Section 1: Querying Data

1. SELECT Statement:

```
SELECT
    select_list
FROM
    table_name;
```

Section 2: Sorting Data

2. ORDER BY Clause:

```
SELECT
    select_list
FROM
    table_name
ORDER BY
    sort_expression [ASC | DESC];
```

Section 3: Filtering Data

3. DISTINCT :

```
SELECT DISTINCT
    column1, column2, ...
FROM
    table1;
```

4. LIMIT:

To limit the number of rows returned by a select statement, you use the LIMIT and OFFSET clauses.

```
SELECT
    column_list
FROM
    table1
ORDER BY column_list
LIMIT row_count OFFSET offset;
```

In this syntax:

- The LIMIT row_count determines the number of rows (row_count) returned by the query.
- The OFFSET offset clause skips the offset rows before beginning to return the rows.

5. FETCH:

```
OFFSET offset_rows { ROW | ROWS }
FETCH { FIRST | NEXT } [ fetch_rows ] { ROW | ROWS }
ONLY
```

Example:

```
SELECT
    employee_id,
    first_name,
    last_name,
    salary
FROM employees
ORDER BY
    salary DESC
OFFSET 0 ROWS
FETCH NEXT 1 ROWS ONLY;
```

6. WHERE Clause:

```
SELECT
    column1, column2, ...
FROM
    table_name
WHERE
    condition;
```

Example:

```
SELECT
        employee_id,
        first_name,
        last_name,
        hire_date
FROM
        employees
WHERE
```

```
        hire_date >= '1999-01-01'
ORDER BY
        hire_date DESC;
```

7. Comparison operators:

Operator	Meaning
=	Equal
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

8. Logical operators:

Operator	Meaning
ALL	Return true if all comparisons are true
AND	Return true if both expressions are true
ANY	Return true if any one of the comparisons is true.
BETWEEN	Return true if the operand is within a range
EXISTS	Return true if a subquery contains any rows
IN	Return true if the operand is equal to one of the value in a list
LIKE	Return true if the operand matches a pattern
NOT	Reverse the result of any other Boolean operator.
OR	Return true if either expression is true

SOME	Return true if some of the expressions are true

Examples:

- SELECT
  first_name, last_name, salary
FROM
  employees
WHERE
  salary >= ALL (SELECT
      salary
    FROM
      employees
    WHERE
      department_id = 8)
ORDER BY salary DESC;

- SELECT
  first_name, last_name, salary
FROM
  employees
WHERE
  salary > 5000 AND salary < 7000
ORDER BY salary;

- SELECT
  first_name, last_name, salary
FROM
  employees
WHERE
  salary = 7000 OR salary = 8000
ORDER BY salary;

- SELECT
  first_name, last_name, salary
FROM
  employees
WHERE
  salary BETWEEN 9000 AND 12000

```
ORDER BY salary;
```

■
```
SELECT
    first_name, last_name, department_id
FROM
    employees
WHERE
    department_id IN (8, 9)
ORDER BY department_id;
```

■
```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    first_name LIKE 'jo%'
ORDER BY first_name;
```

■
```
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE
    salary > ANY(SELECT
            AVG(salary)
        FROM
            employees
        GROUP BY department_id)
ORDER BY first_name , last_name;
```

■
```
SELECT
    first_name, last_name
FROM
    employees e
WHERE
    EXISTS( SELECT
            1
        FROM
```

```
        dependents d
    WHERE
        d.employee_id = e.employee_id);
```

■ SELECT

```
    employee_id,
    first_name,
    last_name,
    salary
FROM
    employees
WHERE
    department_id = 5
AND NOT salary > 5000
ORDER BY
    salary;
```

Section 4:

Conditional Expressions

9. CASE Expression:

- **Simple CASE expression:**

```
CASE expression
WHEN when_expression_1 THEN
    result_1
WHEN when_expression_2 THEN
    result_2
WHEN when_expression_3 THEN
    result_3
...
ELSE
    else_result
END
```

Example:

```
SELECT
   first_name,
   last_name,
```

```
    hire_date,
    CASE (2000 - YEAR(hire_date))
        WHEN 1 THEN '1 year'
        WHEN 3 THEN '3 years'
        WHEN 5 THEN '5 years'
        WHEN 10 THEN '10 years'
        WHEN 15 THEN '15 years'
        WHEN 20 THEN '20 years'
        WHEN 25 THEN '25 years'
        WHEN 30 THEN '30 years'
    END aniversary
FROM
    employees
ORDER BY first_name;
```

