→ DDL (Data Definition Language)Commands

a. **`CREATE`** statement**:**

i. To use any data from a table, first we need to create a database using `CREATE DATABASE` command

ii. Syntax to create database:

```
CREATE DATABASE database_name;
```

Example to create database:

```
CREATE DATABASE hr;
```

iii. The `CREATE TABLE` statement is used to create a new table in a database.

iv. Syntax to create table:

```
CREATE TABLE table_name (
    column1_name datatype,
    column2_name datatype,
    column3_name datatype,
   ....
);
```

Example to create table:

```
CREATE TABLE IF NOT EXISTS `countries` (
  `COUNTRY_ID` varchar(2) NOT NULL,
  `COUNTRY_NAME` varchar(40) DEFAULT NULL,
  `REGION_ID` decimal(10,0) DEFAULT NULL,
  PRIMARY KEY (`COUNTRY_ID`),
  KEY `COUNTR_REG_FK` (`REGION_ID`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

v. To CREATE TABLE using another table

vi. Syntax to create a copy of table:

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

Example to create a copy of table:

```
CREATE TABLE employeescopy AS
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME
FROM employees;
```

b. **`ALTER`** statement**:**

i. It is used to make changes in a table using `ADD`, `DELETE`, `MODIFY` keywords

ii. Using `ALTER`, you can

1. `RENAME` a column,
2. `ADD` a column to the table,
3. `DROP` a column,

4. increase/decrease the width of a column,
5. change data type of a column,
6. copy a table,
7. copy the structure of a table,
8. change position of column in the table structure

iii. `MODIFY` is a clause which helps you in `ALTER` command

iv. Syntax to `RENAME` a column

```
ALTER TABLE table_name
RENAME COLUMN oldcolumn_name to newcolumn_name;
```

This command will rename the table (renaming a table is not recommended)

v. Syntax to `ADD` a column

```
ALTER TABLE table_name ADD column_name data_type;
```

Example:

```
ALTER TABLE employees ADD age int(60);
DESC employees;
```

vi. Syntax to `DROP` a column

```
ALTER TABLE table_name DROP column_name;
```

vii. Syntax to increase the width of a column

```
ALTER TABLE table_name MODIFY column_name new_size;
```

viii. Syntax to decrease the width of a column

```
ALTER TABLE table_name MODIFY column_name new_size_to_decrease;
```

ix. Syntax to change data type of a column

```
ALTER TABLE table_name MODIFY column_name new_data_type;
```

Example:

```
ALTER TABLE employees MODIFY age int(70);
DESC employees;
```

x. Syntax to copy a table

```
CREATE TABLE copy_table_name AS SELECT * FROM
existing_table_name;
```

xi. The structure of copy_table_name is created on the basis of the `SELECT` statement. When `SELECT` is executed, the output of `SELECT` statement will be inserted into table

xii. **<u>Note:</u>** If you don't use `WHERE` clause, all the rows will be affected for that specified column

Example:

```
ALTER TABLE employees ADD age int(60);
DESC employees;
```

xiii. In MySQL, data is truncated, if you try to insert more than the new size/decreased size

c. **`DROP`** statement

i. It removes entire database object, such as table_index, table, view, from the database

ii. This command cannot be undone once executed

iii. All the data stored in the object will be lost

iv. Syntax to drop a table

```
DROP  TABLE table_name;
```

d. **`TRUNCATE`** statement

i. It removes all the data from a table but not the table structure

ii. It is similar to `DELETE` operation, and faster than `DELETE` operation, as it does not has to deal with `WHERE` condition (searching of a particular record)

iii. This statement cannot be undone once executed

iv. Syntax to truncate

```
TRUNCATE TABLE table_name;
```

e. **`RENAME`** statement

i. Sometimes we may want to rename our table to give it a more relevant name. For this purpose we can use `ALTER TABLE` to rename the name of the table.

ii. Syntax to `RENAME` a table

```
RENAME table_name TO new_table_name;
ALTER TABLE old_table RENAME TO new_table;
```

This command will rename the table (renaming a table is not recommended)

→ DQL (Data Query Language) Command

a. **`SELECT`** statement

i. We can view a table using `SELECT` statement

ii. Here `'*'` will give the output, all of the records along with their attributes will be displayed

iii. Syntax:

```
SELECT * FROM table_name;
```

Example:

```
SELECT * FROM employees;
```

→ DML (Data Manipulation Language) Commands

a. **`INSERT INTO`** statement

   i. The `INSERT INTO` statement is used to insert new records in a table.

   ii. It is possible to write the `INSERT INTO` statement in two ways:

      1. Specify both the column names and the values to be inserted
         Syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

         Example:

