Extracting Data using SQL

---

1. Problem Statement
2. Relationships
3. ER Diagram
4. SQL Commands
5. SELECT Query
6. ORDER BY
7. LIMIT & OFFSET
8. Inline Calculation
9. Alias (AS)
10. ROUND() function

Please note that any topics that are not covered in today's lecture will be covered in the next lecture.

Problem Statement:

- Amazon Farmers Market (AFM), a beloved community market, faces a growing threat from large grocery chains that are expanding their local produce offerings. To maintain its customer base and thrive in this competitive environment, AFM needs to leverage data analytics for a deeper understanding of:

    - **Customer Behavior**: Analyze purchase history, spending habits, and visit frequency to identify customer segments, preferences, and potential churn risks.
    - **Vendor Performance:** Analyze sales data and customer attraction metrics to identify top-performing vendors, evaluate product popularity, and optimize vendor selection.

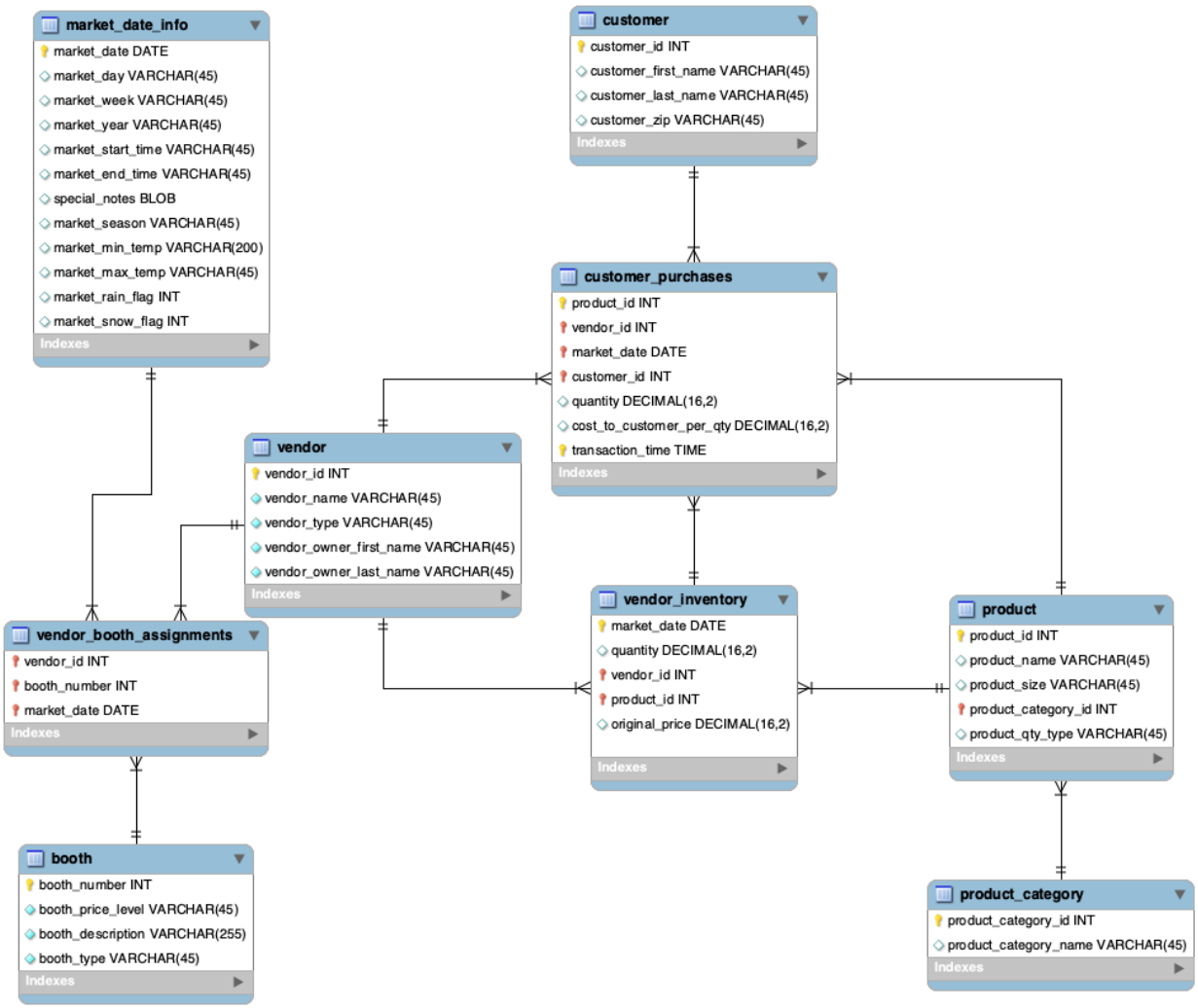
    - **Market Operations:** Analyze traffic trends and seasonal sales patterns to optimize inventory management, staffing levels, and marketing campaigns.
    - **Inventory management:** Analyze future demand for specific products based on historical sales data to optimize the right amount of inventory to meet customer needs while minimizing waste and storage costs.

