MySQL Notes

Jihuan Zhang

1 Selecting Sub-tables

1.1 Selecting Columns

The SELECT ... FROM ... statement is used to select and display data from a database.

```
SELECT * FROM table1; -- select the entire table
SELECT col1, col2, ... FROM table1; -- select all rows for specific columns from a table
```

1.2 Selecting Rows

1.2.1 Filtering Conditions

The WHERE clause is used to filter rows. Logic operators: AND, OR, NOT. Comparison operators: = (not "=="), !=/<>, <, <=, >, >=, IN.

```
SELECT * FROM table1 WHERE col1 <= num AND col2 = 'string1'; -- operators in sql are vectorized
```

1.2.2 Hard Display*

LIMIT and OFFSET are used to restrict the number of rows to display[1].

```
SELECT * FROM table1 ORDER BY col1 LIMIT 3; -- display the first 3 rows (ordered)
SELECT * FROM table1 ORDER BY col1 DESC LIMIT 3; -- display the last 3 rows
SELECT * FROM table1 ORDER BY col1 LIMIT 3 OFFSET 1; -- display first 3 rows ignoring row 1 (i.e., rows 2-4)
```

1.3 Sort

ORDER BY is used to sort the rows. Default is ascending order. Use ASC/DESC to specify order. Also works for strings.

```
SELECT * FROM table1 ORDER BY col1 DESC, col2 ASC, ...; -- lexicographic order
```

1.4 Changing Column Names

AS (Aliases) is Used to change column names (optional).

```
SELECT col1 AS new_colname FROM table1;
SELECT col1 new_colname FROM table1; -- AS is optional
```

2 Modifying Tables Permanently

2.1 Modifying Columns

2.1.1 Changing Column Names

One at a time. You need 1000 lines of commands to change 1000 column names. Use Python to iteratively generate 1000 lines of commands and then paste it in SQL.

```
ALTER TABLE table1 RENAME COLUMN col1 TO new_colname;
```

Data types: DATE, VARCHAR(255), BOOLEAN, DECIMAL(10,2).

[1] They are supported by SQL Server and Oracle, not MySQL

2.1.2 Adding New Column

```
ALTER TABLE table1 ADD COLUMN new_col VARCHAR(255); -- a column of NULL
ALTER TABLE table1 ADD COLUMN new_col VARCHAR(255) DEFAULT value1; -- a column of value1
ALTER TABLE table1 ADD COLUMN new_col VARCHAR(255) NOT NULL DEFAULT value1; -- this column won't allow for
    NULL in the future
```

2.1.3 Deleting Columns

```
ALTER TABLE table1 DROP COLUMN col1, col2, ...
```

2.2 Modifying Rows

2.2.1 Changing Row Values

```
UPDATE table1 SET col1 = val1, col2 = val2, ... WHERE conditions
```

2.2.2 Insert New Rows

```
INSERT INTO table1 (col1, col2, ...) VALUES (val1, val2, ...); -- rest entries will be NULL
```

2.2.3 Delete Rows

3 Operators

- BETWEEN num1 AND num2: num1 $\leq \cdot \leq$ num2. In many SQL versions, num1 must not be greater than num2.

  ```
  SELECT * FROM table1 WHERE col1 BETWEEN 2*PI() AND SQRT(2500);
  ```

  $\pi =$ PI(), $\sqrt{\cdot} =$ SQRT($\cdot$).

- LIKE 'string' (case-insensitive); LIKE BINARY 'string' (case-sensitive).
  Wildcards ( ): % (zero or more arbitrary characters), _ (one arbitrary character).

  ```
  SELECT * FROM table1 WHERE col1 LIKE 'a%'; -- start with 'a'
  SELECT * FROM table1 WHERE col1 LIKE '%a'; -- end with 'a'
  SELECT * FROM table1 WHERE col1 LIKE '%a%'; -- contain 'a'
  SELECT * FROM table1 WHERE col1 LIKE '_a%' -- second character is 'a'
  SELECT * FROM table1 WHERE col1 LIKE '%a_' -- second to last character is 'a'
  SELECT * FROM table1 WHERE col1 LIKE '%a_%' -- contain 'a' followed by at least one character
  SELECT * FROM table1 WHERE col1 LIKE '%_a_%' -- contain 'a' in between two characters
  ```

- A space is a character in SQL.

