Joins

---

1. Problem Statement
2. Intro to Joins
3. Types of Joins
    a. INNER JOIN
    b. LEFT JOIN
    c. RIGHT JOIN
    d. FULL OUTER JOIN
        i. Union
            1. UNION DISTINCT
            2. UNION ALL
4. Summary

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

Company Name: CineFlix Entertainment Store

CineFlix Entertainment Store is a premier rental service provider specialising in CDs, DVDs, and Blu-rays. With a wide range of movies, TV shows, and exclusive collections, CineFlix has been a household name for entertainment enthusiasts. Despite the growing popularity of streaming services, CineFlix continues to attract a loyal customer base due to its diverse inventory, personalized customer service, and flexible rental plans.

Challenges: CineFlix is facing increased competition from digital streaming platforms, which offer convenience and instant access to content.

To remain relevant and competitive, CineFlix needs to:

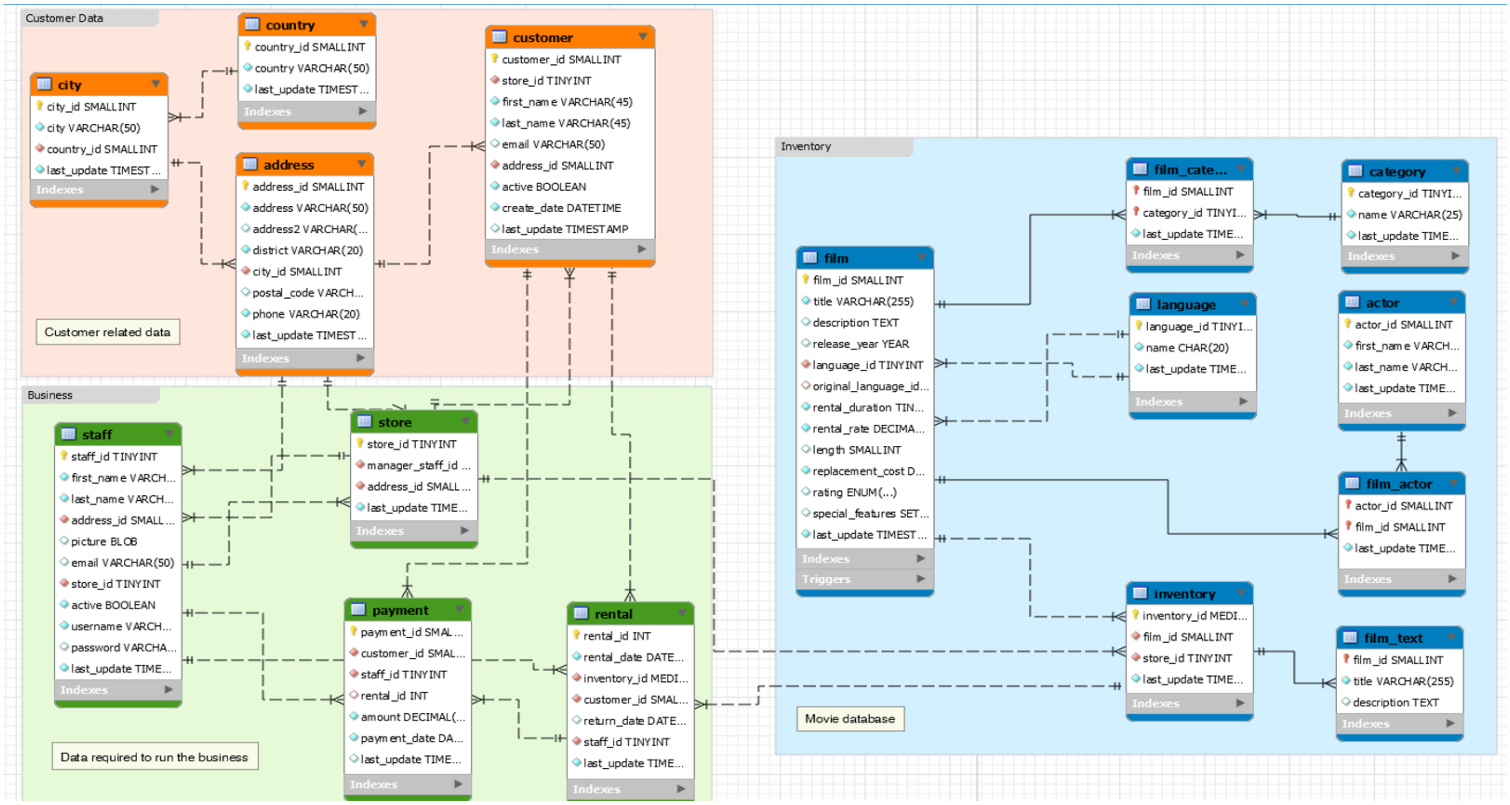
1. **Understand Rental Patterns**: Analyze customer rental behaviours to identify trends, peak rental periods, and preferences for specific genres or formats.
2. **Customer Behavior Analysis**: Gain insights into customer demographics, retention rates, and preferences to tailor marketing strategies and improve customer satisfaction.
3. **Store Performance**: Evaluate the performance of different store locations to identify areas for improvement and optimise operations.
4. **Inventory Management**: Optimize inventory levels to ensure popular titles are available while minimizing costs associated with overstock and outdated inventory.

Problem Statement

CineFlix Entertainment Store is a leading CD, DVD and Blu-ray rental company facing increasing competition from streaming services. To stay competitive, the company aims to improve its services by understanding rental patterns, customer behaviour, and store performance.
It also wants to optimize its inventory management to ensure they have the right films in stock to meet customer demand while minimizing costs associated with excess inventory. They have tasked their data analytics team to derive insights from their database.

