SQL Level 2

Part 2

Grouping Data

Organizing Statistical Results In Categories

GROUP BY

GROUP BY combines rows into groups (multiple rows become one combined row).

1. Select the column you want to group by (dept_id), and the column you want to apply an aggregate function to (salary).
2. List the column you want to group (dept_id) in **GROUP BY**

emp_id	name	dept_id	salary
1001	Mary	2	40000
1002	John	1	49000
1003	Alice	1	51000
1004	Steve	2	62500
1005	John	2	55000
1006	Dan	3	45000
1007	Scott	3	87000

```
SELECT dept_id, AVG(salary)
FROM employees
GROUP BY dept_id;
```

dept_id	AVG(salary)
1	50000
2	52500
3	66000

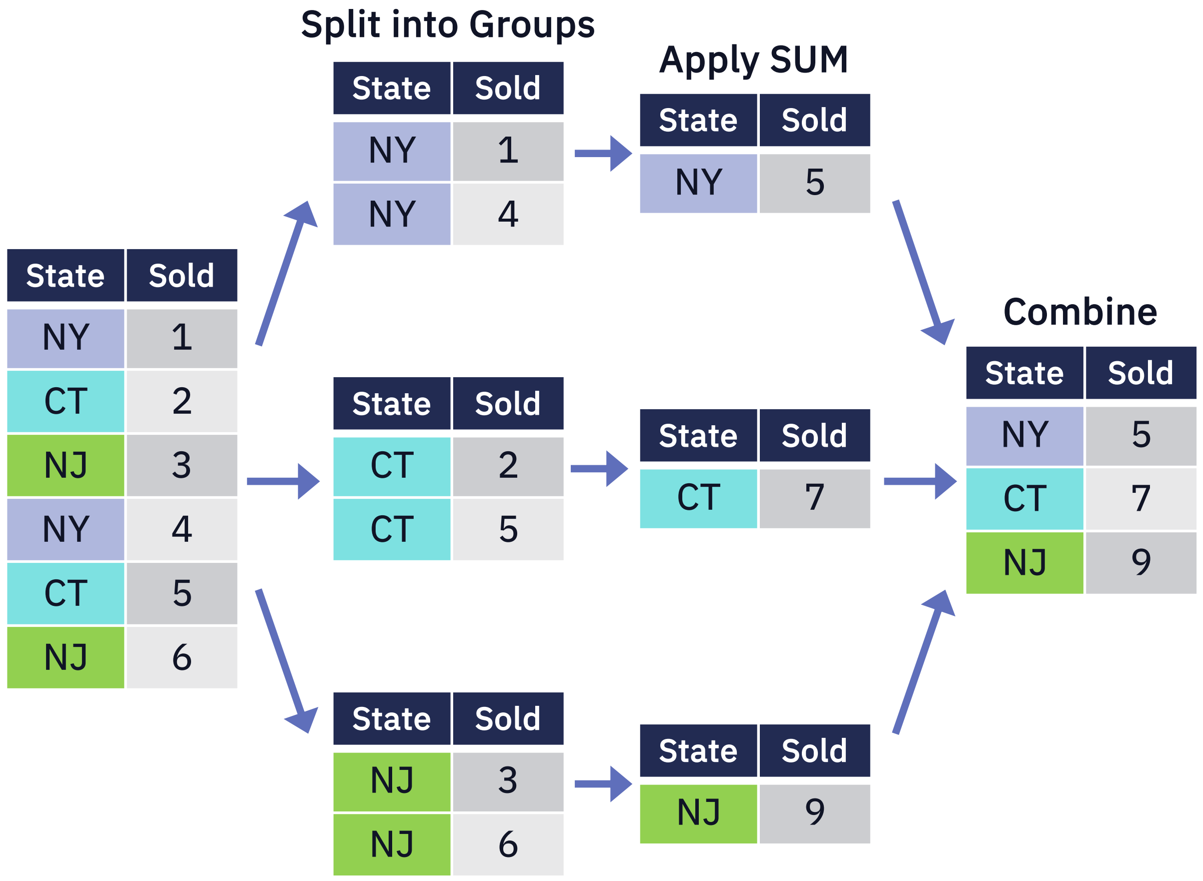
GROUP BY Syntax

```
SELECT key, SUM(DATA)
FROM table GROUP BY key;
```

Key is the column you want to group by.