- **Searched CASE expression:**

```
CASE
WHEN boolean_expression_1 THEN
        result_1
WHEN boolean_expression_2 THEN
        result_2
WHEN boolean_expression_3 THEN
        result_3
ELSE
        else_result
END;
```

Example:

```
SELECT
    first_name,
    last_name,
    CASE
        WHEN salary < 3000 THEN 'Low'
        WHEN salary >= 3000 AND salary <= 5000 THEN
'Average'
        WHEN salary > 5000 THEN 'High'
    END evaluation
FROM
    employees;
```

Section 5: Joining Multiple Tables

10. SQL Aliases:

SQL column aliases:

column_name AS alias_name
column_name alias_name
column_name AS 'Alias Name'

SQL table aliases:

table_name AS table_alias
table_name table_alias

11. INNER JOIN:

Suppose, you have two tables: A and B.

Table A has four rows: (1,2,3,4) and table B has four rows: (3,4,5,6)

When table A joins with table B using the inner join, you have the result set (3,4) that is the intersection of table A and table B.

```
SELECT a

FROM A

INNER JOIN B ON b = a;
```

Example:

```
SELECT

    first_name,

    last_name,

    employees.department_id,

    departments.department_id,

    department_name

FROM
```

```
    employees

        INNER JOIN

    departments ON departments.department_id =
employees.department_id

    WHERE

        employees.department_id IN (1 , 2, 3);
```

12. LEFT OUTER JOIN:

Suppose we have two tables A and B. The table A has four rows 1, 2, 3 and 4. The table B also has four rows 3, 4, 5, 6.

When we join table A with table B, all the rows in table A (the left table) are included in the result set whether there is a matching row in the table B or not.

```
SELECT

    A.n

FROM

    A

LEFT JOIN B ON B.n = A.n;
```

Example:

```
SELECT

    c.country_name,

    c.country_id,

    l.country_id,

    l.street_address,

    l.city

FROM

    countries c
```

```
LEFT JOIN locations l ON l.country_id = c.country_id

WHERE

    c.country_id IN ('US', 'UK', 'CN')
```

13. FULL OUTER JOIN:

```
SELECT column_list
FROM A
FULL OUTER JOIN B ON B.n = A.n;
```

Example:

```
SELECT
    basket_name,
    fruit_name
FROM
    fruits
FULL OUTER JOIN baskets ON baskets.basket_id =
fruits.basket_id;
```

14. CROSS JOIN:

```
SELECT column_list
FROM A
CROSS JOIN B;
```

Example:

```
SELECT
    sales_org,
    channel
FROM
    sales_organization
CROSS JOIN sales_channel;
```

15. SELF JOIN:

We join a table to itself to evaluate the rows with other rows in the same table. To perform the self-join, we use either an inner join or left join clause.

```
SELECT
    column1,
    column2,
```

```
        column3,
        ...
    FROM
        table1 A
    INNER JOIN table1 B ON B.column1 = A.column2;
```

Example:

```
SELECT
   e.first_name || ' ' || e.last_name AS employee,
   m.first_name || ' ' || m.last_name AS manager
FROM
   employees e
      INNER JOIN
   employees m ON m.employee_id = e.manager_id
ORDER BY manager;
```

Section 6: Aggregate Functions

```
SELECT c1, aggregate_function(c2)

FROM table

GROUP BY c1;
```

The following are the commonly used SQL aggregate functions:

- AVG() – returns the average of a set.
- COUNT() – returns the number of items in a set.
- MAX() – returns the maximum value in a set.
- MIN() – returns the minimum value in a set
- SUM() – returns the sum of all or distinct values in a set

Except for the COUNT() function, SQL aggregate functions ignore null.