Dataset link: LINK

Dataset Schema:



Formulating questions to be explored based on the data provided:

- **Rental Patterns:**
  - Get details of the top 5 most rented films.
  - Find the customers & their rental count for each customer where both rentals were rented by the same customer and later returned.

- **Customer Behavior:**
  - Get a list of all customers who have rented more films than the average number of rentals.
  - List all customers and the total amount they have spent on rentals, including customers who have never rented a film.

    - For each customer, identify the names of other customers who have rented the same inventory item(s) at least once.

- **Film Inventory management:**
    - List all film categories along with the number of films in each category, including categories with no films.
    - List all films with rental rates higher than the average rental rate.

- **Store Performance:**
    - List all stores and the number of customers, including stores with no customers & customers with missing store information.
    - For each store, list the total number of films and the number of currently rented films.

These questions will help CineFlix understand which films are popular and which customers rent frequently & delve into customer behaviour, analyzing spending patterns and identifying both active and potentially inactive customers. It will also address store performance, providing insights into customer distribution across stores.

So far...

- You've developed the ability to retrieve data from **a single database table** and refine it by filtering for the desired rows.
- But you might wonder what to do if the data you need is spread across multiple related tables in the database.

Question: Get details of the top 5 most rented films.

Intuition:

Now, what tables do we require?

1. to get the film details like film name & film ID - the ***film*** table.
2. To get the details about film rental, see the ***rental*** table.
3. To get the details about which film was rented most - the ***inventory*** table

So you need all this combined information from 3 tables here.

How do we get the combined information from the 3 tables?

This is where **SQL JOINs** come in.

Q. Now that we have the combined information, how do we solve this problem?

Answer: Now, we just need to count how many times the movies have been rented, and then organise them to see which are among the top 5 most in-demand.

Detailed step-wise intuition :

- We're choosing specific information we want to see - film ID, film title, and rental count.
- We're looking at data from the 'rental' table because that's where rental transactions are recorded.
- We're connecting data from multiple tables: 'rental', 'inventory', and 'film'. We do this to link rentals with the films they're for.

- We group the results based on the film ID and title. This ensures that we count rentals for each film separately.
- We then sort the films based on their rental counts, with the most popular ones appearing first.
- Finally, we restrict the output to only show the top 5 films with the highest rental counts.

Solution Query:

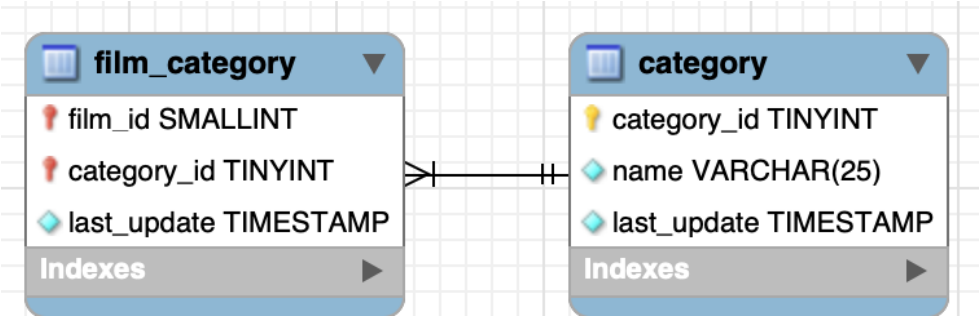
```
SELECT
    f.film_id,
    f.title AS film_title,
    COUNT(r.rental_id) AS rental_count
FROM rental r
INNER JOIN inventory i
  ON r.inventory_id = i.inventory_id
INNER JOIN film f
  ON i.film_id = f.film_id
GROUP BY f.film_id, f.title
ORDER BY rental_count DESC
LIMIT 5;
```

Note that ER Diagrams are crucial in identifying which tables we can join and which key fields connect them.

Question: List all the films along with their film category name.

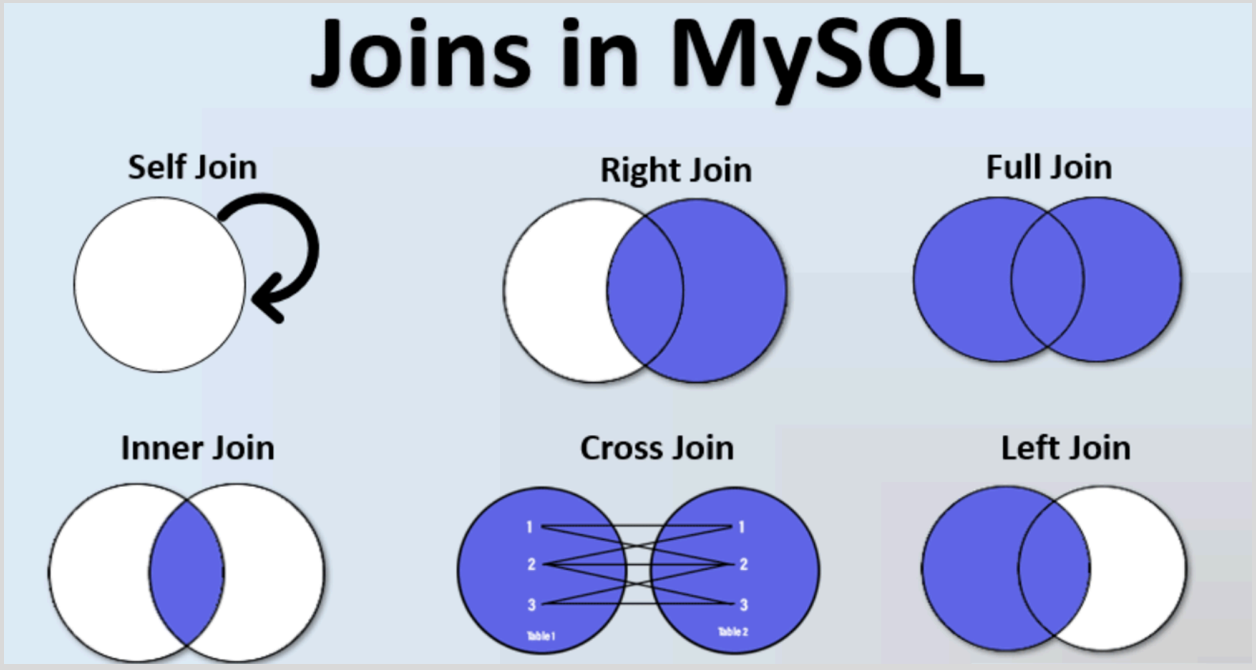
Since only the ID of the category exists in the ***film_category*** table, and the film category's name is in the ***category*** table,

