7.3.1 Query Dates

Now Bobby is ready for analysis. He has created tables based on his entity relationship diagram, and he's successfully imported the data to each table. That's a pretty big chunk of work he's completed. Not to mention the analysis—that's a big step, too.

The next part is to perform the analysis he was tasked with: future-proofing the company by determining how many people will be retiring and, of those employees, who is eligible for a retirement package.

Bobby will need to dive into the data and perform queries to learn when employees were hired as well as their age. To do this, he will need to create statements that use conditionals.

In Python, conditionals such as "if" and "else," and the logical operator "and," are similar to conditional expressions used in SQL. We're querying the database, and the returned results are based on the conditions we have set. For example, one of Bobby's tasks is to search for folks who are retiring soon. The query he builds will include a condition involving employee birthdays. We need to know when they were born to determine when they'll retire, right?

Determine Retirement Eligibility

Bobby's boss has determined that anyone born between 1952 and 1955 will begin to retire. The first query we need to help Bobby write will return a list of those employees. At the bottom of your query editor, type the following code:

```
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1955-12-31';
```

At a glance, we can tell we're searching for the first and last names of employees born between January 1, 1952, and December 31, 1955. Let's break the code down into its components, though, so we can really understand what it's doing:

- The `SELECT` statement is more specific this time. Instead of an asterisk to indicate that we want all of the records, we're requesting only the first and last names of the employees.
- `FROM employees` tells SQL in which of the six tables to look.
- The `WHERE` clause brings up even more specifics. We want SQL to look in the birth_date column for anyone born between January 1, 1952, and December 31, 1955.

Notice how `BETWEEN` and `AND` are both capitalized in our statement? This is part of the SQL syntax. It not only signals the conditions present, but also makes the code easier to read.

Pewlett Hackard has a lot of employees getting ready to age out of the program. This is going to create a considerable amount of openings. Refine this list further by looking only at how many employees were born in 1952. Create another query that will search for only 1952 birth dates.

```
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1952-12-31';
```

This query is almost the same as the last. We've only changed a single digit: the year was switched from 1955 to 1952 after the `AND` clause.

SKILL DRILL

Create three more queries to search for employees who were born in 1953, 1954, and 1955.

Narrow the Search for Retirement Eligibility

There are quite a few folks getting ready to retire. Each of those new queries has a lengthy list of people. Let's see if we can narrow it down a bit more by adding another condition to the query. We'll start with our original query, shown below.

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1955-12-31';
```

We'll modify this query to include a specific hiring range. This time, we're looking for employees born between 1952 and 1955, who were also hired between 1985 and 1988. The modification is subtle, too. We're going to adjust one line of the code block and add another to the end.

The first piece, an adjustment, is to place parentheses around the `WHERE` clause (without including the keyword itself). We'll also remove the semicolon since the code block isn't complete yet. The updated code should look as follows:

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
```

Next, we'll add the final line of code. Our current code has a single condition in place that tells Postgres to search only for people born between 1952 and 1955. The next line of code is our second condition that they were also hired between 1985 and 1988.