- Now that you have understood the fundamental concepts of databases, your manager has assigned you the task of studying the Farmer's Market database to derive valuable insights from it.
- As an intern, you're not sure how to proceed and have several doubts, such as:
    - How are the tables connected or related?
    - How can I retrieve data that spans across multiple tables?
    - What kind of queries do I need to write to get meaningful insights?
- To address these doubts, your manager advises you to:
    - First, understand how the tables in the Farmer's Market database are connected. This involves learning about the relationships between different tables.
    - Learn how to interact with the database and extract data from each table.
    - Then explore various SQL commands and queries to retrieve and manipulate the data effectively which will help you understand customer behavior, vendor performance, market operations and inventory management.

- The main goal of AFM is to leverage the data analytics team to gain a competitive advantage, optimize inventory management, tailor marketing efforts, improve the customer experience for long-term success and solidify AFM's position as a vital community hub for fresh, local produce.

Let’s have a look at the schema of the Farmer’s market database which we already saw in the last lecture:

Dataset: Farmer’s Market database



Formulating questions to be explored based on the data provided:

- **Market Operations:**

    - How to sort the purchases by market date and transaction time to identify recent purchases?
    - Your manager is only interested in looking at the 10 most recent transactions
    - What if the manager asks you to fetch the 2nd & 3rd most recent purchase?
- **Customer Behavior:**
    - Find the total amount the customer has paid along with the date, customer ID, vendor_id, qty, cost per qty and the total amount.

These questions will help us understand customer behaviour and market trends which will help us to optimize inventory management, tailor marketing efforts, improve the customer experience for long-term success and solidify AFM's position as a vital community hub for fresh, local produce.

Relationships in a Schema

Question: How are the tables related to each other?

In the previous lecture, we learnt about different types of keys.

Now let's look at how tables are related to each other and how we define the relationships between two or more tables.

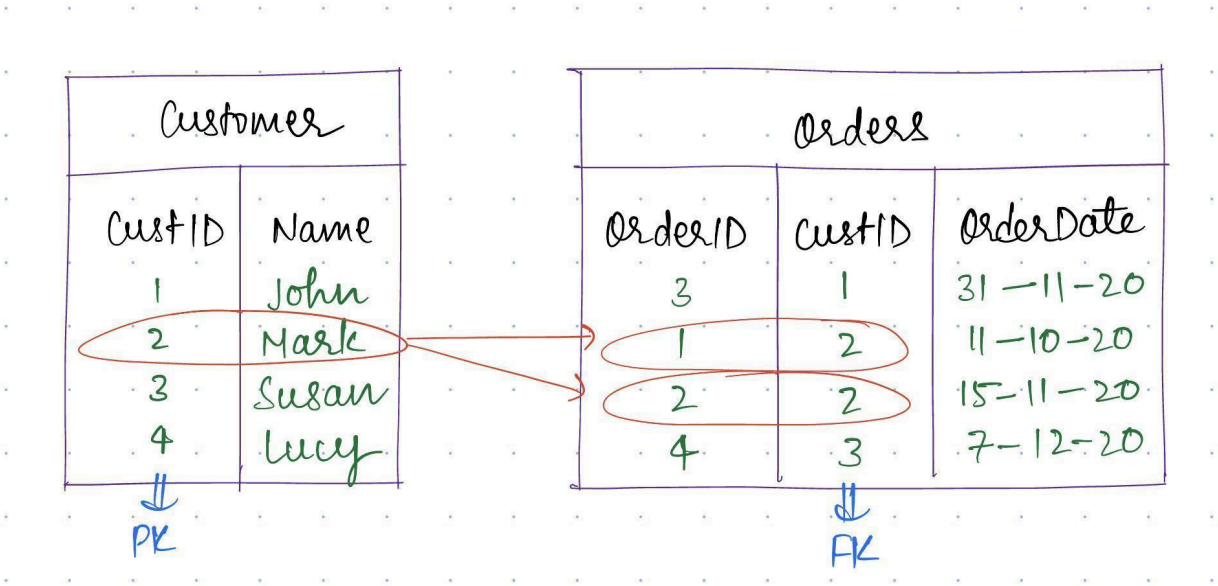
There are multiple types of relationships that two entities (or tables) can share with each other.

