Masterclass SQL for DA

Problem Statement

- It’s your first day as a Data Analyst at **Reliance Fresh.**

- Reliance Fresh has recently decided to open big stores where vendors can directly sell their produce from their stores.

- **All the data is stored in a Database**

- Your manager reaches out to you and gives you a **DB schema of the backend system** that looks like this

- *<show image below>*

First thoughts that come to your mind - explain on the image:

- Where is this data stored? DB?
- What is a DB schema?
- What does each box represent in this diagram?
- What do those lines represent?

Market Story

Now, in a farmer’s market, you’ll have multiple things:

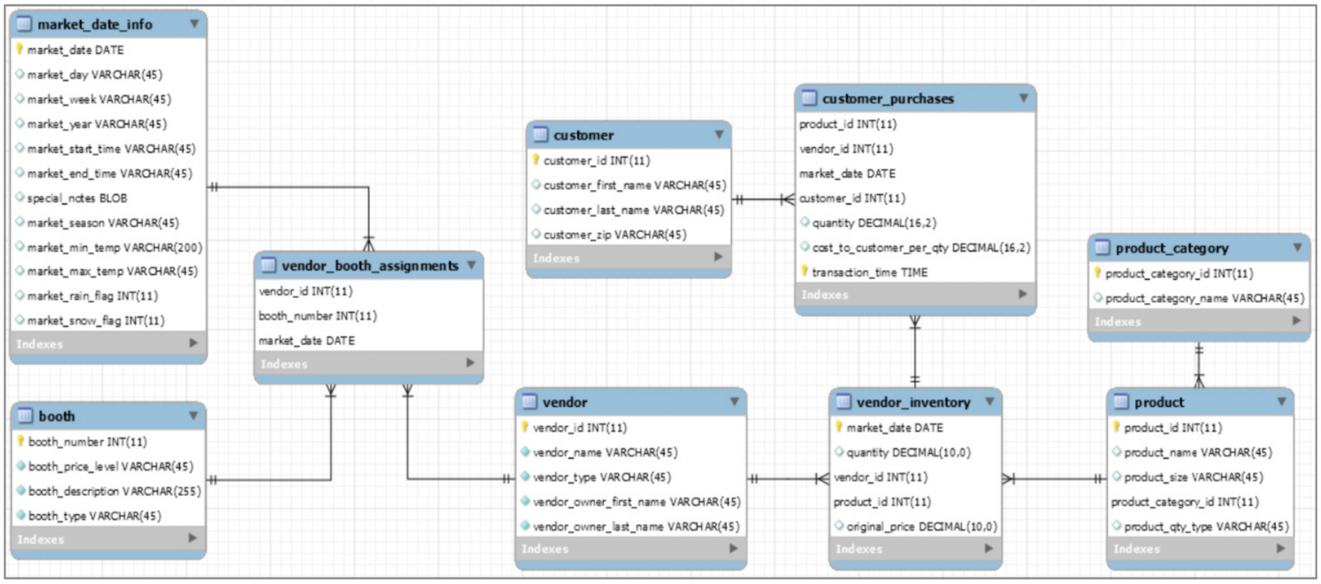
1. Customers
2. Vendors
3. Booths/Stalls
4. Products <-> Product Categories
5. Market information - date, day, week, start time, end time, rain, etc.

So, the story here will become:

- The market opens up.
- Vendors get their slots assigned to them.
- Vendors will also maintain their inventory
- In the inventory, vendors will have multiple products to sell each of which will belong to a category, stored in the product category

Schema: Farmer's Market Database

Farmer's market



We can answer all sorts of business-related questions like

- "How many people visit the market weekly throughout the year, and when do our sales peak?"
- "How much does bad weather impact attendance?"
- "When is each type of fresh fruit or vegetable in season, locally?" and
- "Is this proposed new vendor likely to take business away from existing vendors, or is there enough demand to support another vendor selling these goods."

Some basic terminologies and basic syntax to start writing queries

- **SELECT:** A SELECT statement is the SQL code that retrieves data from the database.
    - When used in combination with other SQL keywords, SELECT can be used to query data
        - from a set of columns in a database table,
        - combine data from multiple tables,
        - filter the results,
        - perform calculations, and more.

