SQL in R

Chapter 8

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Introduction

The sqldf Package

The DBI Package

The dbplyr Package

Resources for Further Learning

Introduction

Introduction

As we have seen, SQL is the primary programming language for querying and interacting with relational databases through a database management system (DBMS).

While learning SQL on its own is essential, it is also common to work with relational databases through programming languages like R and Python, often combining SQL queries with code for exploratory data analysis, visualization, and automation.

In this lecture, we will introduce how R can interface with SQL in different ways, further expanding our toolkit for working with relational data.

The sqldf Package

The sqldf Package

The first basic way to use SQL with R is by using the **sqldf** package, which allows R data frames to be queried using SQL as if they are tables in a database.

```
library(sqldf)
```

The main function in the sqldf package is the **sqldf()** function that queries a data frame.

The sqldf() function loads the data frame into an in-memory SQLite database, so queries are often faster than other methods (i.e., the Tidyverse).

Note: The sqldf() function is only for using SQL queries (i.e., SELECT statements, including joins). It cannot be used to insert, update, or delete data.

Example: The sqldf() Function

The `sqldf()` function inputs a character string representing a SQL `SELECT` statement or character vector whose components each represent a successive SQL statement to be executed.

The table(s) used in the `FROM` clause should be data frames in the calling environment. Remember that even though SQL keywords are not case sensitive, R object names are case sensitive.

For example, we can find the number of flowers of each species in the `iris` data below:

```r
sqldf("SELECT Species, COUNT(*) AS num_flowers
       FROM iris
       GROUP BY Species;")
```

```
     Species num_flowers
1     setosa          50
2 versicolor          50
3  virginica          50
```

The output of `sqldf()` is always a data frame.

Names With Periods

If a column has a name that includes a period (`.`), the `sqldf()` function will throw an error if queried only by name.

```r
sqldf("SELECT AVG(Petal.Length) AS mean_petal_length
       FROM iris
       GROUP BY Species;")
```

```
Error: no such column: Petal.Length
```

Names with periods in them must be enclosed with backticks.

```r
sqldf("SELECT AVG(`Petal.Length`) AS mean_petal_length
       FROM iris
       GROUP BY Species;")
```

```
  mean_petal_length
1             1.462
2             4.260
3             5.552
```

Question: Why does a period cause issues here?

Names With Periods: Double Inside Single

Caution: If using single quotes (`''`) to define the character string, double quotes (`""`) around the column name can be used instead of backticks. You cannot use single quotes inside of double quotes here.

```
sqldf('SELECT AVG("Petal.Length") AS mean_petal_length
       FROM iris
       GROUP BY Species;')
```

```
  mean_petal_length
1             1.462
2             4.260
3             5.552
```

```
sqldf("SELECT AVG('Petal.Length') AS mean_petal_length
       FROM iris
       GROUP BY Species;")
```

```
  mean_petal_length
1                 0
2                 0
3                 0
```

The read.csv.sql() Function

The **`read.csv.sql()`** function reads a CSV file into R after applying a SQL query, so only rows from the output of the query will be processed by R. This can be useful when processing large CSV files.

```
sql_query <- "SELECT Gender, Premie, weight
              FROM file
              WHERE Gender = '\"Female\"';"
births <- read.csv.sql("births.csv", sql = sql_query)
head(births)
```

```
    Gender Premie weight
1 "Female"   "No"    177
2 "Female"   "No"    144
3 "Female"   "No"     98
4 "Female"   "No"    104
5 "Female"   "No"    123
6 "Female"   "No"    153
```

Note that the `file` table in the `FROM` clause of `sql_query` is implied from the `file` argument in `read.csv.sql()`.

The DBI Package

The DBI Package

The `sqldf` package is useful for applying SQL queries to data frames and importing data from large CSV files into R. While data frames are treated like tables in a database, we also want to interface with a relational database that might not be feasibly stored on a local computer.

The `DBI` package provides a database interface for R to connect to and interact with databases.

```r
library(DBI)
```

The dbConnect() Function

The **dbConnect()** function establishes a connection to a database (technically to an DBMS).

The function itself does not have the necessary drivers (i.e., the backend communication software) to interact with a DBMS. We need to specify the drivers for the specific DBMS.