Different types of relationships -

- One-to-One Relationship:
    - One row in Table A will be related to one and only one row in Table B.

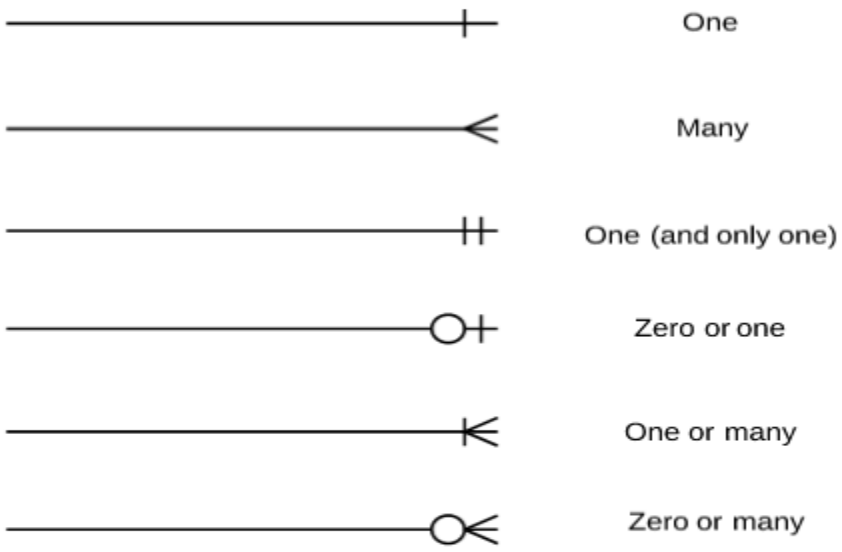
    - **Example** - marriage between a husband and his wife
        - One husband can have only one wife and vice versa.

- One-to-Many Relationship:
    - A single row in Table A is related to multiple rows in Table B.
    - A single row in Table B is related to a single row in Table A.
    - **Example** -
        - relationship between a customer and his orders
            - A customer can place multiple orders.
            - However, one order can only be placed by a single customer.



- Many-to-Many Relationship:
    - One row in Table A is related to many rows in Table B.
    - One row in Table B is related to many rows in Table A.
    - **Example** - the relationship between customers and suppliers
        - One customer can purchase from many suppliers.
        - One supplier can sell to many customers.

ER Diagram edge notations:



Question: What does the Farmer's Market database look like?

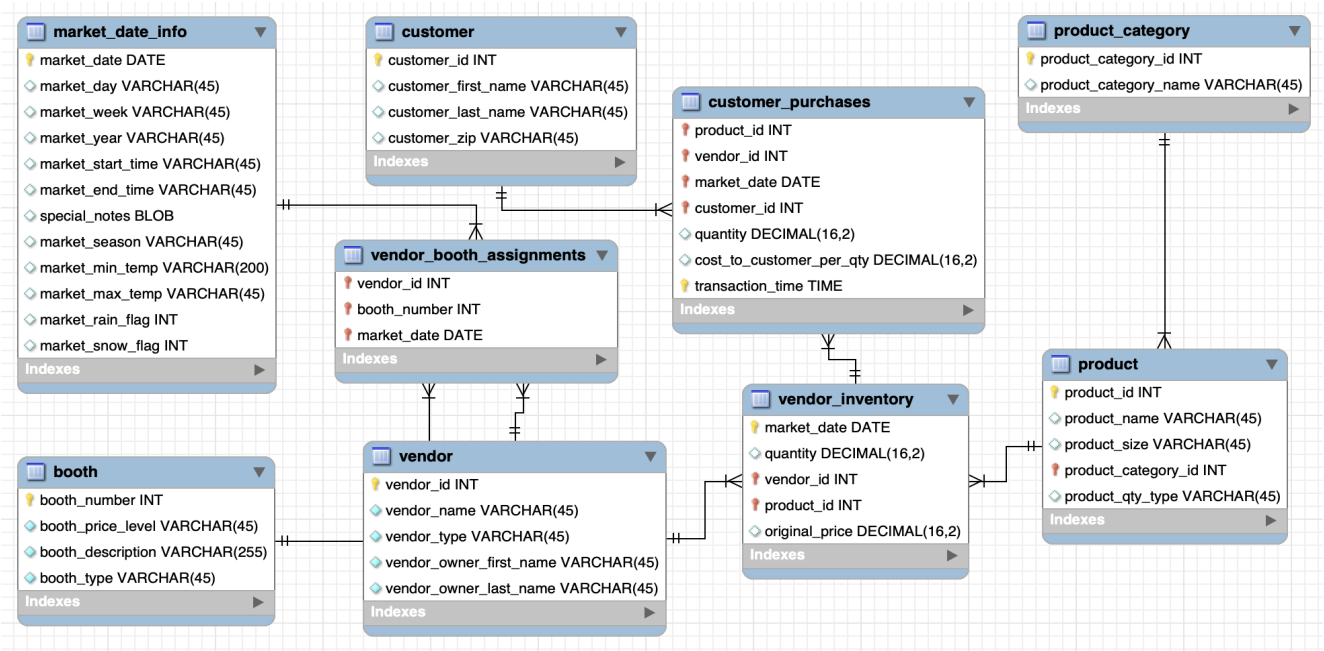
Understanding the ER diagram of the Farmer's Market database.