***Note for Instructor**: Capitalizing main clauses visually differentiates the keyword from other text in your query, such as field names.*

The Fundamental Syntax Structure of a SELECT Query

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]
```

Let's start with a question and we'll understand it better.

Question: Get all the products available in the market.

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 everything from the product table in the **farmers_market schema**.”
- The asterisk represents “all columns,” so technically it’s “Select all columns from the product table in the farmers_market schema,”

Let’s say you don’t want to select all the rows.

LIMIT

- There is another **optional clause**: the 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 an extensive 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.

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. For example,

```
SELECT *
 FROM farmers_market.product
 LIMIT 5
```

Note:

- **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.
- 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, **to select five product IDs and their associated product names from the product table.**

```
SELECT product_id, product_name
FROM farmers_market.product
LIMIT 5
```

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 an input into the next step of a series of code functions without human review, for example).

- 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.

Question: Explore vendor_booth_assignments. List down 10 rows of farmer's market vendor booth assignments, displaying the market date, vendor ID, and booth number from the vendor_booth_assignments table.

```
SELECT market_date, vendor_id, booth_number
 FROM farmers_market.vendor_booth_assignments
 LIMIT 5
```

But this output would make a lot more sense if it were sorted by the market date, right?

Let’s see how to set the sort order of query results.

This is where Order by clause comes into play

The ORDER BY Clause: Sorting Results

- 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 in ascending (ASC) or descending (DESC) order.
- **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 the ascending order.

NOTE: The **sort order is ascending by default**, so if you want your values sorted in ascending order, the **ASC keyword is optional.**

Let’s try to look at the product table and get all the products in the alphabetical order:

```
SELECT product_id, product_name
 FROM farmers_market.product
 ORDER BY product_name
 LIMIT 5
```

Sorting by product_id, from highest to lowest:

```
 SELECT product_id, product_name
 FROM farmers_market.product
 ORDER BY product_id DESC
 LIMIT 5
```

How to sort the products by Date and Vendor

We have the *market_date* column in the product table,

```
SELECT market_date, vendor_id, booth_number
  FROM farmers_market.vendor_booth_assignments
  ORDER BY market_date, vendor_id
  LIMIT 5
```

- We only see rows with the **earliest market date** available in the database, since we sorted by market_date first,
- there are more than 5 records in the table for that same date,
- and we limited this query to only return the first 5 rows.
- After sorting by market date, the records are then sorted by **vendor ID** in ascending order.

You can specify different orders for each variable. For example,

```
SELECT market_date, vendor_id, booth_number
  FROM farmers_market.vendor_booth_assignments
  ORDER BY market_date DESC, vendor_id;
```

How to do calculations on the go? - Inline Calculations

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 price the customer has paid.

What *customer_purchases* looks like:

```
SELECT
      market_date,
      customer_id,
      vendor_id,
      quantity,
      cost_to_customer_per_qty
  FROM farmers_market.customer_purchases
  LIMIT 10
```

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

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity,
    cost_to_customer_per_qty,
    quantity * cost_to_customer_per_qty
 FROM farmers_market.customer_purchases
 LIMIT 10
```

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.

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**. I prefer not to use spaces in my aliases, to avoid having to remember to surround them with quotes if I need to reference them later.

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
 LIMIT 10
```

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

Can we round off the price column to just 2 decimal places. - ROUND function

> 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 **functions inline** 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.

ROUND()

- In the last, the “price” field was displayed with four digits after the decimal point.
- Let’s say we wanted to display the number rounded to the nearest penny (in US dollars), which is two digits after the decimal. That can be accomplished using the ROUND() function.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price
 FROM farmers_market.customer_purchases
 LIMIT 10
```

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:
- **CEIL**
- **FLOOR**

- **LEAST/GREATEST - select greatest(4,5,6,7,1,2,3);**

Transition -> Up until now, we have manipulated just numbers, what about strings?

