Introduction

The merging and transforming of data through SQL operations like Joins, Unions, and Pivots is a critical part of data analysis and management. These techniques allow us to combine data from multiple sources and manipulate it to answer complex business questions. By understanding how to merge tables, handle non-matching rows, and pivot data, you can perform comprehensive data analysis on large datasets.

SQL CRUD Recap and Dataset for Today

We will be using the **Sakila database**, which simulates a DVD rental business. This database is provided by MySQL and contains realistic entities such as films, actors, customers, payments, and rentals. It is a great resource for practicing SQL queries, joins, aggregates, and transactions.

Working with Sakila

1. **Show Databases**:

   ```
   SHOW DATABASES;
   ```

2. **Switch to Sakila Database**:

   ```
   USE sakila;
   ```

3. **Show Tables**:

   ```
   SHOW TABLES;
   ```

4. **View Data**:

   ```
   SELECT * FROM actor;
   SELECT * FROM film;
   ```

CRUD Operations Practice

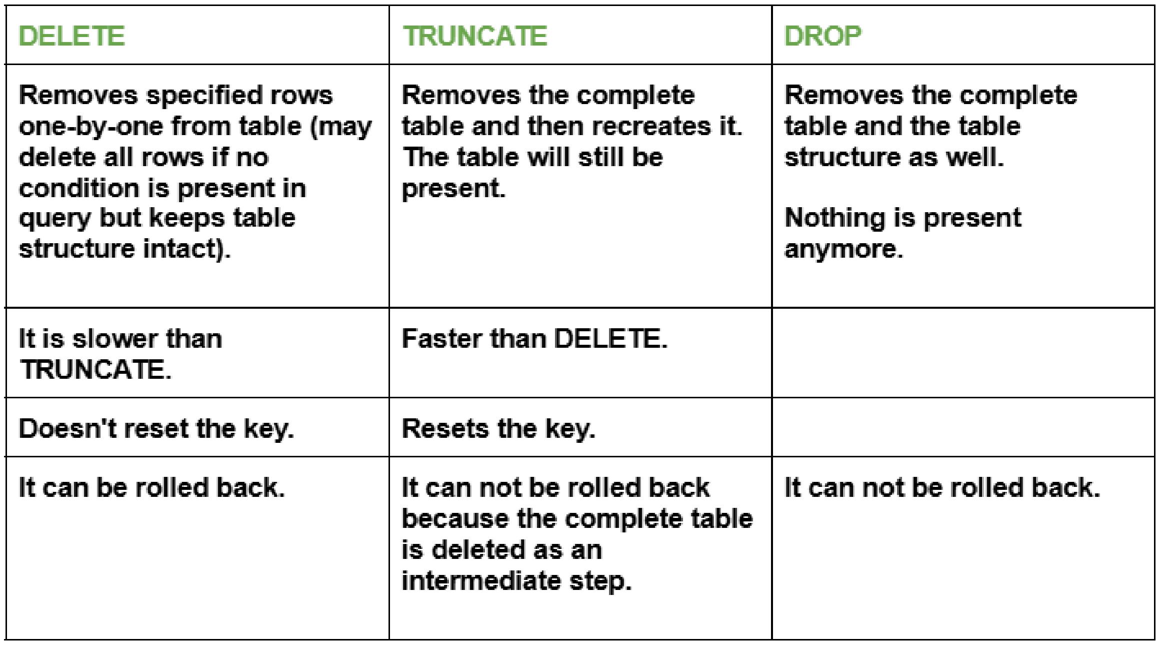
1. **Update Data**: Update the release year to '2006' and rating to 'PG-13' for the movie with ID 1.

   ```
   UPDATE film SET release_year = 2006, rating = 'PG-13' WHERE film_id = 1;
   ```

2. **Delete Data**: Delete the movie with ID 1.

```
DELETE FROM film WHERE film_id = 1;
```

.