2 Step Process

- Gather rows with same value in **GROUP BY** column
- Combine each collection (group) of rows with an **AGGREGATION** function



State	Sold
NY	1
CT	2
NJ	3
NY	4
CT	5
NJ	6

State	Sold
NY	1
NY	4

State	Sold
CT	2
CT	5

State	Sold
NJ	3
NJ	6

State	Sold
NY	5

State	Sold
CT	7

State	Sold
NJ	9

State	Sold
NY	5
CT	7
NJ	9

```
SELECT state, SUM(sold)
FROM sales GROUP BY state
```

GROUP BY Example

How many purchases has each user made?

GROUP BY Example: Step 1 of 2

```
SELECT user_id
FROM purchases
GROUP BY user_id;
```

Select the column you want to group by.

GROUP BY Example: Step 2 of 2

```
SELECT user_id, COUNT(*)
FROM purchases
GROUP BY user_id;
```

Combine the group of rows with an AGGREGATION function.

GROUP BY Another Example

What is the total quantity of each product purchased?

Example: Quantity Per Product

```
SELECT product_id, SUM(quantity)
FROM purchases
GROUP BY product_id;
```

GROUP BY & Aliases

In PostgreSQL you can refer to a column alias in GROUP BY.

In SQL Server you cannot refer to a column alias in GROUP BY.

Exercise

Open the file “2.0 GROUP BY.sql” (in SQL Level 2 folder)

Having

Filtering Grouped Data

HAVING

HAVING is used to further filter down results of a **GROUP BY**

emp_id	name	dept_id	salary
1001	Mary	2	40000
1002	John	1	49000
1003	Alice	1	51000
1004	Steve	2	62500
1005	John	2	55000
1006	Dan	3	45000
1007	Scott	3	87000

dept_id	AVG(salary)
1	50000
2	52500
3	66000

```sql
SELECT dept_id, AVG(salary)
FROM employees
GROUP BY dept_id;
```

dept_id	AVG(salary)
2	52500
3	66000

```sql
SELECT dept_id, AVG(salary)
FROM employees
GROUP BY dept_id
HAVING AVG(salary) > 51000;
```

HAVING

- **HAVING** uses the same operators as **WHERE** and has the same syntax.
- After the data has been grouped and aggregated, the conditions in the **HAVING** clause are applied.
- **HAVING** is evaluated before the **SELECT** clause, so you cannot use column aliases in **HAVING**. Because at the time of evaluating the **HAVING** clause, the column aliases specified in the **SELECT** clause are not available.

SQL Execution Order

SQL	Purpose
FROM	Get the base data from a table
JOIN	Obtain matching data from other table(s)
WHERE	Filter the base data
GROUP BY	Aggregate the base data (collect into groups)
HAVING	Filter the aggregated data
SELECT	Return the final data
DISTINCT	Discard duplicate data
ORDER BY	Sort the final data
LIMIT / TOP	Limit the returned data to a specific number of rows

HAVING with WHERE

The **WHERE** clause can be used with **HAVING** because:

- **WHERE** filter rows *before* any data is grouped
- **HAVING** filters *after* data is grouped

WHERE versus HAVING

SIMILARITIES

WHERE & HAVING are both used to exclude records from the result set.

DIFFERENCES

WHERE:

- **WHERE** is processed *before* the groups are created.
- Therefore, **WHERE** can refer to a value in the original tables.

HAVING:

- **HAVING** is processed *after* the groups are created.
- **HAVING** *cannot* refer to individual columns from a table that are not also part of the group.