You should have a clear understanding of ER diagrams now.
Let's answer a few questions on the ER diagram of our Fresh Market database:

Looking at the diagram, answer the following questions:

1. What is the primary key in the customer, product and vendor tables?
2. What is the relationship between the following tables:
    a. customer and customer_purchase
    b. vendor and vendor_inventory
    c. product and product_category

3. Which attribute is the foreign key in customer_purchase, vendor_inventory and product_category tables?



So, we have defined multiple entities and how they are related to each other using these diagrams and edges to connect them.

This diagram that shows the entities along with their attributes, keys, and relationships is called an Entity-Relationship diagram.

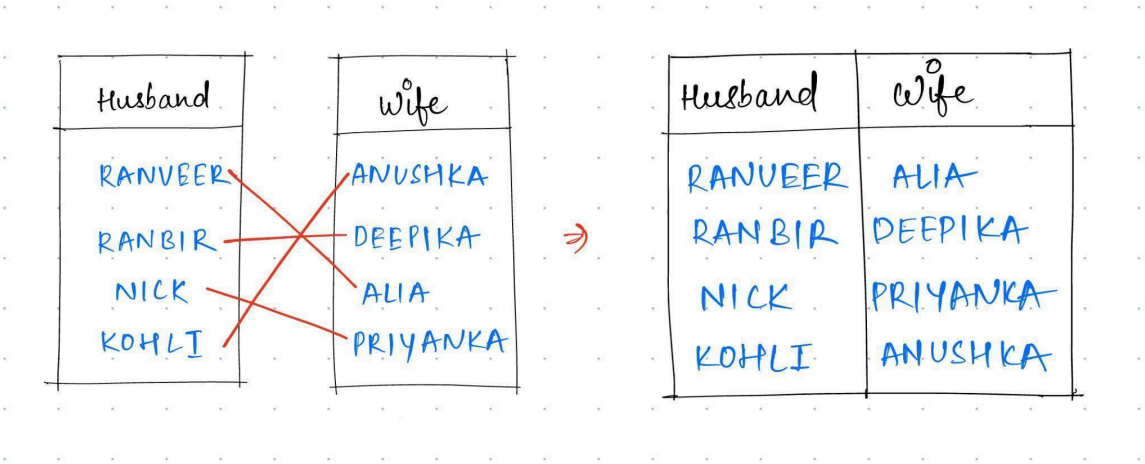
Most of the time, you'll be given this **ER diagram** which is interchangeably referred to as the **DB schema** and you'll have to extract the insights from the tables.

Issues with implementing **1-1** and **m-m** relationships:

Did you notice that we don't have any, **One-to-One** or **Many-to-Many** relationships in our ED diagram?

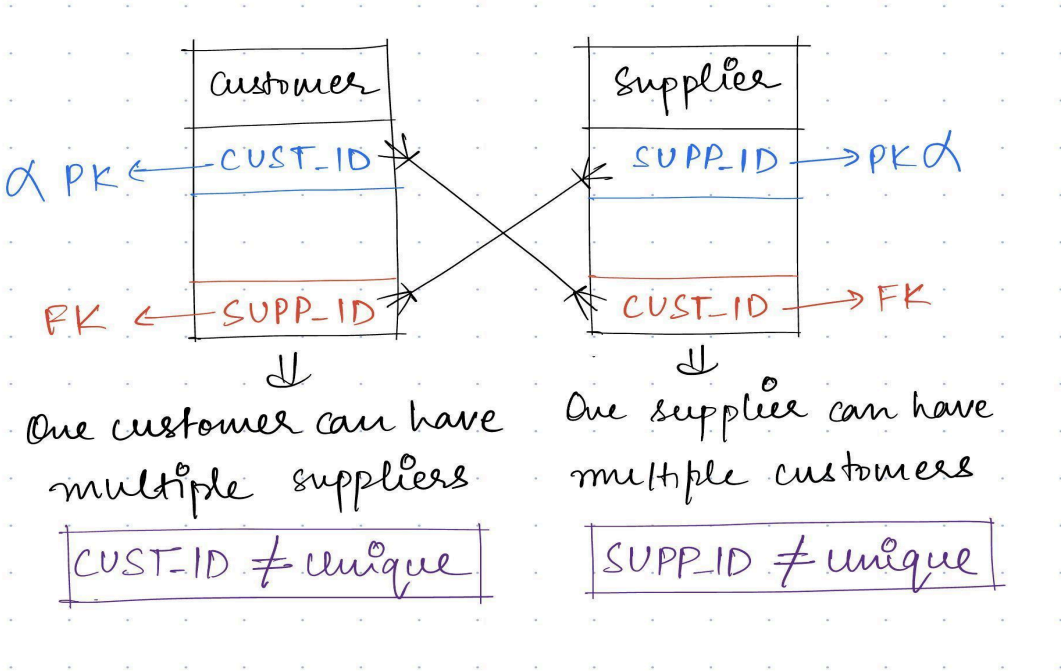
Let's understand why that's the case.