You can use aggregate functions as expressions only in the following:

- The select list of a SELECT statement, either a subquery or an outer query.
- A HAVING clause

16. AVG:

AVG([ALL|DISTINCT] expression)

Example:

```
SELECT
   ROUND(AVG(DISTINCT salary), 2)
FROM
   employees;
```

17. COUNT:

COUNT([ALL | DISTINCT] expression);

Example:

```
SELECT
   COUNT(*)
FROM
   employees;
```

18. SUM:

SUM([ALL|DISTINCT] expression)

Example:

```
SELECT
   SUM(salary)
FROM
   employees;
```

19. MAX:

MAX(expression)

Example:

```
SELECT
   MAX(salary)
FROM
   employees;
```

20. MIN:

MIN(expression)

Example:

```
SELECT
   MIN(salary)
```

```
FROM
    employees;
```

Section 7: Grouping Data

21. GROUP BY:

```
SELECT
        column1,
        column2,
        aggregate_function(column3)
FROM
        table_name
GROUP BY
        column1,
        column2;
```

Example:

```
SELECT
    department_id
FROM
    employees
GROUP BY
    department_id;
```

22. HAVING:

```
SELECT
        column1,
        column2,
        AGGREGATE_FUNCTION (column3)
FROM
        table1
GROUP BY
        column1,
        column2
HAVING
        group_condition;
```

Example:

```
SELECT
```

```
    manager_id,
    first_name,
    last_name,
    COUNT(employee_id) direct_reports
FROM
    employees
WHERE
    manager_id IS NOT NULL
GROUP BY manager_id;
```

23. GROUPING SETS:

```
SELECT
    c1,
    c2,
    aggregate (c3)
FROM
    table
GROUP BY
    GROUPING SETS (
        (c1, c2),
        (c1),
        (c2),
        ()
);
```

Example:

```
SELECT
    warehouse,
    product,
    SUM (quantity) qty
FROM
    inventory
GROUP BY
    GROUPING SETS(
        (warehouse,product),
        (warehouse),
        (product),
        ()
```

```
);
```

24. ROLLUP:

```
SELECT
   c1, c2, aggregate_function(c3)
FROM
   table
GROUP BY ROLLUP (c1, c2);
```

Example:

```
SELECT
   warehouse, product, SUM(quantity)
FROM
   inventory
GROUP BY warehouse, ROLLUP (product);
```

25. CUBE:

```
SELECT
   c1, c2, AGGREGATE_FUNCTION(c3)
FROM
   table_name
GROUP BY CUBE(c1 , c2);
```

Example:

```
SELECT
  warehouse,
  product,
  SUM(quantity)
FROM
  inventory
GROUP BY
  CUBE(warehouse,product)
ORDER BY
  warehouse,
  product;
```

Section 8: SET Operators

26. UNION and UNION ALL:

```
SELECT
   column1, column2
FROM
   table1
UNION [ALL]
SELECT
   column3, column4
FROM
   table2;
```

Example:

```
SELECT
      first_name,
      last_name
FROM
      employees
UNION
SELECT
      first_name,
      last_name
FROM
      dependents
ORDER BY
      last_name;
```

27. INTERSECT:

```
SELECT
      id
FROM
      a
INTERSECT
SELECT
      id
FROM
      b;
```

Example:

```
SELECT
        id
FROM
        a
INTERSECT
SELECT
        id
FROM
        b
ORDER BY id DESC;
```

28. MINUS:

```
SELECT
        id
FROM
        A
MINUS
SELECT
        id
FROM
        B;
```

Example:

```
SELECT
   employee_id
FROM
   employees
MINUS
SELECT
   employee_id
FROM
   dependents
ORDER BY employee_id;
```

Section 9. Subquery

29. Subquery:

■SQL subquery with the IN or NOT IN operator

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    department_id NOT IN (SELECT
            department_id
        FROM
            departments
        WHERE
            location_id = 1700)
ORDER BY first_name , last_name;
```

■SQL subquery in the FROM clause

```
SELECT
    ROUND(AVG(average_salary), 0)
FROM
    (SELECT
        AVG(salary) average_salary
    FROM
        employees
    GROUP BY department_id) department_salary;
```

30. Correlated Subquery:

■SQL correlated subquery in the `WHERE` clause example

```
SELECT
    employee_id,
    first_name,
    last_name,
    salary,
    department_id
FROM
```

```
    employees e
WHERE
    salary > (SELECT
            AVG(salary)
        FROM
            employees
        WHERE
            department_id = e.department_id)
ORDER BY
    department_id ,
    first_name ,
    last_name;
```

SQL correlated subquery in the `SELECT` clause example

```
SELECT
    employee_id,
    first_name,
    last_name,
    department_name,
    salary,
    (SELECT
            ROUND(AVG(salary),0)
        FROM
            employees
        WHERE
            department_id = e.department_id)
avg_salary_in_department
FROM
    employees e
        INNER JOIN
    departments d ON d.department_id = e.department_id
ORDER BY
    department_name,
    first_name,
    last_name;
```

31. EXISTS:

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    EXISTS( SELECT
            1
        FROM
            dependents
        WHERE
            dependents.employee_id =
employees.employee_id);
```