Question: Extract all the product names that are part of product category 1

Filtering data - The WHERE Clause

- The WHERE clause is the part of the SELECT statement in which you list conditions that are used to determine which rows in the table should be included in the results set.
- In other words, the WHERE clause is used for filtering.

The WHERE clause goes after the FROM statement and before any GROUP BY, ORDER BY, or LIMIT statements in the SELECT query

Let's get those product names that are in category 1:

```
SELECT

product_id, product_name, product_category_id

FROM farmers_market.product

WHERE product_category_id = 1

LIMIT 5
```

Question: Print a report of everything ***customer_id 4*** has ever purchased at the farmer's market, sorted by market date, vendor ID, and product ID.

```
SELECT
```

```
        market_date, customer_id,

        vendor_id, product_id,

        quantity,

        quantity * cost_to_customer_per_qty AS price

  FROM farmers_market.customer_purchases

  WHERE customer_id = 4

  ORDER BY market_date, vendor_id, product_id

  LIMIT 5
```

Behind the scenes - How WHERE clause works?

- Each of the conditional statements (like “customer_id = 4”) listed in the WHERE clause will evaluate to TRUE or FALSE for each row, and only the rows for which the combination of conditions evaluates to TRUE will be returned.

Filtering on multiple conditions - using operators - “AND”, “OR”, “NOT”

- Conditions with OR between them will jointly evaluate to TRUE, meaning the row will be returned, if any of the clauses are TRUE.
- Conditions with AND between them will only evaluate to TRUE in combination if all of the clauses evaluate to TRUE. Otherwise, the row will not be returned.
- **Remember that NOT flips the following boolean value to its opposite (TRUE becomes FALSE, and vice versa).**

Question: Find the details of purchases made by customer 4 at vendor 7, we could use the following query:

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty AS price
```

```
 FROM farmers_market.customer_purchases
 WHERE
    customer_id = 4
    AND vendor_id = 7
```

Filtering based on strings

Question: Find the customer detail with the first name of “Carlos” or the last name of “Diaz,”:

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
 FROM farmers_market.customer
 WHERE
    customer_first_name = 'Carlos'
    OR customer_last_name = 'Diaz'
```

LIKE

Suppose you’re searching for a person at the *customer* table but don’t know the spelling of their name.

For example, if someone asked you to look up a customer ID for someone's name (Manuel), but you don’t know how to spell the name exactly, you might try searching against a list with multiple spellings, like this:

Question: You want to get data about a customer you knew as “Jerry,” but you aren’t sure if he was listed in the database as “Jerry” or “Jeremy” or “Jeremiah.”

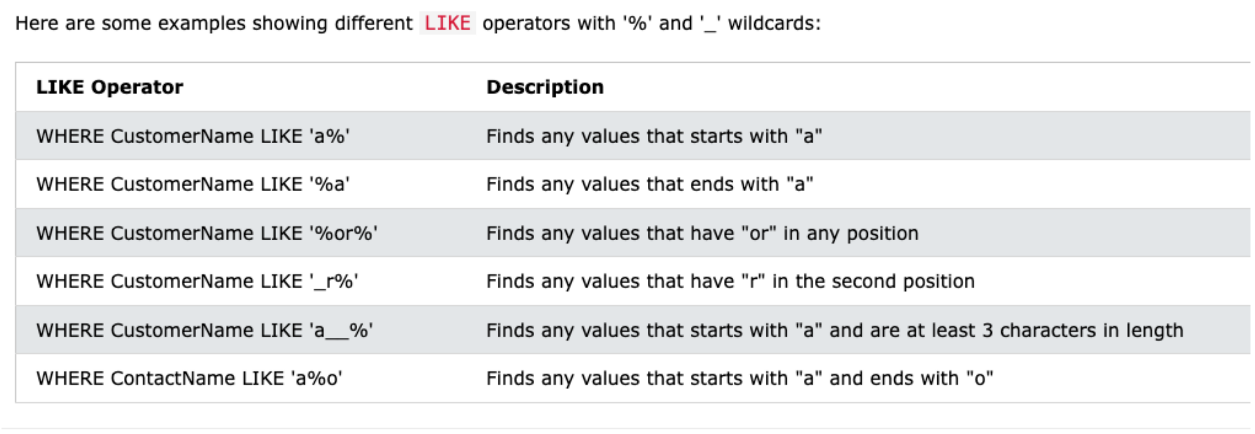
All you knew for sure was that the first three letters were “Jer.”

