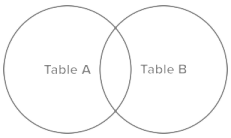
SQL Joins Explained

Basic SQL Join Types

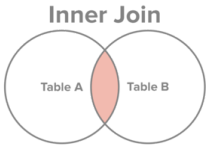
There are four basic types of SQL joins: inner, left, right, and full. The easiest and most intuitive way to explain the difference between these four types is by using a Venn diagram, which shows all possible logical relations between data sets.

Again, it's important to stress that before you can begin using any join type, you'll need to extract the data and load it into an RDBMS like Amazon Redshift, where you can query tables from multiple sources. You build that process manually, or you can use an ETL service like Stitch, which automates that process for you.

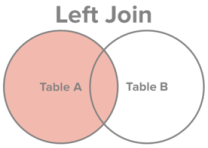
Let's say we have two sets of data in our relational database: table A and table B, with some sort of relation specified by primary and foreign keys. The result of joining these tables together can be visually represented by the following diagram:



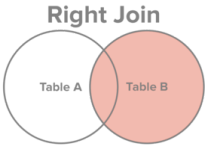
The extent of the overlap, if any, is determined by how many records in Table A match the records in Table B. Depending on what subset of data we would like to select from the two tables, the four join types can be visualized by highlighting the corresponding sections of the Venn diagram:



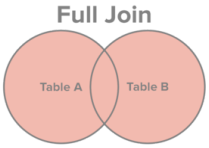
Select all records from Table A and Table B, where the join condition is met.



Select all records from Table A, along with records from Table B for which the join condition is met (if at all).



Select all records from Table B, along with records from Table A for which the join condition is met (if at all).



Select all records from Table A and Table B, regardless of whether the join condition is met or not.

Examples of SQL Join Types

Let's use the tables we introduced in the "What is a SQL join?" section to show examples of these joins in action. The relationship between the two tables is specified by the `customer_id` key, which is the "primary key" in customers table and a "foreign key" in the orders table:

customer_id	first_name	last_name	email	address	city	state	zipcode
1	George	Washington	gwashington@usa.gov	3200 Mt Vernon Hwy	Mount Vernon	VA	22121
2	John	Adams	jadams@usa.gov	1250 Hancock St	Quincy	MA	02169
3	Thomas	Jefferson	tjefferson@usa.gov	931 Thomas	Charlottesville	VA	22902

				Jefferson Pkwy			
4	James	Madison	jmadison@usa.gov	11350 Constitution Hwy	Orange	VA	22960
5	James	Monroe	jmonroe@usa.gov	2050 James Monroe Parkway	Charlottesville	VA	22902

order_id	order_date	amount	customer_id
1	07/04/1776	$234.56	1
2	03/14/1760	$78.50	3
3	05/23/1784	$124.00	2
4	09/03/1790	$65.50	3

Note that (1) not every customer in our customers table has placed an order and (2) there are a few orders for which no customer record exists in our customers table.

Inner Join

Let's say we wanted to get a list of those customers who placed an order and the details of the order they placed. This would be a perfect fit for an inner join, since an inner join returns records at the intersection of the two tables.

```
1  select first_name, last_name, order_date, order_amount
2  from customers c
3  inner join orders o
4  on c.customer_id = o.customer_id
```

inner-join.sql hosted with ♥ by GitHub view raw

first_name	last_name	order_date	order_amount
George	Washington	07/4/1776	$234.56
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	03/14/1760	$78.50
Thomas	Jefferson	09/03/1790	$65.50

Note that only George Washington, John Adams and Thomas Jefferson placed orders, with Thomas Jefferson placing two separate orders on 3/14/1760 and 9/03/1790.

Left Join

If we wanted to simply append information about orders to our customers table, regardless of whether a customer placed an order or not, we would use a left join. A left join returns all records from table A and any matching records from table B.

```sql
select first_name, last_name, order_date, order_amount
from customers c
left join orders o
on c.customer_id = o.customer_id
```

left-join.sql hosted with ♥ by GitHub view raw

first_name	last_name	order_date	order_amount
George	Washington	07/04/1776	$234.56
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	03/14/1760	$78.50
Thomas	Jefferson	09/03/1790	$65.50
James	Madison	NULL	NULL
James	Monroe	NULL	NULL

Note that since there were no matching records for James Madison and James Monroe in our orders table, the `order_date` and `order_amount` are `NULL`, which simply means there is no data for these fields.

So why would this be useful? By simply adding a "where order_date is NULL" line to our SQL query, it returns a list of all customers who have not placed an order:

```sql
select first_name, last_name, order_date, order_amount
from customers c
left join orders o
on c.customer_id = o.customer_id
where order_date is NULL
```

left-join-alt.sql hosted with ♥ by GitHub view raw

Right Join

Right join is a mirror version of the left join and allows to get a list of all orders, appended with customer information.

```
select first_name, last_name, order_date, order_amount
from customers c
right join orders o
on c.customer_id = o.customer_id
```

right-join.sql hosted with ♥ by GitHub view raw

first_name	last_name	order_date	order_amount
George	Washington	07/04/1776	$234.56
Thomas	Jefferson	03/14/1760	$78.50
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	09/03/1790	$65.50
NULL	NULL	07/21/1795	$25.50
NULL	NULL	11/27/1787	$14.40

Note that since there were no matching customer records for orders placed in 1795 and 1787, the `first_name` and `last_name` fields are `NULL` in the resulting set.

Also note that the order in which the tables are joined is important. We are right joining the orders table to the customers table. If we were to right join the customers table to the orders table, the result would be the same as left joining the orders table to the customers table.

Why is this useful? Simply adding a "where first_name is NULL" line to our SQL query returns a list of all orders for which we failed to record information about the customers who placed them:

```
select first_name, last_name, order_date, order_amount
from customers c
right join orders o
on c.customer_id = o.customer_id
where first_name is NULL
```

right-join-alt.sql hosted with ♥ by GitHub view raw

Full Join

Finally, for a list of all records from both tables, we can use a full join.

```
select first_name, last_name, order_date, order_amount
from customers c
full join orders o
on c.customer_id = o.customer_id
```

full-join.sql hosted with ♥ by GitHub view raw

first_name	last_name	order_date	order_amount
George	Washington	07/04/1776	$234.56
Thomas	Jefferson	03/14/1760	$78.50
John	Adams	05/23/1784	$124.00
Thomas	Jefferson	09/03/1790	$65.50
NULL	NULL	07/21/1795	$25.50
NULL	NULL	11/27/1787	$14.40
James	Madison	NULL	NULL
James	Monroe	NULL	NULL

What's next?

The four basic SQL joins described above let you tie the different pieces of data together, and allow you to start asking and answering more challenging questions about it. Yet often it is getting the data into your database or a data warehouse and keeping it up-to-date that is a challenging task. This is especially true if you have multiple sources of data living in completely different places. The next section explains how you can integrate all of your data in a single data warehouse, and ensure its quality and accuracy.

NEXT PAGE →