SQL Execution Order

SQL	Purpose
FROM	Get the base data from a table
JOIN	Obtain matching data from other table(s)
WHERE	Filter the base data
GROUP BY	Aggregate the base data (collect into groups)
HAVING	Filter the aggregated data
SELECT	Return the final data
DISTINCT	Discard duplicate data
ORDER BY	Sort the final data
LIMIT / TOP	Limit the returned data to a specific number of rows

Exercise

Open the file “2.1 HAVING.sql” (in SQL Level 2 folder)

Primary & Foreign Keys

Identifying Rows & Connecting Tables

Primary Key

- The **primary key** uniquely identifies each row in a table.
- A primary key must be a unique value (the same value is never used on multiple rows).
- A primary key cannot have NULL values.
- A table can have only one primary key (which may consist of single or multiple fields). When multiple fields are used as a primary key, they are called a composite key.

primary key

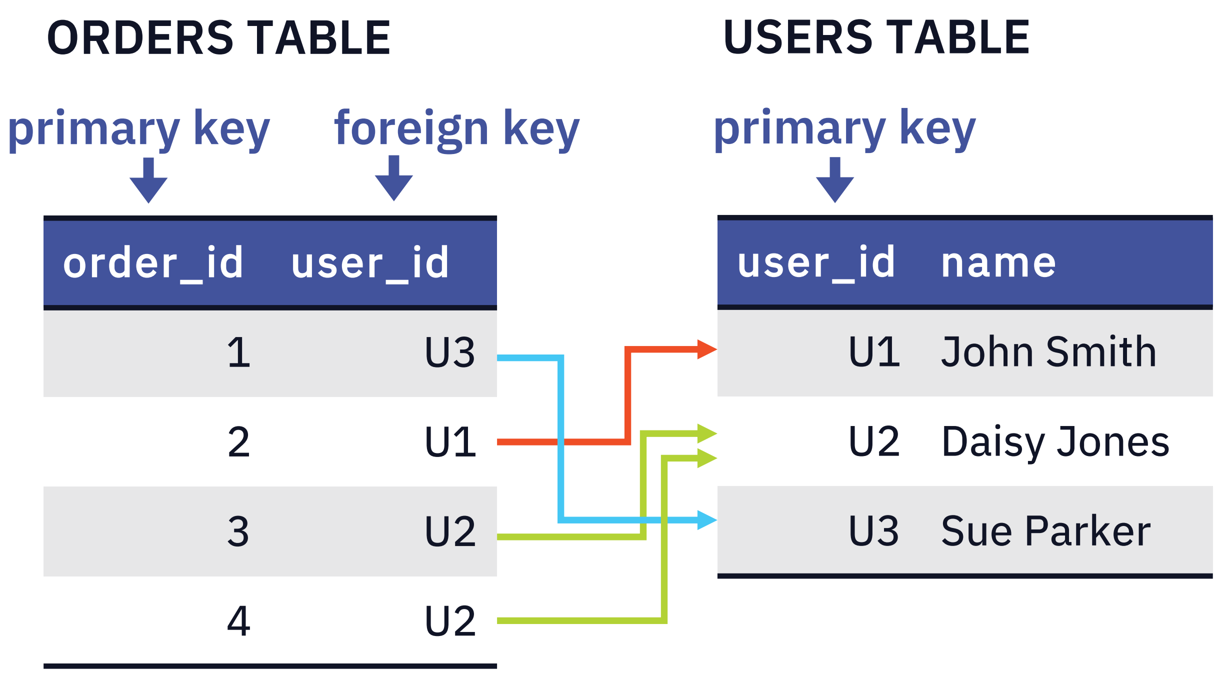
↓

user_id	name	email
1	John Smith	johnsmith@gmail.com
2	Daisy Jones	daisyjones@yahoo.com
3	Sue Parker	sueparker@outlook.com
4	John Smith	john-smith@yahoo.com

Foreign Key

A **foreign key** links 2 tables together (like a cross-reference).