In SQL, instead of listing every variation you can think of, you can search for **partially matched strings** using a **comparison operator called LIKE**, and **wildcard characters**, which serve as a placeholder for unknown characters in a string.

% wildcard

The wildcard character % (percent sign) can serve as a stand-in for any number of characters (including none).

So the comparison LIKE ‘Jer%’ will search for strings that start with “Jer” and have any (or no) additional characters after the “r”:



Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a__%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

IS NULL - missing values

You have learnt how to find and treat missing values in Python but what happens when your **database contains missing values.**

- In the product table, the product_size field is optional, so it’s possible to add a record for a product with no size.

Question: Find all of the products from the product table without sizes.

```
SELECT *
	FROM farmers_market.product
	WHERE product_size IS NULL
```

CASE Statements

Note: If you’re familiar with scripting languages like Python that use “if ” statements, you’ll find that SQL handles conditional logic somewhat similarly, with different syntax.

Conditional flow: “If [one condition] is true, then [take this action].
Otherwise, [take this other action].”

For example: “If the weather forecast predicts it will rain today, then I’ll take an umbrella with me. Otherwise, I’ll leave the umbrella at home.”

In SQL, the code to describe this type of logic is called a CASE statement, which uses the following syntax:

```
CASE
      WHEN [first conditional statement]
       THEN [value or calculation]
      WHEN [second conditional statement]
       THEN [value or calculation]
      ELSE [value or calculation]
END
```

This statement indicates that you want a column to contain different values under different conditions.

GROUP BY

Question: Get a list of customer IDs of customers who made purchases on each market date.

If we write this…

```
SELECT
    market_date,
    customer_id
 FROM farmers_market.customer_purchases
 ORDER BY market_date, customer_id
```

SQL starts becoming powerful for analysis when you use it to aggregate data.

Using the GROUP BY statement, you can specify the level of summarization and then use aggregate functions to summarize values for the records in each group.

- Data analysts can use SQL to build dynamic summary reports that can be automatically updated.
- Dashboards and reports built using software like Tableau and Cognos often rely on SQL queries to get the data they need from the underlying database in an aggregated form that can be used for reporting.

But it all starts with basic SQL aggregation.

- One row per item each customer purchased, **displaying duplicates in the output**, because you're querying the customer_purchases table with no grouping specified.

```
SELECT
    market_date,
    customer_id
 FROM farmers_market.customer_purchases
 GROUP BY market_date, customer_id
 ORDER BY market_date, customer_id
```

Aggregate functions - SUM() and COUNT()

Question: Count the number of purchases each customer made per market date.

```
SELECT
```

```
    market_date,
    customer_id,
    COUNT(*) AS num_purchases
 FROM farmers_market.customer_purchases
 GROUP BY market_date, customer_id
```

Alternate Question: Calculate the total quantity purchased by each customer per market_date.

Here, we'll use the SUM() function

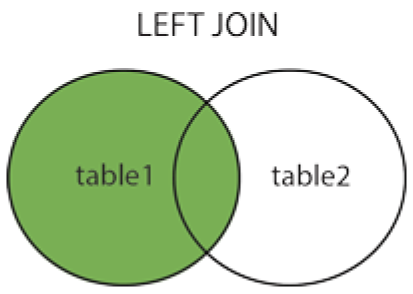
```
SELECT
    market_date,
    customer_id,
    SUM(quantity) AS total_qty_purchased
 FROM farmers_market.customer_purchases
 GROUP BY market_date, customer_id
```

JOINs

Question: Let's say we wanted to list each product name along with its product category name.

- Since only the ID of the product category exists in the ***product*** table, and the product category's name is in the ***product_category*** table,
- we **have to combine the data in the product and product_category tables together in order to generate this list.**

LEFT JOIN



The most frequently used JOIN when building analytical datasets is the **LEFT JOIN.**

- This tells the query to pull all records from the table on the “left side” of the JOIN, and only the matching records (based on the criteria specified in the JOIN clause) from the table on the “right side” of the JOIN.