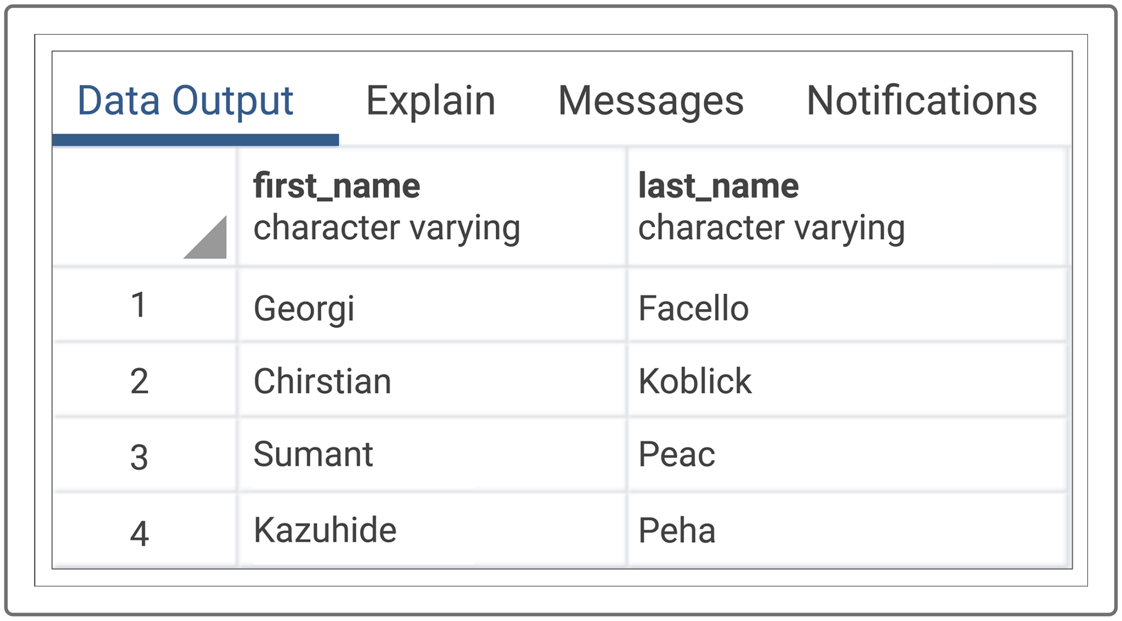
```
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Notice how the second condition is inside parentheses? This is a tuple; in Python, data can be stored inside a tuple and accessed in the same way as a list. In SQL, the tuples in this block of code are part of the syntax. They basically place each condition in a group, and Postgres looks for the first group first, then looks inside the second group to deliver the second condition. Altogether, the completed code block consists of the following:

1. The `SELECT` statement, pulling data from the first and last name columns
2. The `FROM` statement, telling Postgres from which table we're getting the data
3. And two conditional statements: the dates of birth and the dates of hire

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Run the completed query to view the result.



	first_name character varying	last_name character varying
1	Georgi	Facello
2	Chirstian	Koblick
3	Sumant	Peac
4	Kazuhide	Peha

A list of first and last names should appear in your Data Output window. It's a long list, too—too long to scroll to the end in any reasonable amount of time.

REWIND

Much like the `count()` method in Pandas, PostgreSQL has a `COUNT` function that will count the number of rows in a table. Like Python, when we use this function in SQL, we can find the length of a table, or how many viable rows of data exist in it.

Count the Queries

In Postgres, the `COUNT` function works in a similar fashion as the Python `count` method: It counts the rows of a specified column. We can insert the `COUNT` function into our original query, with some adjustments, but because we want a different result, it's better to create a new query instead.

The new query is extremely similar to the original. We only need to use a single column to get the count, though. Copy the original query and paste it at the bottom of the query editor so we can update it.

```
-- Number of employees retiring
SELECT COUNT(first_name)
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Our new query is almost exactly the same as the original with one exception: the `COUNT()` function has been inserted after the `SELECT` statement, and only one column `(first_name)` is included.

With this update, we are telling Postgres to count the number of rows in the first_name column. We could change the column to last_name and get the same results. This is because the length of our query is the same regardless of which of the two columns we use in the `COUNT` function.

Create New Tables

Bobby has a list of retirement-eligible employees to submit to his boss, but it's still on the database. If data was imported to the database using CSV files, why not export it the same way?

We'll need to take a couple of extra steps to help Bobby prepare his data. In Postgres, only data that is saved as a table can be easily exported. We already have a query that returns the results we need, so let's use that to create our table. Copy the query that returns employees born and hired within the correct window, then paste it to the bottom of the query editor again.

For reference, the original query we'll be modifying is as follows:

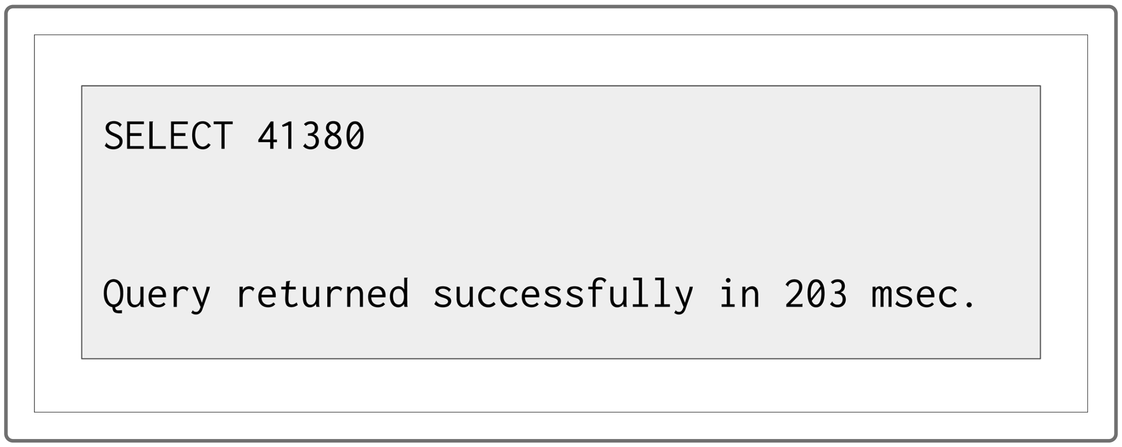
```
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

This time, the change we'll make to the code is also small—we're modifying the `SELECT` statement into a `SELECT INTO` statement. This statement tells Postgres that instead of generating a list of results, the data is saved as a new table completely. Update our code to include the `INTO` portion of the `SELECT` statement.

Insert a new, blank line between the `SELECT` and `FROM` sections of the code. In this vacant space, type in `INTO retirement_info`. With the addition of this line, we're telling Postgres to save the data into a table named "retirement_info." The complete code should look as follows:

```
SELECT first_name, last_name
INTO retirement_info
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

When executing this code, instead of receiving a list of results, you should see a message confirming the query has been successful.



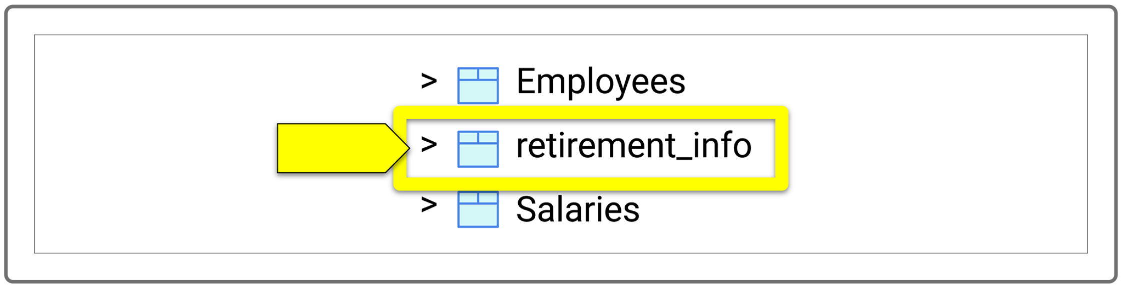
```
SELECT 41380


Query returned successfully in 203 msec.
```

So how can we see what the table looks like? We query it with a `SELECT` statement.

```
SELECT * FROM retirement_info;
```

Our list of data from earlier is now an actual table that we can use with statements and functions to perform analysis. Additionally, if you refresh the list of tables from the dropdown menu on the left, it will now appear in the list.

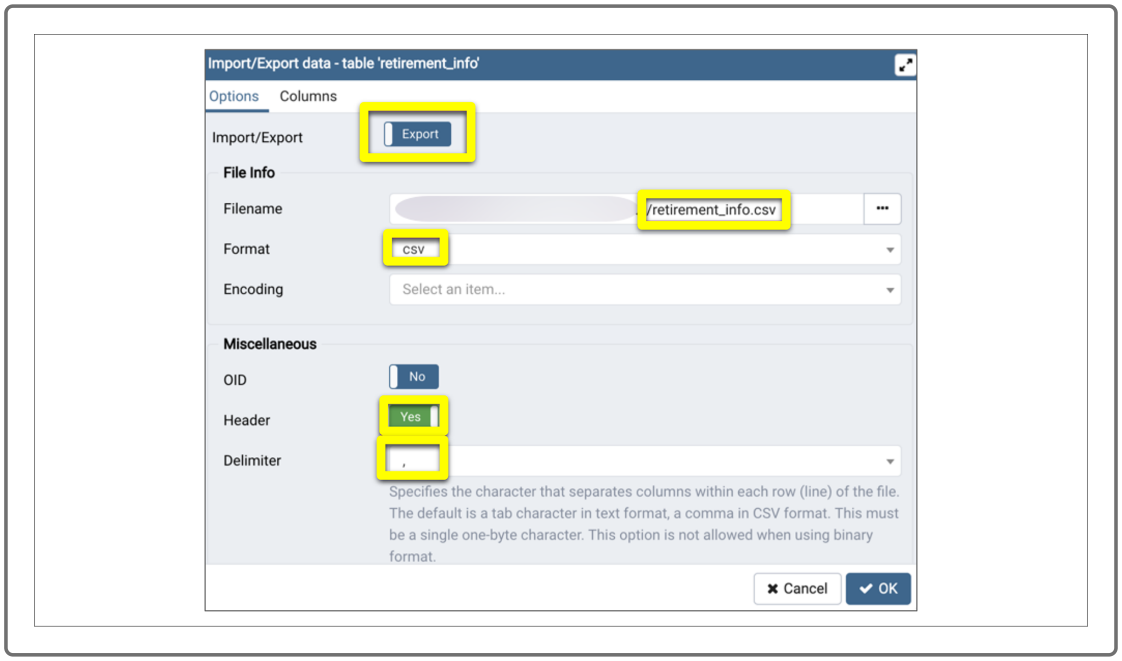


This list of tables is where we'll export the data.

