

# Database Normalization

---

When creating a database, it is really important to think about how data will be stored. This is known as normalization, and it is a huge part of most SQL classes. If you are in charge of setting up a new database, it is important to have a thorough understanding of database normalization.

There are essentially three ideas that are aimed at database normalization:

- 1. Are the tables storing logical groupings of the data?
- 2. Can I make changes in a single location, rather than in many tables for the same information?
- 3. Can I access and manipulate data quickly and efficiently?

This is discussed in detail [here](#).

However, most analysts are working with a database that was already set up with the necessary properties in place. As analysts of data, you don't really need to think too much about data normalization. You just need to be able to pull the data from the database, so you can start making insights.

---

## JOINS: -

---

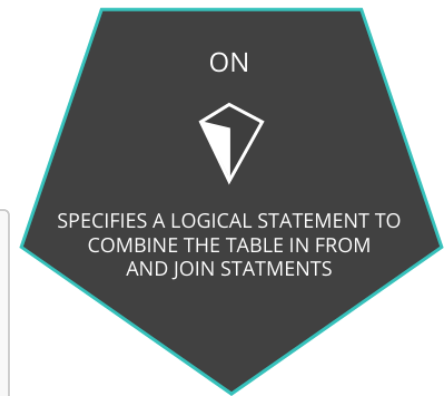


The whole purpose of JOIN statements is to allow us to pull data from more than one table at a time.

JOINS are useful for allowing us to pull data from multiple tables. This is both simple and powerful all at the same time.



With the addition of the JOIN statement to our toolkit, we will also be adding the ON statement.



## JOIN Examples:-

1. 

```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id =accounts.id;
```
2. 

```
SELECT orders.standard_qty, orders.gloss_qty,
       orders.poster_qty,  accounts.website,
       accounts.primary_poc
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```
3. 

```
SELECT *
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id;
```

---

## Primary and Foreign Keys

---

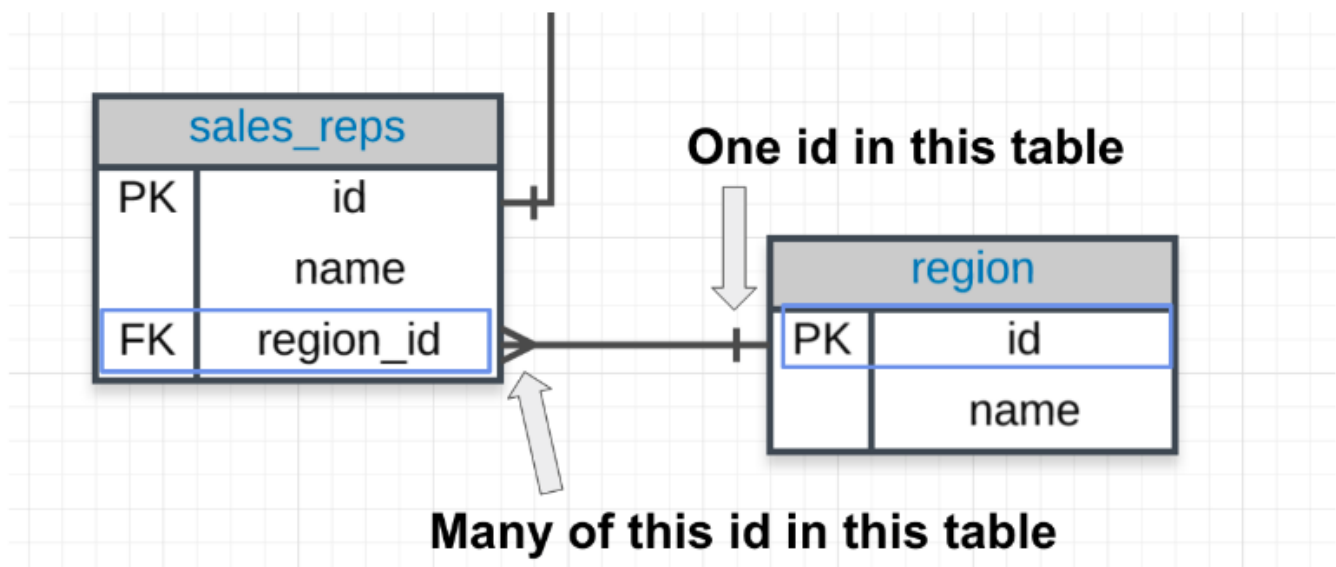
### Primary Key (PK):-

- A primary key is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called id, but that doesn't necessarily have to be the name. It is common that the primary key is the first column in our tables in most databases.

### Foreign Key (FK):-

- A foreign key is a column in one table that is a primary key in a different table. We can see in the Parch & Posey ERD that the foreign keys are:
  - 1. region\_id
  - 2. account\_id
  - 3. sales\_rep\_id

Each of these is linked to the primary key of another table. An example is shown in the image below:



### Primary - Foreign Key Link:-

- In the above image you can see that:
  - The region\_id is the foreign key.
  - The region\_id is linked to id - this is the primary-foreign key link that connects these two tables.
  - The crow's foot shows that the FK can actually appear in many rows in the sales\_reps table.
  - While the single line is telling us that the PK shows that id appears only once per row in this table.

---

## ALIAS

---

When we JOIN tables together, it is nice to give each table an alias. Frequently an alias is just the first letter of the table name. You actually saw something similar for column names in the Arithmetic Operators concept.

### ALIAS Example:-