- Note that the product table will be on the left side of the join we’re setting up, even though it was on the right side of the relationship diagram!

Syntax:

```
SELECT [columns to return]
FROM [left table]
[JOIN TYPE] [right table]
ON [left table].[field in left table to match] = [right table].[field in right table to match]
```

Actual query

```
SELECT * FROM
        product
LEFT JOIN product_category
        ON product.product_category_id = product_category.product_category_id
```

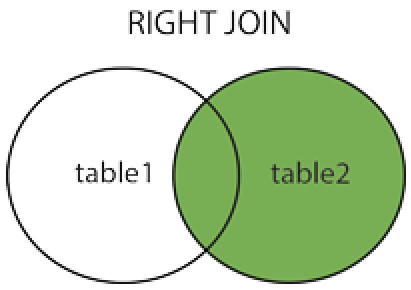
NOTE: You may have noticed two columns called **product_category_id** in the output.

That is because we selected all fields using the **asterisk(*)**, and **there are fields in both tables with the same name**.

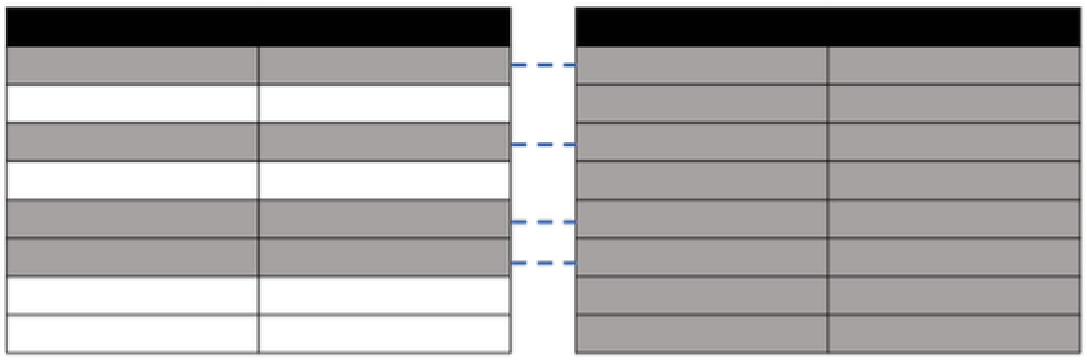
We can use **table aliasing to specify which table a particular column comes out of, in the output.**

```
SELECT
        p.product_id,
        p.product_name,
        pc.product_category_id,
        pc.product_category_name
      FROM product AS p
        LEFT JOIN product_category AS pc
           ON p.product_category_id = pc.product_category_id
```

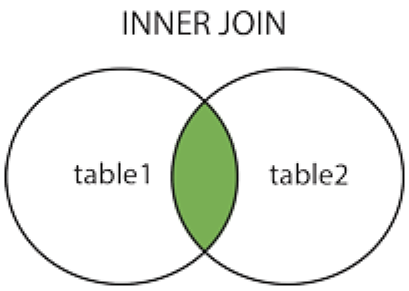
RIGHT JOIN



In a RIGHT JOIN, all of the rows from the "right table" are returned, along with only the matching rows from the "left table," using the fields specified in the ON part of the query.



INNER JOIN



- An INNER JOIN only returns records that have matches in both tables.
- Can you tell which rows from each table will not be returned if we INNER JOIN the **product** and **product_category** tables on *product_category_id*?