Export Data

The export setup is very similar to the import. Let's walk through exporting the "retirement_info" table together.

Right-click on your new table and select "Import/Export." Instead of importing anything, this time we'll be exporting.


Import/Export data - table 'retirement_info'

Options Columns

Import/Export: Export

File Info

Filename: /retirement_info.csv

Format: csv

Encoding: Select an item...

Miscellaneous

OID: No

Header: Yes

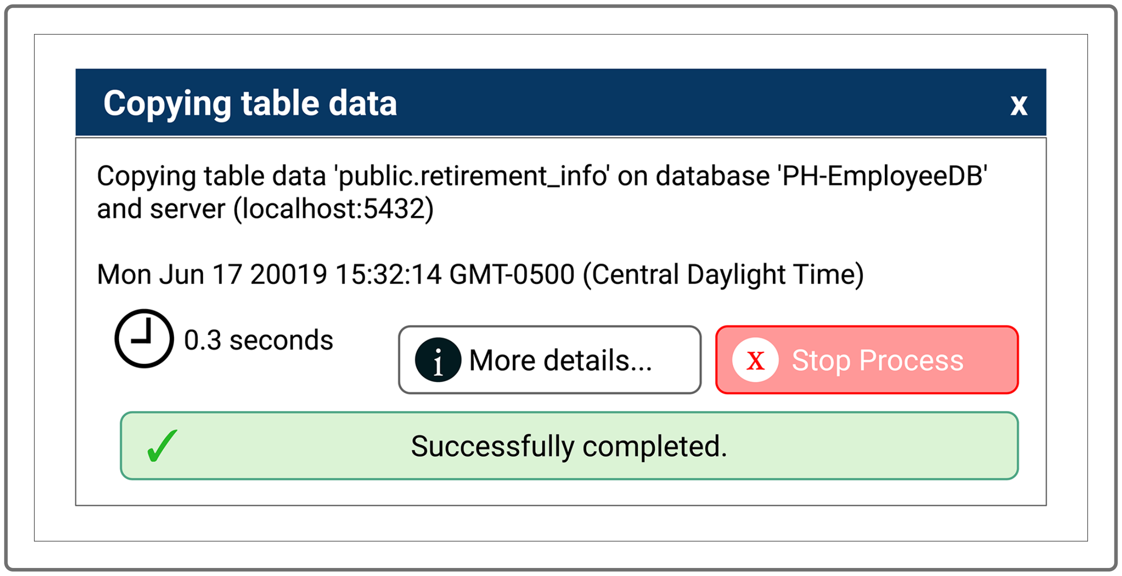
Delimiter: ,

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in CSV format. This must be a single one-byte character. This option is not allowed when using binary format.

Then, follow these steps:

1. Keep the Import/Export button toggled to "Export."
2. Click on the `...` in the Filename field to automatically select the same directory from which you imported the other CSVs. Select a directory, but be sure to rename it to `retirement_info.csv`.
3. Be sure the format is still CSV.
4. Toggle the Header section to "Yes" to include column names in the new CSV files.
5. Select the comma as the delimiter to maintain the same format with all CSV files.

Click OK to start the export. After the file has been created, pgAdmin will confirm our file is ready to be viewed.



Copying table data

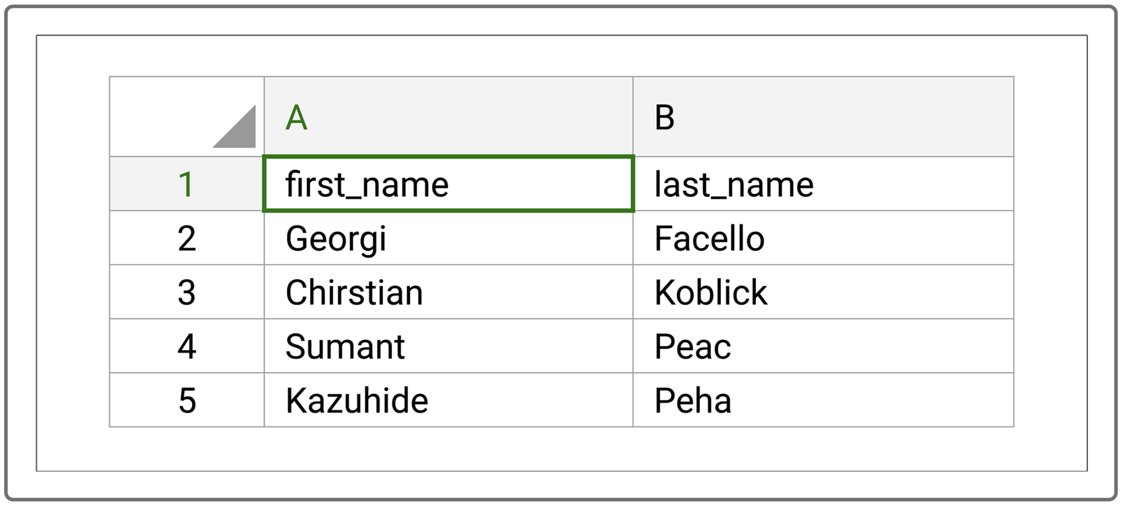
Copying table data 'public.retirement_info' on database 'PH-EmployeeDB' and server (localhost:5432)

