Module 03

Purwadhika
Digital Technology School

Full Stack Web Development

Advanced Database Design and Development

Job Connector Program

Database Design

Database design can be generally defined as a collection of tasks or processes that enhance the designing, development, implementation, and maintenance of enterprise data management system. Designing a proper database reduces the maintenance cost thereby improving data consistency and the cost-effective measures are greatly influenced in terms of disk storage space. Therefore, there has to be a brilliant concept of designing a database. The designer should follow the constraints and decide how the elements correlate and what kind of data must be stored.

What is good database design?

A good database design is, therefore, one that:

- Divides your information into subject-based tables to reduce redundant data.
- Provides Access with the information it requires to join the information in the tables together as needed.
- Helps support and ensure the accuracy and integrity of your information.
- Accommodates your data processing and reporting needs.

The Design Process

The design process consists of the following steps:

- **Determine the purpose of your database** - This helps prepare you for the remaining steps.
- **Find and organize the information required** - Gather all of the types of information you might want to record in the database, such as product name and order number.
- **Divide the information into tables** - Divide your information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.
- **Turn information items into columns** - Decide what information you want to store in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.

The Design Process Continue

- **Specify primary keys** - Choose each table’s primary key. The primary key is a column that is used to uniquely identify each row. An example might be Product ID or Order ID.
- **Set up the table relationships** - Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.
- **Refine your design** - Analyze your design for errors. Create the tables and add a few records of sample data. See if you can get the results you want from your tables. Make adjustments to the design, as needed.
- **Apply the normalization rules** - Apply the data normalization rules to see if your tables are structured correctly. Make adjustments to the tables, as needed.

Database Normalization

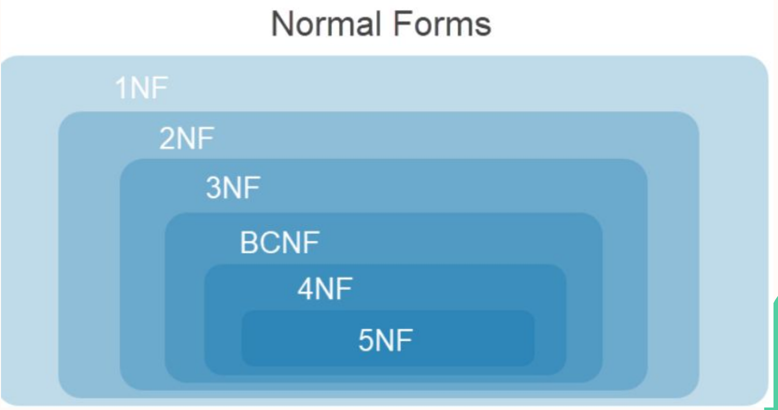
The elementary concepts used in database normalization are:

- **Keys**. Column attributes that identify a database record uniquely.
- **Functional** Dependencies. Constraints between two attributes in a relation.
- **Normal Forms**. Steps to accomplish a certain quality of a database.

Database Normal Form

Normalizing a database is achieved through a set of rules known as normal forms. The central concept is to help a database designer achieve the desired quality of a relational database.

A database is normalized when it fulfills the third normal form. Further steps in normalization make the database design complicated and could compromise the functionality of the system.



Stage of Normalization

Stage	Redundancy Anomalies Addressed
Unnormalized Form (UNF)	The state before any normalization. Redundant and complex values are present.
First Normal Form (1NF)	Repeating and complex values split up, making all instances atomic.
Second Normal Form (2NF)	Partial dependencies decompose to new tables. All rows functionally depend on the primary key.
Third Normal Form (3NF)	Transitive dependencies decompose to new tables. Non-key attributes depend on the primary key.
Boyce-Codd Normal Form (BCNF)	Transitive and partial functional dependencies for all candidate keys decompose to new tables.
Fourth Normal Form (4NF)	Removal of multivalued dependencies.
Fifth Normal Form (5NF)	Removal of JOIN dependencies.

Database Key

What is a KEY?

A database key is an attribute or a group of features that uniquely describes an entity in a table. The types of keys used in normalization are:

- **Super Key.** A set of features that uniquely define each record in a table.
- **Candidate Key**. Keys selected from the set of super keys where the number of fields is minimal.
- **Primary Key**. The most appropriate choice from the set of candidate keys serves as the table's primary key.
- **Foreign Key**. The primary key of another table.
- **Composite Key**. Two or more attributes together form a unique key but are not keys individually.