1. **One-to-One**
    a. With one-to-one relationships, do we even need two separate tables?
    b. No. Instead, we can keep this data in two different columns of a single table.



2. **Many-to-Many**

    A many-to-many relationship is hard to maintain. Why?



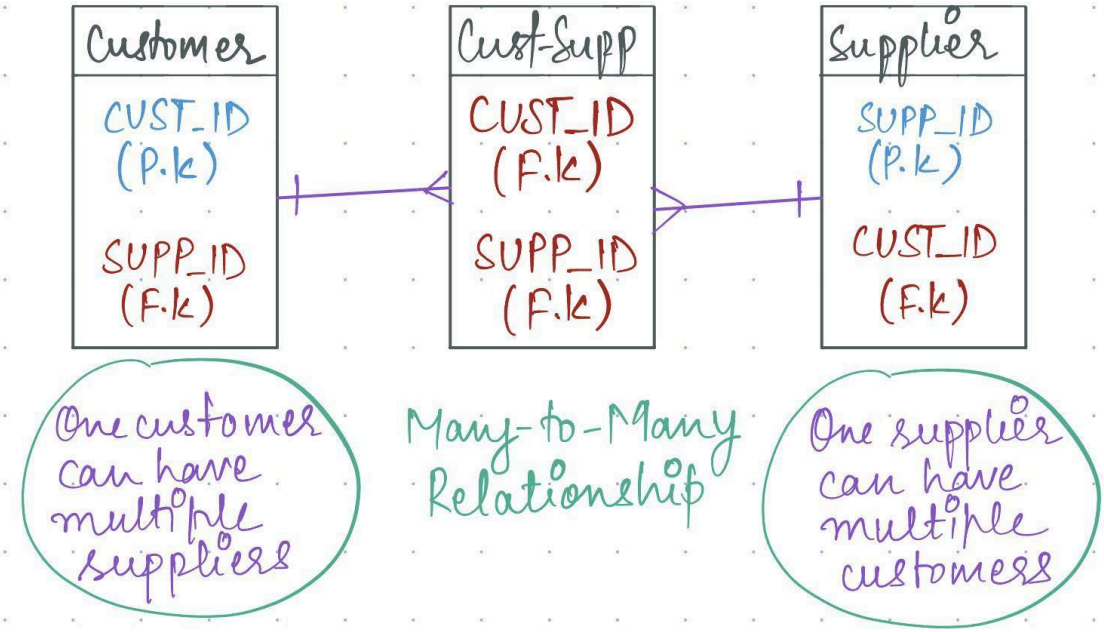
That's why to resolve this issue, a **junction table** is introduced.

- The junction table serves as an intermediary table that connects the two tables, representing the relationships between them.
- By introducing a junction table, you can break down the many-to-many relationship into two one-to-many relationships.
- Each record in the junction table represents a unique combination of a customer and a supplier, linking them together.

Based on the provided tables for Customers and Suppliers, we can create a junction table to link them.

Junction Table (Cust_Supp)

Column Name	Description
CUST_ID	CUST_ID is a foreign key referencing CUST_ID in the Customer table
SUPP_ID	SUPP_ID is a foreign key referencing SUPP_ID in the Supplier table.



- One-to-Many relationship between `Customer` and `Cust_Supp` (one customer can have multiple suppliers).
- Many-to-One relationship between `Cust_Supp` and `Supplier` (one supplier can have multiple customers).

Question: What's the relationship between vendor and product tables?

The many-to-many relationship is established using a **junction table** (vendor_inventory).