```
INSERT INTO regions (REGION_ID, REGION_NAME)
VALUES (5, “Australia”);
```

      2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table.
         Syntax:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

         Example:

```
INSERT INTO regions
VALUES (5, “Australia”);
```

   iii. To copy data from one table to another, `INSERT INTO` statement is used

```
INSERT INTO new_table SELECT * FROM existing_table;
```

   <u>Note:</u> The structure of new table and existing table should be same, otherwise data won’t be inserted

b. **`UPDATE`** statement

   i. It is used to modify an existing record using `SET` keyword

   ii. Syntax:

```
UPDATE <table_name> SET <column_name>=<value> WHERE
<condition>;
```

   Example:

```
SELECT * FROM employees WHERE EMPLOYEE_ID=110;
UPDATE employees SET SALARY = 100000 WHERE EMPLOYEE_ID=110;
SELECT * FROM employees WHERE EMPLOYEE_ID=110;
```

   iii. Interview Problem / challenge

`UPDATE`	`ALTER`
It is used to modify the specific	It is used to modify the structure of

column	database object such as table_name, column_name
It uses `SET` keyword to specify new value for one or more columns and uses `WHERE` clause to specify the condition	Keywords like `ADD`, `DROP`, `MODIFY` are used to make the changes in the structure
This command can be undone using `ROLLBACK`	This command cannot be undone

c. **`DELETE`** statement

i. It is used to remove the records from a table

ii. It falls under Data Manipulation Language(DML)

iii. We can undo the statement by using ROLLBACK operation

iv. Syntax to delete

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

v. Interview Problem / challenge: `DELETE` vs `TRUNCATE`

`DELETE`	`TRUNCATE`
It is DML command	It is DDL command
`WHERE` clause is used	`WHERE` clause is not needed
It is slower as it has `WHERE` clause	It is faster as it has no `WHERE` clause
Free space is retained by table even after command execution	Free space is deallocated after command execution

→ `GROUP BY` clause

a. Under aggregate functions, we saw avg, sum, count, min, max

```
SELECT SALARY FROM employees WHERE SALARY >avg(SALARY);
```

In above query, We cannot use aggregate functions in `WHERE` clause

b. `GROUP BY` clause is used to group two rows to have same value in one or more columns

c. It typically works with aggregate functions viz. avg, sum, count, min, max

d. `GROUP BY` clause should be written after `WHERE` clause to specify which column should be grouped together

e. When you write `SELECT` statement, the column which you've grouped, should be present in `SELECT` statement

f. Example:

```
SELECT DEPARTMENT_ID, count(*) num_of_employees
FROM employees WHERE DEPARTMENT_ID=50 GROUP BY DEPARTMENT_ID;
```

→ Rules for `GROUP BY` clause:

a. Besides group function or aggregate function, whichever column is present in `SELECT` clause, that column name has to be present in `GROUP BY` clause
b. But, Whichever column is present in `GROUP BY` clause, it may or may not be present in `SELECT` statement
c. Example:

```
SELECT max(SALARY) FROM employee GROUP BY DEPARTMENT_ID;
```

In this case, `DEPARTMENT_ID` will also be bought to server RAM, sorting will be performed department wise, sorting in salary will also be performed but `DEPARTMENT_ID` will not be displayed

d. There is no upper limit in `GROUP BY` clause, if you have large number of columns in `GROUP BY` clause, but it'll be slow in execution because sorting will take time

```
SELECT JOB_ID, DEPARTMENT_ID, sum(SALARY) FROM employee GROUP BY
JOB_ID, DEPARTMENT_ID;
SELECT DEPARTMENT_ID, JOB_ID, sum(SALARY) FROM employee GROUP BY
JOB_ID, DEPARTMENT_ID;
```

e. The position of column in `SELECT` clause and the position of column in `GROUP BY` clause need not to be same
f. The position of the column in the `SELECT` clause will determine the position of the column in the output.
g. The position of column in `GROUP BY` clause will determine sorting order, grouping order
h. Spatial query, n-dimensional queries
    i. If you have 1 column in `GROUP BY` clause, this means 2D query
    ii. If you have 2 columns in `GROUP BY` columns, this means 3D query
    iii. If you have 3 columns in `GROUP BY` columns, this means 4D query
    iv. If you have multiple columns in `GROUP BY` columns, this means spatial query

→ `HAVING` clause

a. It is used to filter the result of a query based on conditions that involve an aggregate function.
b. It is used in combination with `GROUP BY` clause, which groups the row based on one or more columns
c. `HAVING` clause is applied to the grouped row and filters out any group that does not satisfy the condition
d. Syntax for `HAVING` clause:

```
SELECT <column_name_to_be_grouped>, <aggregate_funtion> FROM
<table_name> WHERE <condition> GROUP BY <column_to_be_grouped>
HAVING <condition>;
```

e. `WHERE` clause is used to restrict the row
f. `HAVING` clause works after all searching, sorting and conditioning performed on any SQL statement
g. It is recommended that only group function should be used in `HAVING` clause
h. A statement like
   `SELECT DEPARTMENT_ID, sum(sal) FROM employee GROUP BY DEPARTMENT_ID having sal>17000`
   This will give you error, as `‘sal’` is not a group function

i. `SELECT DEPARTMENT_ID, sum(sal) FROM employee GROUP BY DEPARTMENT_ID having DEPARTMENT_ID=110`
   This above statement will work but it is not an efficient way of using `HAVING` clause
j. It is recommended that only group functions should be used in `HAVING` clause

→ Interview problem / challenge : `WHERE` vs. `HAVING`

`WHERE`	`HAVING`
It filters the row depending on the condition	It filters on the group condition
It is applicable without `GROUP BY` clause	It does not work without `GROUP BY` clause
It gives you row restriction/row function	It gives you column restriction/column function
It is used before `GROUP BY` clause	It is used after `GROUP BY` clause
It is single-row operation	It is multiple-row operation as it uses aggregate function

→ `JOIN`

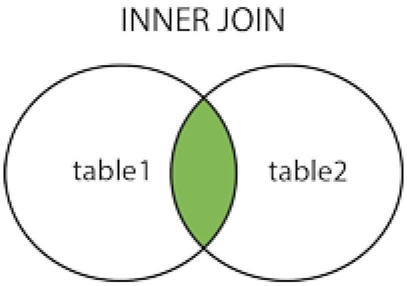
a. `JOIN` statement is used to combine data or rows from two or more tables based on common field between them
b. `JOIN` is used to view columns of two or more tables
c. `JOIN` works from left to Right

→ Types of `JOIN`

a. INNER JOIN / Equi JOIN / Natural Join:

   i. This join is based on equality conditions, so matching rows of both the tables
   ii. Both tables should have same columns / attributes

iii. It fetches common data from the same column as specified in `JOIN` condition



iv. Example – `JOIN` keyword:

```
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s JOIN course_detail c
ON s.STUDENT_ID=c.STUDENT_ID;
```

v. Example – `INNER JOIN` KEYWORD:

```
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s INNER JOIN course_detail c
ON s.STUDENT_ID=c.STUDENT_ID;
```

vi. Example – `WHERE` keyword:

```
SELECT EMPLOYEE_ID, e.DEPARTMENT_ID, DEPARTMENT_NAME
FROM employees e, departments d
WHERE e.DEPARTMENT_ID=d.DEPARTMENT_ID;
```

b. inequi `JOIN`:

i. This joins the table based on inequality conditions

ii. It'll show non-matching rows of both the tables

iii. It is used in exception reports

iv. Example:

```
-- using JOIN & ON keyword, with table alias
SELECT FIRST_NAME, LAST_NAME, EMPLOYEE_ID, e.DEPARTMENT_ID
FROM employees e JOIN departments d
ON e.EMPLOYEE_ID=d.DEPARTMENT_ID AND d.DEPARTMENT_NAME!='hr';
```

c. Cartesian `JOIN`

i. This is a join without `WHERE` clause

ii. It is the fastest join, but of no use

iii. Every row of one table is combined with every row of other table

iv. Basically used to generate combinations

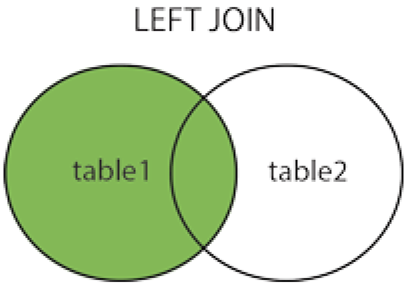
v. Example

```
SELECT s.STUDENT_ID,  c.COURSE_ID
FROM student s, course_detail c;
```

d. LEFT JOIN:

i. It matches all the rows from the left table and matching row from the right table

ii. If there is no matching row in the table, the result will contain NULL value for those columns



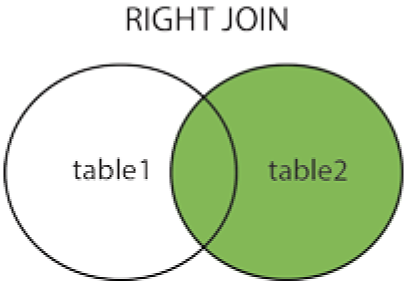
iii. Example:

```
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s LEFT JOIN course_detail c
ON s.student_ID=c.STUDENT_ID;
```

e. RIGHT JOIN

i. It returns all the rows from the right table, and matching rows from the left table

ii. If there is no matching row in the table, the result will contain NULL value for those columns



iii. Example:

```
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s RIGHT JOIN course_detail c
ON s.student_ID=c.STUDENT_ID;
```

f. SELF JOIN

i. It is a type of JOIN that joins a table to itself

ii. You can use SELF JOIN to combine rows from same table based on related columns

iii. When we say, a table joins to itself, this means two copies of the same table are used in this join, but they are treated as separate tables with different alias

iv. To distinguish tables, we use alias so that it will create copies of same table with different alias name

v. We specify JOIN condition based on one more column in a table just like other joins (We are comparing the value within the same table)

vi. SELF JOIN are used when you have table that contains recursive data

vii. Suppose you’re given a table student having attributes student_id, student_name & course_id, and you want to know how many students have the same course_id, so that you can assign the same project to them.

viii. Example:

```
SELECT s1.STUDENT_ID, s2.COURSE_ID
FROM student s1 JOIN student s2
ON s1.COURSE_ID=s2.COURSE_ID;
```

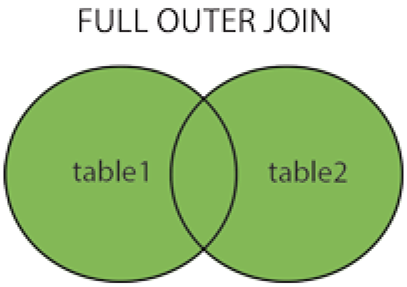
g. `FULL JOIN` / `FULL OUTER JOIN`

i. It combines result of LEFT OUTER JOIN, and RIGHT OUTER JOIN

ii. This will match all the matching rows from both the tables as well as non-matching rows from both the tables

iii. The resulting table will combine all the rows from both the tables where matching rows are combined and non-matching rows are returned with NULL values

iv. `‘FULL JOIN’` and `‘FULL OUTER JOIN’` are same



v. Example:

```
-- cannot implement FULL JOIN in MySQL
SELECT s.COURSE_ID, c.COURSE_ID
FROM student s FULL JOIN course_detail c
ON s.COURSE_ID=c.COURSE_ID;

-- FULL JOIN implemented in MySQL using UNION of LEFT JOIN &
RIGHT JOIN
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s LEFT JOIN course_detail c
ON s.student_ID=c.STUDENT_ID
UNION
SELECT s.STUDENT_ID, c.COURSE_ID
FROM student s RIGHT JOIN course_detail c
ON s.student_ID=c.STUDENT_ID;
```

h. OUTER JOIN

i.

j. employees

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
100	Steven	King	SKING	515.123.4567	1987-06-17	AD_PRES	24000.00	0.00	0	90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	1987-06-18	AD_VP	17000.00	0.00	100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	1987-06-19	AD_VP	17000.00	0.00	100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	1987-06-20	IT_PROG	9000.00	0.00	102	60
104	Bruce	Ernst	BERNST	590.423.4568	1987-06-21	IT_PROG	6000.00	0.00	103	60
105	David	Austin	DAUSTIN	590.423.4569	1987-06-22	IT_PROG	4800.00	0.00	103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	1987-06-23	IT_PROG	4800.00	0.00	103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	1987-06-24	IT_PROG	4200.00	0.00	103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	1987-06-25	FI_MGR	12000.00	0.00	101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	1987-06-26	FI_ACCOUNT	9000.00	0.00	108	100
110	John	Chen	JCHEN	515.124.4269	1987-06-27	FI_ACCOUNT	100000.00	0.00	108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	1987-06-28	FI_ACCOUNT	7700.00	0.00	108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	1987-06-29	FI_ACCOUNT	7800.00	0.00	108	100

k. departments

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700

l. Equi Join

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
100	90	Executive
101	90	Executive
102	90	Executive
103	60	IT
104	60	IT
105	60	IT
106	60	IT

107	60	IT
108	100	Finance
109	100	Finance
110	100	Finance

m. asa

→

→ TO-DOs

f.

g.

h. Date & time methods

i. Transactions