For SQLite, the **RSQLite::SQLite()** function outputs a SQLiteDriver object that provides the drivers to dbConnect().

```r
con <- dbConnect(RSQLite::SQLite(), "TYSQL.sqlite")
```

The second argument is the dbname argument that specifies the file (or path) name that contains the database.

Note: As written, the TYSQL.sqlite file is assumed to be in R's working directory. Make sure you place the file in the correct folder. If there is not already a file of that name in your working directory, the command above will create an empty database file of that name in that location.

Connecting to Other Databases

Most databases are not stored locally on a file. To connect to a server, you would use code more like this:

```r
# Do not run
library(RMariaDB) # (Or whichever DBMS you need)
con <- dbConnect(RMariaDB::MariaDB(),
  host = "host.address"
  user = "username",
  password = "password"
)
```

Some common drivers:

- `RMariaDB::MariaDB()` for MariaDB and MySQL
- `RPostgreSQL::PostgreSQL()` for PostgreSQL
- `odbc::odbc()` for (Microsoft) ODBC
- `bigrquery::bigquery()` for (Google) Big Query

Side Note: The `RMySQL` package is now deprecated and has been replaced by `RMariaDB`.

The dbListTables() Function

Once a connection object has been made (`con` is an S4 object that inherits from `DBIConnection`), we can now access our database using the connection.

The **`dbListTables()`** function outputs a vector of the names of the tables in the database.

```
dbListTables(con)
```

```
[1] "Customers"            "CustomersWithOrders"
[3] "OrderItems"           "OrderItemsBackup"
[5] "Orders"               "OrdersBackup"
[7] "ProductCustomers"     "Products"
[9] "Vendors"
```

The dbListFields() Function

The **`dbListFields()`** function outputs a vector of the names of the fields (columns) of a specific table.

```
dbListFields(con, "Products")
```

```
[1] "prod_id"    "vend_id"    "prod_name"
[4] "prod_price" "prod_desc"
```

```
dbListFields(con, "OrderItems")
```

```
[1] "order_num"  "order_item" "prod_id"
[4] "quantity"   "item_price"
```

The dbReadTable() Function

The **`dbReadTable()`** function reads an entire table and outputs a data frame of that table.

This is equivalent to the SQL query `SELECT * FROM <name>;`.

```r
prod_df <- dbReadTable(con, "OrderItems")
head(prod_df)
```

```
  order_num order_item prod_id quantity item_price
1     20005          1    BR01      100       5.49
2     20005          2    BR03      100      10.99
3     20006          1    BR01       20       5.99
4     20006          2    BR02       10       8.99
5     20006          3    BR03       10      11.99
6     20007          1    BR03       50      11.49
```

The dbGetQuery() Function

The usual (i.e., more general) way to query the database is to use the **dbGetQuery()** function that inputs a SQL query as a character string and returns the results in a data frame.

```
sql_query <- "SELECT cust_name, cust_state, cust_email
              FROM Customers;"
cust_info <- dbGetQuery(con, sql_query)
cust_info
```

```
      cust_name cust_state            cust_email
1  Village Toys         MI sales@villagetoys.com
2    Kids Place         OH                  <NA>
3       Fun4All         IN    jjones@fun4all.com
4       Fun4All         AZ dstephens@fun4all.com
5 The Toy Store         IL   kim@thetoystore.com
6      Toy Land         NY                  <NA>
```

Inline Queries

SQL queries can also be written inline.

```r
dbGetQuery(con, "
   SELECT prod_name, vend_name, prod_price
   FROM Products
   INNER JOIN OrderItems
   ON OrderItems.prod_id = Products.prod_id
   INNER JOIN Vendors
   ON Products.vend_id = Vendors.vend_id
   WHERE order_num = 20007;
")
```

```
            prod_name       vend_name prod_price
1  18 inch teddy bear      Bears R Us      11.99
2   Fish bean bag toy Doll House Inc.       3.49
3   Bird bean bag toy Doll House Inc.       3.49
4 Rabbit bean bag toy Doll House Inc.       3.49
5         Raggedy Ann Doll House Inc.       4.99
```

Beyond Queries: The dbExecute() Function

Most of the SQL commands used in data analytics and data science are queries using `SELECT` statements, but SQL can also be used for the definition and manipulation of data in a database.