Question: How will you extract data from each of these tables?

For that, we have a specific language to interact with this relational DBMS, called **SQL**.

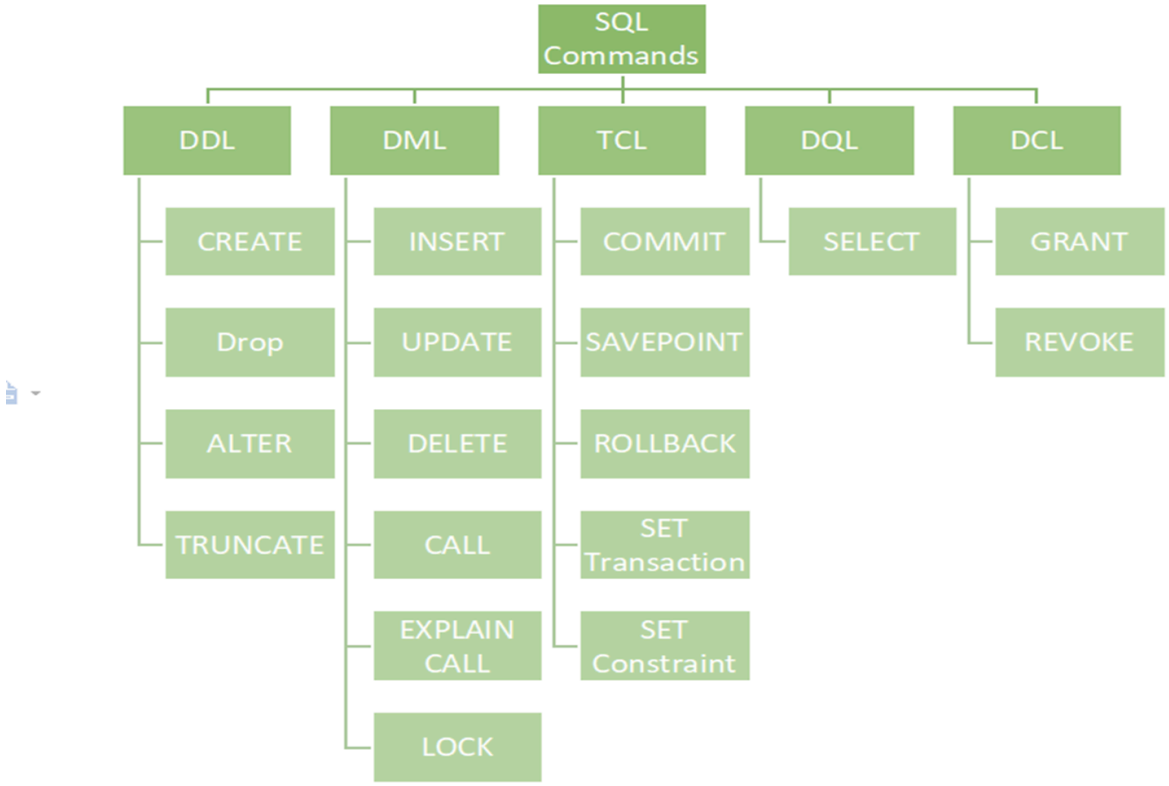
What is SQL?

- SQL stands for **Structured Query Language**

- There are different types of SQL commands depending on the type of task you want to perform.

Types of SQL commands -

- **DDL** - Data Definition Language
- **DML** - Data Manipulation Language
- **TCL** - Transaction Control Language
- **DQL** - Data Query Language - most important
- **DCL** - Data Control Language



Now, If I ask you to get me all the products from the farmer's market database, how will you do that?

Question: Get all the products available in the market.

The Fundamental Syntax Structure of a SELECT Query

In SQL, we have these SELECT statements. SELECT: A SELECT statement is the SQL code that retrieves data from the database.

SQL SELECT queries follow this basic syntax, though most of the clauses are optional:

```
SELECT [columns to return]
FROM [schema.table]
WHERE [conditional filter statements]
GROUP BY [columns to group on]
HAVING [conditional filter statements that are run after grouping]
ORDER BY [columns to sort on]
LIMIT [first x number of rows to be selected]
OFFSET [number of rows to skip]
```

To answer our question, The SELECT and FROM clauses are generally required because they indicate which columns to select and from what table.

For example,

```
SELECT * FROM `farmers_market.product`
```

can be read as:

- "Select all columns from the product table in the farmers_market schema."

Tip: Problems with (*)

- Even if you want all columns returned, it's good practice to list the names of the columns instead of using the asterisk.

- **Production pipelines:** Especially if the query will be used as part of a data pipeline (if the query is automatically run nightly and the results are used as input into the next step of a series of code functions without human review).

