Joins Contd.

---

1. Problem Statement
2. Joining multiple tables
3. Self JOIN
4. Cross JOIN
5. Non-Equi JOINS
6. 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 specializing 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.

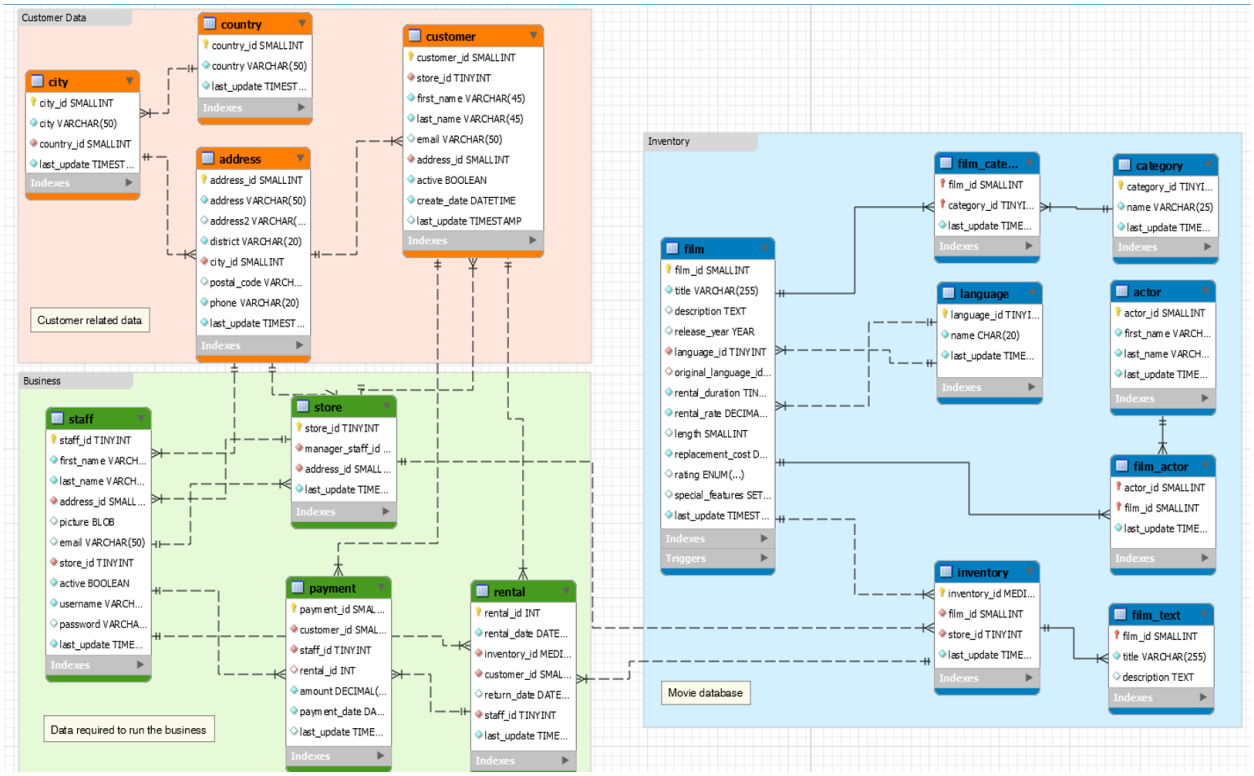
2. **Customer Behavior Analysis**: Gain insights into customer demographics, retention rates, and preferences to tailor marketing strategies and improve customer satisfaction.
3. **Store Performance**: Evaluate the performance of different store locations to identify areas for improvement and optimize 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Optimize inventory levels to ensure popular titles are available while minimizing costs associated with overstock and outdated inventory.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 all rental pairs where both rentals were made by the same customer & later returned it.

- **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.

JOINs with more than two tables

Question: For each store, list the total number of films and the number of currently rented films.

Imagine that you're building an interactive report that lets you filter a store, rentals, and inventory to see the total number of films each store has (including unrented ones) and the number of currently rented films.

So we need a merged dataset that contains all of their records.

This requires joining the three tables:

1. store
2. customer
3. rental
4. inventory
5. film

What kind of JOINs do you think we could use to ensure that all stores are included, even if they haven't rented to any customer?

- We use a LEFT JOIN with the customer table on store_id. This ensures all stores are included even if they haven't had any rentals.