We have to combine the data in the ***category*** and ***film_category*** tables together to generate this list.



- The figure shows the many-to-one relationship between these tables.
- Their primary keys are each identified with a golden key and the foreign key with a red key.
- Each row in the ***category*** table can be associated with many rows in the ***film_category*** table.
- The fields that connect the two tables are *film_category.category_id* and *category.category_id*.

Now, there are multiple ways of joining these tables.



Imagine there are two tables:

Stores Table

store_id	store_name
1	Store A
2	Store B
2	Store B2
3	Store C
4	Store D

Customers Table

customer_id	customer_name	store_id	active
1	Alice	1	1
2	Bob	2	0
3	Charlie	NULL	1
4	Dave	3	1
5	Eve	3	1
6	Frank	2	1

stores table:

The stores table contains information about various store locations. It has the following columns:

- store_id: A unique identifier for each store. This is the primary key of the table.
- store_name: The name of the store.

customers table*:*

The customers table contains information about customers and the stores they are associated with. It has the following columns:

- customer_id: A unique identifier for each customer. This is the primary key of the table.
- customer_name: The name of the customer.
- store_id: A foreign key that references the store_id in the stores table, indicating which store the customer is associated with. This can be NULL if the customer is not associated with any store.
- active: A flag indicating whether the customer is active (1 for active, 0 for inactive).

Now you want to retrieve active customers who belong to any of the stores.

- **INNER JOIN** returns only the rows with a match in both tables. It will return customers who have a valid ***store_id*** that matches an entry in the (stores) table.
- If there are duplicate values in one or both tables, the result will include all possible combinations of these duplicates.

Result

customer_id	customer_name	store_id	active	store_id	store_name
1	Alice	1	1	1	Store A
2	Bob	2	0	2	Store B
2	Bob	2	0	2	Store B2
4	Dave	3	1	3	Store C
5	Eve	3	1	3	Store C
6	Frank	2	1	2	Store B
6	Frank	2	1	2	Store B2

- **LEFT JOIN** returns all rows from the left table (stores), and the matched rows from the right table (customers). If no match is found, NULL values are returned for columns from the right table. This will include all stores and their customers, even if a store has no customers.
- If duplicate values are in the left table, it will produce combinations for each match found in the right table.

Result

customer_id	customer_name	store_id	active	store_id	store_name
1	Alice	1	1	1	Store A
2	Bob	2	0	2	Store B
6	Frank	2	1	2	Store B
2	Bob	2	0	2	Store B2
6	Frank	2	1	2	Store B2
4	Dave	3	1	3	Store C
5	Eve	3	1	3	Store C
NULL	NULL	NULL	NULL	4	Store D

- **RIGHT JOIN** returns all rows from the right table (customers), and the matched rows from the left table (stores). If no match is found, NULL values are returned for columns from the left table. This will include all

customers and their respective stores, even if a customer does not have a store.
- If duplicate values are in the right table, it will produce combinations for each match found in the left table.

Result

customer_id	customer_name	store_id	active	store_id	store_name
1	Alice	1	1	1	Store A
2	Bob	2	0	2	Store B
2	Bob	2	0	2	Store B2
3	Charlie	NULL	1	NULL	NULL
4	Dave	3	1	3	Store C
5	Eve	3	1	3	Store C
6	Frank	2	1	2	Store B
6	Frank	2	1	2	Store B2

- **FULL OUTER JOIN** returns all rows when there is a match in one of the tables. This means it returns all rows from the left table (stores) and the right table (customers), with NULLs in places where there is no match. This will include all stores and all customers, showing NULLs where there is no corresponding match.
- If duplicate values are in both tables, it will result in combinations for each match found.

Result

customer_id	customer_name	store_id	active	store_id	store_name
1	Alice	1	1	1	Store A
2	Bob	2	0	2	Store B
2	Bob	2	0	2	Store B2
3	Charlie	NULL	1	NULL	NULL
4	Dave	3	1	3	Store C
5	Eve	3	1	3	Store C
6	Frank	2	1	2	Store B
6	Frank	2	1	2	Store B2
NULL	NULL	NULL	NULL	4	Store D

Syntax for joining tables:

```
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]
```

Order of Execution of an SQL query:

- **FROM** - The database gets the data from tables in the FROM clause and if necessary, performs the JOINs.
- **JOIN** - Depending on the type of JOIN used in the query and the conditions specified for joining the tables in the **ON** clause, the database engine matches rows from the virtual table created in the FROM clause.
- **WHERE** - After the JOIN operation, the data is filtered based on the conditions specified in the WHERE clause. Rows that do not meet the criteria are excluded.

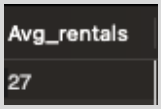
- **GROUP BY** - If the query includes a GROUP BY clause, the rows are grouped based on the specified columns and aggregate functions are applied to the groups created.
- **HAVING** - The HAVING clause filters the groups of rows based on the specified conditions
- **SELECT** - After grouping and filtering are done, the SELECT statement 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.