- This is because returning "all" columns may result in a different output **if the underlying table is modified**, such as when a new column is added, or columns appear in a different order, which could break your automated data pipeline.

- Although it's convenient to query all columns using *, **it increases the unnecessary data transferred between the database server and the application** because the application may need only partial data from the table.

To specify which columns you want to be returned, list the column names immediately after SELECT, separated by commas, instead of using the asterisk (*).

For example, select the product IDs and their corresponding product names & sizes from the `product` table.

```
SELECT product_id, product_name, product_size
FROM `farmers_market.product`
```

How to sort data in SQL?

This is where the **Order By** clause comes into play.

- The ORDER BY clause is used to sort the output rows.
- In it, you list the columns by which you want to sort the results, in order, separated by commas.
- You can also specify whether you want the sorting to be done in ascending **(ASC)** or descending **(DESC)** order.
- The sort order is **ascending by default**, so the ASC keyword is optional.

Illustration:

- Use the **Macros** created in this **GSheet** to demonstrate how **Order By** works for multiple columns.
- Make sure you **Reset the Sorted records** once done with the demonstration.

If we want to sort this entire data in **descending order** of a particular column, in this case by ***product_id*** -

```
SELECT product_id, product_name, product_size
FROM `farmers_market.product`
ORDER BY product_id DESC
```

You can also sort data by multiple columns with different sort order in the following manner -

Syntax:

```
SELECT <col1>, <col2>, ...
FROM `dataset.table_name`
ORDER BY
    <col1> ORDER,
    <col2>,
    <col3>, ...
```

The sorting order is applicable to one single column only.

Here the column written first is given priority so first the data will be sorted on the basis of <col1> then by <col2> and at the end by <col3>

Let's look at the `customer_purchases` table -

```
SELECT *
FROM `farmers_market.customer_purchases`
```

Question: How to sort the purchases by market date and transaction time both in descending order to get the most recent transaction?

For each ***market_date*** (in descending order), we'll sort the data by ***transaction_time*** (again in descending order).

```
SELECT
        market_date,
        transaction_time,
        quantity,
        cost_to_customer_per_qty
 FROM `farmers_market.customer_purchases`
 ORDER BY market_date DESC, transaction_time DESC
```

NOTE:

- **For strings,** ASC sorts text alphabetically and numeric values from low to high, and DESC sorts them in reverse order.
- **In MySQL**, NULL values appear first when sorting is done in ascending order.

Question: Your manager is only interested in looking at the 10 most recent transactions in the `customer_purchases` table.

LIMIT

By using LIMIT, you can get a preview of your current query's results without having to wait for all of the results to be compiled.

We do have the ***market_date*** column in the `customer_purchases` table.

```
SELECT
        market_date,
        transaction_time,
        quantity,
        cost_to_customer_per_qty
 FROM `farmers_market.customer_purchases`
 ORDER BY market_date DESC
 LIMIT 10
```

- **Optional clause:** LIMIT clause. You'll frequently use LIMIT while developing queries. LIMIT sets the maximum number of rows that are returned, in cases when you don't want to see all of the results you would otherwise get.
- **Computationally inexpensive:** Useful when you're developing queries that pull from a large database when queries take a long time to run since the query will stop executing and show the output once it's generated the set number of rows.

Question: What if the manager asks you to fetch the 2nd & 3rd most recent purchase (skip the 1st one)?

OFFSET

OFFSET allows you to **omit a specified number of rows before the beginning** of the result set.

```
SELECT
        market_date,
        quantity
 FROM `farmers_market.customer_purchases`
 ORDER BY market_date DESC
 LIMIT 2
 OFFSET 1
```

NOTE:

BigQuery uses the LIMIT and OFFSET clauses separately.

- The general syntax is LIMIT <count> OFFSET <offset>,
- where <count> specifies the maximum number of rows to return,
- and <offset> specifies the number of rows to skip.
- Eg. LIMIT 2 OFFSET 1

In **MySQL**,

- The OFFSET clause can either be used separately (as shown above).
- Or we can combine it with the LIMIT clause (as shown below).
    - The general syntax is LIMIT <offset>, <count>
        - where <offset> specifies the number of rows to skip
        - and <count> specifies the maximum number of rows to return.
        - E.g., LIMIT 1, 2.

Tips:

- Line breaks and tabs don't matter to SQL code execution and are treated as spaces, so you can indent your SQL and break it into multiple lines for readability without affecting the output.
- SQL reserved keywords like SELECT, FROM, WHERE, etc. are case insensitive. Do not confuse the column names, dataset names or table names to be case insensitive as well.
- Semicolon (;) is not required when you're writing a single query in the editor. In case you have multiple queries in the same tab, then only you need to add a semicolon at the end of a query.