Database Key - Super Key

employeeID	name	age	email
1	Adam A.	30	adam.a@email.com
2	Jacob J.	27	jacob.j@email.com
3	David D.	35	david.d@email.com

Some examples of super keys in the table are:

- employeeID
- (employeeID, name)
- email

All super keys can serve as a unique identifier for each row. On the other hand, the employee's name or age are not unique identifiers because two people could have the same name or age.

Database Key - Candidate Key

employeeID	name	age	email
1	Adam A.	30	adam.a@email.com
2	Jacob J.	27	jacob.j@email.com
3	David D.	35	david.d@email.com

The candidate keys come from the set of super keys where the number of fields is minimal. The choice comes down to two options:

- employeeID
- email

Both options contain a minimal number of fields, making them optimal candidate keys.

Database Key - Primary Key

- The Primary keys constraint uniquely identifies each record in a database table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only one primary key, which may consist of single or multiple fields. **Example: No_KTP, product_key, ID_sidikjari**

Database Key - Foreign Key

A foreign key is a field or a column that is used to establish a link between two tables. In simple words you can say that, a foreign key in one table used to point primary key in another table.

Foreign Key

First table:

S_Id	LastName	FirstName	CITY
1	MAURYA	AJEET	ALLAHABAD
2	JAISWAL	RATAN	GHAZIABAD
3	ARORA	SAUMYA	MODINAGAR

Second table:

O_Id	OrderNo	S_Id
1	99586465	2
2	78466588	2
3	22354846	3
4	57698656	1

The "S_Id" column in the 1st table is the PRIMARY KEY in the 1st table.

The "S_Id" column in the 2nd table is a FOREIGN KEY in the 2nd table.

Join Table

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Join Table

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Join Table

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Join Table Syntax

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

In this part we would like to join the tables between Orders with Customers table.

The Orders refer to a table name, while the OrderID refers to a column name inside the Orders table.

The Customers refer to a table name, while the CustomerName refers to a column name inside the Customers table.

Join Table Syntax

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

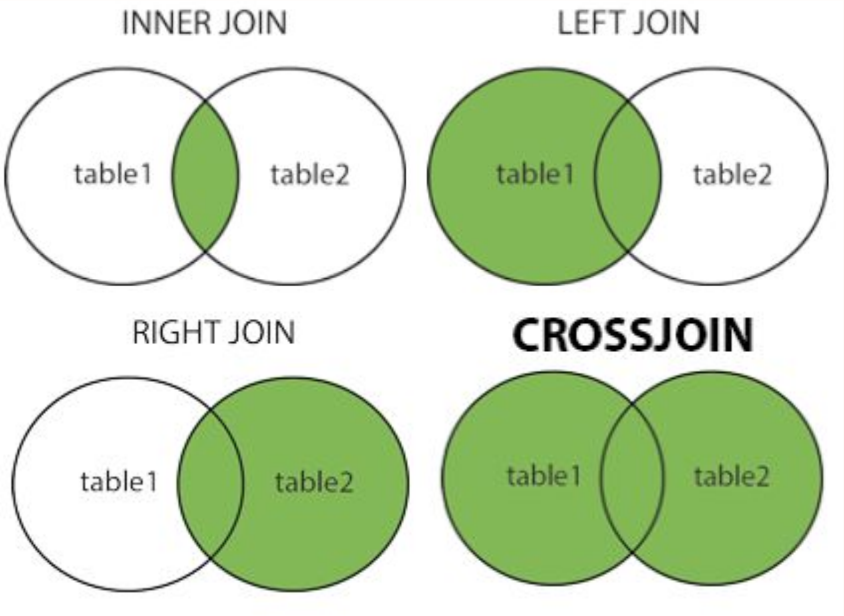
“FROM Orders” this mean source of the table, while “INNER JOIN” represent to type of join syntax and following by table name “INNER JOIN Customers” this mean that Customers table would be join with INNER JOIN type to table Orders.