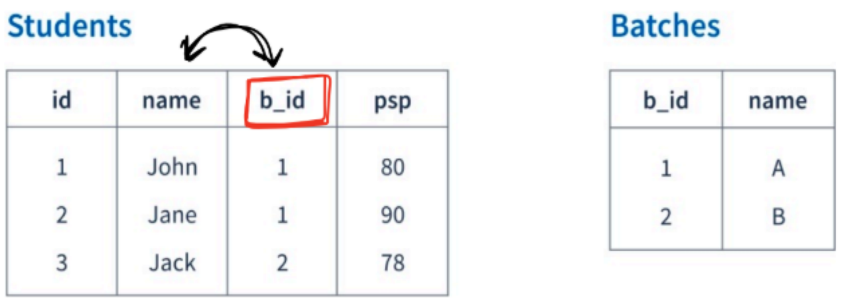
DELETE	TRUNCATE	DROP
Removes specified rows one-by-one from table (may delete all rows if no condition is present in query but keeps table structure intact).	Removes the complete table and then recreates it. The table will still be present.	Removes the complete table and the table structure as well. Nothing is present anymore.
It is slower than TRUNCATE.	Faster than DELETE.	
Doesn't reset the key.	Resets the key.	
It can be rolled back.	It can not be rolled back because the complete table is deleted as an intermediate step.	It can not be rolled back.

Note: In most MySQL setups (InnoDB), TRUNCATE behaves like DDL and cannot be rolled back.

Joins

Inner Join

An **Inner Join** combines rows from two or more tables based on a related column, returning only the rows that have matching values in both tables.



Students

id	name	b_id	psp
1	John	1	80
2	Jane	1	90
3	Jack	2	78

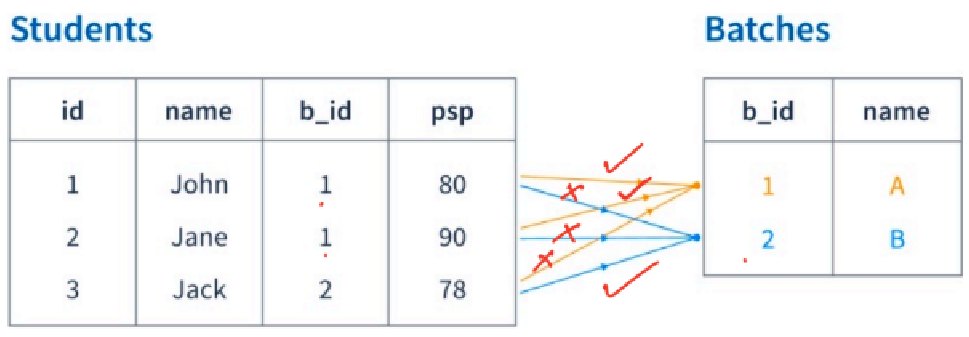
Batches

b_id	name
1	A
2	B

Example

Suppose you need to print the name of every student along with the name of their batch:

```
SELECT students.first_name, batches.batch_name
FROM students
JOIN batches
ON students.batch_id = batches.batch_id;
```

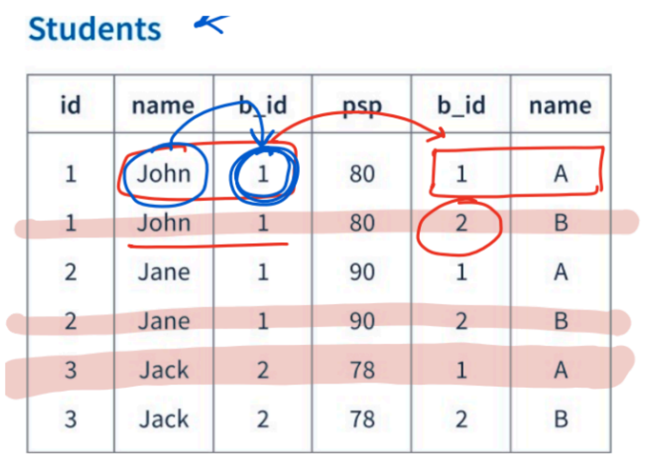


Students

id	name	b_id	psp
1	John	1	80
2	Jane	1	90
3	Jack	2	78

Batches

b_id	name
1	A
2	B



Students

id	name	b_id	psp	b_id	name
1	John	1	80	1	A
1	John	1	80	2	B
2	Jane	1	90	1	A
2	Jane	1	90	2	B
3	Jack	2	78	1	A
3	Jack	2	78	2	B

Here, the query joins the `students` table with the `batches` table, matching the `batch_id` column in both tables.

