When you're working as a data analyst, you will often need to use join queries to combine *three or more* tables. To do this, you use the same syntax as with two tables, but with more **JOIN**s added at the end of the **FROM** clause. Each **JOIN** should have its own **ON** keyword and join condition.

For example, to join the customers, orders, and employees tables, you could use this query:

```
SELECT c.name AS customer_name,
    o.total AS order_total,
    e.first_name AS employee_name
  FROM customers c
    JOIN orders o ON c.cust_id = o.cust_id
    JOIN employees e ON o.empl_id = e.empl_id;
```

Notice how the final two lines of this query have the same structure: the **JOIN** keyword, a table reference, a table alias, the **ON** keyword, and a join condition. You can add arbitrarily many lines like this to the **FROM** clause, to join together arbitrarily many tables.

The result the above query is:

customer_name	order_total	employee_name
Arfa	28.54	Sabahattin
Brendon	48.69	Virginia
Brendon	-16.39	Virginia
Chiyo	24.78	Ambrosio

The arrangement of the rows in the result is arbitrary. Each result row represents an order, and gives the name of the customer who placed the order, the total amount of the order, and the employee who recorded the order. Because this is an inner join (the default type), all non-matching rows are excluded from the result. You can specify other types of joins, in any combination. For example, to include the order placed by the customer who is not in the customers table, change the first **JOIN** to **RIGHT OUTER JOIN**.

The sequence of the tables in a multi-table join does not matter, except with left and right outer joins, where it affects which table's non-matching rows are included. Each join condition can refer to join key columns in any of the tables mentioned earlier in the **FROM** clause.

Join queries are computationally expensive and can be slow, especially when you're joining three or more tables, or if you're working with very large data. One strategy that can be used to remedy this problem is to join the data in advance and store the pre-joined result in a separate table, which you can then query. (If you completed the first course in this specialization, you might recall this is one way of *denormalizing* a normalized database, to make it easier to run analytic queries.) This approach of pre-joining tables has some costs, but it can make analytic queries easier to write and faster to run. A later course in this specialization will go into more details about how you can do that.