Join Table Syntax

```sql
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

“ON” represent to highlight which column that could be relate one table to another using the **foreign key**. In this case CustomerID column on table Orders is the foreign key.

Type of SQL Join



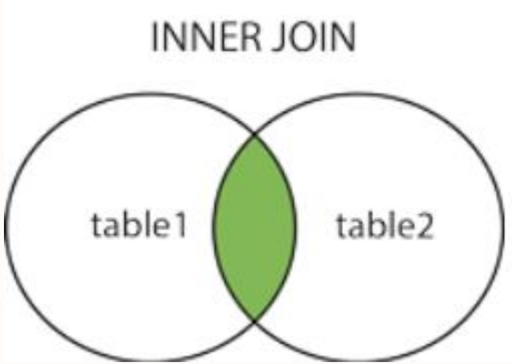
Go to https://dev.mysql.com/doc/index-other.html and download sakila db. Extract and import sakila data into your MySQL.

We would use this sample data to learn about join

Inner Join

The INNER JOIN keyword selects records that have matching values in both tables.

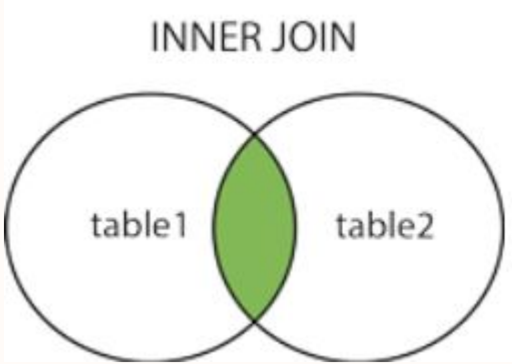
```sql
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



Inner Join Example

The INNER JOIN keyword selects records that have matching values in both tables.

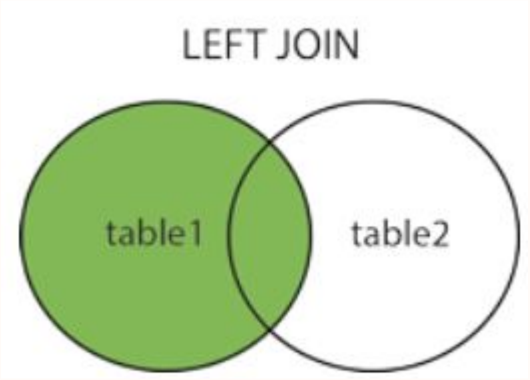
```sql
SELECT * FROM country
INNER JOIN city
ON country.country_id = city.country_id;
```



Left Join

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

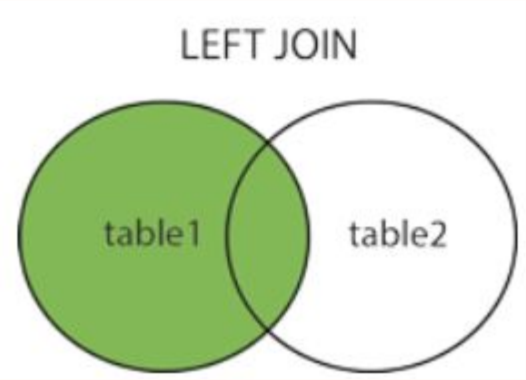
```sql
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```



Left Join Example

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

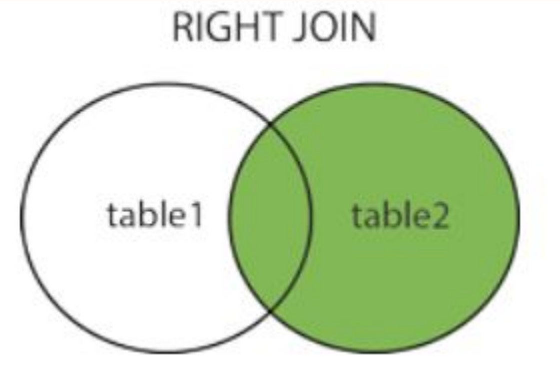
```
SELECT * FROM customer
LEFT JOIN actor
ON customer.last_name = actor.last_name;
```



Right Join

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

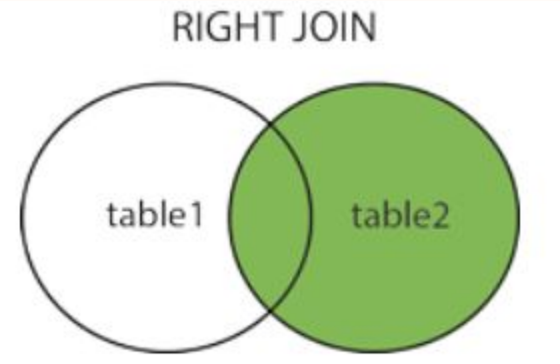
```sql
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```



Right Join Example

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

```sql
SELECT * FROM customer
RIGHT JOIN actor
ON customer.last_name = actor.last_name;
```



Cross Join

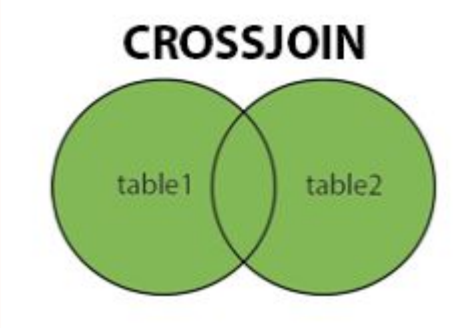
The CROSS JOIN keyword returns all records from both tables (table1 and table2).