  Operator presedence ( ): () > everything else > NOT > AND > OR.

- IN($\cdot$):

  ```
  SELECT * FROM table1 WHERE col1 IN (num1, num2, num3);
  SELECT * FROM table1 WHERE col1 = num1 OR col1 = num2 OR col1 = num3; -- equivalent
  SELECT col1 FROM table11 WHERE col2 IN (SELECT col3 FROM table2 WHERE col4 = 'string1'); -- subqueries
      can be used
  ```

- IS NULL:

  ```
  SELECT * FROM table1 WHERE col1 IS NULL;
  SELECT * FROM table1 WHERE col1 IS NULL AND col2 IS NULL;
  ```

4 Modifying a Table

- The `INSERT INTO ... VALUES ...` statement is used to insert new rows in a table.

```
INSERT INTO table1 VALUES (val1, val2, ...); -- insert a row of full length
INSERT INTO table2 (col1, col2, ...) VALUES (val1, val2, ...); -- insert a sub-row
INSERT INTO table1 VALUES (val1, val2, ...), (val1, val2, ...), ...; -- insert multiple rows
```

- The `UPDATE ... SET` statement is used to modify the existing rows.

```
UPDATE table1 SET col1 = val1, col2 = val2, ... WHERE condition;
```

- The `DELETE FROM` statement is used to delete existing rows in a table. The `DROP TABLE` statement delete the entire table.

```
DELETE FROM table1 WHERE condition; -- delete rows satisfying the condition
DELETE FROM table1; -- delete all rows
DROP TABLE table1; -- delete the table
```

- Reverse or commit the modifications using `BEGIN`, `ROLLBACK`, and `COMMIT`.

```
BEGIN;  -- Start a transaction/modification

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

-- If you change your mind:
ROLLBACK;  -- undo both updates

-- If you're sure:
COMMIT;    -- make both updates permanent
```

4.1 Aggregate Functions

- `DISTINCT` finds unique values in rows; can be combined with `COUNT()`.

```
SELECT COUNT(*) FROM table1; -- count the number of rows
SELECT DISTINCT col1, col2, ... FROM table1; -- select unique rows wrt (col1, col2, ...)
SELECT COUNT(DISTINCT col1, col2, ...) FROM table1; -- count the number of unique rows wrt (col1, col2
    , ...)
```

- `COUNT()`, `MIN()`, `MAX()`, `AVG()`, and `SUM()` ignore `null` values, except for `COUNT(*)`.

```
SELECT MAX(col1) FROM table1 WHERE condition;
SELECT SUM(Quantity * 10) FROM OrderDetails;
SELECT OrderID, SUM(Quantity) AS TotalQuantity FROM OrderDetails GROUP BY OrderID;
SELECT * FROM Products WHERE price > (SELECT AVG(price) FROM Products); -- Return all products with a
    price above average
```

5 Join

5.1 Basic Usage

`JOIN ... ON ...`, or `INNER JOIN ... ON ...`, is used to select rows that have matching values in both tables. Use "`.`" in `table1.col1` and `table2.col2`. `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN` are used in the same way.

```
SELECT table1.col1, table2.col2, ...
FROM table1 JOIN table2 ON table1.key1 = table2.key1 AND table1.key2 = table2.key2 AND ...
```

Join three tables:

```
SELECT table1.col1, table2.col2, ...
FROM ((table1 JOIN table2 ON table1.key1 = table2.key1) JOIN table3 ON table1.key2 = table3.key2);
```

Self join example:
Select pairs of customer names that are from the same city

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

Select and concatenate all customer names that are from the same city

```
SELECT City,
GROUP_CONCAT(CustomerName SEPARATOR ', ') AS AllCustomersInCity
FROM Customers
GROUP BY City;

-- sort the names and cities
SELECT City,
GROUP_CONCAT(CustomerName ORDER BY CustomerName SEPARATOR ', ') AS AllCustomersInCity
FROM Customers
GROUP BY City
ORDER BY City;

-- ignore city that only has one customer
SELECT City,
GROUP_CONCAT(CustomerName ORDER BY CustomerName SEPARATOR ', ') AS AllCustomersInCity
FROM Customers
GROUP BY City HAVING COUNT(*) > 1 -- only group cities that appear more than once
ORDER BY City;
```

`UNION`: merge columns from two tables and drop all duplicates. `UNION ALL`: merge columns from two tables without dropping duplicates.