Question: Get a list of all customers who have rented more films than the average number of rentals.

Query to get the average number of rentals:

```
SELECT ROUND(AVG(num_rentals)) Avg_rentals
FROM (
    SELECT COUNT(*) AS num_rentals
    FROM rental
    GROUP BY customer_id
  ) AS rental_counts;
```

Output:

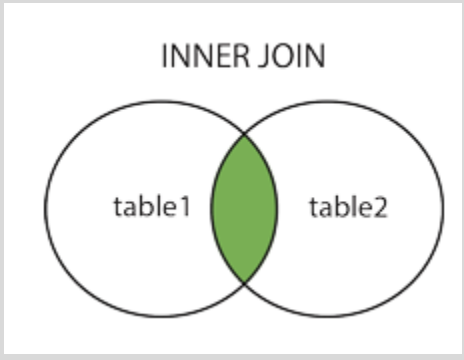


Suppose we need customer details of all the customers who have rented more than the average number. In that case, we only require an **intersection of customers** whose details are present in the customer & rental tables.

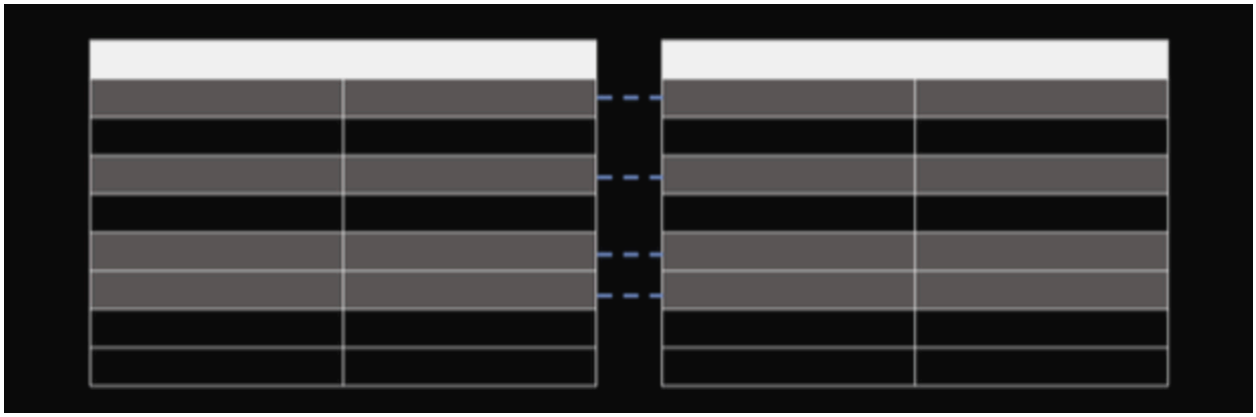
Will you need to join here?

What type of join are we going to use here?

Inner JOIN



An INNER JOIN only returns records that have matches in both tables.



Query:

```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    COUNT(r.rental_id) AS rental_count
FROM customer c
INNER JOIN rental r
```

```
  ON c.customer_id = r.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING rental_count > 27
ORDER BY rental_count DESC;
```

Breaking down **Inner JOIN** :

- An inner join is a type of join in SQL that returns only the rows from both tables that have matching values in the specified columns.
- In this case, the inner join is performed on the "customer" and "rental" tables using the "customer_id" column as the matching column.
- The "INNER JOIN" clause specifies that we want to retrieve only the rows that have matching values in both tables. In other words, we only want to retrieve the details of customers who have rented a film.
- The "ON" clause specifies the condition for the join. In this case, we want to match the "customer_id" column in both tables. By joining this column, we can link each rental to the corresponding customer.
- Once the join is performed, we can retrieve the customer's first_name & last_name as customer_name & customer_id of the customers who have rented a film from the "customer" table.
- We will also count the number of rentals for each customer using the rental_is column from the “rental” table.
- At last, we will filter the records of the customers who have rented more than 27 films.

USING keyword

The USING keyword simplifies the join syntax when you want to join tables based on a common column name.

It specifies which column should be used in the join condition.

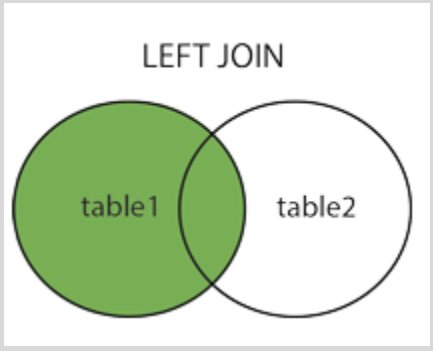
The above query can also be written in the following manner -

```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    COUNT(r.rental_id) AS rental_count
FROM customer c
INNER JOIN rental r
      USING (customer_id)
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING rental_count > 27
ORDER BY rental_count DESC;
```

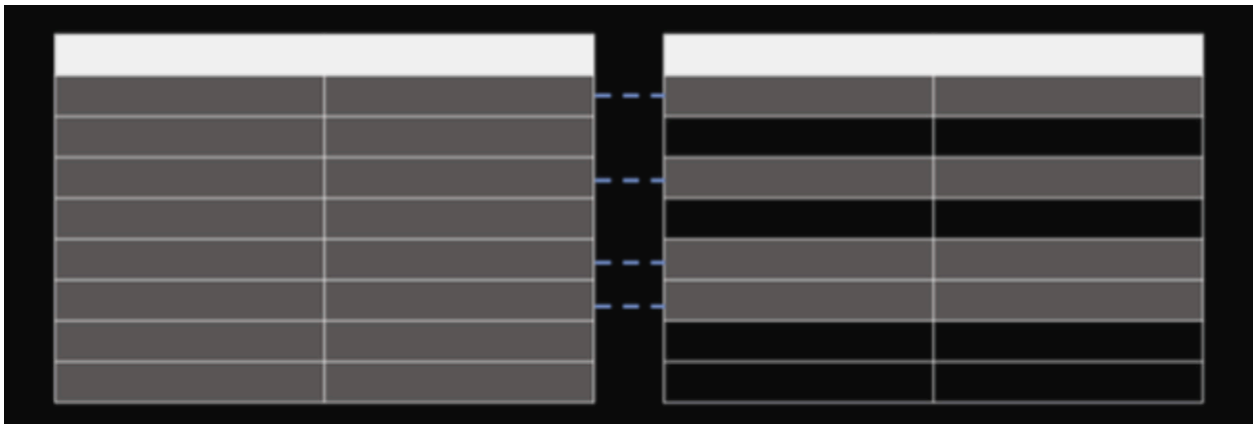
Please note that the USING keyword can only be used when joining columns with the same name in both tables, and it's typically used for inner joins and natural joins.

