CREATING DATABASE

```
CREATE DATABASE  _databaseName;
```

TO USE A DATABASE

```
USE _databasename;
```

TO DROP A DATABASE

```
DROP DATABASE _databasename;
```

TO SET A DATABASE TO READ-ONLY

```
DROP DATABASE _databasename;
```

TO SET A DATABASE TO READ-ONLY

This means that we can't modify the database, but we can still access it.

```
ALTER DATABASE _databasename READ ONLY = 1;
```

TO DISABLE THE READ-ONLY MODE OF DATABASE

```
ALTER DATABASE _databasename READ ONLY = 0;
```

TABLES

TO CREATE A TABLE

Each column is separated by a comma

```
CREATE TABLE _tableName (
        columnName _dataType,
        columnName _dataType,
        columnName _dataType
);
```

TO RENAME A TABLE

```
RENAME TABLE _oldTableName to _newTableName;
```

TO DROP A TABLE

```
DROP TABLE _tableName;
```

ALTERING TABLE

TO ADD COLUMN TO AN EXISTING TABLE

```
ALTER TABLE _tableName
ADD _newColumnName _dataType;
```

TO RENAME COLUMN OF AN EXISTING TABLE

```
ALTER TABLE _tableName
RENAME COLUMN _oldColumnName to _newColumnName;
```

TO CHANGE THE DATA TYPE OF A COLUMN FROM AN EXISTING TABLE

```
ALTER TABLE _tableName
MODIFY COLUMN _columnName _newDataType;
```

TO MOVE COLUMNS

```
ALTER TABLE _tableName
MODIFY _columnName _dataType;
AFTER _nameOfTheColumnWeWantOurColumnToComeAfter;
```

TO MOVE COLUMNS AT THE FIRST PLACE

```
ALTER TABLE _tableName
MODIFY _columnName _dataType;
FIRST;
```

TO DROP A COLUMN

```
ALTER TABLE _tableName
DROP COLUMN _columnName;
```

TO INSERT A ROW

The values passed to the VALUES must also match the data type of that column.

```
INSERT INTO _tableName
VALUES (value1, value2, value3);
```

TO INSERT MULTIPLE ROW

The values passed to the VALUES must also match the data type of that column.

```
INSERT INTO _tableName
VALUES (value1, value2, value3),
       (value1, value2, value3)
       (value1, value2, value3) ;
```

TO ADD A ROW WITH INCOMPLETE PARAMETERS

Scenario: *The table has (value1, value2, value3, value4), but you only want to pass first the 3. Follow the syntax below:*

The parentheses after the _tableName determines the values you only want to pass.

INSERT INTO _tableName (columnName1, columnName2, columnName3)
VALUES (value1, value2, value3);

TO SELECT DATA FROM TABLE

"" denotes all columns from the table.*
Change the "" to the name of the column you only want to get.*

SELECT * from _tableName;

SELECT _columnName from _tableName;

Note: *You can add DISTINCT after the SELECT to remove all duplicates from the result set.*

TO SELECT DATA USING CONDITIONS FROM TABLE

SELECT * from _tableName;
WHERE (condition);

Example:
*SELECT * from _tableName;*
WHERE employee_id = 1;

!= -> to select from the data that is not equal to
is -> instead of =, we use is (but = still works).

TO UPDATE DATA FROM THE TABLE

You can modify more than one values at one time.

UPDATE _tableName
SET _columnToBeUpdated = value;

Note: This will update all the values in the specified column.

Use a condition to only change the value of a row within a column.

UPDATE _tableName
SET _columnToBeUpdated = value
WHERE employee_id = 6;

TO DELETE A ROW FROM THE TABLE

DELETE FROM _tableName;
WHERE (condition);

Note: *Always put a where to specify what row to delete since, without it, all of the rows will be deleted.*

PUTTING A SAFEPOINT

SAFEPOINT

SET AUTOCOMMIT = OFF;

Use the following for each safepoint you wish to mark:
COMMIT;

To return to the most recent safepoint:
ROLLBACK;

CURRENT DATE AND TIME

TO GET THE CURRENT DATE AND TIME

Datatypes:
date = DATE
time = TIME

To get the current date:
CURRENT_DATE()

To get the value of tomorrow or yesterday:
CURRENT_DATE() + 1 or CURRENT_DATE() - 1

To get the current time:
CURRENT_TIME()

