One-to-Many and Many-to-Many Joins ¶

Introduction

Previously, you learned about the typical case where one joins on a primary or foreign key. In this section, you'll explore other types of joins using one-to-many and many-to-many relationships!

Objectives

You will be able to:

- Explain one-to-many and many-to-many joins as well as implications for the size of query results
- Query data using one-to-many and many-to-many joins

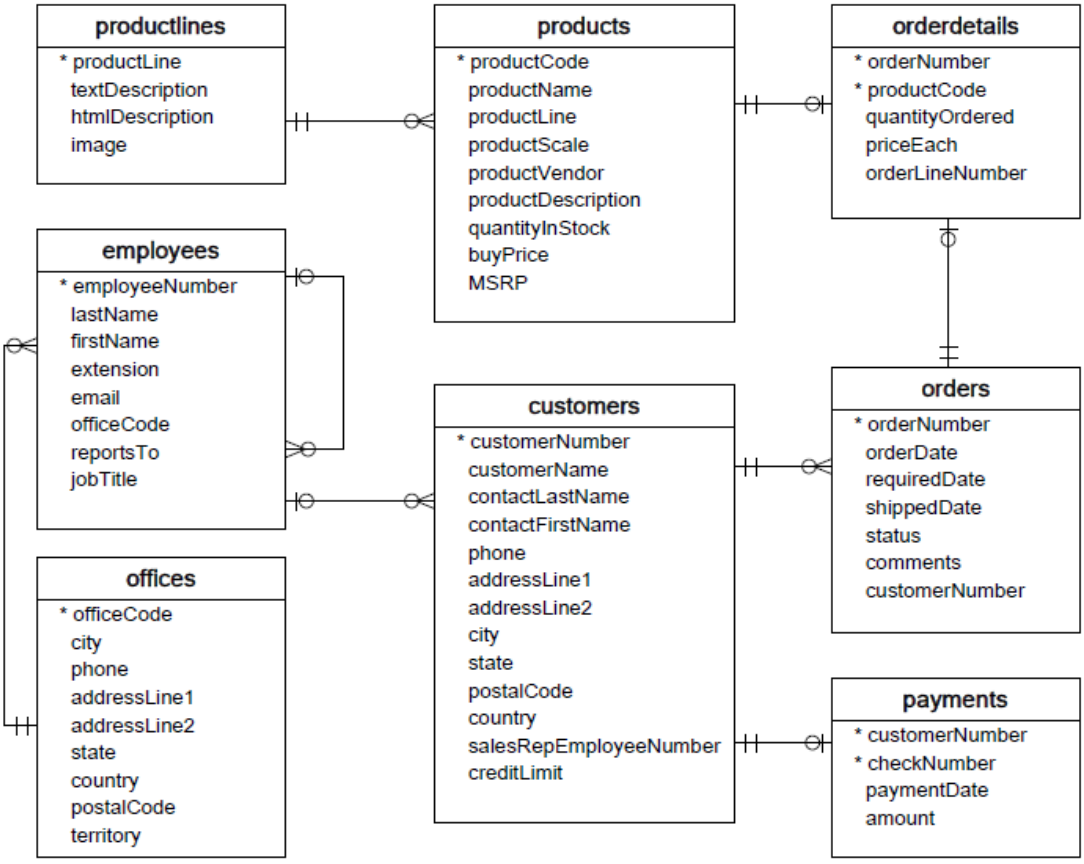
One-to-Many and Many-to-Many Relationships

So far, you've seen a couple of different kinds of join statements: `LEFT JOIN` and `INNER JOIN` (aka, `JOIN`). Both of these refer to the way in which you would like to define your join based on the tables and their shared information. Another perspective on this is the number of matches between the tables based on your defined links with the keywords `ON` or `USING`.

You have also seen the typical case where one joins on a primary or foreign key. For example, when you join on `customerID` or `employeeID`, this value should be unique to that table. As such, your joins have been very similar to using a dictionary to find additional information associated with that record. In cases where there are multiple entries in either table for the field (column) you are joining on, you will similarly be given multiple rows in your resulting view, one for each of these entries.

Restaurants Case Study

We'll start with this familiar ERD:



For example, let's say you have another table `restaurants` that has many columns including `name`, `city`, and `rating`. If you were to join this `restaurants` table with the offices table using the shared city column, you might get some unexpected behavior. That is, in the office table, there is only one office per city. However, because there will likely be more than one restaurant for each of these cities in your second table, you will get unique combinations of offices and restaurants from your join. If there are 513 restaurants for Boston in your restaurant table and 1 office for Boston, your joined table will have each of these 513 rows, one for each restaurant along with the one office.

If you had 2 offices for Boston and 513 restaurants, your join would have 1026 rows for Boston; 513 for each restaurant along with the first office and 513 for each restaurant with the second office. Three offices in Boston would similarly produce 1539 rows; one for each unique combination of restaurants and offices. This is where you should be particularly careful of many to many joins as the resulting set size can explode drastically, potentially consuming vast amounts of memory and other resources.

Connecting to the Database

In [1]:
```
import sqlite3
import pandas as pd
```

In [2]:
```
conn = sqlite3.connect('data.sqlite')
```

One-to-One Relationships

Sometimes, a `JOIN` does not increase the number of records at all. For example, in our database, each employee is only associated with one office. So if our original query included information about all employees with a `jobTitle` of "Sales Rep", then our joined query also added the city location of their offices, we would get the same number of results both times.

