1. CREATE TABLE: Making a Table

- **Purpose**: Define a new table.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
);
```

Example: Create a table for students.

```
CREATE TABLE students (
  student_id INT NOT NULL,
  name VARCHAR(50),
  age INT,
  PRIMARY KEY (student_id)
);
```

---

2. INSERT INTO: Add Data to a Table

- **Single Row**:

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

Example: Add a student.

```
INSERT INTO students (student_id, name, age) VALUES (1, 'Alice', 20);
```

- **Multiple Rows**:

```
INSERT INTO table_name (column1, column2) VALUES
(value1, value2),
(value3, value4);
```

Example: Add multiple students.

```
INSERT INTO students (student_id, name, age) VALUES
(2, 'Bob', 22),
(3, 'Charlie', 21);
```

3. CREATE TABLE with Primary and Foreign Keys

- **Primary Key**: Ensures each row is unique.
- **Foreign Key**: Links to another table.
- **Example**: Create `students` and `courses` tables:

```
CREATE TABLE students (
  student_id INT NOT NULL,
  name VARCHAR(50),
  age INT,
  PRIMARY KEY (student_id)
);

CREATE TABLE courses (
  course_id INT NOT NULL,
  course_name VARCHAR(50),
  student_id INT,
  PRIMARY KEY (course_id),
  FOREIGN KEY (student_id) REFERENCES students(student_id)
);
```

Explanation:

- `student_id` in `students` is the **Primary Key** (unique identifier).
- `student_id` in `courses` is the **Foreign Key**, linking to `students(student_id)`.

---

4. INSERT INTO with Foreign Key

- Add data that respects the link between tables:

```
INSERT INTO students (student_id, name, age) VALUES (1, 'Alice', 20);
INSERT INTO courses (course_id, course_name, student_id) VALUES (101, 'Math', 1);
```

---

5. View the Table

- See the structure of a table:

```
DESCRIBE table_name;
```

Example: View `students` table:

```
DESCRIBE students;
```

- Fetch all data:

```
SELECT * FROM students;
```

---

1. SELECT: Fetch Data

- Get specific columns:

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

Example: Get names and ages of employees.

- Get all columns:

```
SELECT * FROM table_name;
```

Example: Show all data in the table.

---

2. DISTINCT: Remove Duplicates

- Fetch unique values:

```
SELECT DISTINCT column1 FROM table_name;
```

Example: Find all unique departments.

---

3. WHERE: Filter Data

- Fetch rows that match a condition:

```
SELECT * FROM table_name WHERE column = 'value';
```

Example: Get employees in the "HR" department.

- Use comparison operators:

```
SELECT * FROM table_name WHERE age > 30;
```

Operators: `=`, `!=`, `<`, `>`, `<=`, `>=`

---

4. Logical Operators

- Combine conditions:

```
SELECT * FROM table_name WHERE age > 30 AND city = 'London';
```

  - **AND**: All conditions must be true.
  - **OR**: At least one condition is true.
  - **NOT**: Opposite of a condition.

---

5. LIKE: Pattern Matching

- Use wildcards for patterns:

```
SELECT * FROM table_name WHERE name LIKE 'A%';
```

Example: Names starting with "A".
  - `%`: Any characters.
  - `_`: One character.

---

6. IN and BETWEEN

- **IN**: Match a list of values.

```
SELECT * FROM table_name WHERE city IN ('London', 'Paris');
```

Example: People in London or Paris.

- **BETWEEN**: Match a range.

```
SELECT * FROM table_name WHERE age BETWEEN 20 AND 30;
```

---

7. ORDER BY: Sort Data

- Arrange rows in order:

```
SELECT * FROM table_name ORDER BY column1 ASC, column2 DESC;
```

Example: Sort by age (ascending) and salary (descending).

---

8. LIMIT: Control Results

- Fetch a specific number of rows:

```
SELECT * FROM table_name LIMIT 5;
```

Example: Get the first 5 rows.

- Skip rows with OFFSET:

```
SELECT * FROM table_name LIMIT 5 OFFSET 10;
```

Example: Skip 10 rows, fetch the next 5.

---

9. Aggregate Functions

- Perform calculations:

```
SELECT COUNT(column), SUM(column), AVG(column), MAX(column), MIN(column) FROM table_name;
```

Example: Count employees, find total and average salary.

---

10. GROUP BY and HAVING

- Group rows and calculate for each group:

```
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

Example: Count employees in each department.

- Filter groups:

```sql
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

Example: Show departments with more than 5 employees.

---

11. JOINS: Combine Tables

- **Inner Join**: Match rows in both tables.

```sql
SELECT A.name, B.order_date
FROM customers A
INNER JOIN orders B ON A.id = B.customer_id;
```

- **Left Join**: All rows from the left table.

```sql
SELECT A.name, B.order_date
FROM customers A
LEFT JOIN orders B ON A.id = B.customer_id;
```

- **Right Join**: All rows from the right table.

```sql
SELECT A.name, B.order_date
FROM customers A
RIGHT JOIN orders B ON A.id = B.customer_id;
```

- **Full Join**: All rows from both tables.

```sql
SELECT A.name, B.order_date
FROM customers A
FULL OUTER JOIN orders B ON A.id = B.customer_id;
```

---

12. Subqueries

- Query inside another query:

```sql
SELECT name FROM students WHERE marks = (SELECT MAX(marks) FROM students);
```

Example: Find the top scorer.

- Correlated Subquery:

```sql
SELECT name
FROM employees A
WHERE salary > (SELECT AVG(salary) FROM employees B WHERE A.department = B.department);
```

---

13. INSERT, UPDATE, DELETE

- **Insert**: Add new data.

```sql
INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');
```

- **Update**: Modify data.

```sql
UPDATE table_name SET column1 = 'new_value' WHERE condition;
```

- **Delete**: Remove data.

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

---

14. CASE: Conditional Data

- Add conditional logic:

```sql
SELECT name,
       CASE
         WHEN marks >= 50 THEN 'Pass'
         ELSE 'Fail'
       END AS result
FROM students;
```

Example: Show pass/fail status.

---

15. Views

- Create a virtual table:

```sql
CREATE VIEW view_name AS
SELECT column1, column2 FROM table_name WHERE condition;
```

Example: Create a view of high-salary employees.

- Use a view:

```
SELECT * FROM view_name;
```

---

16. Indexes

- Speed up queries:

```
CREATE INDEX index_name ON table_name (column1);
```

---

17. Transactions

- Use to ensure data consistency:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

---

18. Data Types

- Common types:
  - `INT` : Numbers
  - `VARCHAR(n)` : Text
  - `DATE` : Dates
  - `BOOLEAN` : True/False

---

19. Constraints

- Enforce rules on columns:
  - `NOT NULL` : Cannot be empty.
  - `UNIQUE` : Unique values only.
  - `PRIMARY KEY` : Unique + Not Null.
  - `FOREIGN KEY` : Links to another table.

---

20. Stored Procedures

- Save a query for reuse:

```
CREATE PROCEDURE procedure_name()
BEGIN
  SELECT * FROM table_name;
END;
```