For the combined current date and time:
NOW()

UNIQUE, NOT NULL, CHECK

SETTING VALUES IN COLUMN TO BE UNIQUE

When a column has a unique constraint, duplicate values in that column are restricted.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType UNIQUE
),
```

When modifying an existing table to add a constraint,

```
ALTER TABLE _tableName
ADD CONSTRAINT
UNIQUE(_columnName);
```

SETTING VALUES IN COLUMN TO BE NOT NULL

When the column is not null, the values in it can't be null.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType NOT NULL
),
```

When modifying an existing table,

```
ALTER TABLE _tableName
MODIFY _columnName _dataType NOT NULL;
```

PUTTING A CONDITION IN EACH VALUES IN A COLUMN

When the column has a not null, values in it can't be null.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType,
    CONSTRAINT _constraintName CHECK (condition)
),
```

Example:

```
CREATE TABLE _tableName(
    hourlyPay int,
    CONSTRAINT chk_hourly_pay CHECK (hourlyPay <= 10)
);
```

When modifying an existing table,

```
ALTER TABLE _tableName
ADD CONSTRAINT _constraintName CHECK (condition)
```

SETTING A DEFAULT VALUE FOR A VALUE IN COLUMN

When the column has a default value, you don't need to include that column; just specify which values you want to add to the column.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType, DEFAULT _defaultValue
);
```

When modifying an existing table,

```
ALTER TABLE _tableName
ALTER _columnName SET DEFAULT _defaultValue;
```

When inserting values,

Scenario: *You have 3 columns, and the last one has a default value.*

```
INSERT INTO _tableName() (_columnName1, _columnName2)
VALUES (value1, value2);
```

PRIMARY KEY, AUTO INCREMENT, FOREIGN KEY

SETTING A PRIMARY KEY

A column must be unique and not null when designated as the primary key.
There is only one primary key per column. Unique identifier.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType PRIMARY KEY
);
```

When modifying an existing table,

```
ALTER TABLE _tableName
ADD CONSTRAINT
PRIMARY KEY(_columnName);
```

SETTING AUTO INCREMENT

You can only set auto increment to a column that is a key.
When using auto-increment, the table starts at 1.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType PRIMARY KEY AUTO_INCREMENT
);
```

When modifying an existing table,

```
ALTER TABLE _tableName
ADD CONSTRAINT
PRIMARY KEY(_columnName);
```

Scenario: *We have a table that has 2 columns (transaction_id, amount).*

```
INSERT INTO transactions (amount)
VALUES (3.38)
```

The transaction ID column will automatically set the first value to 1. We will increment the next value by 1.

To set the initial starting point of the auto increment,

```
ALTER TABLE _tableName
AUTO_INCREMENT = _startingValue; (e.g. 1000)
```

SETTING UP A FOREIGN KEY

To obtain a reference from the primary key of another table, use a foreign key.
This means that the key allows us to create a link between two tables.

Example:
TABLE transactions
TABLE customers

If TABLE customers has a primary key of customers_id, the TABLE transactions can get a reference from that key.

When creating a table,

```
CREATE TABLE _tableName(
    _columnName _dataType
    FOREIGN KEY(_columnName) REFERENCES _tableName(_columnNameInThisTable)
);
```

When dropping a foreign key,

To get the name of the foreign key, go to your database in MySQL, click the drop-down menu of your table, and then select the foreign key.

```
ALTER TABLE _tableName
DROP FOREIGN KEY _foreignKeyName;
```

When modifying an existing table,

```
ALTER TABLE _tableName
ADD CONSTRAINT _foreignKeyUniqueName
FOREIGN KEY(_columnName) REFERENCES _tableName(_columnNameInThisTable);
```

Scenario: *We have a table that has 2 columns (transaction_id, amount).*

```
INSERT INTO transactions (amount)
VALUES (3.38)
```

The transaction ID column will automatically set the first value to 1. We will increment the next value by 1.

To set the initial starting point of the auto-increment.

```
ALTER TABLE _tableName
AUTO_INCREMENT = _startingValue; (e.g. 1000)
```

Note: It will return an error if you delete data from the parent table that another table uses as a foreign key.

One way to solve that issue is to set the foreign key check to 0. Set it to 1 if you want to turn it on.

```
SET foreign_key_checks = 0;
```

The second way is to use ON DELETE NULL or ON DELETE CASCADE.