Flow using JOINs:

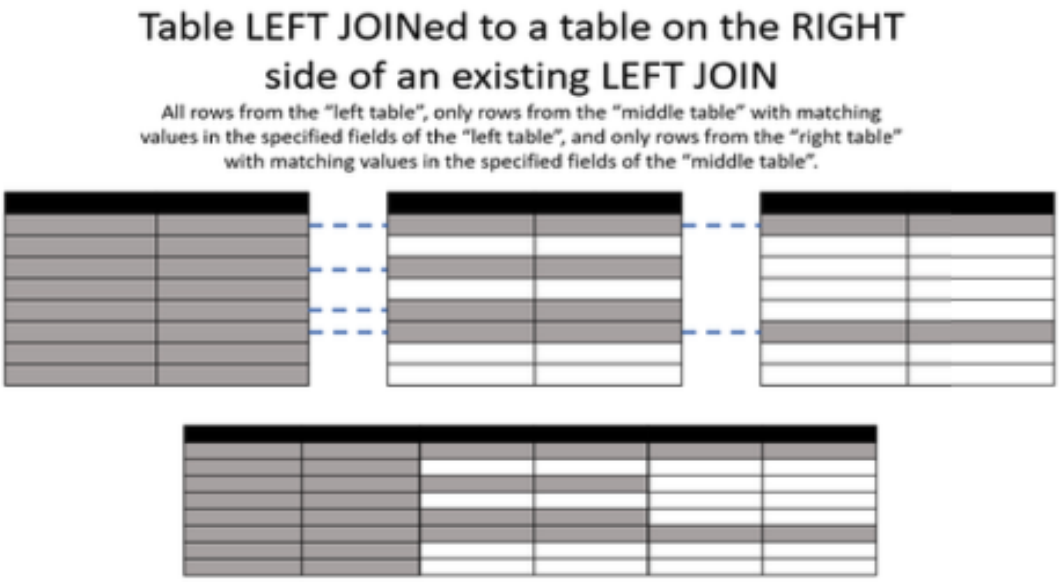
1. We begin with the store table, as it represents the stores for which we want film information.
2. LEFT JOIN with the customer: Since a store might not have any rentals, we use a LEFT JOIN with the customer table on store_id. This ensures all stores are included even if they haven't had any rentals.
3. LEFT JOIN with rental: We then LEFT JOIN the rental table with the customer table on customer_id. This connects rental information to stores through customers. Similar to the previous join, this ensures all stores are considered regardless of rentals.
4. LEFT JOIN with inventory: We LEFT JOIN the inventory table with the rental table on inventory_id. This connects film inventory details to rental information.
5. LEFT JOIN with film: Finally, we LEFT JOIN the film table with the inventory table on film_id. This brings film information into the query.

The query to accomplish these joins looks like this -

```
SELECT
```

```
    s.store_id,
    COUNT(f.film_id) AS total_films,
    COUNT(CASE WHEN r.return_date IS NULL AND inv.inventory_id IS NOT NULL THEN 1 ELSE NULL END) AS rented_films
FROM store s
LEFT JOIN customer c
  ON s.store_id = c.store_id
LEFT JOIN rental r
  ON c.customer_id = r.customer_id
LEFT JOIN inventory inv
  ON inv.inventory_id = r.inventory_id
LEFT JOIN film f
  ON inv.film_id = f.film_id
GROUP BY s.store_id;
```

You can think of the second JOIN as being merged into the result of the first JOIN. resembling something like this:



Self JOIN

Question: Find the customers and how many pairs of rentals they have, where the same customer rented both rentals and later returned.

We want to reward our most loyal customers at the Entertainment Store with discount coupons.

For that, we need to analyse rental pairs where a customer rents and then returns a movie, which can help us identify frequent renters.

Understanding how often customers rent multiple movies simultaneously can inform future inventory management and marketing strategies.

Now, how would you solve this problem?

- **Self JOIN**

What is self-join?

Self-join is a regular join that is used to join a table with itself. It basically allows us to combine the rows from the same table based on some specific conditions. It is very useful and easy to work with, and it allows us to retrieve data or information that involves comparing records within the same table.

Why self-join?

We use a self-join on the rental table because we're analysing rental data for the same customer. We want to compare two **separate** rentals within the same table to see if the same customer made both and later returned them. A self-join allows us to match records within the same table based on a specific column (customer ID in this case).

