

# SQL for R via SQLDF

Roberto Romo

10/27/2017

## Introduction

In this course we have learned how to manipulate, tidy, and query data via matrices, vectors, lists, and data frames. With vectors we can subscript with bracket notation to query data. Similarly, by using the “dplyr” package we can query and manipulate data frames. R is a powerful language and like many other languages benefits from libraries (packages). Packages simplify calculations (with elegance) and extend the flexibility of our programs while abstracting junky and lengthy calculations. They also allow us to personalize our working environment, one that we may be comfortable with and as a result be more efficient with.

Amongst the myriad of languages, a developer has a tough choice in selecting a tool that best fits their need. While one language may offer more flexibility another might provide efficiency. A developer first keeps a couple of languages in mind that best fits their need, but how to they then proceed and pick one? It is key that a developer is flexible and can adapt to different working environments, however; developers find comfort in working environments and tools that are more native and natural to them. One language might have a slight edge over another but this doesn't mean the developer should tend to the “better” one. Often, the developer tends to the one they are more comfortable with and the one that has the necessary and sufficient power to carry out the tasks.


Some packages give developers the best of both worlds; they allow a developer to use a language that they are comfortable with while using the utility of another. This post is about extending R to use SQL commands via the “sqldf” package to manipulate and query data frames. Although this post will draw many parallels to the “dplyr” package, it will not compare both packages hand in hand, but rather inform of a tool that you might prefer. So let's get started!

## Introduction to SQL

What is SQL? SQL, Structured Query Language, is a programming language that is used to manage relational databases and to perform operations on data. SQL allows the user to forgo indices to access a record and allows the user to access many records with a single command. The fundamental utility of SQL is the ability to query and manage database systems with minimal effort.

For simplification we will only focus on the querying part of SQL and abstract the database management. We will be using the following reference as we learn more about SQL and it's place in R.

### SQL cheat sheet



#### Basic Queries

- filter your columns  
**SELECT** col1, col2, col3, ... **FROM** table1
- filter the rows  
**WHERE** col4 = 1 **AND** col5 = 2
- aggregate the data  
**GROUP** by ...
- limit aggregated data  
**HAVING** count(\*) > 1
- order of the results  
**ORDER** BY col2

Useful keywords for **SELECTS**:

- DISTINCT** - return unique results
- BETWEEN a AND b** - limit the range, the values can be numbers, text, or dates
- LIKE** - pattern search within the column text
- IN** (a, b, c) - check if the value is contained among given.

#### Data Modification

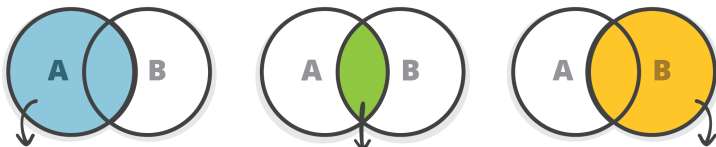
- update specific data with the **WHERE** clause  
**UPDATE** table1 **SET** col1 = 1 **WHERE** col2 = 2
- insert values manually  
**INSERT INTO** table1 (**ID**, **FIRST\_NAME**, **LAST\_NAME**)  
**VALUES** (1, 'Rebel', 'Labs');
- or by using the results of a query  
**INSERT INTO** table1 (**ID**, **FIRST\_NAME**, **LAST\_NAME**)  
**SELECT** id, last\_name, first\_name **FROM** table2

#### Views

A **VIEW** is a virtual table, which is a result of a query. They can be used to create virtual tables of complex queries.

**CREATE VIEW** view1 **AS**  
**SELECT** col1, col2  
**FROM** table1  
**WHERE** ...

#### The Joy of JOINS



**LEFT OUTER JOIN** - all rows from table A, even if they do not exist in table B

**INNER JOIN** - fetch the results that exist in both tables

**RIGHT OUTER JOIN** - all rows from table B, even if they do not exist in table A

#### Updates on JOINed Queries

You can use **JOINS** in your **UPDATE**:

```
UPDATE t1 SET a = 1  
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id  
WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

NB! Use database specific syntax, it might be faster!

#### Semi JOINS

You can use subqueries instead of **JOINS**:

```
SELECT col1, col2 FROM table1 WHERE id IN  
(SELECT t1_id FROM table2 WHERE date >  
CURRENT_TIMESTAMP)
```

#### Indexes

If you query by a column, index it!