ON DELETE NULL: *When an fk is deleted, replace the foregin key with NULL.*
ON DELETE CASCADE: *When an FK is deleted, delete the whole row.*

```
CREATE TABLE _tableName(
    _columnName _dataType
    FOREIGN KEY(_columnName) REFERENCES _tableName(_columnNameInThisTable)
    ON DELETE NULL
);
```

The third method involves initially deleting the rows in the child table that use a foreign key. The pattern is similar to the child-to-parent table.

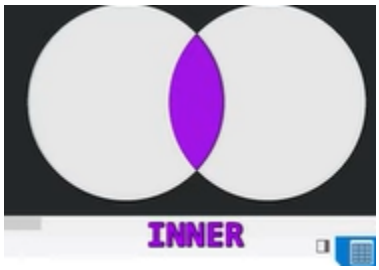
JOIN TABLES

Imagine we have these two tables:

transactions		
transaction_id	amount	customer_id
1000	4.99	3
1001	2.89	2
1002	3.38	3
1003	4.99	1
1004	1.00	

customers		
customer_id	first_name	last_name
1	Fred	Fish
2	Larry	Lobster
3	Bubble	Bass
4	Poppy	Puff

INNER/LEFT/RIGHT JOIN



SELECT *
FROM _tableYouWantOnTheLeft INNER/LEFT/RIGHT JOIN _tableYouWantOnTheRight
ON **_tableNameWithForeignKey(LEFT)._columnNameThatHasForeignKey = _tableName(RIGHT)._columnNameThatHasPrimaryKey**

Example:

SELECT * FROM customers INNER/LEFT/RIGHT JOIN transactions ON customers.customer_id = transactions.customer_id

SELF JOIN

Self-join is a process where we join another copy of a table to itself. In essence, we attached a duplicate of the current table to itself.

Note: *Any joining pattern would work here, whether it's LEFT, RIGHT or INNER. Additionally, we need to use an alias for the table and its copy.*

SELECT * FROM _table AS _alias INNER JOIN _table as _anotherAlias ON _table.(attributeToCompare) = _table.(attributeToCompare);

SELECT * FROM customers AS a INNER JOIN customers AS b ON a.customer_id = b.referral_id

WILD CARD CHARACTERS

Note: *LIKE is used here instead of =. We check for patterns rather than value equality.*

%
We use this to check or find data inside the database that begins or ends with the specified requirements.

Scenario: *Suppose we want all customers to have a first name that starts with s.*

SELECT * FROM customers where name LIKE “s%”.

%(word) *– ends with*
(word)% *- starts with*

_

This is used to check or find the data inside the database that contains specific data with a given length. To better understand, it's like filling in the blanks.
.

Scenario: *Suppose we want to get all the customers that end with ook but only a four-letter word.*

SELECT * FROM customers where name LIKE "_ook".

This checks all the data for all of the data that starts with any letter but ends with ook.

Moreover, we can combine these two.

Scenario: *Suppose we want to find a word inside our database that has the second letter "a.".*

SELECT * FROM customers WHERE name LIKE "_a%".

This simply means that the first letter can be any letter, but the following letter must start with the letter a.

ORDER BY

SELECT * FROM _tableName ORDER BY _columnYouWantToBeOrdered;

The default arrangement for this is ascending ASC (low to high). If you want to reverse the arrangement into a descending one, just add DESC after the column name.

SELECT * FROM _tableName ORDER BY _columnYouWantToBeOrdered DESC;

Furthermore, you can apply the order by method to multiple arguments.

Scenario: *Suppose you want to order the transactions based on the amount of their transactions, but if two amounts are the same, order them more by using their customer ID.*

SELECT * FROM _tableName ORDER BY _columnYouWantToBeOrdered, _secondColumnToBeUsed;

LIMIT

We use LIMIT to limit the data fetched from a table.

Scenario: *Assume we only want to get the first two data points from the database.*

SELECT * FROM customers LIMIT 2;

Furthermore, we can use limits and offsets.

Scenario: *Let's say we wish to obtain data that is adjacent to our target data.*

SELECT * FROM customers LIMIT 1, 1

The first one is the offset, and the second one is the limit.

UNION

To combine the results from two select statements, use UNION.

Note: *In order for you to use the UNION, the results must have the same columns; otherwise, it will not work.*

SELECT * FROM customers
UNION
SELECT * from employees;

This will automatically remove all duplicates. But if you want to have the duplicates available, add ALL after the UNION (UNION ALL).