CROSS JOIN can potentially return very large result-sets!

The CROSS JOIN keyword returns all matching records from both tables whether the other table matches or not, those rows will be listed as well.

If you add a WHERE clause (if table1 and table2 has a relationship), the CROSS JOIN will produce the same result as the INNER JOIN clause:

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```



Cross Join

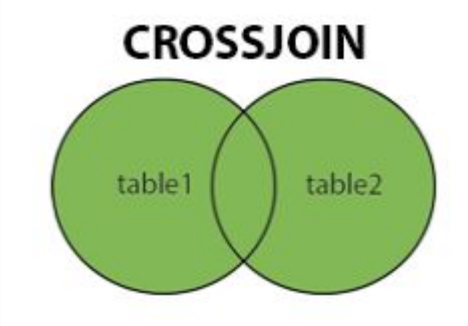
The CROSS JOIN keyword returns all records from both tables (table1 and table2).

CROSS JOIN can potentially return very large result-sets!

The CROSS JOIN keyword returns all matching records from both tables whether the other table matches or not, those rows will be listed as well.

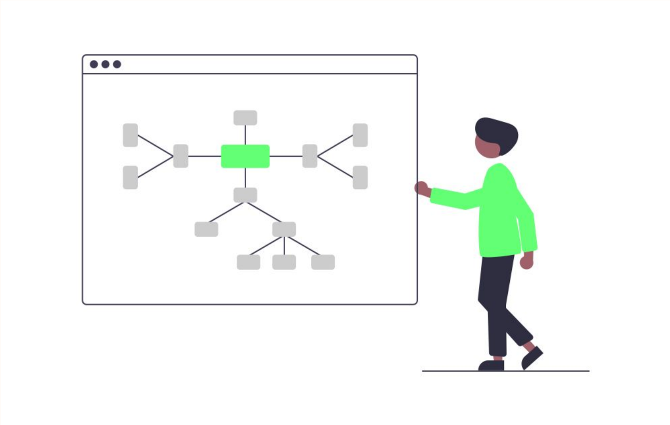
If you add a WHERE clause (if table1 and table2 has a relationship), the CROSS JOIN will produce the same result as the INNER JOIN clause:

```
SELECT * FROM customer
CROSS JOIN actor;
```



Database Relationships in MySQL

Database relationship means how the data in one table is related to the data in another table. In RDBMS (Relational Database Management System). The term “Relational” refers to the tables with Relations. Relationships between two tables are created using keys. A key in one table will normally relate to a key in another table. Two tables in a database may also be unrelated.



Database Relationships in MySQL

There are mainly 3 types of database relationships:

1. One-to-one (1:1) Relationship: If only one data in one table relates to the only one data in another table it is known as one-to-one (1:1) relationship.
2. One-to-many (1:M) Relationship: If only one data in one table relates to the multiple data in another table it is known as the one-to-many (1:M) relationship.
3. Many-to-many (M:M) Relationship: And if multiple data in one table relates to the multiple data in another table it is known as many-to-many (M:M) relationship.

One-to-one (1:1) Relationship

In a One-to-One (1:1) Relationship only one data in one table relates to only one data in another table. Take a look at the example tables, we will use sakila database. We have two tables: customer and address. First table stores customer_id, first_name, last_name, address_id. The second table stores address_id and address column values which are shown in the below image.

