SQL Notebook 1: What are the Characteristics of Jobs by Census Block?

In these workbooks, we will start with a motivating question, then walk through the process we need to go through in order to answer the motivating question. Along the way, we will walk through various SQL commands and develop skills as we work towards answering the question.

If you have not yet watched the "Introduction to DBeaver" video, watch it before you proceed!

You should stop at the checkpoints and try doing the exercises and answering the questions posed in these sections.

Longitudinal Employer-Household Dynamics (LEHD) Data

In these workbooks, we will be using LEHD data. These are public-use data sets containing information about employers and employees. Information about the LEHD Data can be found at https://lehd.ces.census.gov/ (https://lehd.ces.census.gov/).

We will be using the LEHD Origin-Destination Employment Statistics (LODES) datasets in our applications in this workbook. Each state has three main types of files: Origin-Destination data, in which job totals are associated with a home and work Census block pair, Residence Area Characteristic data, in which job totals are by home Census block, and Workplace Area Characteristic data, in which job totals are by workplace Census block. In addition to these three, there is a "geographic crosswalk" file with descriptions of the Census Blocks as they appear the in the LODES datasets.

You can find more information about the LODES datasets here (https://lehd.ces.census.gov/data/lodes/LODES7/LODESTechDoc7.3.pdf).

Motivating Question

The LODES data has a wealth of information about jobs at the census block level. We want to explore this, so that we can characterize the data that is available to us. That is, for any given state, we want to answer, for example, some of the following questions:

- **How many census blocks contain workplaces?**
- **What were the most jobs in a census block?**
- **How many census blocks had over 50 jobs? Over 100?**
- **Among census blocks containing workplaces, what is the average number of jobs per census block?**

These, as well as other questions about the data we might answer, can help us better understand the distribution of jobs by location. In this notebook, try to keep these types of questions in mind as we explore the data.

Starting Out: Introduction to SQL and Relational Databases

SQL is a language designed for a very specific purpose: to interact with relational databases.

- **Database**: A database is a structured collection of data. There are various different ways of structuring the database, and there may or may not be information about the relationship between entities in the database.
- **Query**: A query is a request for data from the database.
- **Database Management System (DBMS)**: A DBMS is a system of storing and managing databases, including querying the database.
- **Relational Database Management System (RDBMS)**: In an RDBMS, data records are stored in *tables*, each of which has a predefined set of *columns*, the pieces of information captured for each record in a table, and *rows* in the table, where each row has a place to store a value for every column in the table.

Tables, including their columns, column types and relationships with other tables, are defined in a database **schema**. Many times, tables will contain a **primary key**, one or more columns that uniquely define a row. You can think of the primary key as a kind of ID, in which each row is given a unique ID. Tables can also contain **foreign keys**, which are column(s) that comprise the primary key in another table and, thus, provides a way of matching between multiple tables.

Tables may be located by first locating your database (`ds_public_1`), then your schema (`dbo`), and finally the table as defined by the schema (`lodes_ca_wac_S000_JT00_2015`). Generally, a table's complete location will be identifiable by those three components: database.schema.table. We will begin by using `ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015` .

Writing a Basic Query

In order to analyze the data in a database, we need to query it. Let's start with some basics. We'll start by retrieving all columns from the California Workplace Area Characteristic (`lodes_ca_wac_S000_JT00_2015`) table. Try running the following query:

```
SELECT TOP 10 * FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

You should see 10 rows of the `lodes_ca_wac_S000_JT00_2015` dataset. Let's go over the basics of this SQL command.

- **SELECT:** We start out with the `SELECT` statement. The `SELECT` statement specifies which variables (columns) you want.
  - Here, we used `SELECT TOP 10 *` . The " `*` " just says that we want all the variables and show ten observations.
  - If we wanted a few columns, we would use the column names separated by commas instead of " `*` " (for example, `w_geocode, createdate`).

- **FROM:** Now, let's look at the next part of the query, `lodes_ca_wac_S000_JT00_2015` . This part of the query specifies the table, `ca_wac_2015` , from which we want to retrieve the data. Most of your queries will begin in this fashion, describing which columns you want and from which table.

In this case, we've put everything in one line, but that's not necessary. We could have split the code up into multiple lines, like so:

```
SELECT TOP 10 *
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

This gives the same output as our original query. Generally, once queries start getting longer, breaking up the code into multiple lines can be very helpful in organizing your code and making it easier to read.

Along those lines, note that we used a semi-colon at the end of the query to mark the end of the query. That isn't absolutely necessary here, but it does help mark the end of a query and is required in other applications of SQL, so it's good practice to use it.

Side note about capitalization

If you notice, we've been using all caps for SQL commands and all lowercase for data table and schema names. This is simply a convention, as SQL is not case sensitive. For example, we could have run `select top 10 * from ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;` and it would have given us the exact same output as the first query.

This does mean you need to be careful when using column names. If your column name has capital letters in it, you need use double quotes (e.g. `"C000"`) to preserve the capitalization. For this reason, you might find that using all lowercase letters in column names is preferable, which is what we've done here.