If the columns have different names or if you need to perform more complex joins, you should stick to the ON keyword to specify the join condition explicitly.

Left JOIN



This tells the DBMS 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.



Question: List all customers and the total amount they have spent on rentals, including customers who have never rented a film.

Ques. Which table should we use as the left table if we use LEFT JOIN?
Ans: The customer table should be on the left and payment should be on the right.

Query:

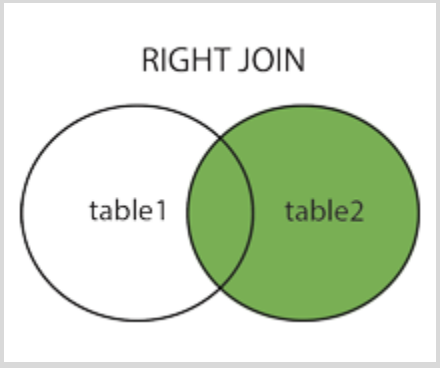
```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
```

```
    IFNULL(SUM(p.amount), 0) AS total_spent
FROM customer c
LEFT JOIN payment p
  ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spent, customer_name;
```

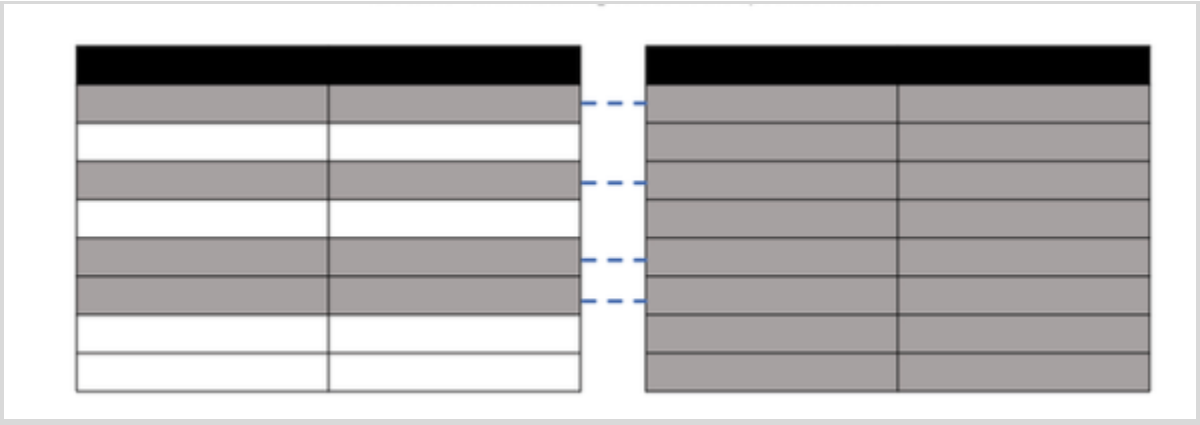
Breaking down **Left JOIN** :

- The Left JOIN indicates that we want all rows from the **customer** table (which is listed on the left side of the JOIN keyword) and
- Only the associated rows from the **payment** table. So, if a payment is not associated with any of the customers, it will not be included in the results.
- If a customer is without a rental payment, it would be included in the results, with the fields on the **payment** side being NULL.
- The ON part of the JOIN clause tells the query to match up the rows in the two tables using each table's values in the customer_id field.
- We can specify which table each column is from since it's possible to have identically named columns in different tables.

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.



To write the same query (the one we saw earlier) using a RIGHT JOIN, you can simply reverse the order of the tables and use a RIGHT JOIN instead of a LEFT JOIN.

Query:

```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    IFNULL(SUM(p.amount), 0) AS total_spent
FROM payment p
RIGHT JOIN customer c
  ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
```

```
ORDER BY total_spent, customer_name;
```

- In this query, the "payment" table is on the left side of the RIGHT JOIN, and the "customer" table is on the right side.
- The rest of the query remains the same as in your original LEFT JOIN query.
- This RIGHT JOIN query will return all customer details, including those without any rental payments, and it will also display customers when they have a matching category.

Question: List all film categories along with the number of films in each category, including categories with no films.