#	customer_id	store_id	first_name	last_name	email	address_id
1	1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5
2	2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacusto...	6
3	3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer...	7
4	4	2	BARBARA	JONES	BARBARA.JONES@sakilacustome...	8
5	5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacusto...	9

#	address_id	address	address2	district	city_id	postal_code	phone
1	1	47 MySakila Drive	NULL	Alberta	300		
2	2	28 MySQL Boulevard	NULL	QLD	576		
3	3	23 Workhaven Lane	NULL	Alberta	300		14033335568
4	4	1411 Lillydale Drive	NULL	QLD	576		6172235589
5	5	1913 Hanoi Way		Nagasaki	463	35200	28303384290

One-to-one (1:1) Relationship

One-to-one relationships in databases are used for various reasons and serve specific purposes. Here are a few reasons why one might choose to use a one-to-one relationship in a database:

1. Security and access control: In some cases, you may want to restrict access to certain sensitive information. By placing sensitive data in a separate table with a one-to-one relationship, you can apply stricter access controls to that table, ensuring that only authorized users can view or modify the data.
2. Performance optimization: One-to-one relationships can help improve query performance. When you have large tables with a mix of frequently and infrequently accessed columns, splitting them into separate tables can reduce the overall data volume and improve query execution time.
3. Flexibility and scalability: One-to-one relationships provide flexibility and scalability in database design. If you anticipate that certain attributes may change or expand in the future, having separate tables allows you to easily add or modify columns without affecting the structure of the main table.

One-To-Many (1:M) Database Relationship

One-to-many relationship is a type of relationship between two entities in a database. It's called "one-to-many" because it describes a relationship where one entity (let's call it Entity A) can be associated with multiple instances of another entity (Entity B), but each instance of Entity B is associated with only one instance of Entity A.

One-To-Many (1:M) Database Relationship

Take a look at the example between the "customers" and "rentals" tables. Each customer can have multiple rentals, but each rental belongs to only one customer. Here's an example SQL query that retrieves the rentals for a specific customer:

```sql
SELECT c.customer_id, r.rental_id, r.rental_date, r.return_date
FROM rental r
JOIN customer c ON r.customer_id = c.customer_id
WHERE c.customer_id = 130;
```

#	customer_id	rental_id	rental_date	return_date
1	130	1	2005-05-24 22:53:30	2005-05-26 22:04:30
2	130	746	2005-05-29 09:25:10	2005-06-02 04:20:10
3	130	1630	2005-06-16 07:55:01	2005-06-19 06:38:01
4	130	1864	2005-06-17 01:39:47	2005-06-24 19:39:47
5	130	2163	2005-06-17 23:46:16	2005-06-22 22:48:16
6	130	2292	2005-06-18 07:37:48	2005-06-20 02:45:48

Many-To-Many (M:M) Database Relationship

Many-to-many relationship is a type of relationship between two entities where each entity can be associated with multiple instances of the other entity. In this type of relationship, the cardinality is "many" on both sides.

Many-To-Many (M:M) Database Relationship

Let's consider the Sakila database and demonstrate a many-to-many relationship using the "film" and "actor" tables. Each film can have multiple actors, and each actor can participate in multiple films. Here's an example query:

```sql
SELECT f.film_id, f.title, a.actor_id, a.first_name, a.last_name
FROM film_actor fa
JOIN film f ON fa.film_id = f.film_id
JOIN actor a ON fa.actor_id = a.actor_id
WHERE f.title = 'ACADEMY DINOSAUR';
```

Many-To-Many (M:M) Database Relationship

In this example, we have the following tables involved:

- The "film" table contains film information, including the film_id and title.
- The "actor" table contains actor information, including the actor_id, first_name, and last_name.
- The "film_actor" table serves as the junction table, connecting films and actors by storing the film_id and actor_id as foreign keys.
- The query retrieves the film_id, title, actor_id, first_name, and last_name for the film with the title 'ACADEMY DINOSAUR. By joining the "film_actor" table with the "film" and "actor" tables, we establish the many-to-many relationship between films and actors.

```
SELECT f.film_id, f.title, a.actor_id, a.first_name, a.last_name
FROM film_actor fa
JOIN film f ON fa.film_id = f.film_id
JOIN actor a ON fa.actor_id = a.actor_id
WHERE f.title = 'ACADEMY DINOSAUR';
```

Aggregate Function

An aggregate function in SQL performs a calculation on multiple values and returns a single value. SQL provides many aggregate functions that include avg, count, sum, min, max, etc. An aggregate function ignores NULL values when it performs the calculation, except for the count function.

Aggregate Function

An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Various types of SQL aggregate functions are:

- Count()
- Sum()
- Avg()
- Min()
- Max()

Aggregate Function - Count

The COUNT() function returns the number of rows in a database table

```
COUNT(*)

COUNT( [ALL|DISTINCT] expression )
```

Aggregate Function - Count

In this example, we have the following tables involved:

- The "category" table contains category information, including the category_id and name.
- The "film_category" table serves as the junction table, connecting categories and films by storing the category_id and film_id as foreign keys.
- The "film" table contains film information, including the film_id.