Sales Rep Employees

In [3]:
```
q = """
SELECT firstName, lastName, email
FROM employees
WHERE jobTitle = 'Sales Rep'
;
"""
df = pd.read_sql(q, conn)
print("Number of results:", len(df))
```

```
Number of results: 17
```

In [4]:
```
# Displaying only 5 for readability
df.head()
```

Out[4]:

	firstName	lastName	email
0	Leslie	Jennings	ljennings@classicmodelcars.com
1	Leslie	Thompson	lthompson@classicmodelcars.com
2	Julie	Firrelli	jfirrelli@classicmodelcars.com
3	Steve	Patterson	spatterson@classicmodelcars.com
4	Foon Yue	Tseng	ftseng@classicmodelcars.com

Cities for Sales Rep Employees

Now we'll join with the `offices` table in order to display the city name as well.

In [5]:
```
q = """
SELECT firstName, lastName, email, city
FROM employees
JOIN offices
    USING(officeCode)
WHERE jobTitle = 'Sales Rep'
;
"""
df = pd.read_sql(q, conn)
print("Number of results:", len(df))
```

```
Number of results: 17
```

In [6]:
```
# Displaying only 5 for readability
df.head()
```

Out[6]:

	firstName	lastName	email	city
0	Leslie	Jennings	ljennings@classicmodelcars.com	San Francisco
1	Leslie	Thompson	lthompson@classicmodelcars.com	San Francisco
2	Julie	Firrelli	jfirrelli@classicmodelcars.com	Boston
3	Steve	Patterson	spatterson@classicmodelcars.com	Boston
4	Foon Yue	Tseng	ftseng@classicmodelcars.com	NYC

As you can see, we got the same number of results as the original query, we just have more data now.

One-to-Many Relationships

When there is a one-to-many relationship, the number of records will increase to match the number of records in the larger table.

Product Lines

Let's start with selecting the product line name and text description for all product lines.

In [7]:
```
q = """
SELECT productLine, textDescription
FROM productlines
;
"""
df = pd.read_sql(q, conn)
print("Number of results:", len(df))
```

```
Number of results: 7
```

In [8]:
```
df
```

Out[8]:

	productLine	textDescription
0	Classic Cars	Attention car enthusiasts: Make your wildest c...
1	Motorcycles	Our motorcycles are state of the art replicas ...
2	Planes	Unique, diecast airplane and helicopter replic...
3	Ships	The perfect holiday or anniversary gift for ex...
4	Trains	Model trains are a rewarding hobby for enthusi...
5	Trucks and Buses	The Truck and Bus models are realistic replica...
6	Vintage Cars	Our Vintage Car models realistically portray a...

Joining with Products

Now let's join that table with the products table, and select the vendor and product description.

In [9]:
```
q = """
SELECT productLine, textDescription, productVendor, productDescription
FROM productlines
JOIN products
    USING(productLine)
;
"""
df = pd.read_sql(q, conn)
print("Number of results:", len(df))
```

```
Number of results: 110
```

In [10]:
```
# Displaying only 5 for readability
df.head()
```

Out[10]:

	productLine	textDescription	productVendor	productDescription
0	Classic Cars	Attention car enthusiasts: Make your wildest c...	Autoart Studio Design	Hood, doors and trunk all open to reveal highl...
1	Classic Cars	Attention car enthusiasts: Make your wildest c...	Carousel DieCast Legends	Features include opening and closing doors. Co...
2	Classic Cars	Attention car enthusiasts: Make your wildest c...	Carousel DieCast Legends	The operating parts of this 1958 Chevy Corvett...
3	Classic Cars	Attention car enthusiasts: Make your wildest c...	Carousel DieCast Legends	This diecast model of the 1966 Shelby Cobra 42...
4	Classic Cars	Attention car enthusiasts: Make your wildest c...	Classic Metal Creations	1957 die cast Corvette Convertible in Roman Re...

As you can see, the number of records has increased significantly, because the same product line is now appearing multiple times in the results, once for each actual product.

Many-to-Many Relationships

A many-to-many join is as it sounds; there are multiple entries for the shared field in both tables. While somewhat contrived, we can see this through the example below, joining the offices and customers table based on the state field. For example, there are 2 offices in MA and 9 customers in MA. Joining the two tables by state will result in 18 rows associated with MA; one for each customer combined with the first office, and then another for each customer combined with the second option. This is not a particularly useful join without applying some additional aggregations or pivots, but can also demonstrate how a poorly written query can go wrong. For example, if there are a large number of occurrences in both tables, such as tens of thousands, then a many-to-many join could result in billions of resulting rows. Poorly conceived joins can cause a severe load to be put on the database, causing slow execution time and potentially even tying up database resources for other analysts who may be using the system.

Just Offices

In [11]:
```
q = """
SELECT *
FROM offices
;
"""

df = pd.read_sql(q, conn)
print('Number of results:', len(df))
```

```
Number of results: 8
```

Just Customers

In [12]:
```
q = """
SELECT *
FROM customers
;
"""

df = pd.read_sql(q, conn)
print('Number of results:', len(df))
```

```
Number of results: 122
```

Joined Offices and Customers

In [13]:
```
q = """
SELECT *
FROM offices
JOIN customers
    USING(state)
;
"""

df = pd.read_sql(q, conn)
print('Number of results:', len(df))
```

```
Number of results: 254
```

In [14]:
```
# Displaying only 5 for readability
df.head()
```

Out[14]:

	officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
0	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
1	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
3	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
4	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA

5 rows × 21 columns

Whenever you write a SQL query, make sure you understand the unit of analysis you are trying to use. Getting more data from the database is not always better! The above query might make sense as a starting point for something like "what is the ratio of customers to offices in each state", but it's not there yet. Many-to-many joins can be useful, but it's important to be strategic and understand what you're really asking for.

Summary

In this section, you expanded your join knowledge to one-to-many and many-to-many joins!