32. ALL:

```
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE
    salary > ALL (SELECT
            salary
        FROM
            employees
        WHERE
            department_id = 2)
ORDER BY salary;
```

33. ANY:

```
SELECT
    first_name,
    last_name,
    salary
FROM
    employees
WHERE
    salary = ANY (
```

```
        SELECT
            AVG(salary)
        FROM
            employees
        GROUP BY
            department_id)
    ORDER BY
        first_name,
        last_name,
        salary;
```

Section 10: Modifying data

34. INSERT:

■ Insert one row into a table

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

Example:

```
INSERT INTO dependents (
    first_name,
    last_name,
    relationship,
    employee_id
)
VALUES
    (
        'Dustin',
        'Johnson',
        'Child',
        178
    );
```

■ Insert multiple rows into a table

```
INSERT INTO table1
VALUES
    (value1, value2,...),
```

```
        (value1, value2,...),
        (value1, value2,...),
        ...;
```

Example:

```
INSERT INTO dependents (
        first_name,
        last_name,
        relationship,
        employee_id
)
VALUES
        (
                'Cameron',
                'Bell',
                'Child',
                192
        ),
        (
                'Michelle',
                'Bell',
                'Child',
                192
        );
```

Copy rows from other tables

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

Example:

```
INSERT INTO dependents_archive
SELECT
        *
FROM
```

dependents;

35. UPDATE:

```
UPDATE table_name
SET column1 = value1,
 column2 = value2
WHERE
    condition;
```

Example:

```
UPDATE dependents
SET
   last_name = 'Lopez'
WHERE
   employee_id = 192;
```

36. DELETE:

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

Example:

```
DELETE FROM dependents
WHERE
   employee_id IN (100 , 101, 102);
```

Section 11: Working with table structures

37. CREATE TABLE:

```
CREATE TABLE table_name(
    column_name_1 data_type default value
column_constraint,
    column_name_2 data_type default value
column_constraint,
    ...,
```

```
    table_constraint
);
```

Example:

```
CREATE TABLE courses (
    course_id INT AUTO_INCREMENT PRIMARY KEY,
    course_name VARCHAR(50) NOT NULL
);
```

38. ALTER TABLE:

SQL ALTER TABLE ADD column

```
ALTER TABLE table_name
ADD new_colum data_type column_constraint [AFTER
existing_column];
```

Example:

```
ALTER TABLE courses
ADD fee NUMERIC (10, 2) AFTER course_name,
ADD max_limit INT AFTER course_name;
```

SQL ALTER TABLE MODIFY column

```
ALTER TABLE table_name
MODIFY column_definition;
```

Example:

```
ALTER TABLE courses
MODIFY fee NUMERIC (10, 2) NOT NULL;
```

SQL ALTER TABLE DROP columns

```
ALTER TABLE table_name
DROP column_name,
DROP colum_name,
…
```

Example:

```
ALTER TABLE courses
DROP COLUMN max_limit,
DROP COLUMN credit_hours;
```

39. DROP TABLE:

```
DROP TABLE [IF EXISTS] table_name;
```

Example:

DROP TABLE emergency_contacts;

40. TRUNCATE TABLE:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE big_table;

SQL TRUNCATE TABLE vs. DELETE

Logically the TRUNCATE TABLE statement and the DELETE statement without the WHERE clause gives the same effect that removes all data from a table. However, they do have some differences:

- When you use the DELETE statement, the database system logs the operations. And with some efforts, you can roll back the data that was deleted. However, when you use the TRUNCATE TABLE statement, you have no chance to roll back except you use it in a transaction that has not been committed.
- To delete data from a table referenced by a foreign key constraint, you cannot use the TRUNCATE TABLE statement. In this case, you must use the DELETE statement instead.
- The TRUNCATE TABLE statement does not fire the delete trigger if the table has the triggers associated with it.
- Some database systems reset the value of an auto-increment column (or identity, sequence, etc.) to its starting value after you execute the TRUNCATE TABLE statement. It is not the case for the DELETE statement.
- The DELETE statement with a WHERE clause deletes partial data from a table while the TRUNCATE TABLE statement always removes all data from the table.