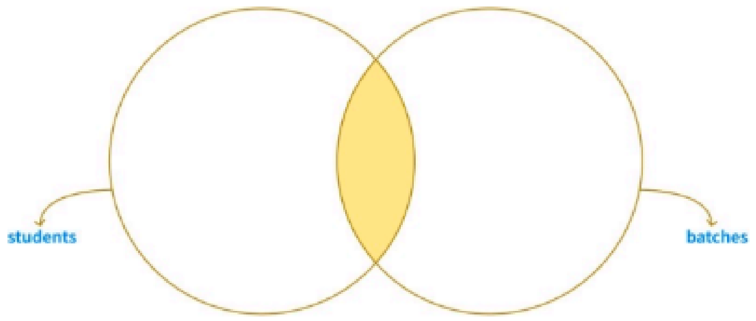
Sakila Example: Film and Language

To print each film's title and language, you join the `film` table with the `language` table:

```
SELECT film.title, language.name
FROM film
JOIN language
ON film.language_id = language.language_id;
```

Left Join

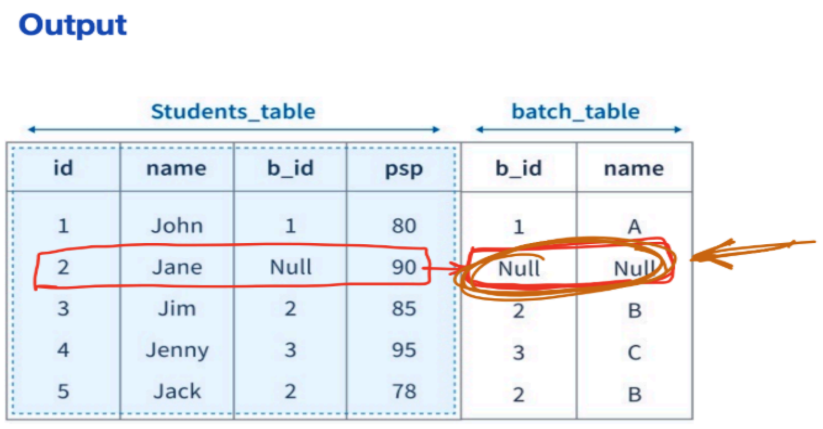
A **Left Join** returns all rows from the left table and the matching rows from the right table. If there is no match, the result is `NULL` on the right side.



Example

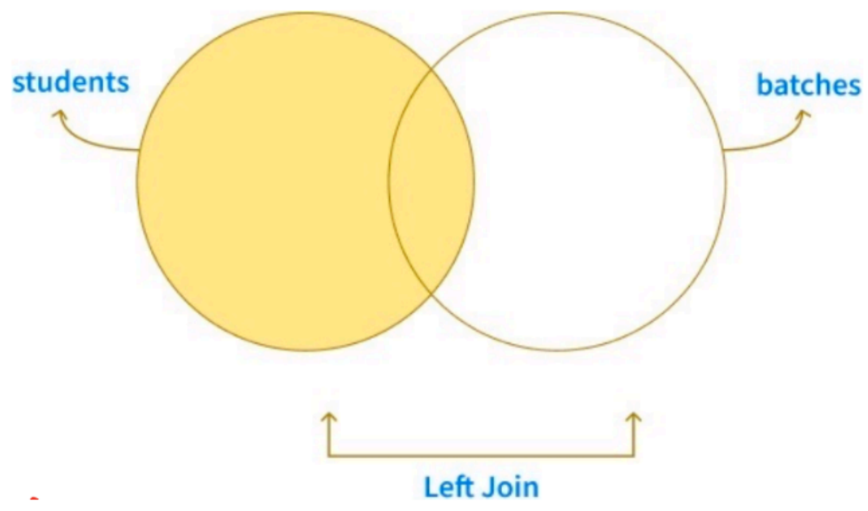
To get every student along with their batch, including students without a batch:

SELECT students.first_name, batches.batch_name
FROM students
LEFT JOIN batches
ON students.batch_id = batches.batch_id;



Output

Students_table				batch_table	
id	name	b_id	psp	b_id	name
1	John	1	80	1	A
2	Jane	Null	90	Null	Null
3	Jim	2	85	2	B
4	Jenny	3	95	3	C
5	Jack	2	78	2	B



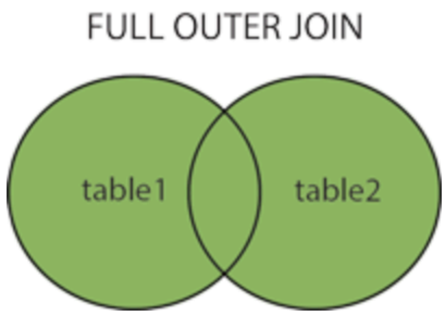
If a student does not have a batch, the `batch_name` will be `NULL`.