Query:

```
SELECT
    rental1.customer_id,
    COUNT(*) AS rental_pair_count
FROM rental AS rental1
INNER JOIN rental AS rental2
    ON rental1.customer_id = rental2.customer_id
    AND rental1.rental_id < rental2.rental_id
    AND rental1.return_date IS NOT NULL
    AND rental2.return_date IS NOT NULL
GROUP BY rental1.customer_id
ORDER BY rental_pair_count DESC, rental1.customer_id;
```

Breaking down **Self JOIN** :

- The core operation is a SELF JOIN on the rental table. This means the table is joined to itself, but with aliases to differentiate between the two instances (rental1 and rental2).
- The ON clause specifies the condition for joining the two instances of the rental table. Here, it ensures both rentals are connected to the same customer:
    - ***rental1.customer_id = rental2.customer_id***: This condition guarantees both rentals in the pair belong to the same customer by matching their customer IDs.
    - ***AND rental1.rental_id < rental2.rental_id:*** This ensures that we do not compare the same rental record and avoid

duplicate pairs. It also ensures that rental1 occurs before rental2. Guarantees the first rental (lower ID) is returned before the second.
    - ***rental1.return_date IS NOT NULL*** and ***rental2.return_date IS NOT NULL***: Confirms successful return of both rentals.
- **GROUP BY rental1.customer_id** aggregates the rental count for each unique customer.
- Finally, sort the output first by rental_pair_count (highest to lowest) and then by customer ID for better organisation.

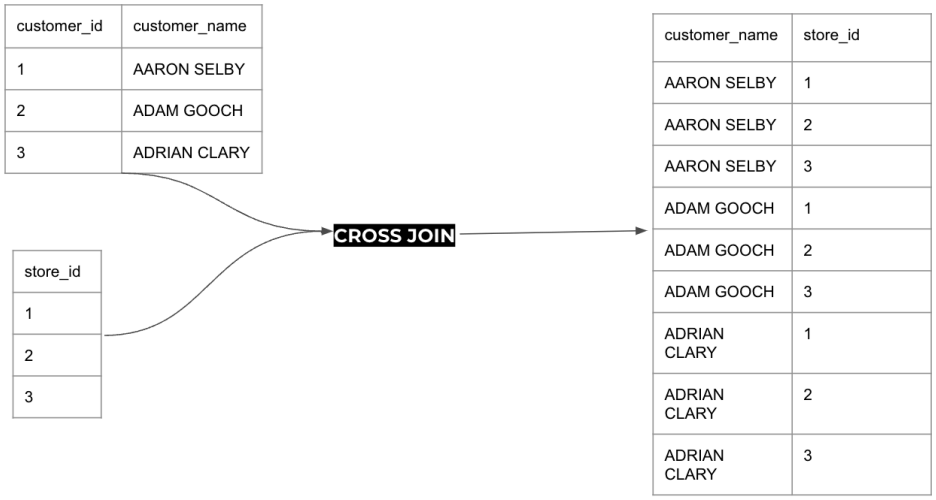
Cross JOIN

Question: List all possible pairs of customers and stores.

To list all possible pairs of customers and stores, you need to perform a Cartesian join between the customer and store tables. This type of join will pair every row from the customer table with every row from the store table, generating all possible combinations.

A **Cross JOIN,** also known as the Cartesian JOIN, matches each record from one table (say T1) with each record from the other table (say T2).

So if we have ‘n’ no. of rows in T1 and ‘m’ no. of rows in T2 then the resulting table (say T3) will have ‘nxm’ no. of rows.




```
SELECT
        CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
        s.store_id
FROM customer c
CROSS JOIN store s
ORDER BY customer_name, s.store_id;
```

Another example of Cross JOINs could be: You want to calculate the total sales of a product in every store in a country. For every product, we'd need to create a combination of product and store mapping.

Thus, we'd need to use cross-join for that.

Equi & Non-equi Joins

We have seen equi-joins which are the regular joins.

- **Non-equi Join** is also a type of Join in which we need to retrieve data from multiple tables.

- **Non-Equi Join** matches the column values from different tables based on an inequality based on the operators like, **<,>,<=,>=,!=, BETWEEN**, etc.