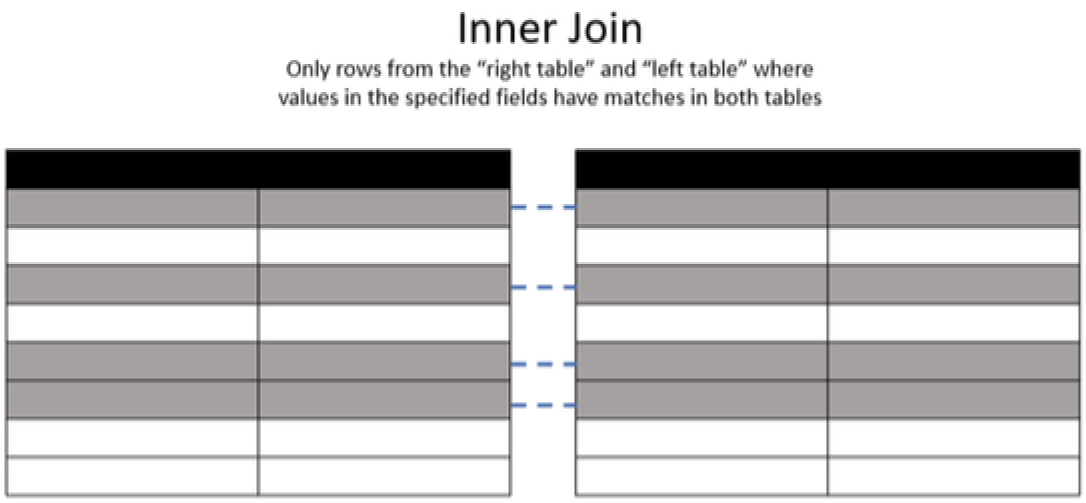

Inner Join
Only rows from the "right table" and "left table" where values in the specified fields have matches in both tables

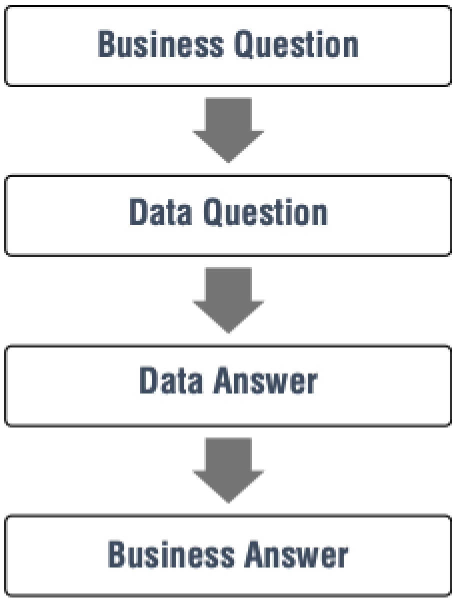
Figure 5.10

Job of a Data Analyst / Scientist

- As an Analyst, you should be able to anticipate potential follow-up questions is a skill that analysts develop over time, through experience.
- For example, if the manager of the farmer's market asked me "What were the total sales at the market last week?"
- I would expect to be asked more questions after delivering the answer to that one, such as **"How many sales were at the Wednesday market versus the Saturday market last week?"** or **"Can you calculate the total sales over another period?"** or **"Let's track these weekly market sales over time,"** or **"Now that we have the total sales for last week, can we break this down by vendor?"**
-

Your Task to build a single dataset

- Given time, you could build a single dataset that could be imported into a reporting system like Tableau and used to answer all of these questions.



In the Farmer's Market database, the sales are tracked in the **customer_purchases** table, **which has one row per item purchased** (with multiples of the same item potentially included in a single row, since there is a quantity column).

Each row in **customer_purchases** contains the

- **ID** of the product purchased,
- the ID of the vendor the product was purchased from,
- the ID of the customer making the purchase,
- the transaction date and time, and
- the quantity and cost per quantity.

Goal: **We want to design the dataset to have one row per date and vendor, we do not need to include detailed information about the customers or products.**

And because our most granular time dimension is ***market_date***, we do not need to consider the field that tracks the time of purchase.

Step 1: Filter out the data we need

Write a query that pulls only the fields I need, leaving out unnecessary information and allowing us to summarize at the selected level of detail.