VIEWS

Views are similar to virtual tables in that they are based on one or more columns from the database tables (result set), but they can be interacted with just like the tables.

The main purpose of views is to prevent creating another table that already contains the same data.

TABLE
customer_id, first_name, last_name

VIEWS

Scenario: *Suppose we want to get the first and last names from the table. It is not practical to create another table for that set of data, given that we already have that data inside a table.*

CREATE VIEW _viewName AS (SQL statement that will return a result set)
CREATE VIEW _viewName AS SELECT first_name, last_name FROM employees;

It's crucial to remember that the values within the view update instantly. This implies that any changes made to the table underlying the views will automatically update the values.

DROP VIEW _viewName

INDEXES

We use indexes to quickly find values within a specific column.

Basically, the main purpose of the indexes is to shorten the time needed to find data from a database table column.

Note: *Indexes are useful for searching but undesirable for updating. Searching (fast) Updating (slow)*

To show all of the current indexes,

SHOW INDEXES FROM _tableName

To add index,
CREATE INDEX _indexName ON _tableName(_columnNames)

To drop an index,
ALTER TABLE _tableName DROP INDEX _indexName

GROUP BY

We use GROUP BY to aggregate all rows by a specific column. This is often used with aggregate functions such as SUM, AVG, COUNT, and so on.

Scenario: *Suppose we have this table:*

transaction_id	customer_id	amount	order_date
1001	4.99	1	2023-01-01
1002	4.89	2	2023-01-02

Suppose we want to find the total amount of transactions per day.

SELECT SUM(amount), order_data FROM transactions GROUP BY order_date

Note: *If we want to use WHERE together with the GROUP BY, it will not work; use HAVING instead.*

Scenario: *Suppose we want to get the dates that has a total transaction amount of greater than 4.*

SELECT SUM(amount), order_data FROM transactions GROUP BY order_date HAVING SUM(amount) < 4;

ROLLUP

Roll-up is an extension of GROUP BY. It adds another row after the total result set, showing the grand total.

Scenario: *Suppose we have this table:*

transaction_id	customer_id	amount	order_date
1001	4.99	1	2023-01-01
1002	4.89	2	2023-01-02

Suppose we want to find the total number of transactions per day.

SELECT SUM(amount), order_data FROM transactions GROUP BY order_date

Now, if we want to get the total amount from the result set, we can use ROLLUP.

What we can do is just put WITH ROLLUP after the GROUP BY statement.

SELECT SUM(amount), order_data FROM transactions GROUP BY order_date WITH ROLLUP.

PROCEDURE

A procedure is a type of prepared SQL code that you can save. It is like a function that can be simply called to execute a set of statements.

Scenario: *Suppose we have this table:*

transaction_id	customer_id	amount	order_date
1001	4.99	1	2023-01-01
1002	4.89	2	2023-01-02

Suppose we want to create a procedure to get lists of order dates.

```
CREATE PROCEDURE _procedureName()
BEGIN
   SELECT order_date FROM transactions;
END;
```

Now, we are faced with a dilemma: we have two semicolons, but we want the statement to terminate at END, not at the transaction. To fix this, we will change the delimiter from (;) to (// or $$).

```
DELIMITER $$
CREATE PROCEDURE _procedureName()
BEGIN
   SELECT order_date FROM transactions;
END $$
DELIMITER ;
```

Also, we can pass data inside the procedure by setting a parameter during its creation.

```
DELIMITER $$
CREATE PROCEDURE _procedureName(IN _parameterName _parameterType)
BEGIN
   SELECT order_date FROM transactions;
END $$
DELIMITER ;
```

To drop a procedure name,
```
DROP PROCEDURE _procedureName;
```

To invoke a procedure,
```
CALL _procedureName(parameters if available);
```

TRIGGERS

A trigger is used to execute an action when an event happens, such as insert, delete, or update.

CREATE TRIGGER _triggerName
TIME(BEFORE/AFTER) EVENT(UPDATE/DELETE/INSERT) ON _tableName
FOR EACH ROW
(STATEMENT)

Note: *We ensure that we change everything for each row because we may be working with more than one row.*

Additionally, prior to altering the data, we used the terms NEW or OLD. This is to tell SQL what data we are using, whether it is old or new.

Example: *NEW.salary or OLD.salary*