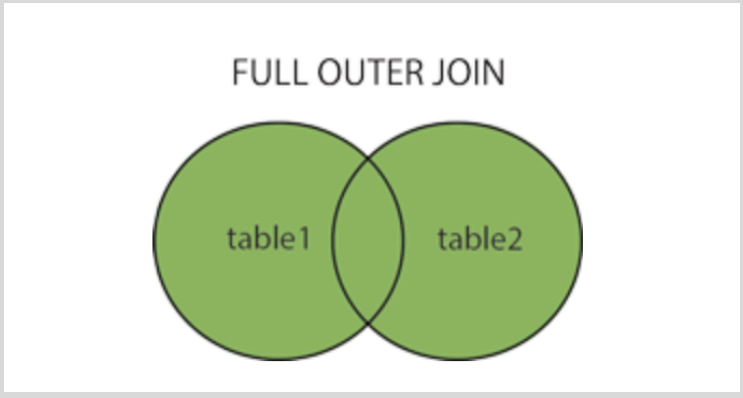
```
SELECT
    c.name AS category_name,
    COUNT(fc.film_id) AS film_count
FROM film_category fc
RIGHT JOIN category c
  ON fc.category_id = c.category_id
GROUP BY c.category_id, c.name
ORDER BY film_count DESC, c.name;
```

Breaking down **Right JOIN** :

- The query starts by specifying the tables involved. The ***film_category*** table is aliased as fc and the ***category*** table is aliased as c.
- A **RIGHT JOIN** is used here, which ensures all categories from the category table are included, even if there are no corresponding entries in the ***film_category*** table.

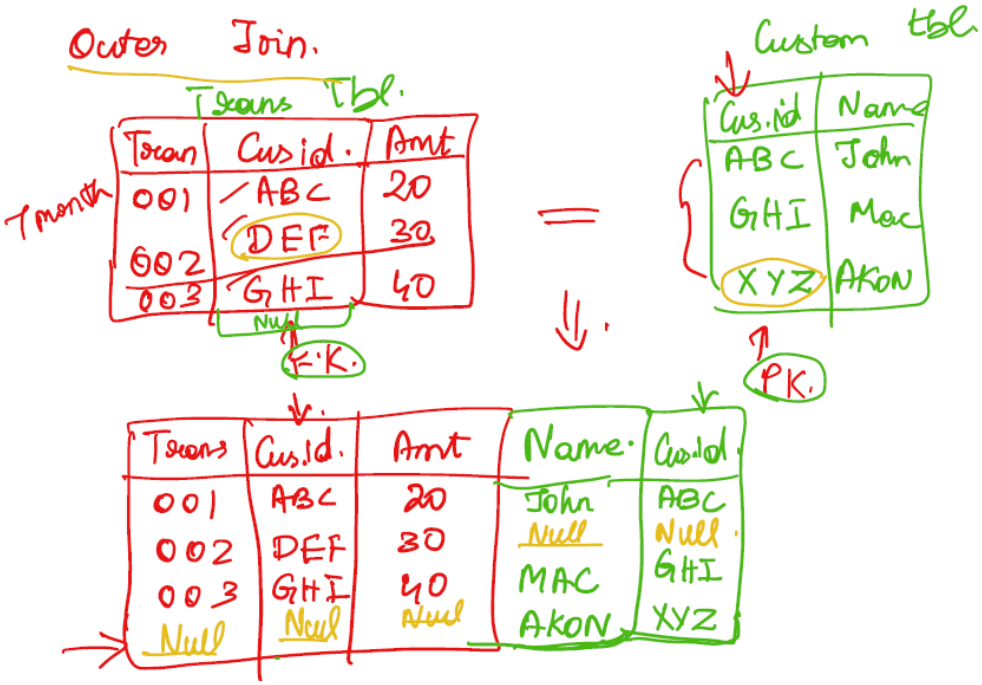
- This specifies the condition for the RIGHT JOIN. It matches rows from the film_category table with the category table where the category_id is the same in both tables.
- c.name AS category_name selects the name column from the category table and renames it as category_name in the output.
- COUNT(fc.film_id) AS film_count counts the number of film_id entries in the film_category table for each category. This count is renamed as film_count in the output. The COUNT function will return 0 for categories with no films because of the RIGHT JOIN.
- This groups the results by category_id and name from the category table. Grouping is necessary when using aggregate functions like COUNT to ensure counts are calculated per category.
- This sorts the results first by film_count in descending order (DESC). Categories with more films will appear first.
- If there are categories with the same number of films, it further sorts them alphabetically by category_name.

Full Outer Join



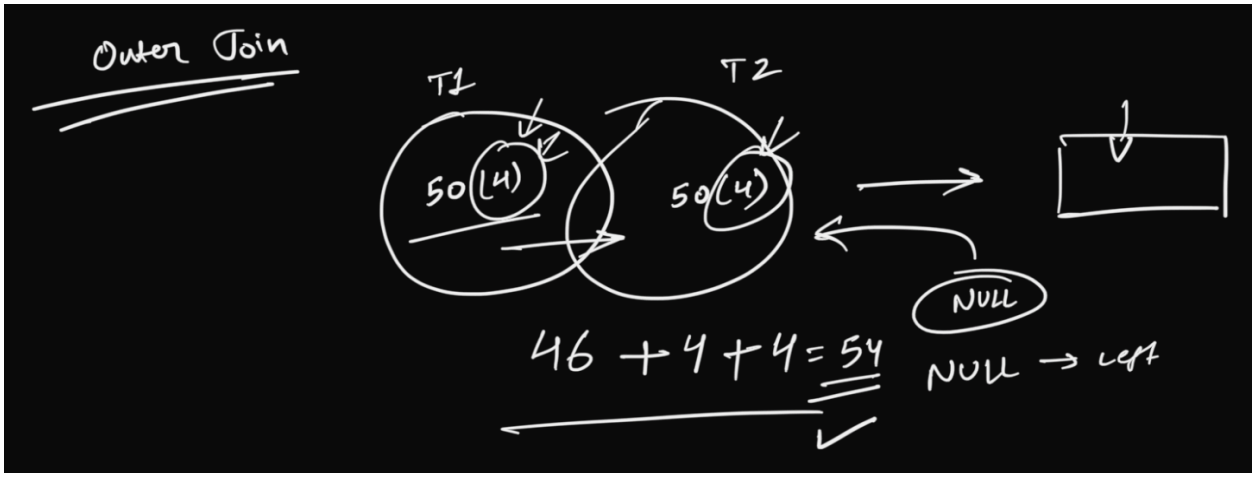
A full outer join returns all rows from both the tables whether there is a match in the left (table1) or right (table2) table records.