The query uses the COUNT() function to count the number of films (film_id) in each category. By joining the "category," "film_category," and "film" tables, we establish the necessary relationships.

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

Aggregate Function - Count

The result set will include the category_id, category_name, and film_count columns, where film_count represents the count of films in each category. The GROUP BY clause is used to group the results by category_id and category_name.

#	category_ic	category_name	film_count
1	1	Action	64
2	2	Animation	66
3	3	Children	60
4	4	Classics	57
5	5	Comedy	58
6	6	Documentary	68

Aggregate Function - Sum

The SUM() function returns the total sum of a numeric column.

```
SUM()

SUM( [ALL|DISTINCT] expression )
```

Aggregate Function - Sum

Here's an example of using the SUM syntax in the Sakila database to calculate the total rental revenue for each customer:

- The "customer" table contains customer information, including the customer_id, first_name, and last_name.
- The "payment" table contains payment information, including the customer_id and amount.
- The query uses the SUM() function to calculate the total rental revenue (amount) for each customer. By joining the "customer" and "payment" tables based on the customer_id, we establish the necessary relationship.

```sql
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS total_revenue
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name;
```

Aggregate Function - Sum

The result set will include the customer_id, first_name, last_name, and total_revenue columns, where total_revenue represents the sum of payment amounts for each customer. The GROUP BY clause is used to group the results by customer_id, first_name, and last_name.

This query allows you to retrieve the total revenue generated from rentals for each customer in the Sakila database.

#	customer_id	first_name	last_name	total_revenue
1	1	MARY	SMITH	118.68
2	2	PATRICIA	JOHNSON	128.73
3	3	LINDA	WILLIAMS	135.74
4	4	BARBARA	JONES	81.78
5	5	ELIZABETH	BROWN	144.62
6	6	JENNIFER	DAVIS	93.72

Aggregate Function - Avg

The AVG() function calculates the average of a set of values.

```
AVG()

AVG( [ALL|DISTINCT] expression )
```

Aggregate Function - Avg

In this example, we only need the "film" table, which contains information about films, including the length of each film.

The query uses the AVG() function to calculate the average length of films in the "film" table.

```sql
SELECT AVG(length) AS average_length
FROM film;
```

Aggregate Function - Avg

The result set will include a single column, average_length, which represents the average length of all films in the database.

This query provides a simple and straightforward example of calculating the average length of films in the Sakila database.

#	average_lengt
1	115.2720

Aggregate Function - Min

The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.

```
MIN()

MIN( [ALL|DISTINCT] expression )
```

Aggregate Function - Min

In this example, we are working with the "film" table, which contains information about films, including the length of each film.

The query uses the MIN() function to retrieve the minimum length among all films in the database.

```
SELECT MIN(length) AS minimum_length
FROM film;
```

Aggregate Function - Min

The result set will include a single column, minimum_length, which represents the minimum length among all films.

This query provides a simple and straightforward example of retrieving the minimum length in the Sakila database.

#	minimum_length
1	46

Aggregate Function - Max

The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.

```
MAX(*)

MAX( [ALL|DISTINCT] expression )
```

Aggregate Function - Max

In this example, we are working with the "film" table, which contains information about films, including the length of each film.

The query uses the MAX() function to retrieve the maximum length among all films in the database.

```
SELECT MAX(length) AS maximum_length
FROM film;
```

Aggregate Function - Max

The result set will include a single column, minimum_length, which represents the minimum length among all films.

This query provides a simple and straightforward example of retrieving the minimum length in the Sakila database.

maximum_length
185

Aggregate Function with Group By and Having - Count

Still using sakila db as a sample data. In this case, we would like to create a feature about promotion based on table customer, and payment. For those customers who made a transaction more than 20 times, we would give some promotions to the next payment orders made. The question is, how we get list of customer with order payment more than 20?

Aggregate Function with Group By and Having - Count

Based on that situation, we can combine several clauses in MySQL to provide that data. Please take a look at this query:

```sql
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Count

We can combine **COUNT, GROUP BY, and HAVING** in order to provide the data.