```
FROM tablename AS t1
JOIN tablename2 AS t2
```

Before, you saw something like:

```
SELECT col1 + col2 AS total, col3
```

Frequently, you might also see these statements without the **AS** statement. Each of the above could be written in the following way instead, and they would still produce the **exact same results**:

```
FROM tablename t1
JOIN tablename2 t2
```

and

```
SELECT col1 + col2 total, col3
```

## Aliases for Columns in Resulting Table

While aliasing tables is the most common use case. It can also be used to alias the columns selected to have the resulting table reflect a more readable name.

Example:

```
Select t1.column1 aliasname, t2.column2 aliasname2
FROM tablename AS t1
JOIN tablename2 AS t2
```

The alias name fields will be what shows up in the returned table instead of t1.column1 and t2.column2

aliasname	aliasname2
example row	example row
example row	example row


```
1.  SELECT r.name region, s.name rep, a.name account
    FROM sales_reps s
    JOIN region r
    ON s.region_id = r.id
    JOIN accounts a
    ON a.sales_rep_id = s.id
    ORDER BY a.name;
```

```
2.  SELECT r.name region, a.name account,
      o.total_amt_usd/(o.total + 0.01) unit_price
    FROM region r
    JOIN sales_reps s
```

```
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id;
```

# INNER JOIN

Only returns rows that appear in both tables.




**INNER JOIN**

Only returns rows that appear in both tables.

ORDERS

id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539
19	1021	558
24	1031	1363

INNER JOIN

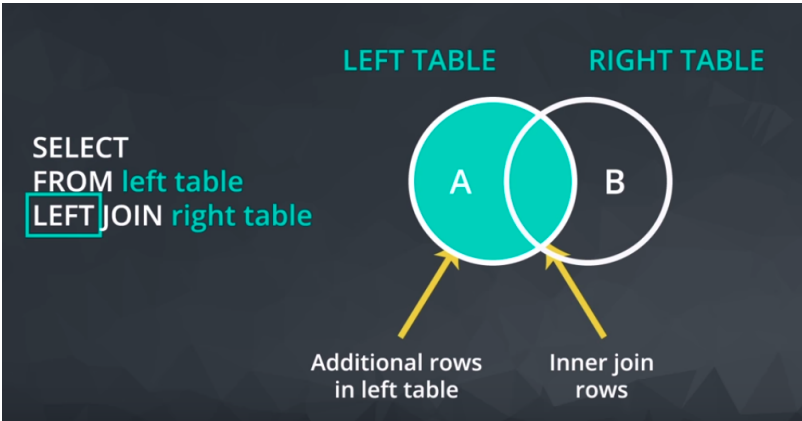


ACCOUNTS

id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple


```
SELECT a.id, a.name, o.total
FROM orders o
JOIN accounts a
ON o.account_id = a.id
```

Left & Right Tables:-



LEFT JOIN:-

ORDERS



ACCOUNTS

id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple
1031	Berkshire Hathaway
1041	McKesson
1051	UnitedHealth Group
1061	CVS Health

```
SELECT a.id, a.name, o.total
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
```

LEFT JOIN RESULT (SAME AS INNER JOIN)

id	account_id	total
1001	Walmart	169
1001	Walmart	288
1011	Exxon Mobil	541
1021	Apple	539

ORDERS

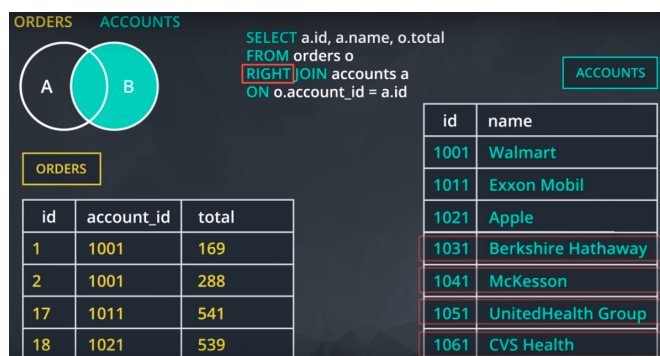
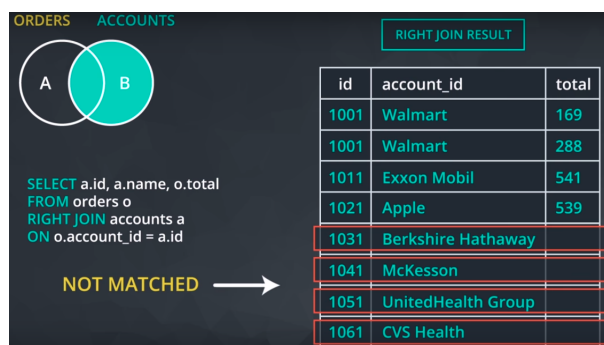
id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539

ACCOUNTS

id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple
1031	Berkshire Hathaway
1041	McKesson
1051	UnitedHealth Group
1061	CVS Health

```
SELECT a.id, a.name, o.total
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
```

## Right JOIN:-



If there is not matching information in the JOINED table, then you will have columns with empty cells. These empty cells introduce a new data type called NULL. You will learn about NULLs in detail in the next lesson, but for now you have a quick introduction as you can consider any cell without data as NULL.

## Other JOIN Notes:-

### INNER JOINS

Notice **every** JOIN we have done up to this point has been an **INNER JOIN**. That is, we have always pulled rows only if they exist as a match across two tables.

Our new **JOINS** allow us to pull rows that might only exist in one of the two tables. This will introduce a new data type called **NULL**. This data type will be discussed in detail in the next lesson.

### Quick Note

You might see the SQL syntax of

LEFT OUTER JOIN

OR

RIGHT OUTER JOIN

These are the exact same commands as the **LEFT JOIN** and **RIGHT JOIN** we learned about in the previous video.

## OUTER JOINS:

- The last type of join is a full outer join. This will return the inner join result set, as well as any unmatched rows from either of the two tables being joined.
- Again this returns rows that do not match one another from the two tables. The use cases for a full outer join are very rare.
- You can see examples of outer joins at the link [here](#) and a description of the rare use cases [here](#). We will not spend time on these given the few instances you might need to use them.

- Similar to the above, you might see the language **FULL OUTER JOIN**, which is the same as **OUTER JOIN**.

A simple rule to remember this is that, when the database executes this query, it executes the join and everything in the **ON** clause first. Think of this as building the new result set. That result set is then filtered using the **WHERE** clause.

The fact that this example is a left join is important. Because inner joins only return the rows for which the two tables match, moving this filter to the **ON** clause of an inner join will produce the same result as keeping it in the **WHERE** clause.

### Pro Tips:-

#### Pro Tip

LOGIC IN THE WHERE CLAUSE  
OCCURS AFTER THE  
JOIN OCCURS

#### Pro Tip

THESE EXTRA ROWS ARE  
BECAUSE THIS IS A LEFT JOIN.

#### Pro Tip

LOGIC IN THE ON CLAUSE  
REDUCES THE ROWS BEFORE  
COMBINING THE TABLES

### JOIN Examples:-

1. Provide a table that provides the region for each sales\_rep along with their associated accounts. This time only for the Midwest region. Your final table should include three columns: the region name, the sales rep name, and the account name. Sort the accounts alphabetically (A-Z) according to account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
WHERE r.name = 'Midwest'
ORDER BY a.name;
```