Example -



Suppose you have two tables, T1 and T2, both with 50 records each.

When we try to match these two tables, we find that there are 4 mismatching rows.



How many rows would we get after joining these two tables using a full outer join?

Ans. 54

Important:

- MySQL does not support FULL OUTER JOIN, we have to combine the records we get from LEFT JOIN and RIGHT JOIN to get an equivalent using **UNION**.
- Those columns that exist in only one table will contain NULL in the opposite table.

How do we combine the records, where all the columns are the same, in columns in the same order with the same data types?

- We perform **UNION.**

Now, let's learn about **UNION**.

Example:

Imagine you're a financial analyst at Uber headquarters. The CEO has requested a report on Uber's (i.e., Uber Cabs, Uber Eats) overall revenue. You know Uber offers ride-hailing services through its core "Uber Cabs" platform, but it has also expanded into food delivery with "Uber Eats." To get a complete picture of Uber's financial health, you need to consider revenue from both sectors.

If you want to look at revenue from all types of sales - Uber Cabs, Uber Eats. How will you get that?

How to do it in MySQL:

```
SELECT *
FROM A
```

```
LEFT JOIN B
        ON A.id = B.id

UNION

SELECT  *
FROM A
RIGHT JOIN B
        ON A.id = B.id;
```

Instructor Note: *[In MySQL, by default, UNION returns distinct values from both tables and gets rid of duplicate values.]*

How to use UNION in BigQuery:

- UNION is only supported in MySQL, not in BigQuery.
- For BigQuery you'll have to use **UNION_DISTINCT**.
- Note that UNION_DISTINCT gets rid of any duplicate records.
- If you wish to include the duplicates, use **UNION_ALL** instead.

Q1: List all stores, including stores with no active customers.

What type of JOIN should we use here?
Answer: Left JOIN

What table should be on the left?

Answer: Store table

Query:

```
SELECT
    s.store_id,
FROM store s
LEFT JOIN customer c
ON s.store_id = c.store_id;
```

- There can be customers without any purchases.
- The customer table has details of all the customers regardless of their purchase history.
- Since we did a LEFT JOIN, we're getting a list of all stores, and their associated customers, if there are any.
- Customers with purchases & active will show up in the output multiple times for each store if one store has multiple customers.
- Customers without purchases will have NULL values in all fields displayed from the stores table.

List all stores and the number of customers, including stores with no customers.

Query 1:

```
SELECT
    s.store_id,
    COUNT(c.customer_id) AS customer_count
FROM store s
LEFT JOIN customer c
ON s.store_id = c.store_id
GROUP BY s.store_id
```

- A LEFT JOIN is performed between the ***store*** table and the ***customer*** table.
- The join condition ON ***s.store_id = c.store_id*** specifies that rows from the store table are matched with rows from the customer table where the store_id is the same in both tables.

- The **LEFT JOIN** ensures that all rows from the store table are included in the result, even if there are no matching rows in the customer table. If a store has no customers, the corresponding customer_id will be NULL.
- Stores with no customers will have a count of 0 because COUNT on NULL values results in 0.

Q2: Get all the customers with missing store information.

What type of JOIN should we use here?
Answer: Right JOIN

Query:

```
SELECT *
FROM store s
RIGHT JOIN customer c
ON s.store_id = c.store_id
```

- There can be customers with missing store information.
- The customer table has details of all the customers.
- Since we did a RIGHT JOIN, we're getting a list of all customers, and their associated stores.
- Customers with store information & active will show up in the output multiple times for each store if one store has multiple customers.
- Customers without any store information will have NULL values in all fields displayed from the customers table.

List all stores and the number of customers, including customers with missing store information.

Query 2:

```
SELECT
    s.store_id,
    COUNT(c.customer_id) AS customer_count
FROM store s
RIGHT JOIN customer c
ON s.store_id = c.store_id
GROUP BY s.store_id;
```

- A RIGHT JOIN is performed between the ***store*** table and the ***customer*** table.
- The join condition ON ***s.store_id = c.store_id*** specifies that rows from the customer table are matched with rows from the store table where the store_id is the same in both tables.
- The **RIGHT JOIN** ensures that all rows from the customer table are included in the result, even if there are no matching rows in the store table. If a customer has no store information, the corresponding store_id will be NULL.
- Customers with no store information will have a count of 0 because COUNT on NULL values results in 0.

Now let's combine these two queries i.e. Query 1 & Query 2, to obtain the final solution query.

Soln using UNION -

List all stores and the number of active customers, including stores with no active customers and customers with missing store information.

```
SELECT s.store_id,
    COUNT(c.customer_id) AS customer_count
FROM store s
LEFT JOIN customer c
ON s.store_id = c.store_id
```

```
GROUP BY s.store_id

UNION

SELECT s.store_id,
      COUNT(c.customer_id) AS customer_count
FROM store s
RIGHT JOIN customer c
ON s.store_id = c.store_id
GROUP BY s.store_id;
```

Output:

store_id	customer_count
1	327
2	275
3	0
4	0
NULL	4

Now let's also try to write the solution query using a Full Outer Join in **Bigquery**

Soln using Full Outer Join -

```
SELECT
      s.store_id,
     COUNT(c.customer_id) AS customer_count
FROM `cineflix_store.store` as s
FULL OUTER JOIN `cineflix_store.customer` as c
          ON s.store_id = c.store_id
GROUP BY s.store_id
ORDER BY s.store_id;
```