A **foreign key** in one table, refers to a **primary key** in another table.



ORDERS TABLE

order_id	user_id
1	U3
2	U1
3	U2
4	U2

USERS TABLE

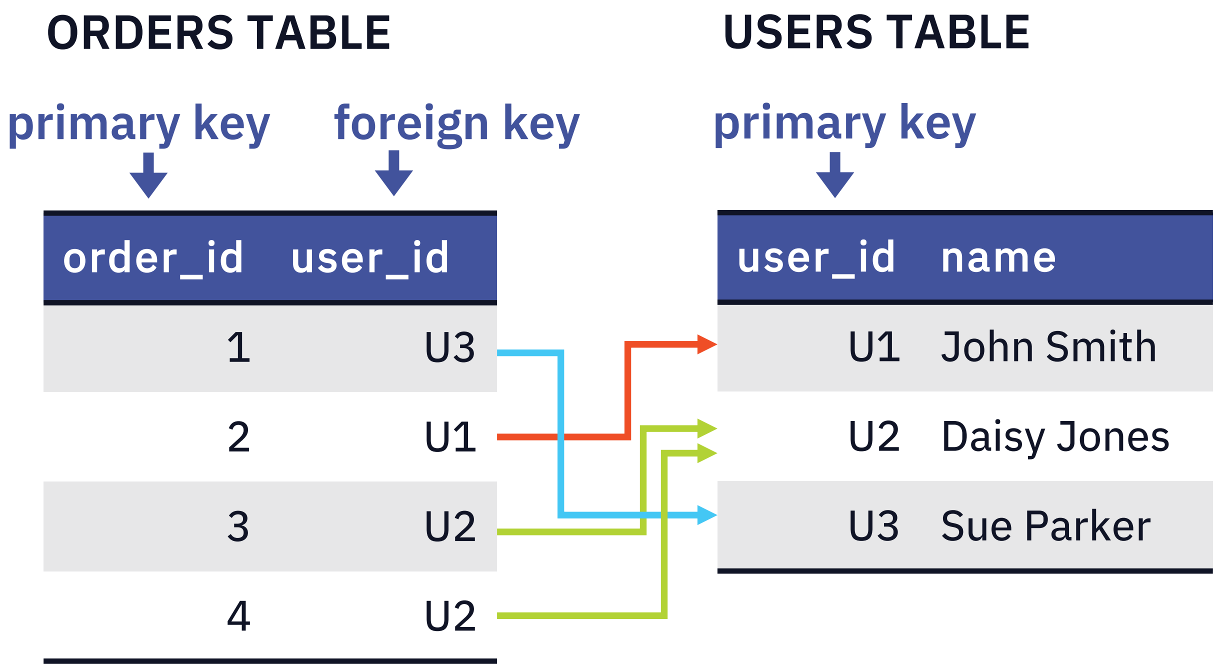
user_id	name
U1	John Smith
U2	Daisy Jones
U3	Sue Parker

Each order is made be a user.
That user's info is stored in a different table.

We connect the order to the user's info using foreign and primary keys.

Foreign Key

A **foreign key** value can be used multiple times in that column.