2. Provide a table that provides the region for each sales\_rep along with their associated accounts. This time only for accounts where the sales rep has a first name starting with S and in the Midwest region. Your final table should include three columns: the region name, the sales rep name, and the account name. Sort the accounts alphabetically (A-Z) according to account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
```

```
WHERE r.name = 'Midwest' AND s.name LIKE 'S%'
ORDER BY a.name;
```

3. Provide a table that provides the region for each sales\_rep along with their associated accounts. This time only for accounts where the sales rep has a last name starting with K and in the Midwest region. Your final table should include three columns: the region name, the sales rep name, and the account name. Sort the accounts alphabetically (A-Z) according to account name.

```
SELECT r.name region, s.name rep, a.name account
FROM sales_reps s
JOIN region r
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
WHERE r.name = 'Midwest' AND s.name LIKE '% K%'
ORDER BY a.name;
```

4. Provide the name for each region for every order, as well as the account name and the unit price they paid (total\_amt\_usd/total) for the order. However, you should only provide the results if the standard order quantity exceeds 100. Your final table should have 3 columns: region name, account name, and unit price.

```
SELECT r.name region, a.name account, o.total_amt_usd/(o.total + 0.01)
unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
WHERE o.standard_qty > 100;
```

5. Provide the name for each region for every order, as well as the account name and the unit price they paid (total\_amt\_usd/total) for the order. However, you should only provide the results if the standard order quantity exceeds 100 and the poster order quantity exceeds 50. Your final table should have 3 columns: region name, account name, and unit price. Sort for the smallest unit price first.

```
SELECT r.name region, a.name account, o.total_amt_usd/(o.total + 0.01)
unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.id
```



```
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
WHERE o.standard_qty > 100 AND o.poster_qty > 50
ORDER BY unit_price;
```

6. Provide the name for each region for every order, as well as the account name and the unit price they paid ( $\text{total\_amt\_usd}/\text{total}$ ) for the order. However, you should only provide the results if the standard order quantity exceeds 100 and the poster order quantity exceeds 50. Your final table should have 3 columns: region name, account name, and unit price. Sort for the largest unit price first.

```
SELECT r.name region, a.name account, o.total_amt_usd/(o.total + 0.01)
unit_price
FROM region r
JOIN sales_reps s
ON s.region_id = r.id
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
WHERE o.standard_qty > 100 AND o.poster_qty > 50
ORDER BY unit_price DESC;
```

7. What are the different channels used by account id 1001? Your final table should have only 2 columns: account name and the different channels. You can try SELECT DISTINCT to narrow down the results to only the unique values.

```
SELECT DISTINCT a.name, w.channel
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
WHERE a.id = '1001';
```

8. Find all the orders that occurred in 2015. Your final table should have 4 columns: occurred\_at, account name, order total, and order total\_amt\_usd.

```
SELECT o.occurred_at, a.name, o.total, o.total_amt_usd
FROM accounts a
JOIN orders o
ON o.account_id = a.id
WHERE o.occurred_at BETWEEN '01-01-2015' AND '01-01-2016'
ORDER BY o.occurred_at DESC;
```

There are a few more advanced JOINS that we did not cover here, and they are used in very specific use cases. **UNION** and **UNION ALL**, **CROSS JOIN**, and the tricky **SELF JOIN**. These are more advanced than this course will cover, but it is useful to be aware that they exist, as they are useful in special cases.