The `dbGetQuery()` function is for any `SELECT` (or `SELECT`-type) statement that returns a query result set.

For any other statements in SQL (e.g., `CREATE`, `INSERT`, `UPDATE`, or `DELETE`), the **`dbExecute()`** command is used instead.

We will not focus on these tasks here, but there is a reference page here: https://dbi.r-dbi.org/reference/dbExecute.html

The dbDisconnect() Function

When establishing a connection with `dbConnect()`, we are opening a connection to a database, possibly on a remote server.

An open connection to a database uses resources (e.g., memory, connection slots, temporary tables, etc.), both on our local machine and on the database server. Leaving a connection open when it is no longer needed can limit resources, cause bugs, or even crash R.

The **`dbDisconnect()`** function is used to close a connection. If the connection is not explicitly closed, the connection remains open until R quits or the session ends.

For best practices, especially when working with a database on a remote server, **always disconnect** from the database when your work is done.

```r
dbDisconnect(con)
```

The dbplyr Package

The dbplyr Package

The DBI package allows us to connect R with databases and run SQL queries using R as an interface. Using DBI, queries to the database must be written in SQL syntax first, then the query is passed to dbGetQuery() to retrieve the results from the database.

Is there a purely R way to interact with a database that does not require writing SQL commands at all?

Yes! The **dbplyr** package enables us to write database queries using dplyr syntax. The dbplyr functions translate the dplyr commands into SQL for us, so we no longer need to write SQL queries directly.

The Power Within

The dbplyr package is installed as part of the Tidyverse, so install.packages("tidyverse") will include dbplyr, or it can be installed on its own with install.package("dbplyr").

After installation, dbplyr does not need to be explicitly loaded with library(dbplyr) like most other packages.

It turns out that the usual dplyr package will detect when you are working with a database and automatically load dbplyr for you.

So the more important package to load is dplyr.

```r
library(dplyr, warn.conflicts = FALSE)
```

Establishing a Connection

Even when working with `dbplyr`, we first need to establish a connection to a database, which is still done with the `DBI::dbConnect()` function.

```r
con <- dbConnect(RSQLite::SQLite(), "TYSQL.sqlite")
```

Side Note: If you are starting an R session from this section of the lecture, you would need the fully qualified function name `DBI::dbConnect()`.

The tbl() Function

The `tbl()` function retrieves a table and returns a `tbl_dbi` object.

```
orders_tbl <- tbl(con, "Orders")
orders_tbl
```

```
# Source:   table<`Orders`> [5 x 3]
# Database: sqlite 3.47.1 [/Users/mike/Dropbox/School/Teach
  order_num order_date cust_id
      <int> <chr>      <chr>
1     20005 2020-05-01 1000000001
2     20006 2020-01-12 1000000003
3     20007 2020-01-30 1000000004
4     20008 2020-02-03 1000000005
5     20009 2020-02-08 1000000001
```

Once we have a `tbl_dbi` object, we can apply `dplyr` functions to it to construct a query to the database.

dbplyr is Lazy

Even though the `orders_tbl` object we created shows the first few rows of the `Orders` table, the SQL query has not been sent to the DBMS yet. All queries written using `dplyr` are generated **lazily**.

When working with a database, all `dplyr` function calls use **lazy evaluation**: No data is queried from the database until it is explicitly retrieved. This also means that all operations are pushed to and executed in the database (DBMS or server) rather than being done in R.

The show_query() Function

As an example, we can write a query to find the total price of each order, sorted from highest to lowest.

```
total_query <- tbl(con, "OrderItems") |>
   group_by(order_num) |>
   summarize(total_price = sum(item_price * quantity)) |>
   arrange(desc(total_price))
```

The **show_query()** function shows the SQL query that was (lazily) generated.

```
total_query |> show_query()
```

```
<SQL>
SELECT `order_num`, SUM(`item_price` * `quantity`) AS `total_price`
FROM `OrderItems`
GROUP BY `order_num`
ORDER BY `total_price` DESC
```

The collect() Function

Once a `dplyr` query is ready, the **`collect()`** function is used to execute the query and retrieve the results from the database.

```
total_query |> collect()
```

```
# A tibble: 5 x 2
  order_num total_price
      <int>       <dbl>
1     20009       1868.
2     20007       1696
3     20005       1648
4     20006        330.
5     20008        190.
```