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers;

SELECT 'Customer' AS Type, ContactName, City, Country -- create a column named "Type" such that all its
    elements are "Customer" strings.
FROM Customers
UNION
SELECT 'Supplier', ContactName, City, Country
FROM Suppliers;

SELECT City FROM Customers1
UNION
SELECT City FROM Customers2
UNION
SELECT City FROM Suppliers; -- can union multiple tables
```

5.2 Explanation

Details of `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`:

Let the `employee` table be

ID	Name	DeptID
1	Alice	10
2	Bob	20
3	Carol	10
4	David	40

and the `department` table be

DeptID	DeptName
10	Sales
20	Marketing
30	IT
50	Finance

Then, "`employee JOIN department ON employee.DeptID = department.DeptID`" creates the following table:

employee.ID	employee.Name	employee.DeptID	department.DeptID	DeptID.DeptName
1	Alice	10	10	Sales
2	Bob	20	20	Marketing
3	Carol	10	10	Sales

"`employee RIGHT JOIN department ON employee.DeptID = department.DeptID`" creates the following table:

employee.ID	employee.Name	employee.DeptID	department.DeptID	DeptID.DeptName
1	Alice	10	10	Sales
2	Bob	20	20	Marketing
NULL	NULL	NULL	30	IT
NULL	NULL	NULL	50	Finance

"`employee LEFT JOIN department ON employee.DeptID = department.DeptID`" creates the following table:

employee.ID	employee.Name	employee.DeptID	department.DeptID	DeptID.DeptName
1	Alice	10	10	Sales
2	Bob	20	20	Marketing
3	Carol	10	10	Sales
4	David	40	NULL	NULL

"`employee FULL JOIN department ON employee.DeptID = department.DeptID`" creates the following table:

employee.ID	employee.Name	employee.DeptID	department.DeptID	DeptID.DeptName
1	Alice	10	10	Sales
2	Bob	20	20	Marketing
3	Carol	10	10	Sales
4	David	40	NULL	NULL
NULL	NULL	NULL	30	IT
NULL	NULL	NULL	50	Finance

Therefore, combined with `SELECT ... FROM ...`, it is easy to understand what's happening. Also, it is clear that table1 LEFT JOIN table2 is the same as table2 RIGHT JOIN table1, with the only difference be the column order.

Now, observe that in the above example, the matching columns have unrepeated values in at least one of the two tables (in the example, Dept.ID in `department` is unrepeated). What if both have repeated values?

When the join columns aren't unique in either table, you get a Cartesian product of the matching rows - every matching row from table1 is combined with every matching row from table2. For example, let the `department` table be

DeptID	DeptName
10	Sales
20	Marketing
10	Sales

instead.

Then, "`employee JOIN department ON employee.DeptID = department.DeptID`" creates the following table:

employee.ID	employee.Name	employee.DeptID	department.DeptID	DeptID.DeptName
1	Alice	10	10	Sales
1	Alice	10	10	Sales
2	Bob	20	20	Marketing
3	Carol	10	10	Sales
3	Carol	10	10	Sales

5.3 Group by

- `GROUP BY`: this is very simple and straightforward.

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

The `HAVING` clause is used after `GROUP BY` to place condition on which values of column being grouped by to display.

```
SELECT col1, col2, ...
FROM table1
WHERE condition1
GROUP BY col3
HAVING condition2;
```

6 Time Series Data

Create a table consisting of stock name (ticker), time (ts), and prices.

```
CREATE TABLE stock_prices (
  ticker VARCHAR(10),
  ts TIMESTAMP,
  price DECIMAL(10,2)
  );

INSERT INTO stock_prices VALUES
  ('TSLA', '2025-09-01 09:30:00', 185.20),
  ('TSLA', '2025-09-01 09:31:00', 185.35),
  ('TSLA', '2025-09-01 09:32:00', 185.10);

SELECT ts, price,
AVG(price) OVER (ORDER BY ts ROWS 4 PRECEDING) AS moving_avg
FROM stock_prices;
```