Chapter 15: Database Querying Using SQL

- Small data sets can easily fit in a computer's memory.
- Medium data sets can fit on a computer's hard disk, but perhaps not in its memory.
- SQL (Structured Query Language) is a database querying language developed in the 1970s that is ideal for retreiving medium data from a database.
- SQL is so ideal for its purpose that it's very commonly used by data scientists (often data science job interviews include quizzes about SQL queries).

From `dplyr` to SQL

- Considering the airline flight data, if we want to retrieve the top on-time carriers with at least 100 flights arriving at JFK in September 2016, we could write R code in `dplyr` using functions like `filter`, `inner_join`, `summarize`, and `arrange` (see example code).
- However, the full `flights` data set has 169 million individual flights (around 20 GB of memory).
- Difficult to store in R in the computer memory, so it's better to store data on disk or on a remote server.
- Then we can use querying statements to retrieve only the rows we need.

tbl_sql Objects

- The dbConnect_scidb function from the mdsr package can be used to connect to a remote server where the airlines data sets are stored.
- Using the tbl function, we create an R object that is not exactly a data frame, but rather a tbl_sql object.
- When working with tbl_sql objects, dplyr actually internally converts the pipeline into an SQL query, which can be seen explicitly with the show_query function.
- Certain R functions (e.g., filter, n()) translate directly to SQL operations (WHERE, COUNT(*)).
- The translate_sql function translates simple R commands into SQL commands (see examples).
- In fact, the dbplyr package is designed to take dplyr code and make it work as SQL queries do.

Flat-file databases, Memory, and Disk Space

- A *flat-file* database consists of rows and columns of data, usually with a header row of column names.
- Data in memory can be accessed at once and quickly, but computers typically can hold only a few GB of memory.
- Data in the hard disk is stored permanently; it can be accessed but much more slowly. Computers typically can hold thousands of GB in hard disk space.
- All objects in the R workspace are stored in memory, so dealing with very large data in R makes performance sluggish.
- It's more efficient with large data sets to store them externally and use *relational databases*.

Implementations of SQL

- SQL (Structured Query Language) is a programming language developed in the 1970s for relational database management systems.
- There are numerous implementations of SQL, such as *SQLite*, *MySQL*, and even SAS's `PROC SQL`.
- In these classroom examples, we will run the SQL queries in R using the `sqldf` package.
- This doesn't cover all the aspects of SQL, but it handles the major querying tools.

Getting Started with SQL

- With SQL, there is a database server (which may be a remote server) that stores the data and executes queries.
- Appendix F gives information about setting up a MySQL database, but we will not cover that here.
- For the examples showing SQL queries, we will use a remote SQL relational database, already set up and containing multiple tables, using the `airlines` database from the `nycflights13` package.

SQL Queries

- Queries in SQL are the main way to access data.
- All queries start with the `SELECT` keyword and contain one or more *clauses*.
- Many of the clauses have direct analogues in `dplyr` verbs.
- `SELECT` lists the columns that you want to retrieve (like `select` in `dplyr`).
- `SELECT *` selects *all* the columns from a table — be careful when doing this.
- You can also select functions or transformations of the columns of a table (like with `mutate` in `dplyr`).

Other SQL Clauses

- `FROM` specifies the table from which you are selecting the columns.
- `WHERE` allows you to pick only the records satisfying some criteria (like the `filter` verb in `dplyr`).
- `GROUP BY` allows you to aggregate the records according to some grouping variable (like the `group_by` verb in `dplyr`).
- `HAVING` operates similarly to `WHERE` in that it filters results based on some logical condition, but `HAVING` operates on the *result set* rather than the records themselves.
- So `HAVING` is a filter applied after the rows have already been aggregated via `GROUP BY`.
- `ORDER BY` specifies a condition for ordering the rows of the result set (like the `arrange` verb in `dplyr`).
- `LIMIT` restricts the number of rows that are printed in the output (like `head` or `slice` in R).

Basic SQL Queries

- The only *required* clauses in a query are `SELECT` and `FROM`.
- The other clauses are optional.
- Typically, SQL queries end in a semicolon (although if we run them with `sqldf` in R, the query is quoted, so that the semicolon is not needed).
- The clauses don't have to be on different lines, although it can make long queries more readable if you place the clauses on different lines.
- The keyword `AS` allows you to specify a column name in the result set that is different from the column name in the original table.
- See the examples of basic queries on the `airlines` data set.

The `WHERE` Clause

- Using the `WHERE` clause, we can retrieve only the data records that meet a certain condition.
- This could be a *compound* condition, specified with `AND` and `OR` keywords.
- Besides equality (or inequality) conditions, we can use `BETWEEN` or `IN` keywords to specify that we want records with values between two values or in some list of values.
- When using `AND` and `OR` keywords, be careful: The criteria in the `WHERE` clause are not evaluated left to right, but rather the `AND`s are evaluated first, before the `OR`s.
- Use parentheses, where needed, to guide the query about which conditions to evaluate first (see examples).

The `GROUP BY` Clause

- The `GROUP BY` clause allows you to aggregate rows so that the output has results given for the distinct levels of some grouping variable.
- You must use an `aggregate` function when using `GROUP BY`.
- Some aggregate functions in SQL are `COUNT()`, `SUM()`, `MIN()`, `MAX()`, and `AVG()`.
- See examples on the airlines data.

