF	Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned
SESSION_USER	Returns the current MySQL user name and host name
VERSION	Returns the current version of the MySQL database