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

- After reviewing the output without aggregation, we'll **group** and **sort** by ***vendor_id*** and ***market_date***,
- **SUM** the calculated cost column, **round** it to two decimal places, and give it an **alias of sales.**

```
SELECT
    market_date,
    vendor_id,
    ROUND(SUM(quantity * cost_to_customer_per_qty),2) AS sales
 FROM farmers_market.customer_purchases
 GROUP BY market_date, vendor_id
 ORDER BY market_date, vendor_id
```

From the output, will you be able to answer the **anticipated questions**:

- **What were the total sales at the market last week?**
- **How many of last week's sales were made on Wednesdays versus on Saturdays?**
- **Can we calculate the total sales over another time period?**
- **Can we track the weekly market sales over time?**
- **Can we break down the weekly sales by a vendor?**

Step 2: Add some more information from other tables to answer the questions

- Some additional information that might be valuable to have on hand for reporting from other tables, including ***market_day***, ***market_week***, and ***market_year*** from the **market_date_info** table
- And ***vendor_name*** and ***vendor_type*** from the **vendor** table.

Let's **LEFT JOIN** those into our dataset, so we keep all the existing rows and add more columns where available.

```
SELECT
    cp.market_date,
    md.market_day,
    md.market_week,
    md.market_year,
    cp.vendor_id,
    v.vendor_name,
    v.vendor_type,
    ROUND(SUM(cp.quantity * cp.cost_to_customer_per_qty),2) AS sales
FROM farmers_market.customer_purchases AS cp
    LEFT JOIN farmers_market.market_date_info AS md
        ON cp.market_date = md.market_date
    LEFT JOIN farmers_market.vendor AS v
        ON cp.vendor_id = v.vendor_id
GROUP BY cp.market_date, cp.vendor_id
ORDER BY cp.market_date, cp.vendor_id
```

Now we can use this custom dataset to create reports and conduct further analysis.

Using custom analytics datasets in SQL using CTEs

There are multiple ways to store queries (and the results of queries) for reuse in reports and other analyses.
Here, we will cover two approaches for more easily querying from the results of custom dataset queries you build:

1. **Common Table Expressions**
2. **views.**

Most **database systems**, including MySQL since version 8.0, support Common Table Expressions (CTEs), also known as **"WITH clauses."**

CTEs allow you to create an alias for an entire query, allowing you to reference it in other queries like any database table.

The syntax for CTEs is:

```
WITH [query_alias] AS (
[query]
),
[query_2_alias] AS (
[query_2]
)
SELECT [column list]
FROM [query_alias]
```

... [remainder of query that references aliases created above]

-> where "[**query_alias**]" is a placeholder for the name you want to use to refer to a query later, and "[**query**]" is a placeholder for the query you want to reuse.

Let's use it for our use case.

Question: if we wanted to reuse the previous query we wrote to generate the dataset of sales summarized by date and vendor for a report that summarizes sales by market week, we could put that query inside a WITH clause.

```
WITH sales_by_day_vendor AS
       (
          SELECT
             cp.market_date,
             md.market_day,
             md.market_week,
             md.market_year,
             cp.vendor_id,
             v.vendor_name,
             v.vendor_type,
             ROUND(SUM(cp.quantity * cp.cost_to_customer_per_qty),2) AS sales
          FROM farmers_market.customer_purchases AS cp
             LEFT JOIN farmers_market.market_date_info AS md
                ON cp.market_date = md.market_date
             LEFT JOIN farmers_market.vendor AS v
                ON cp.vendor_id = v.vendor_id
          GROUP BY cp.market_date, cp.vendor_id
          ORDER BY cp.market_date, cp.vendor_id
)
       SELECT
          s.market_year,
          s.market_week,
          SUM(s.sales) AS weekly_sales
       FROM sales_by_day_vendor AS s
       GROUP BY s.market_year, s.market_week
```

Break down:

- Notice how the SELECT statement at the bottom references the **sales_by_ day_vendor Common Table Expression using its alias**, treating it just like a table, and even giving it an even shorter alias, s.
- You can filter it, perform calculations, and do anything with its fields that you would do with a normal database table.

Note:

- If you want to alias multiple queries in the WITH clause, you put each query inside its own set of parentheses, separated by commas.

- You only use the WITH keyword once at the top, and enter “[alias_name] AS” before each new query you want to reference later. (The AS is not optional in this case.)
- Each query in the WITH clause can reference any query that preceded it, by using its alias.