Mon Jun 17 20019 15:32:14 GMT-0500 (Central Daylight Time)

Successfully completed.

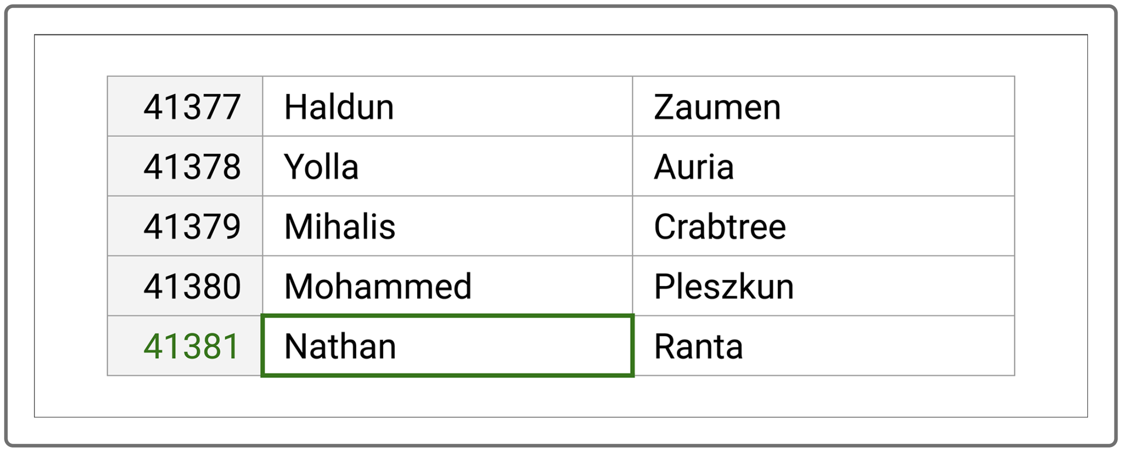
Open your project folder and navigate to the resource file holding your CSVs, then open the new file to verify the data was successfully exported.

The first five names look the same as our earlier query results.



	A	B
1	first_name	last_name
2	Georgi	Facello
3	Chirstian	Koblick
4	Sumant	Peac
5	Kazuhide	Peha

Did the entire list get exported? In Excel, use the hotkey to move to the bottom of the spreadsheet (Command + down arrow for a Mac, or CTRL + down arrow for Windows).



41377	Haldun	Zaumen
41378	Yolla	Auria
41379	Mihalis	Crabtree
41380	Mohammed	Pleszkun
41381	Nathan	Ranta

The Excel index shows there are 41,381 rows, including the header row. That's everything. This file holds the information requested: the names of all employees who were born between 1952 and 1955 and who were hired by Pewlett Hackard between 1985 and 1988.

This hefty list of employees is now ready for Bobby to present to his manager. Let's keep going.

IMPORTANT

> Much like we created a `.txt` file to save our ERD schema creation text, we should save our queries as we go as well. This is for a couple of reasons.
>
> First, if our computer crashes, pgAdmin may not save the text in our query editor. While the database is static and new tables will remain in place, it's a good idea to keep the queries on hand to reference later.
>
> Second, referencing the queries is helpful for us because we may be able to adjust and alter existing code to reuse it later. It also is useful for other people to understand how we created our queries and new tables.
>
> Copy the SQL queries we've created so far in our pgAdmin query editor into a new VSCode window. Save this file as `queries.sql` in our repo

> folder. Make sure to add the new file and commit it to our repo. This way, we won't need to worry about our work getting lost or deleted.
>
> You'll want to save new code we create to this file and commit it as we continue to progress through the module.

ADD/COMMIT/PUSH

Remember to add, commit, and push your work!