Example: Multiple Join (Generate Query)

Suppose we want to show the product name, vendor name, and product price for all items bought in order number 20007.

```r
# Create tables
vendors <- tbl(con, "Vendors")
products <- tbl(con, "Products")
orderitems <- tbl(con, "OrderItems")

# Generate query
multiple_join <- vendors |>
   inner_join(products, join_by(vend_id)) |>
   inner_join(orderitems, join_by(prod_id)) |>
   filter(order_num == 20007) |>
   select(prod_name, vend_name, prod_price)
```

Example: Multiple Join (Collect)

```r
# Execute query and retrieve data
multiple_join |> collect()
```

```
# A tibble: 5 x 3
  prod_name           vend_name       prod_price
  <chr>               <chr>                <dbl>
1 18 inch teddy bear  Bears R Us           12.0 
2 Fish bean bag toy   Doll House Inc.       3.49
3 Bird bean bag toy   Doll House Inc.       3.49
4 Rabbit bean bag toy Doll House Inc.       3.49
5 Raggedy Ann         Doll House Inc.       4.99
```

Example: Multiple Join (Show Query)

The equivalent SQL query for the previous multiple join:

```r
# Show query
multiple_join |> show_query()
```

```
<SQL>
SELECT `prod_name`, `vend_name`, `prod_price`
FROM (
  SELECT
    `Vendors`.*,
    `Products`.`prod_id` AS `prod_id`,
    `prod_name`,
    `prod_price`,
    `prod_desc`,
    `order_num`,
    `order_item`,
    `quantity`,
    `item_price`
  FROM `Vendors`
  INNER JOIN `Products`
    ON (`Vendors`.`vend_id` = `Products`.`vend_id`)
  INNER JOIN `OrderItems`
    ON (`Products`.`prod_id` = `OrderItems`.`prod_id`)
) AS `q01`
WHERE (`order_num` = 20007.0)
```

Example: Anti-Join

Suppose we are interested in finding out which of our vendors have we not yet bought products from.

```r
# Create tables
vendors <- tbl(con, "Vendors")
products <- tbl(con, "Products")
# Generate query
anti_vendors <- vendors |>
   anti_join(products, join_by(vend_id)) |>
   select(vend_id, vend_name)
# Execute query and retrieve data
anti_vendors |> collect()
```

```
# A tibble: 3 x 2
  vend_id vend_name
  <chr>   <chr>
1 BRE02   Bear Emporium
2 FRB01   Furball Inc.
3 JTS01   Jouets et ours
```

Example: Anti-Join (Show Query)

The equivalent SQL query for the previous anti-join:

```r
# Show query
anti_vendors |> show_query()
```

```
<SQL>
SELECT `vend_id`, `vend_name`
FROM `Vendors`
WHERE NOT EXISTS (
  SELECT 1 FROM `Products`
  WHERE (`Vendors`.`vend_id` = `Products`.`vend_id`)
)
```

Limitations of dbplyr

The purpose of `dbplyr` is to translate the most common and useful data wrangling and data manipulation functions in `dplyr` to their equivalent commands in SQL.

Nearly (if not) all SQL queries we have seen so far can be written using `dbplyr`, but SQL has a lot of functionality beyond what `dplyr` can do. In particular, `dplyr` can only read from the database (i.e., `SELECT` statements or the DQL part of SQL), it cannot write to it (e.g., the DDL and DML parts of SQL).

For more complex queries or other operations (such as insert, update, or delete) on the database, SQL statements still need to be written explicitly (and executed with `DBI::dbExecute()`).

One Last Thing

Don't forget to disconnect!

```
dbDisconnect(con)
```

Resources for Further Learning

References and Resources

Much of these slides come from the R documentation or official websites of the R packages. We have covered the most common and practical functions in each package, but there is more advanced functionality when using R to interface with SQL that we will not be able to cover.

Vignettes for `DBI`

- Introduction to DBI
- Advanced DBI Usage

Articles on `dbplyr`

- Using `dplyr` with Databases
- Writing SQL with `dbplyr`

Articles on the `pool` Package (for managing connections)

- Why `pool`?
- Pooling database connections in R