In **SQL**, selecting data across multiple tables is called a **JOIN**. There are many types of joins depending on how you want to combine rows. There are **INNER** and **OUTER** joins. You can also perform **LEFT**, **RIGHT**, **FULL**, and **CROSS** joins. But the easiest is the **NATURAL JOIN**, which is what we will start with here.

Use LEFT JOIN and GROUP BY to count the number of items won for each bidder.  
Select the surname and the count.  
Count the item\_id.  
Use bidders.bidder\_id for the grouping.  
Write the join to include bidders with zero items won.  
Note: What you choose as the LEFT table matters.

🡪SELECT surname, COUNT(item\_name)

FROM bidders

LEFT JOIN auction

on bidders.bidder\_id = auction.winner\_id

GROUP BY bidders.bidder\_id

1. **Countries + Geoloc**

Let's start by **joining** the **countries** and **geoloc** tables. The geoloc table has **latitude** and **longitude** values for the center of each country.

Enter the **NATURAL JOIN** query to see how it combines the data into a single table:

🡪SELECT \* FROM countries NATURAL JOIN geoloc

1. **Southern Hemisphere**

For each country, we now have all the columns from both tables. Awesome! This works because **NATURAL JOIN** is able to use the **ccd** column to match up the two tables.

Let's add a **WHERE** clause to select only the countries that are **south** of the **equator**. In other words, countries where **latitude < 0**.

🡪SELECT \* FROM countries NATURAL JOIN geoloc

WHERE latitude< 0;

1. **Eastern Countries**

How would you select all the countries **east** of the **prime meridian**?

🡪SELECT \* FROM countries NATURAL JOIN geoloc

WHERE longitude > 0;

1. **Coastal Countries**

Now let's join **countries** with the **coastlines** tables.

Write a query to select only the countries that have a coastline greater than 0.

🡪SELECT \* FROM countries NATURAL JOIN coastlines

WHERE coast\_km > 0;

1. **Different Column Names**

The **NATURAL JOIN** clause is easy to use, but it only works if the corresponding columns are **named the same** in both tables.

Take a look at the **shapes** and the **perimeters** tables. Notice that the IDs are called **id** in the shapes table, but are called **shape\_id** in the perimeters table.

Since the columns have different names, the natural join will not work the way we want. Try it out and **see what happens**:

🡪Since the key column names are different, the **NATURAL JOIN** didn't know how to match rows. As a result, it matched everything to everything, meaning it matched every column in the first table with every column in the second table creating a very large table. This result is called a **CROSS JOIN**and it's not what we want here.

1. **Basic Join**

We want each shape to line up with its corresponding perimeter. To do this, we need to use a basic **JOIN** clause and explicitly state which columns to equate using the **ON** keyword

🡪SELECT \* FROM shapes JOIN perimeters

ON id=shape\_id;

1. **Large Perimeters**

Write a query to select the shapes that have a **perimeter greater than 400**.

🡪SELECT \* FROM shapes JOIN perimeters

ON id=shape\_id

WHERE perimeter > 400;