```
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Count

First, put **COUNT** at the beginning on the select statement to show the field as the expected output. Customer first_name and last_name would be shown as additional information.

```
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Count

Second, since we need to know detail information about customers, we join the table from payment to customer table using **INNER JOIN** and **ON** with customer_id as a foreign key in

```
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Count

The **GROUP BY** clause in SQL is used to group rows with similar values together based on one or more columns. It is typically used in combination with aggregate functions to perform calculations on groups of rows rather than individual rows. In this case we combine with **COUNT** aggregate clause

```
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Count

At the end of the line, we uses **HAVING** clause in order to give a conditional result. Using comparison operator is part of **HAVING** clause in this case, we use more than 20 as a condition.

```
SELECT COUNT(payment.customer_id), customer.first_name, customer.last_name
FROM payment
INNER JOIN customer
ON customer.customer_id = payment.customer_id
GROUP BY payment.customer_id
HAVING COUNT(customer.customer_id) > 20;
```

Aggregate Function with Group By and Having - Sum

Still using sakila db as a sample data. Last time, we just created a feature about discount promotion based on table customer, and payment based on 20 transaction minimum who would get the promotions. In this case is a little bit different, what if the one who got the discount promotions is based on how much total payment that someone made?

Aggregate Function with Group By and Having - Sum

Please take a look at this query:

- The "customer" table contains customer information, including the customer_id, first_name, and last_name.
- The "payment" table contains payment information, including the amount and customer_id.

```sql
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(p.amount) > 200;
```

Aggregate Function with Group By and Having - Sum

The query uses the SUM() function to calculate the total payment made by each customer. By joining the "customer" and "payment" tables based on the customer_id, we establish the necessary relationship.

```
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(p.amount) > 200;
```

Aggregate Function with Group By and Having - Sum

The result set will include the customer_id, first_name, last_name, and total_payment columns, where total_payment represents the sum of payments made by each customer. The GROUP BY clause is used to group the results by customer_id, first_name, and last_name.

```
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(p.amount) > 200;
```

Aggregate Function with Group By and Having - Sum

The HAVING clause is then used to filter the results and only include customers who have made a total payment greater than 200. The SUM() function is used to calculate the sum of payments for each customer and compare it to the threshold value of 200.

This query allows you to find customers in the Sakila database who have made a total payment greater than 200.

```
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(p.amount) > 200;
```

Subquery

- A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.
- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Subquery Example Clause

```
SELECT column_name [, column_name ] FROM table1 [, table2 ]
WHERE column_name OPERATOR
(SELECT column_name [, column_name ] FROM table1 [, table2 ]
[WHERE])
```

```
SELECT film_id, title, rental_duration
FROM film
WHERE rental_duration > (
    SELECT AVG(rental_duration)
    FROM film
);
```

Subquery Example Clause

In this example, we have the "film" table involved.

The subquery is used to calculate the average rental duration of all films. It is nested within the main query.

The main query then retrieves the film_id, title, and rental_duration columns from the "film" table for films where the rental duration is greater than the average rental duration obtained from the subquery.

This query allows you to retrieve film information for films that have a rental duration greater than the average rental duration.