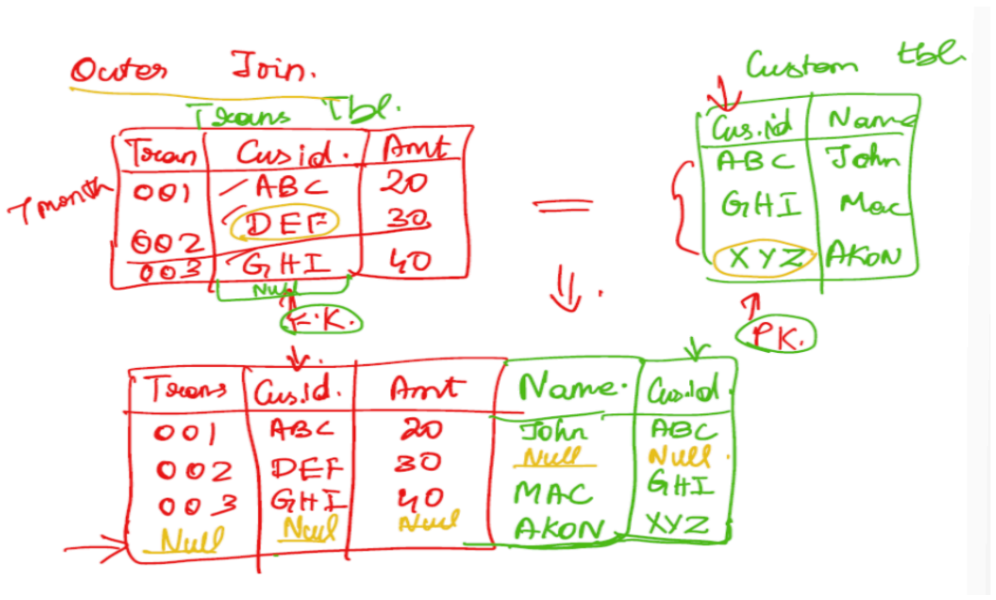
Mini Example Output:

first_name	batch_name
John	DataSci
Alice	WebDev
Bob	NULL

Full-Outer Join (Emulated in MySQL)

MySQL does not support **FULL OUTER JOIN** directly, but you can emulate it by combining `LEFT JOIN` and `RIGHT JOIN` with a `UNION`.





Example:

To find customers who have either made a purchase or signed up but never made a purchase:

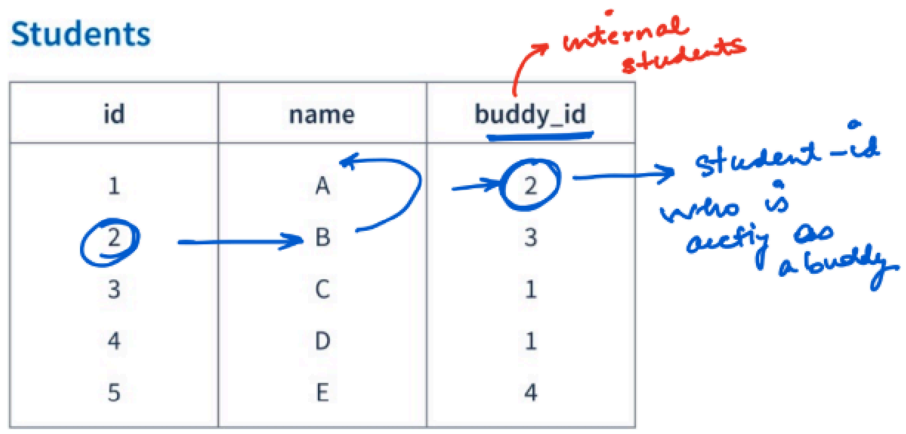
```
SELECT *
FROM customer c
LEFT JOIN purchase p
ON c.customer_id = p.customer_id
UNION
SELECT *
FROM customer c
RIGHT JOIN purchase p
ON c.customer_id = p.customer_id;
```

Decision Framework for Joins

- **Inner Join**: Use when you need only the rows where there is a match between the tables.
- **Left Join**: Use when you want to include all rows from the left table, even if there is no match in the right table.
- **Right Join**: Use when you want to include all rows from the right table, even if there is no match in the left table.
- **Full Join**: Use when you want to include all rows from both tables, even if there is no match.

Self-Join

A **Self-Join** is when a table is joined with itself. This is useful when you have a hierarchical relationship within the same table, such as an employee-manager relationship.