The `ORDER BY` Clause

- The `ORDER BY` clause can take the output and order the rows by some variable.
- Using the `DESC` keyword after the variable name will return the rows in descending order of that variable, rather than in ascending order which is the default.
- If the variable is a character variable, the ordering will be alphabetical.
- `ORDER BY` may be used in a query either with or without a `GROUP BY` clause.

The HAVING Clause

- The HAVING clause will filter the *result set* based on some specified condition.
- Note that WHERE operates on the original data in the table and HAVING operates on the result set.
- The implementations *MySQL* and *SQLite* allow you to use derived column aliases in HAVING clauses, but not all SQL implementations support this.
- You cannot move the conditions that operate on the result set from the HAVING clause to the WHERE clause — this will give an error.
- You can move the conditions that operate on the original data from the WHERE clause to the HAVING clause, but this will result in a loss of efficiency since the aggregation will be performed on *all* the records.
- See examples of the correct use of WHERE and HAVING.

The `LIMIT` Clause

- A `LIMIT` clause truncates the output to a specified number of rows (like the `head` or `slice` functions in R).
- The first number in the `LIMIT` clause indicates the number of rows to skip, and the latter number gives the number of rows to retrieve.
- So which rows will be returned if we use the following?
- `LIMIT 5, 10`
- `LIMIT 0, 4`

Joining Tables in SQL

- Recall that several joining operators were available in `dplyr`, such as `inner_join`, `left_join`, etc.
- In SQL, we can similarly join tables using the `JOIN` keyword.
- We must specify:
  1. the name of the first table to join
  2. the type of join
  3. the name of the second table to join
  4. the condition on which to join (the `ON` condition)

Types of Joins

- The types of joins available differ across SQL implementations.
- The basic `JOIN` keyword will do an inner join, which will include all records that are in *both* tables.
- `LEFT JOIN` will include all the records in the first table, and records that have no match in the second table will have missing values in the resulting joined table for the columns that appear only in the second table.
- `RIGHT JOIN` will include all the records in the second table, and records that have no match in the first table will have missing values in the result.

Other Types of Joins

- The `CROSS JOIN` is a combination of every row from the first table with every row from the second table to form the result set.
- If the first table has n rows and the second table has m rows, the result set from the `CROSS JOIN` will have $n \times m$ rows.
- The `CROSS JOIN` is mainly useful to create a large table on which to do further processing.
- The `FULL JOIN`, also called `FULL OUTER JOIN`, is available in some SQL implementations but not all of them.

Flights and Airports example

- The `flights` table contains information on individual flights, including the departure airport and destination airport.
- However, the airports are only identified with their three-letter FAA codes.
- It may be desirable to have the full airport name in a table with the flight information.
- The `airports` data set has both a column with the FAA code and a column with the full airport name.
- We can join the `flights` and `airports` tables to get all the information we need in one table.
- See examples of an inner join (and of a left join that includes some flights to destinations that are not in the `airports` table).

Table Aliases

- We have seen the use of column aliases, in which we rename (using the `AS` keyword) a column in the result table to make the column header more descriptive.
- We can also make code for joins more readable by using *table aliases*.
- Customarily, table alias is a single letter following the `AS` that can later be used as a shortcut reference for the table name.
- Note that in SQL, we often refer to a column using both the table alias and the column name.
- For example: We create the table alias via `flights AS o` and then refer to the `dest` column in `flights` using `o.dest` in the `ON` condition.
- This is very useful when columns in two different tables have the same name, i.e., `ON a.idnumber = b.idnumber` where `idnumber` is the column name in both tables `a` and `b`.

Joining Multiple Tables

- We can join more than two tables together using multiple `JOIN` clauses.
- In the airlines example, the airline carriers have codes, but the `airlines` table has information on the full airline names associated with these codes.
- Joining the `flights`, `airports`, and `airlines` tables will allow us the have information on individual flights, full airport names, and full airline names all in one table.
- Why not simply store the whole airport and airline names in the `flights` data table instead of codes, if we're going to need the names?

More on Joining Multiple Tables

- Storage reasons: 169 million rows in `flights` — abbreviations take up less space.
- Also, airport names could change in the future, but FAA codes are permanent.
- We can also join with the same table twice in one SQL query: We must give the table different aliases in the different `JOIN` clauses.
- In that case, using table aliases becomes necessary.
- See examples with the flights data.

The UNION Keyword and Subqueries

- We can use the UNION keyword to bind together the records from the results of two queries.
- We can also do a query on the result set from a previous query.
- The initial query is called a subquery and is typically set apart with parentheses.
- See examples with the flights data where a subquery produces a list of values that are used in a subsequent query.

Thoughts on R and SQL

- Many common tasks can be performed with either tools in the R package `dplyr` or with SQL queries.
- R will typically be preferred with smaller data sets, since it has many more statistical functions available.
- SQL is preferred if the data set is large enough that it does not fit easily in the computer's memory, but SQL has only the most rudimentary functions for data analysis.
- Overall, SQL shines in *data management*, while R is better for *data analysis*, so it is helpful for a data scientist to be familiar with both.