Now, consider the following query. What do you think it will do?

```
SELECT TOP 100 w_geocode, createdate
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

We've changed the original query by using `w_geocode, createdate` instead of `*` , so we'll only get the values from two columns, `w_geocode` and `createdate` . In addition, we've changed the value after `TOP` to be 100 instead of 10, so we'll get the first 100 rows instead of the first 10 rows.

```
SELECT TOP 100 w_geocode, createdate
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

Checkpoint 1: Running Basic Queries

Consider the following queries.
What do you think they will do?
Try figuring out what the output will look like, then run the code to see if you're correct.

- `SELECT TOP 25 * FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;`
- `SELECT TOP 100 c000,ca01,ca02,ca03 FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;`
- `SELECT TOP 100 * FROM ds_public_1.dbo.ca_od_2015;`
- `SELECT TOP 40 * FROM ds_public_1.dbo.lodes_ca_rac_S000_JT00_2015;`

Think about the following scenarios. What is the query you would use to answer these questions? Try them out.

- You want to see the first 100 rows of the origin and destination geocodes for each census block in California.
- You want to see the top 1000 rows of census blocks containing workplaces and the number of jobs for workers of each race.

Checking Number of Rows and Duplicates

Let's say we want to find out how many rows there are. You can do this by using a `COUNT`.

```
SELECT COUNT(*)
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

Here, we used `COUNT(*)`, which does a count of all rows, regardless of `NULL` values. We can instead do a count of all non-`NULL` values of a certain variable by including that variable instead of `*`.

```
SELECT COUNT(w_geocode)
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

But wait; what if there are duplicates in the data? We can check for them by using `DISTINCT`.

```
SELECT DISTINCT TOP 100 w_geocode
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

This shows us all of the rows with distinct `w_geocode` values; that is, all of the distinct census block ids. Let's count how many there are. To count them, all we have to do is put `COUNT()` around the `DISTINCT` part.

```
SELECT COUNT(DISTINCT(w_geocode))
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

Building Up a Query

Notice that we wanted to count the number of distinct rows, but we first started from querying the rows with distinct `w_geocode` first before adding in the `COUNT`. Though this is a simple example, this process of building up a query as we go is important, especially when we get to more complicated tasks. When writing a query, try to think about the basic parts first, and feel free to run intermediate steps (making sure to include `TOP`) as you go.

Using Conditional Statements

Suppose we want to look at a subset of the data. We can use conditional statements to do this. For example, suppose we are interested in looking at Census blocks with a specific total number of jobs.

```
SELECT TOP 1000 *
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
WHERE c000 < 100;
```

Using a query like the one above can also be useful for finding if there are any data entry errors or missing values. Since it's not possible to have job totals less than 0, if there are any rows with negative job totals, this is likely an error or the method used to code missing values (e.g. `-1`).

We can also use more complicated conditional statements.

```
SELECT count(*)
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
WHERE (c000 > 50) AND (c000 < 100);
```

This subsets to rows in which `c000` is greater than 50 and `c000` is less than 100. That is, this subsets to census blocks with between 50 and 100 total jobs. You can structure `OR` statements in a similar manner.

```
SELECT count(*)
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
WHERE (c000 <= 50) OR (c000 >= 100);
```

This subsets to rows in which `c000` is less than or equal to 50 or `c000` is greater than or equal to 100. This query should, in other words, capture the rest of the rows.

Common Comparison Operators

Though there are some more complicated comparison operators (if you're curious, feel free to look up what `LIKE` and `IN` do), these should cover most of what you want to do.

- **=** : equal to
- **!=** or " **<>** ": not equal to
- **<** : less than
- **<=** : less-than-or-equal-to
- **>** : greater than
- **>=** : greater-than-or-equal-to
- **IS NULL** and **IS NOT NULL** : The signifier of a row in a column not having a value is a special keyword: `NULL` . To check for `NULL` , you use `IS NULL` or `IS NOT NULL` , rather than "=" or "!=". For example, to count the number of rows with `NULL` values for `c000` we might use the following:

```
SELECT count(*)
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
WHERE c000 IS NULL;
```

Using Case When

There are instances where you will find it helpful to use `CASE WHEN` to create a categorical variable. We could have, in the above example, just created a categorical variable that takes 1 if the number of total jobs in a census block is over 100 and takes value 0 if not. In the example below, we select ten observations of the columns `w_geocode` and `over100` . `over100` is a variable that is 1 if `c000` >100 and 0 otherwise.

```
SELECT TOP 10 w_geocode, (CASE WHEN c000>100 THEN 1 ELSE 0 END) AS over100
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015;
```

Side Note: Aliasing

You may have noticed that we included a part using " `AS` ," followed by a new name, in the first line. When you ran the code, you might have noticed that the column labels were changed to these new names. This is called aliasing, and is done for readability and ease of access. Later on, aliasing will also help us more easily reference tables within the same query.