Students

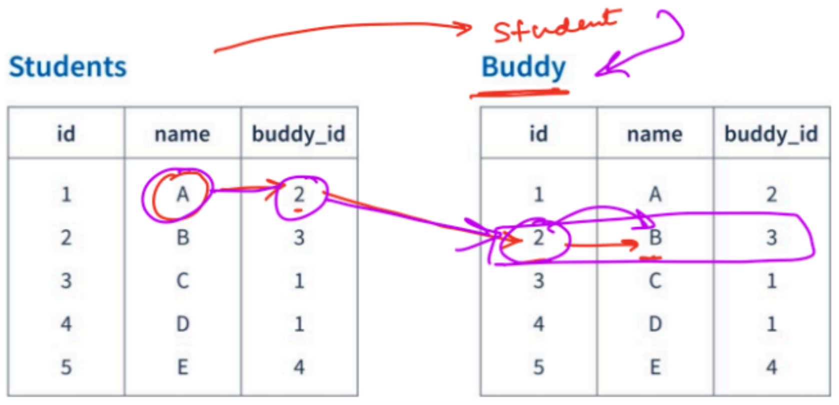
id	name	buddy_id
1	A	2
2	B	3
3	C	1
4	D	1
5	E	4

internal students
student-id who is acting as a buddy

Example: Buddy System in Students

For each student, print their name along with their buddy's name:

```
SELECT s1.name AS student_name, s2.name AS buddy_name
FROM students s1
JOIN students s2
ON s1.buddy_id = s2.student_id;
```



Students

id	name	buddy_id
1	A	2
2	B	3
3	C	1
4	D	1
5	E	4

Buddy

id	name	buddy_id
1	A	2
2	B	3
3	C	1
4	D	1
5	E	4

This query joins the `students` table with itself, using the `buddy_id` column to find the buddy of each student.

Example: Comparing Movies Released in the Same Year

To compare the ratings of films released within the same year:

```
SELECT DISTINCT
   f1.title AS Film1_Title,
   f1.rating AS Film1_Rating,
   f2.title AS Film2_Title,
   f2.rating AS Film2_Rating,
   f1.release_year AS Release_Year
FROM film f1
INNER JOIN film f2 ON
   f1.film_id < f2.film_id
   AND f1.release_year = f2.release_year
ORDER BY f1.release_year, Film1_Title, Film2_Title;
```

Union Join

UNION

The **UNION** operation combines the result sets of two SELECT queries into a single result set, ensuring no duplicates.

Example: List Active Staff and Customers with Last Name Starting with 'S':

```
SELECT first_name, last_name, 'STAFF' AS person_type
FROM staff
WHERE active = 1
UNION
SELECT first_name, last_name, 'CUSTOMER' AS person_type
FROM customer
WHERE active = 1 AND last_name LIKE 'S%'
ORDER BY last_name, first_name;
```

Handling Duplicates: `UNION` vs. `UNION ALL`

- **UNION** removes duplicate rows.
- **UNION ALL** keeps all rows, including duplicates.

Operator	**Purpose**	**Duplicate Handling**	**Performance**
UNION	Combines results and acts as an implicit DISTINCT.	Automatically **removes** duplicate rows, ensuring a unique list.	Slower, as the database must perform sorting and de-duplication.
UNION ALL	Simply appends one result set to another.	**Preserves** all rows, including exact duplicates.	Faster, as it avoids the de-duplication step.

Pivot Questions

Example: Rearranging Products Table

To rearrange the `Products` table so that each row contains `product_id`, `store`, and `price`, you can use `UNION`:

```
SELECT product_id, 'store1' AS store, store1 AS price
FROM Products
WHERE store1 IS NOT NULL
UNION
SELECT product_id, 'store2' AS store, store2 AS price
FROM Products
WHERE store2 IS NOT NULL
UNION
SELECT product_id, 'store3' AS store, store3 AS price
FROM Products
WHERE store3 IS NOT NULL
ORDER BY product_id, store;
```

Note: This is a manual pivot achieved using UNION. MySQL does not provide a native PIVOT operator.

Conclusion

Key Insights

- **Joins** allow you to combine data from multiple tables, with `INNER JOIN` returning only matching rows, and `LEFT JOIN` including unmatched rows from the left table.
- **Self-Join** enables comparisons or hierarchical relationships within the same table.
- **UNION** combines results from multiple queries, while **Full-Outer Join** (emulated in MySQL) combines all rows from both tables.

Key Terms

- **Inner Join**: Combines rows from two tables where there is a match.
- **Left Join**: Includes all rows from the left table, with matching rows from the right table.
- **Self-Join**: Joins a table with itself to find relationships within the same table.
- **Union**: Combines the result sets of multiple queries, removing duplicates.