Question 1: List all films with rental rates higher than the average rental rate.

```
SELECT
        f.film_id,
    f.title AS film_title,
    f.rental_rate
FROM film f
JOIN (
  SELECT AVG(rental_rate) AS avg_rental_rate
  FROM film
) AS avg_table
ON f.rental_rate > avg_table.avg_rental_rate
ORDER BY f.rental_rate DESC;
```

This subquery calculates the average rental rate:

- AVG(rental_rate): This function calculates the average of the rental_rate column from all rows in the film table.
- AS avg_rental_rate: This aliases the result of the subquery as avg_rental_rate.
- FROM film: This specifies the film table as the source for the average calculation.

Joining Films with Average Rate:

- JOIN: This keyword initiates the join operation.
- (...) AS avg_table: This references the subquery we created earlier, containing the average rental rate calculation.
- ON f.rental_rate > avg_table.avg_rental_rate: This specifies the join condition. It ensures only films with a rental rate (f.rental_rate) greater than the average rental rate (avg_table.avg_rental_rate) are included in the results.

Question 2: For each customer, identify the names of other customers who have rented the same inventory item(s) at least once.

```
SELECT
    CONCAT(c1.first_name,' ',c1.last_name) AS customer1,
    CONCAT(c2.first_name,' ',c2.last_name) AS customer2
FROM customer c1
INNER JOIN rental r1
    ON c1.customer_id = r1.customer_id
INNER JOIN rental r2
    ON r1.inventory_id = r2.inventory_id
    AND r1.rental_id <> r2.rental_id
INNER JOIN customer c2
    ON c2.customer_id = r2.customer_id
ORDER BY customer1, customer2;
```

The query joins four tables:

- customer: Contains customer information (including customer_id, first_name, and last_name).
- rental: Stores rental data (including customer_id and inventory_id).
    - Two instances of rental are aliased as r1 and r2 for clarity in the join conditions.

Connecting Rentals and Customers:

- c1.customer_id = r1.customer_id: This joins the first customer (c1) with their rentals (r1) based on customer ID.
- r1.inventory_id = r2.inventory_id: This connects rentals from r1 to rentals in r2 that share the same inventory item (identified by inventory_id).

Filtering for Different Customers:

- r1.rental_id <> r2.rental_id: This crucial condition ensures we don't include a customer renting the same item twice. It excludes self-joins and focuses on identifying different customers who rented the same inventory item.

Joining the Second Customer:

- c2.customer_id = r2.customer_id: This connects the second rental (r2) to a different customer (c2) who also rented the same inventory item.

Selecting Customer Names:

- CONCAT(c1.first_name,' ',c1.last_name) AS customer1: This concatenates the first and last name of the first customer (c1) with a space in between, creating an alias customer1 for better readability in the output.
- CONCAT(c2.first_name,' ',c2.last_name) AS customer2: This follows the same logic for the second customer (c2), creating an alias customer2.

Ordering the Results:

- ORDER BY customer1, customer2: This sorts the final output alphabetically first by customer1 (the first customer's name) and then by customer2 (the second customer's name).

Insights & Recommendations

Total Films per Store:

This provided a comprehensive view of film inventory across all stores. We can see how many films each store has in total and how many are currently rented.

- Analyze stores with low film counts or high rental rates to identify potential stock shortages or high-demand films. Consider allocating inventory based on rental trends and store locations. Consider adjusting inventory levels to ensure sufficient stock of these popular films.

Rental Pairs by Same Customer:

This identified customers who have rented the same inventory items as other customers, potentially indicating similar preferences or interests.

- Customers have the highest rental pair counts, the business can target these loyal customers with personalized promotions or rewards to further enhance their rental experience and encourage repeat business.
- Understanding how often customers rent multiple movies simultaneously can inform decisions about stocking movie titles to meet rental demand and optimize inventory management.

Films with Rental Rates Above Average:

This query provides insights into rental status, allowing for better inventory management and understanding of store performance.

- Optimize inventory levels based on rental trends to ensure stores have popular films in stock. Consider special pricing strategies for these films. Offer promotions or discounts on currently rented films to incentivize customers to return rentals promptly, freeing up inventory for new rentals.

Customers Who Rented the Same Inventory Items:

This query identifies customers who have rented the same movies, revealing potential shared preferences.

- Leverage this information to suggest similar movies to customers based on their rental history. This can be done through targeted email campaigns or in-store recommendations. Implement personalized marketing campaigns targeting these loyal customers with tailored offers and recommendations.

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.