Checkpoint 2: Counting Rows, Using Conditional Statements and Creating Variables

We've included the 2015 OD, RAC, WAC, and geography crosswalk data for both California and Illinois for you in tables.

The California tables are named the following:

- `lodes_ca_wac_S000_JT00_2015` : CA Workplace Area Characteristics
- `lodes_ca_wac_S000_JT00_2015` : CA Residence Area Characteristics
- `ca_od_main_JT00_2015` : CA Origin-Destination
- `lodes_ca_xwalk` : CA Geography Crosswalk

The Illinois data follow the same format as the California, except with `il` replacing `ca` (e.g. `il_wac_2015` for the Illinois 2015 WAC). Try using the methods described in this section to further explore the tables. Answer the questions below, making sure to write out the queries used to answer the questions.

- How many census blocks contain more than 200 jobs?
- How many census blocks contain residences of fewer than 25 workers?
- How many census blocks contain workplaces with more than 10 workers with a Bachelor's degree or higher?
- How many counties are there?
- How many total census blocks are there?
- How many Metropolitan/Micropolitan areas are there?

Using Aggregation Functions

We've created a variable that indicates whether the census block had over 100 total jobs. What if we are interested in number of jobs within a census block with monthly earnings at or below $1250 or not. What if we wanted to know how many blocks had over 100 jobs and how many didn't, and to display this information in one table? We can now use the `GROUP BY` statement. The `ORDER BY` statement orders the rows that it displays according to whatever you put after it. In this case, we chose the count of `over100`.

```
SELECT (CASE WHEN CE01>100 THEN 1 ELSE 0 END) AS over100, COUNT(w_geocode)
as ct
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
GROUP BY (CASE WHEN CE01>100 THEN 1 ELSE 0 END)
ORDER BY ct DESC;
```

Using GROUP BY with Multiple Variables

For the next few queries, let's try using a different table. The `ca_xwalk` table in the same `lodes` database contains information about each of the census blocks in California. We can use this to, for example, look at aggregation by CBSA (metropolitan/micropolitan area) name and by county name.

```
SELECT TOP 100 cbsaname, ctyname, COUNT(*)
FROM ds_public_1.dbo.lodes_ca_xwalk
GROUP BY cbsaname, ctyname
ORDER BY COUNT(*) DESC;
```

This first groups by CBSA (`cbsaname`) name, then it groups by county (`ctyname`), in that order. In this case, county is nested completely in the metropolitan/micropolitan area. In other cases in which we don't have complete nesting, we would be able to see all possible combinations that exist in the data.

Further, notice that we used `DESC` after `ORDER BY` . This orders in descending order instead of ascending order, so that we can see the areas with the most census blocks at the top.

Conditional Statements After Aggregation

Suppose we wanted to display only certain counts. We can use `HAVING` to do this.

```
SELECT TOP 100 ctyname, cbsaname, COUNT(cbsaname)
FROM ds_public_1.dbo.lodes_ca_xwalk
GROUP BY ctyname, cbsaname
HAVING count(cbsaname) > 20000
ORDER BY COUNT(*) DESC;
```

This will only display the counts for which the count of `cbsaname` is greater than 20000. Note that this is different from using `WHERE` , since the conditional statement comes after the `GROUP BY` statement. Basically, `HAVING` gives us a way of using the same types of conditional statements after we do our aggregation.

Using Different Aggregation Functions

What if we wanted to find the sum within each group, or the minimum or maximum value? We can use the appropriate aggregation function. To show this, let's go back to our `lodes_ca_wac_S000_JT00_2015` table.

```
SELECT (CASE WHEN CE01>100 THEN 1 ELSE 0 END) as over100, COUNT(*) AS ct,
AVG(cast(c000 AS INT)) AS avg_jobs, MIN(cast(c000 AS INT)) AS min_jobs,
MAX(cast(c000 AS INT)) AS max_jobs
FROM ds_public_1.dbo.lodes_ca_wac_S000_JT00_2015
GROUP BY (CASE WHEN CE01>100 THEN 1 ELSE 0 END)
ORDER BY over100;
```

Here, we're finding the counts, average, minimum, and maximum value of the total jobs in each census block within each group. Now, we're not doing anything very insightful here, since the groups already split the blocks by how many jobs there are. However, as we'll see later on, these aggregation functions can be very useful. For example, suppose we had the county data that's in `lodes_ca_xwalk` in this table. We could find the average number of jobs per census block for each county in this way.

Checkpoint 3: Checking Your Dataset

Using the above methods, explore the tables we've provided or your own state's data to answer the questions below. As before, make sure to include the queries with your answers.

- Which county has the most census blocks?

- Which Metropolitan/Micropolitan area has the most census blocks?
- Which Origin census block - Destination census block combination has the most workers? How many workers are in this combination?
- How would you find all counties containing at least 1000 census blocks?
- For California, how many census blocks are there with a latitude above +36?
- For California, which county has the most census blocks above the +36 latitude line? Which county has the most below?