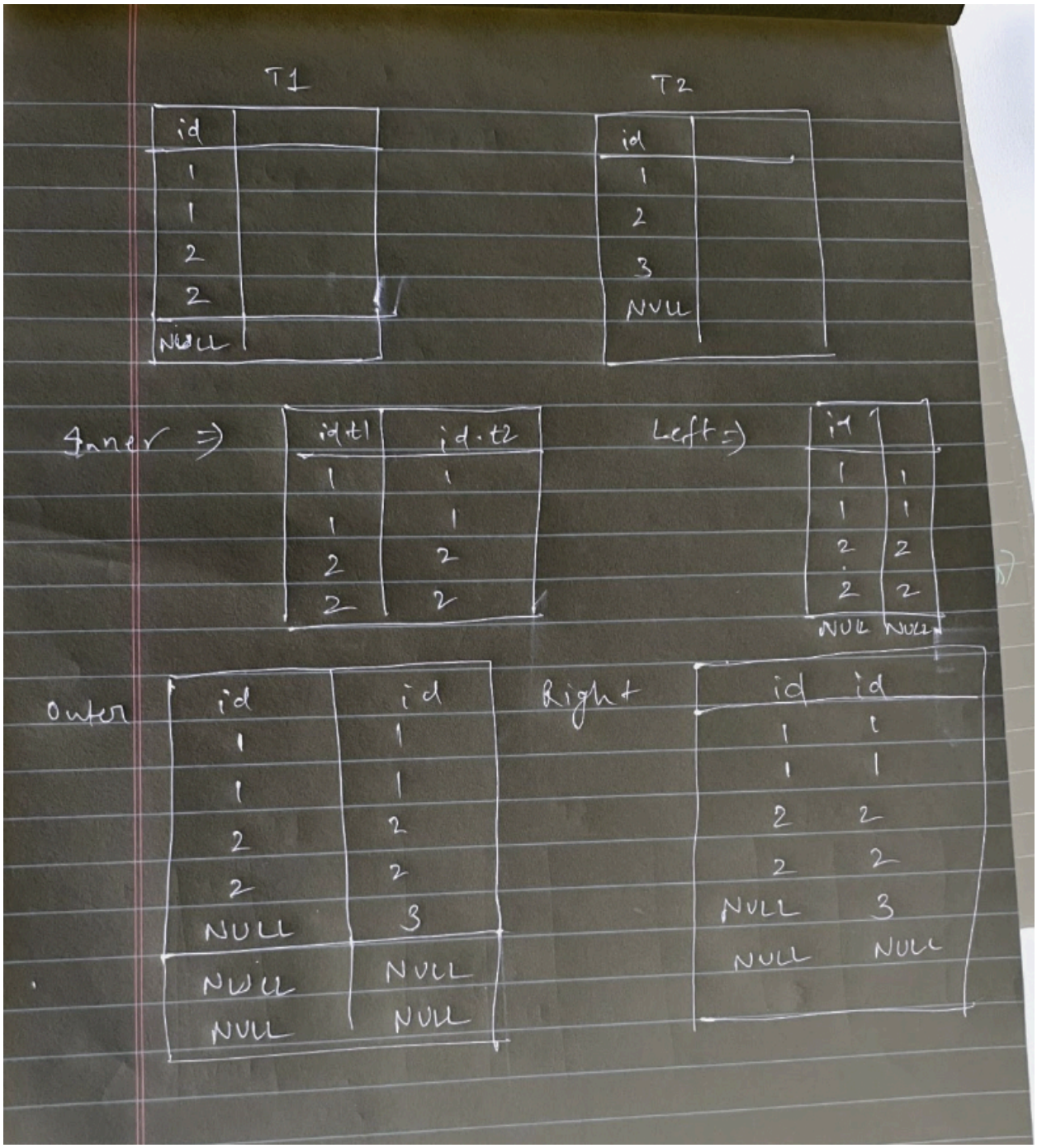
Output:

Query results

JOB INFORMATION | RESULTS | CHA

Row	store_id	customer_count
1	*null*	4
2	1	327
3	2	275
4	3	0
5	4	0

Examples of different types of JOINs



Insights & Recommendations

1. Top Rented Films:

This revealed the most popular films based on rental frequency. CineFlix can use this information to:

- Prioritize stocking these films in sufficient quantities to meet demand. Consider promoting these films through in-store displays or targeted marketing campaigns.

2. High-Volume Renters:

This identified customers who rent more films than the average. These are likely loyal customers who contribute significantly to CineFlix's revenue. CineFlix can leverage this information to:

- Develop targeted loyalty programs or special offers to retain these high-value customers. Gather feedback from these customers to understand their preferences and improve the rental experience.

3. Customer Spending:

This provided insights into customer spending habits, It includes customers who haven't rented any films. CineFlix can use this information to:

- Identify potential inactive customers who haven't rented recently and develop strategies to re-engage them (e.g., special discounts). Understand the overall revenue generated from rentals.

4. Film Categories:

This revealed the number of films in each category, including categories with no films. CineFlix can use this information to:

- Identify popular film categories and potentially adjust their inventory allocation accordingly. Explore opportunities to expand their film selection in under-represented categories if there's customer demand.

5. Store Performance:

Left Join (First Part): This includes all stores, even those with no customers. It helps identify stores that might require additional marketing efforts or location adjustments.

Right Join (Second Part): This includes all customers, even those with missing store information. It provides a more comprehensive view of customer distribution, considering customers who might have rented from stores with missing data.

CineFlix can use this information to:

- Evaluate the performance of individual stores based on customer traffic. Identify stores that might benefit from resource allocation adjustments (staffing, inventory). Investigate reasons behind missing store information for some customers and improve data collection practices.

Overall Summary:

By analyzing this data, CineFlix can gain valuable insights into customer behaviour, film popularity, store performance, and overall revenue generation. This data-driven approach can help them optimize their film selection, marketing strategies, inventory management, and store operations to remain competitive in the face of streaming services.