order_id	user_id
1	U3
2	U1
3	U2
4	U2

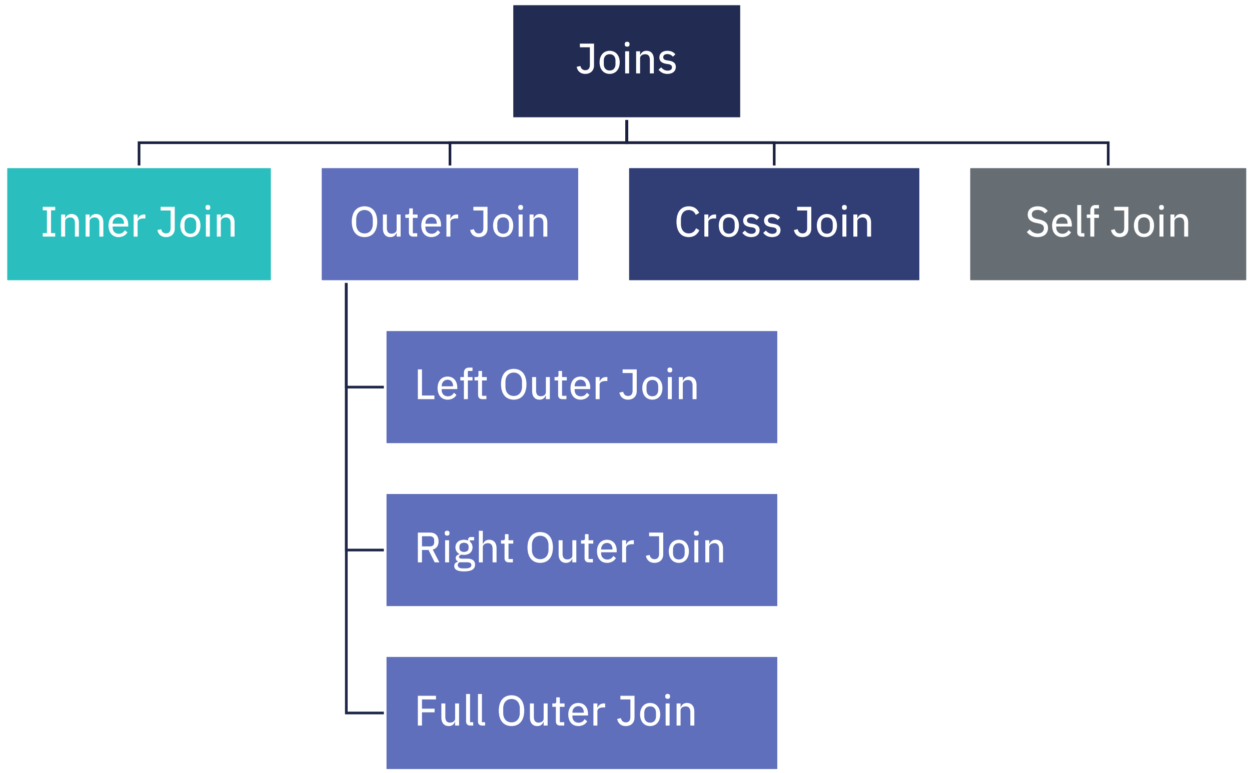
user_id	name
U1	John Smith
U2	Daisy Jones
U3	Sue Parker

Each order has a unique order_id (primary key), that is associated with a user_id (foreign key).

While each order is unique, the same user can place multiple orders. So the user_id may be repeated across different order_ids (because it refers to the same user many times).

Joins

- **INNER JOIN:** return matching rows (data matches in both tables).
- **OUTER JOIN:** returns matching and non-matching rows (missing values appear as NULL).
- **CROSS JOIN**: Returns a paired combination of each row of the 1st table with each row of the 2nd table.
  Less used, so we're not going to cover.
- **SELF JOIN**: Matches one part of a table with another part of the same table.



Self Joins

Joining Two Parts of a Single Table

Self Joins

- SELF JOINS let you join one part of a table to another part of the same table.
- They are useful when you want to find records that have values in common with other rows in the same table.
- In order to join a table to itself, you must use table aliases. Table aliases are created just like column aliases. By creating table aliases, SQL perceives the table being joined to itself as an additional separate table.

Unions

Combining Results Into a Single Table

UNION

UNION combines the result set of 2 or more SELECT statements, which results in new **rows**. It's different than a JOIN, which combines **columns** from different tables.

- Both queries must output the same number and order of columns.
- The data types of those columns must be compatible.

UNION removes all duplicates rows from the combined data set.

UNION ALL does NOT remove duplicates.

To sort rows in the result set, use ORDER BY in the second query.

Exercise

Open the file “2.2 Self Joins and Unions.sql” (in SQL Level 2 folder)