```sql
SELECT film_id, title, rental_duration
FROM film
WHERE rental_duration > (
    SELECT AVG(rental_duration)
    FROM film
);
```

SQL Transaction

A **transaction** is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

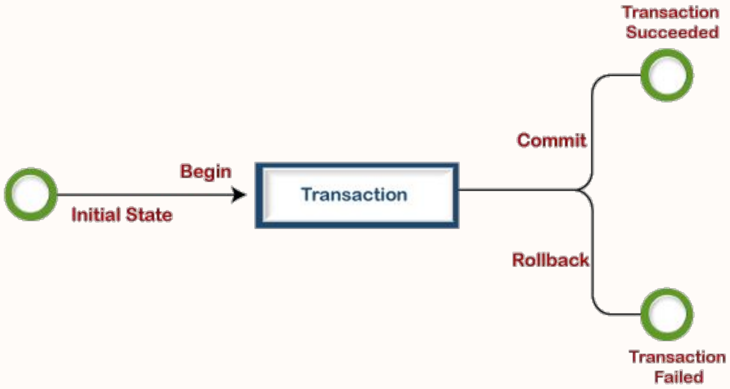
SQL Transaction - Control

COMMIT – to save the changes.

ROLLBACK – to roll back the changes.

SAVEPOINT – creates points within the groups of transactions in which to ROLLBACK.

SET TRANSACTION – Places a name on a transaction.



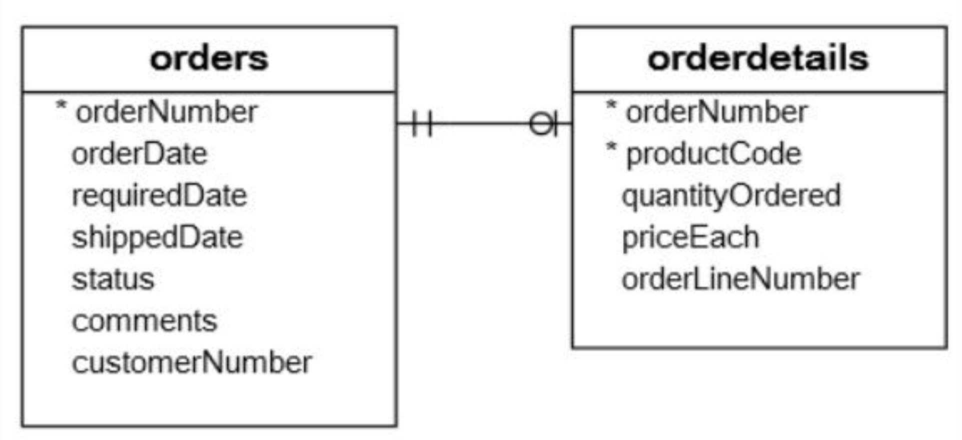
MySQL Transaction Statements

MySQL provides us with the following important statement to control transactions:

- To start a transaction, you use the START TRANSACTION statement. The BEGIN or BEGIN WORK are the aliases of the START TRANSACTION.
- To commit the current transaction and make its changes permanent, you use the COMMIT statement.
- To roll back the current transaction and cancel its changes, you use the ROLLBACK statement.
- To disable or enable the auto-commit mode for the current transaction, you use the SET autocommit statement.

MySQL Transaction Example

We will use the orders and orderDetails table for the demonstration



MySQL Transaction Commit Example

The following illustrates the step of creating a new sales order:

- First, start a transaction by using the START TRANSACTION statement.
- Next, select the latest sales order number from the orders table and use the next sales order number as the new sales order number.
- Then, insert a new sales order into the orders table.
- After that, insert sales order items into the orderdetails table.
- Finally, commit the transaction using the COMMIT statement.
- Optionally, you can select data from both orders and orderdetails tables to check the new sales order.

MySQL Transaction Commit Example

```sql
-- 1. start a new transaction
START TRANSACTION;
-- 2. Get the latest order number
SELECT @orderNumber:=MAX(orderNUmber)+1 FROM orders;
-- 3. insert a new order for customer 145
INSERT INTO orders(orderNumber,
                   orderDate,
                   requiredDate,
                   shippedDate,
                   status,
                   customerNumber)
VALUES(@orderNumber,'2005-05-31','2005-06-10','2005-06-11','In Process',145);
-- 4. Insert order line items
INSERT INTO orderdetails(orderNumber,
                         productCode,
                         quantityOrdered,
                         priceEach,
                         orderLineNumber)
VALUES(@orderNumber,'S18_1749', 30, '136', 1),
      (@orderNumber,'S18_2248', 50, '55.09', 2);
-- 5. commit changes
COMMIT;
```

MySQL Transaction Rollback Example

First, log in to the MySQL database server and delete data from the orders table. As you can see from the output, MySQL confirmed that all the rows from the orders table were deleted.

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> DELETE FROM orders;
Query OK, 327 rows affected (0.03 sec)
```

MySQL Transaction Rollback Example

Second, log in to the MySQL database server in a separate session and query data from the orders table. In this second session, we still can see the data from the orders table.

We have made the changes in the first session. However, the changes are not permanent. In the first session, we can either commit or roll back the changes.

For the demonstration purpose, we will roll back the changes in the first session.

```
mysql> SELECT COUNT(*) FROM orders;
+----------+
| COUNT(*) |
+----------+
|      327 |
+----------+
1 row in set (0.00 sec)
```

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.04 sec)
```

SQL Indexing

The **CREATE INDEX** statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

SQL Indexing - Syntax

Create Index Syntax,

Creates an index on a table. Duplicate values are allowed:

```sql
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

SQL Indexing - Syntax

Create Unique Index Syntax,

Creates an unique index on a table. Duplicate values are allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

SQL Indexing - Example

```sql
CREATE INDEX idx_lastname
ON Persons (LastName);

CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

Thank You!