Order of Execution of a SQL query:

- **FROM** - The database gets the data from tables in the FROM clause.
- **SELECT** - It determines which columns to include in the final result set.
- **ORDER BY** - It allows you to sort the result set based on one or more columns, either in ascending or descending order.
- **OFFSET** - The specified number of rows are skipped from the beginning of the result set.
- **LIMIT** - After skipping the rows, the LIMIT clause is applied to restrict the number of rows returned.

How do you do calculations on the go? - **Inline Calculations**

Question: In the customer purchases, we have quantity and cost per qty separate, query the total amount that the customer has paid along with the date, customer ID, vendor_id, qty, cost per qty and the total amount.

In the `customer_purchases` table, we have a ***quantity*** column and a ***cost_to_customer_per_qty*** column, so we can multiply those to get the total amount the customer has paid.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity,
    cost_to_customer_per_qty,
    quantity * cost_to_customer_per_qty
 FROM farmers_market.customer_purchases
```

The query demonstrates how the values in two different columns can be multiplied by one another by putting an asterisk between them.

When used this way, the asterisk represents a multiplication sign.

NOTE:
Along with multiplication (*), we can also perform some other arithmetic operations like addition (+), subtraction (-) and division (/).

Alias

- To give the calculated column a meaningful name, we can create an *alias* by adding the keyword **AS** after the calculation and then specifying the new name.
- If your alias includes spaces, it should be surrounded by single quotes.

Here, we'll give the alias "**price**" to the result of the "quantity times cost_to_customer_per_qty" calculation.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty AS price
 FROM farmers_market.customer_purchases
```

NOTE:
In MySQL syntax, the AS keyword is actually optional. For clarity, we will use the AS convention for assigning column aliases in this book.

Functions in SQL

- A SQL function is a piece of code that takes inputs that you give it (which are called parameters), performs some operation on those inputs, and returns a value.
- You can use **inline functions** in your query to modify the raw values from the database tables before displaying them in the output.

Function Syntax

FUNCTION_NAME([parameter 1],[parameter 2], ...[parameter n])

- Each bracketed item shown is a placeholder for an input parameter.
- Parameters go inside the parentheses following the function name and are separated by commas.
- The input parameter might be a field name or a value.

Question: Can we round off all the values in the result set to 2 decimal places?

We can easily do this by using the **ROUND()** function.

Query:

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price
 FROM farmers_market.customer_purchases
```

TIP:
The ROUND() function can also accept negative numbers for the second parameter, to **round digits that are to the left of the decimal point**.

For example, SELECT ROUND(1245, -2) will return a value of 1200.

Other similar functions are:

1. **CEIL -** rounds a number up to the nearest integer greater than or equal to the input.
   Eg. CEIL(5.4) -> 6

2. **FLOOR -** rounds a number down to the nearest integer less than or equal to the input.
   Eg. FLOOR (5.7) -> 5

Query:

```
SELECT
```

```
market_date,
customer_id,
vendor_id,
CEIL(quantity * cost_to_customer_per_qty) AS ceil_price
FLOOR(quantity * cost_to_customer_per_qty) AS floor_price
FROM farmers_market.customer_purchases
```

Insights and Recommendations:

1. **Purchasing Patterns:**
   Focusing on recent transactions allows for quick insights into current purchasing patterns and customer behavior.

   - Recommendation: Extract and analyze these transactions to identify any emerging trends, popular products, or customer preferences. This analysis can inform immediate adjustments in inventory, pricing, or marketing strategies to capitalize on current trends and enhance customer satisfaction

2. **Analyzing Immediate Preferences:**
   Fetching the 2nd and 3rd most recent purchases allows for a deeper analysis of immediate customer preferences and buying patterns, skipping the very latest transaction.

   - Recommendation: Retrieve and scrutinize these transactions to pinpoint evolving trends or shifts in customer behaviour. This focused analysis can guide adjustments in inventory management and marketing strategies to capitalize on emerging preferences

3. **Customer Spending Analysis:**
   Querying the total amount paid by each customer provides valuable insights into individual customer spending habits and overall revenue generation.

    - Recommendation: Aggregate and analyze this data to identify high-value customers, tailor personalized marketing strategies to enhance their loyalty, and optimize pricing strategies to maximize revenue

Overall Summary:

By analyzing this data, Amazon Fresh can gain valuable insights into customer behaviour, which will help them to optimize inventory management, tailor marketing efforts, improve the customer experience for long-term success and solidify AFM's position as a vital community hub for Fresh, local produce.