```
CREATE INDEX index1 ON table1 (col1)
```

Don't forget:

- Avoid overlapping indexes
- Avoid indexing on too many columns
- Indexes can speed up **DELETE** and **UPDATE** operations

#### Useful Utility Functions

- convert strings to dates:  
**TO\_DATE** (Oracle, PostgreSQL), **STR\_TO\_DATE** (MySQL)
- return the first non-NULL argument:  
**COALESCE** (col1, col2, "default value")
- return current time:  
**CURRENT\_TIMESTAMP**
- compute set operations on two result sets  
**SELECT** col1, col2 **FROM** table1  
**UNION / EXCEPT / INTERSECT**  
**SELECT** col3, col4 **FROM** table2;

**Union** - returns data from both queries

**Except** - rows from the first query that are not present in the second query


**Intersect** - rows that are returned from both queries

#### Reporting

Use aggregation functions

- COUNT** - return the number of rows
- SUM** - cumulate the values
- AVG** - return the average for the group
- MIN / MAX** - smallest / largest value

BROUGHT TO YOU BY



For more comprehensive SQL tutorials the following sources might come of help: <https://www.w3schools.com/sql/>, <https://www.tutorialspoint.com/sql/>. When querying data with SQL we want to think of the data as being stored in different tables (much like a data frame), in the following sections we will learn how to query data by leveraging these column names and types.

## The Thrift Store

Suppose we manage a small thrift store shop and maintain a database of our inventory. Our database is split into two tables “clothing” and “shoes”. Our “clothing” table has columns “price”, “size”, “brand”, “category”, “gender” where category is “top” or “bottom”. Our “shoes” table has columns “price”, “size”, “brand”, and “gender”. Now suppose our thrift store is thriving and we receive daily shipments of inventory that is logged onto the database before being put out for sale. As the owners of the thrift store we know that some of the inventory we receive is worth much more than the retail price and we have often picked out a couple items for ourselves. However, per company policy and that of our distributors, we are obliged to put all of our inventory for sale before we purchase any of it. We are not always supervising the store and we don't log the inventory so it is impossible for us to determine when juicy inventory is out for sale. Luckily we have a database of the inventory managed

via SQL where we can view what is up for sale instead of going shelf by shelf.

Not super important for our purposes but here is how our tables were created in SQL:

```
/* Create a table called clothing */
CREATE TABLE clothing(price integer, size integer,
    brand text, category text, gender text);
/* Create a table called shoes*/
CREATE TABLE shoes(price integer, size integer, brand
    text, category text, gender text);
```

and here is how our inventory is updated:

```
INSERT INTO clothing VALUES(8,'L', 'Old Navy', 'bottom',
    , 'women');
INSERT INTO clothing VALUES(9,'S', 'GAP', 'bottom',
    'women');
INSERT INTO clothing VALUES(2,'M', 'Levy', 'bottom',
    'women');
INSERT INTO clothing VALUES(10,'S', 'Levy', 'bottom',
    'women');
INSERT INTO clothing VALUES(13,'L', 'Levy', 'bottom',
    'women');
INSERT INTO clothing VALUES(12,'L', 'Nike', 'bottom',
    'women');
INSERT INTO clothing VALUES(17,'L', 'Nike', 'bottom',
    'women');

INSERT INTO shoes VALUES(20, 10, 'Nike', 'men');
INSERT INTO shoes VALUES(22, 11, 'Nike', 'men');
INSERT INTO shoes VALUES(10, 12, 'Nike', 'men');
INSERT INTO shoes VALUES(13, 8, 'Adidas', 'men');
INSERT INTO shoes VALUES(27, 9, 'Adidas', 'men');
INSERT INTO shoes VALUES(17, 11.5, 'Adidas', 'men');
INSERT INTO shoes VALUES(29, 12, 'Puma', 'men');
INSERT INTO shoes VALUES(20, 7, 'Puma', 'men');
```

## SQL Commands

At its core, a SQL command is structured as: SELECT columns WHERE columns meet the imposed conditions GROUPed by a column HAVING an imposed limitation ORDERed by a column/condition. You first filter the columns then you filter the rows, followed by aggregating the data, limited the data and ordering the results. We will learn about this structure in this post, so don't worry if it sounds complicated.

### Select (Querying Columns)

Let's synthesize the above: The SELECT clause allows us to "select" data from our database given a column name and a table. Instead of an index or row and column, SQL allows us to directly access the database if we know the column names of the tables in the database and if we know the [datatype](#) of each column. (Note: this is very similar to the select() function from the 'dplyr' package)

Back to our example, we know that we have two tables in our database, "clothing" and "shoes", and we know their columns. Before the store opens, we want to check our inventory for both clothing and shoes. The following SQL command returns our current inventory:

```
select * from clothing;
select * from shoes;
```

And the following is returned

6	M	Nike	top	men
8	M	Nike	top	men
15	L	Nike	top	men
5	L	Adidas	top	men
10	S	Adidas	top	men
7	S	Puma	top	women
9	S	GAP	top	men
5	M	Puma	top	men
3	XL	Express	bottom	men
8	L	Old Navy	bottom	women
9	S	GAP	bottom	women
2	M	Levy	bottom	women
10	S	Levy	bottom	women
13	L	Levy	bottom	women
12	L	Nike	bottom	women
17	L	Nike	bottom	women
20	10	Nike		men
22	11	Nike		men
10	12	Nike		men
13	8	Adidas		men
27	9	Adidas		men
17	11.5	Adidas		men
29	12	Puma		men
20	7	Puma		men
31	7	Converse		men
20	8	Nike		men
14	13	Nike		men
25	11	Nike		men
15	11	Nike		women
19	8	Adidas		women
22	9	Adidas		women

where the first column is price followed by size, brand, category (for clothing only), and gender. The "" takes place of the COLUMN placeholder in the skeleton command in this example and it means "get all the columns" of this table. But now suppose we only wanted certain columns, say we only want to view the available brands. We can replace "" and specify a column name, for example

```
select brand from clothing;
```

where brand is a column of clothing and we get:

```
$sqlite3 database.sdb < main.sql
```

```
Nike  
Nike  
Nike  
Adidas  
Adidas  
Puma  
GAP  
Puma  
Express  
Old Navy  
GAP  
Levy  
Levy  
Levy  
Nike  
Nike
```

Just like that, boom! We are querying the data that we need with simple lines and without specifying an index. The SQL syntax is fairly straightforward and intuitive once you have become familiar with it. The neat property is that it reads like a sentence: “select column from table”. Additionally, we can specify multiple columns of the same table, for example we might want to query the cost and brand column of the shoes table:

```
select cost,brand from shoes;
```

We’ve learned to filter columns (select them), we will now proceed to filter rows under a condition(s).

## Where (Filtering Rows)

Suppose now we want a more complicated query. Our example database is relatively small, in the case that we ran a huge thrift store the queries mentioned earlier might return thousands upon thousands of columns. We might want to be more specific in our queries. Suppose we are both fanatics of the “Nike” brand and we often find mint condition shoes in our inventory. So now we want to query our database to return only the Nike branded shoes. From our previous SQL command structure: select column from table, we will add “WHERE CONDITION”. That is, WHERE is a clause (like Select) and CONDITION is a logical condition on the columns that filter the returned rows. From the previous example “select \* from shoes” our new query will be:

```
SELECT * from shoes WHERE brand == 'Nike';
```

Here our condition is “brand == 'Nike'” and the following rows are returned:

```
$sqlite3 database.sdb < main.sql
```

```
20 | 10 | Nike | men  
22 | 11 | Nike | men  
10 | 12 | Nike | men  
20 | 8 | Nike | men  
14 | 13 | Nike | men  
25 | 11 | Nike | men  
15 | 11 | Nike | women
```

However, let’s assume that this query still returns too many rows and we specifically just want to view Nike shoes that are equal to or more than \$20. We add the condition “price >= 20” to our query and we get:

```
SELECT * from shoes where brand == 'Nike', price >= 20;
```

```
$sqlite3 database.sdb < main.sql
```

```
20 | 10 | Nike | men
22 | 11 | Nike | men
20 | 8 | Nike | men
25 | 11 | Nike | men
```

We conditioned our query to return the columns we wanted and rows that fit the condition(s). (Note: this is very similar to `filter()` from the “dplyr” package)

## GROUP BY (grouping columns), HAVING, and ORDER BY

We progress into more complicated and perhaps more useful SQL queries. We have filtered our columns and our rows, now we will group our columns. Before proceeding it is important to note that these clauses do not all have to go in one SQL query, we use the components of the SQL skeleton command as needed. As a reminder the general structure of the SQL query we have learned so far is:

```
select col1,col2... from table1,table2.. where CONDITION1,CONDITION2..
```

We will now add on to this query and our general structure will be:

```
select col1,col2... from table1,table2.. where CONDITION1,CONDITION2.. group by col3, having CONDITION3 order by
CONDITION4 ASC | DESC
```

The GROUP BY clause groups columns, the HAVING clause is similar to the WHERE clause except that it is now used with the aggregate function GROUP BY, and finally the ORDER BY clause is used to sort the result-set (or columns) by descending or ascending order. Back to our example, suppose we wanted to know how many items of each brand we had in our inventory. In order to query that result we want to use the GROUP BY clause to group all the items under a “brand” (a column) and keep a count of them. SQL has several [functions](#) that are imposed to columns. In our scenario, we will make use of the `count()` function that returns the number of rows that match our criteria, the brand. If we run the following query:

```
select count(brand), brand from clothing GROUP BY brand;
```

The following is returned:

```
$sqlite3 database.sdb < main.sql
```

```
2 | Adidas
1 | Express
2 | GAP
3 | Levy
5 | Nike
1 | Old Navy
2 | Puma
```

where the query groups columns with the same brand, proceeds to count the aggregate columns and return that selection. Similarly, we could have grouped our columns or any other column like “size” or “gender”. (Note: the GROUP BY clause is very similar to `group_by()` from the “dplyr” package).

We proceed to slowly build up our (elegant) query and impose a condition on our aggregate query. Suppose now that we want to query the brands that have less than or equal to 2 items in our inventory. (Who knows we might find a rare and expensive piece of clothing or an unusual brand). We again, add to our query the HAVING clause followed by a condition, where our condition here is: `count(brand) <= 2`

```
select count(brand), brand from clothing group by brand HAVING COUNT(brand) <=2;
```

The following is returned:

```
$sqlite3 database.sdb < main.sql
```

```
2 | Adidas
1 | Express
2 | GAP
1 | Old Navy
2 | Puma
```

As a reminder the HAVING clause is not necessary to proceed with the aggregate query [by aggregate query I am referring to the joining of columns under the GROUP BY clause], it is a clause that helps to narrow down the data that you are trying to fetch. The HAVING clause does not have to be condition on the columns that are selected, it can be on any column as long as it is part of the aggregate query and can use the aforementioned SQL [functions](#). However, it is important to note that the HAVING clause is not a direct substitute of the WHERE clause, the

HAVING clause is used as a direct result of the aggregating cause by the GROUP BY clause. To finalize our basic SQL querying, we will add to our query an ORDER BY clause that is independent from the previous aggregation. That is, we can utilize the ORDER BY CLAUSE in a query as simple as:

```
select * from shoes ORDER BY brand, price;
```

resulting in the following:

```
$sqlite3 database.sdb < main.sql
```

```
13 | 8 | Adidas | men
17 | 11.5 | Adidas | men
19 | 8 | Adidas | women
21 | 6 | Adidas | women
22 | 9 | Adidas | women
27 | 9 | Adidas | men
27 | 7 | Adidas | women
5 | 5 | Converse | women
31 | 7 | Converse | men
34 | 7 | Converse | women
10 | 12 | Nike | men
14 | 13 | Nike | men
15 | 11 | Nike | women
20 | 10 | Nike | men
20 | 8 | Nike | men
22 | 11 | Nike | men
25 | 11 | Nike | men
7 | 4 | Puma | women
8 | 5 | Puma | women
20 | 7 | Puma | men
29 | 12 | Puma | men
32 | 9 | Puma | women
55 | 9 | UGG | women
60 | 8 | UGG | women
```

Which orders our query by first the brand (in alphabetical order) and then into price. Now if we wanted to list either (or both) the brand and price by a certain order (ASCENDING or DESCENDING) we can with the DESC and ASC key words:

```
select * from shoes ORDER BY brand DESC, price ASC;
```

```
$sqlite3 database.sdb < main.sql
```

```
55 | 9 | UGG | women
60 | 8 | UGG | women
7 | 4 | Puma | women
8 | 5 | Puma | women
20 | 7 | Puma | men
29 | 12 | Puma | men
32 | 9 | Puma | women
10 | 12 | Nike | men
14 | 13 | Nike | men
15 | 11 | Nike | women
20 | 10 | Nike | men
20 | 8 | Nike | men
22 | 11 | Nike | men
25 | 11 | Nike | men
5 | 5 | Converse | women
31 | 7 | Converse | men
34 | 7 | Converse | women
13 | 8 | Adidas | men
17 | 11.5 | Adidas | men
19 | 8 | Adidas | women
21 | 6 | Adidas | women
22 | 9 | Adidas | women
27 | 9 | Adidas | men
27 | 7 | Adidas | women
```

Or we can use the ORDER BY clause in a more complex and complete query that returns the items of each brand in our inventory that have more than or equal to 2 ordered by their count in ascending order :

```
SELECT count(brand), brand FROM clothing GROUP BY brand HAVING count(brand) >= 2 ORDER by count(brand) DESC;
```

```
$sqlite3 database.sdb < main.sql
```

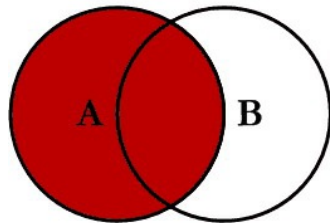
```
5 | Nike
3 | Levy
2 | Adidas
2 | GAP
2 | Puma
```

## Joins

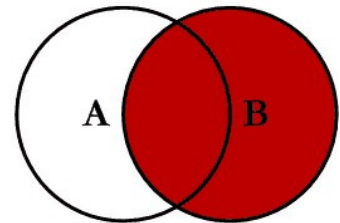
A key utility of SQL is the ability to query data by joining tables. For a more complete overview of SQL joins [here](#) is a good source. We will use the following graphic as reference.



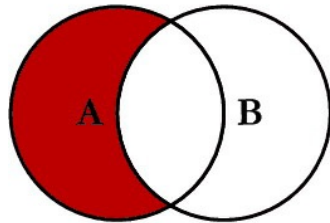
# SQL JOINS



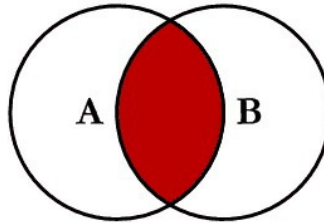
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



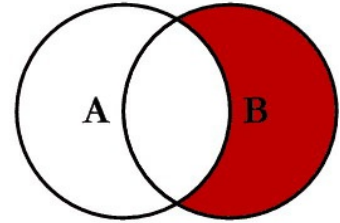
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



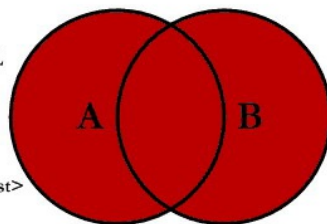
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



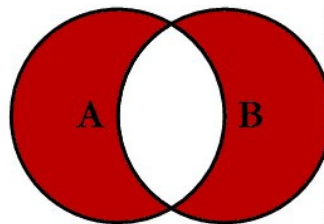
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

As shown above in the Venn Diagrams there are multiple types of joins in SQL. For simplicity we will only be focusing on the Right Join illustrated at the top most of the reference image. Supposed we have two tables one of our “clothing” inventory and a new table “transactions” with columns “purchaseDate”, “price”, “item”, “itemID”. The “transactions” table keeps track of what items have been sold, when, the price they were sold for, and all items have a unique itemID. From our previous examples, we have also modified the “clothing” table to include an “itemID” for each item. The itemID’s for both tables correspond to the same items. Now, suppose we wanted to know which items have been sold. We have a table that keeps track of our inventory and another of the transactions so how can we go about JOINING these two tables to query what we want? Taking a look at the top right most Venn Diagram, we see that when A and B are joined with RIGHT JOIN the rows that are queried back are those that are part of B and those where B and A intersect. Back to our example, A would be the “clothing” table and B would be the “transactions” table since we don’t want to query items in A that are not in B, or items that have not been sold.

The construction of our “transactions” table and our new “clothing” table:

```
CREATE TABLE transactions(purchaseDate integer, price integer, item
text, itemID integer);

CREATE TABLE clothing(price integer, size integer, brand text,
category text, gender text, itemID integer);
```

Now suppose the tables were filled accordingly and we ran the following query:

```
select * from clothing RIGHT JOIN transactions B where A.itemID = B.itemID;
```

The above query translates to: Select all the columns from clothing where the itemID of the clothing table and the itemID of the transaction table match. We perform a Right Join since we do not want to query the items that have not been sold. Now you might notice a syntactical change, the A after clothing and B after transactions. Referencing A and B is a form of table prefix that is used so that the join runs successfully (think of it as a maintenance practice), this is not to be confused with [aliasing](#). The different joins are used to query different data from different tables and to condition them to restrict the query. [Note: SQL Join shares some ground with the merge\(\) R function.](#)

[Note: In this particular example, INNER JOIN would have also worked!]

## SQL in R

That is SQL! It allows us to query a lot of data with a single command (and communicate with a (relational) database). So why in the world do I want to use SQL in R when I can just use SQL on its own environment? We don’t use Python to use Java or use Python to use Matlab for example, or do we? The “sqldf” [package](#) supports a SQL backend that allows us to use the previous SQL commands in R with the sqldf() function. Sqldf() takes in a string that is equivalent to the previous SQL query examples. For example, to query all of the inventory in our shoes table we would call sqldf(“select \* from shoes”) and to query only the shoes that are branded Nike with a cost of 20 or more we would call sqldf(“select \* from shoes where cost >20 and brand == ‘Nike’”). All of the SQL queries we have just learned can be applied to the sqldf() function. For further reference and more advanced usage refer to the [documentation](#) and for more examples refer [here](#) and [here](#).



```

SELECT * FROM clothing;
SELECT * FROM shoes;

select * from shoes where brand == "Nike" and price >=20;

SELECT category FROM clothing where brand == 'Nike';

SELECT count(brand), brand FROM clothing GROUP BY brand HAVING count(brand) >=
  2 ORDER by count(brand) DESC;

```

```

sqldf("SELECT * FROM clothing")
sqldf("SELECT * FROM shoes")

sqldf("select * from shoes where brand == \"Nike\" and price >=20")

sqldf("SELECT category FROM clothing where brand == 'Nike'")

sqldf("SELECT count(brand), brand FROM clothing GROUP BY brand HAVING count
(brand) >= 2 ORDER by count(brand) DESC")

```

Above is the comparison between simple SQL queries from our example and the same queries used in R with sqldf() package. In essence, if you know SQL this package may come in handy if you find the SQL querying more intuitive than say the “dplyr” package, if not then this is a good starting to learn about the SQL world and all its derivatives. Happy Querying!

## Conclusion

Are you still not convinced that you may want to give SQL a try or SQL in R? It narrows down to preference. Perhaps you are more comfortable with SQL than you are with R and using the “sqldf” improves your efficiency. Perhaps you use R for features that aren’t in SQL but would still like to use some core SQL feature and database keeping. Or Maybe you want to learn SQL and using the “sqldf” package will help you in your journey. The ‘sqldf’ package is a tool that may improve your productivity or simply give a refreshing change to your working environment. There are many parallels between the ‘dplyr’ package and the utility of the ‘sqldf’ package that allow you to pick based off of preference and comfort. That is the beauty of libraries, that they personalize our development experience while facilitating our productivity. Perhaps SQL isn’t for you, but I would encourage you to explore other packages in any programming language you are familiar with. I encourage you to find packages that can be a “replacement” for packages you are currently using; you never know; your productivity might increase or you might find it dreadful but at the end you will have learned to appreciate the power that libraries give to our programming languages.

## Citations

<https://github.com/ggrothendieck/sqldf#example-4-join> <https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html>  
<https://www.w3schools.com/sql/> <https://www.tutorialspoint.com/sql/>  
[http://www.cs.toronto.edu/~nn/csc309/guide/pointbase/docs/html/htmlfiles/dev\\_datatypesandconversionsFIN.html](http://www.cs.toronto.edu/~nn/csc309/guide/pointbase/docs/html/htmlfiles/dev_datatypesandconversionsFIN.html)  
[https://www.w3schools.com/sql/sql\\_count\\_avg\\_sum.asp](https://www.w3schools.com/sql/sql_count_avg_sum.asp) [https://www.w3schools.com/sql/sql\\_alias.asp](https://www.w3schools.com/sql/sql_alias.asp)  
<https://www.statmethods.net/management/merging.html> <https://cran.r-project.org/web/packages/sqldf/README.html#examples>  
[https://jasdumas.github.io/tech-short-papers/sqldf\\_tutorial.html#join\\_queries](https://jasdumas.github.io/tech-short-papers/sqldf_tutorial.html#join_queries) <https://www.r-bloggers.com/use-sql-to-operate-r-data-frames/>