A Comparative Analysis of the use of SQLite and dplyr for Data Analytics

1. Introduction

Both the SQLite[1] and dplyr[2] packages both have a multitude of beneficial and detrimental attributes when used for data analysis using the programming language r. Relational databases such as those implemented using SQLite provide many advantages such as data manipulation abilities and serverless, easily-implemented connection as well as the well know, terse query language, SQL. Dplyr[2] mainly places its focus on the use of data frames for the query and analysis of data, with a specific attention to the pipeline function (%>%) which will be detailed further throughout this report. Dplyr[2] features specifically optimised methods written in Rcpp[3] allowing for C++ integration in r, resulting in increased performance speed when compared to base r. This raises the question of speed vs organisation. It is this, usability, and applications of each package that will be detailed and cross examined in this report.

2. Observational Differences

When importing data from the .csv file the processes is relatively similar for each package. Data is loaded using the read.table() method for each package into either the table or dataframe. The main difference occurs when creating the database table for SQLite[1]. A CREATE TABLE query is required which allows the declaration of column datatypes resulting in enforced data restrictions for the table [0]. One advantage to this is the adhesion to the ACID properties required for a relational database. The enforcement of strict datatypes allow checks for data consistencies before manipulation or aggregation can occur, allowing users to check for abnormalities or other unexpected issues with the data. Conducting such checks using dplyr[2] or R is possible, however they are not required or as intuitive as the implementation in SQLite[1]. SQLite[1] also trumps dplyr[2] when automatically incrementing the SS_ID column via the use of a primary key, and the AUTOINCREMENT statement which allows for easy consistent indexing, however the obligation of a primary key may be considered a disadvantage based on the level of organisation needed. The SS_ID was incremented in dplyr[2] using the rownames() function to initiate the SS_ID to the row number, which only required slightly more code than the SQLite[1] alternative.

```
dbSendQuery(con,
            "CREATE TABLE census_income
        (SS_ID INTEGER PRIMARY KEY AUTOINCREMENT,
        AAGE INTEGER,
```

Figure 1: Demonstrating the strict column type, primary key and the AUTOINCREMENT function.

When creating tables and loading data using the SQLite[1] package, the database is created in memory rather than using disk space [1] . The main advantage of this is performance, due to the faster read and write times using RAM when compared to secondary storage devices. Similarly dplyr[2] boasts “blazing fast performance for in-memory data”[2] meaning both packages start off on somewhat equal ground in regards to the analysis task and speed. Fortunately for speed, the census data [4] provided fits into memory and the use of disk storage is not required although it is possible in both packages.

Both packages provide different frameworks for the filtering, aggregating and sorting of data. Queries for task 1 using dplyr were mainly composed using the Forward-Pipe Operator to pipe objects into functions or expressions, allowing a chain of data manipulation.

```
##Mean income for each race
print(incomeByRace<-select(data, ARACE, AHRSPAY, WKSWORK) %>%
        group_by(ARACE) %>%
        summarise(avgIncome =mean(AHRSPAY * 40)) %>%
        arrange(desc(avgIncome)))
```

Figure 2: Dplyr query to select the mean income for each race using the forward pipe-operator.

As seen in Figure 2, the pipeline operator is used to pass objects, in this case an in-memory dataframe to the filtering or aggregation functions such as select(), group by() and summarise(). The pipeline operator allows similar functionality to SQL queries used without the constrains of the SQLite syntax, providing an easily readable, flexible structure for data analytic queries.

SQLite queries provide a set syntax for data retrieval and aggregation, following the traditional framework SELECT * FROM Database WHERE *Conditions* (See figure 3) Initially for the simple query example above the SQLite[1] declarative, English like syntax proves easily read and comprehended when compared to the Dplr[2] query, especially without prior knowledge of the

```
#Selecting the average pay for each race
print(dbGetQuery(con, "SELECT AVG(AHRSPAY*40),ARACE
                       FROM census_income
                       WHERE AHRSPAY !=0
                       GROUP BY ARACE"))
```

Figure 3: SQLite query to select the mean income for each race using the forward pipe-operator.

pipe operator. However, when looking at more advanced data queries the intent and capabilities of each package become evident. Drawing on an example excluded of task 1, calculating the median using each package is very different.

In dplyr[2], the median is simply calculated via the use of the summarise function (Figure 4) in one line of code; demonstrating the power and flexibility of the summarise function in conjunction with pipelining data. Built in functions are also available for typical operations such as max(), min() and mean which were used throughout task 1.

```
data.raw %>% group_by(date) %>% summarise(median = median(count, na.rm = TRUE))
```

Figure 4: Calculating the median using dplyr

SQLite[1] lacks a distinct median function; instead convoluted, verbose code is required which lacks inherent readability therefore creating code which is harder to maintain, update and alter after development, further highlighting the flexibility and increased functionality of the summarise() function.

A similar disparity is displayed in the difference between the alteration of data via either the mutate() function for dplyr[2] or the ALTER table equivalent for SQLite[1]. In Part 1 task 2 the SS_ID needed to be appended and incremented after the table creation. The SQLite package required a full declaration of the table including all columns to and data types to add a new column which is time consuming and arduous especially when compared to the simple use of the mutate function in dplyr[2]. Mutate() simple accepts the dataset to be edited, the column name and the contents of the new column which is greatly preferred.

```
SELECT AVG(x)
FROM (SELECT x
      FROM MyTable
      ORDER BY x
      LIMIT 2
      OFFSET (SELECT (COUNT(*) - 1) / 2
              FROM MyTable))
```

Figure 5: Calculating the median when the number of records is even using SQLite[1]

Task 5 included the creation of 3 new, separate tables from the existing census data table. The implementation for dplyr[2] is extremely intuitive using the pipe operator to select the desired columns from the original data. SQLite[1] is also intuitive however requires slightly more code using the CREATE TABLE AS …. FROM original_table.

For the completion of task 6, for both packages the inner join method needs to be utilised in order to draw on and filter data which occurs in multiple tables. Inner join was specifically chosen to draw records based on their SS_ID that occur in each table. After implementing the new tables and conducting an inner join for both packages problems occurred with the column names when using dplyr[2]. Default suffixes are applied to columns which are seen as duplicates in the creation of the join, meaning column names were appended with .x or .y which caused issues with the retrieval of data. In an attempt to circumvent this, a character vector is passed in the inner_join method enabling new columns to be appended with our suffix of choice, in this case - (_new) (See figure 6). Implementation of the inner join using SQLite[1] was seamless and required no renaming of columns or otherwise edits.

```
#Created a combined table using inner join
combinedTbl<-data %>% inner_join(personDf, payDf,
  by= 'SS_ID',suffix = c("_original", "_new")) %>%
  inner_join(., jobDf, by= 'SS_ID', suffix = c("_original", "_new") )
```

Figure 6: Using a character vector to circumvent the default suffixes applied by dplyr

While extracting figures such as the maximum wage in question 6. Dplyr[2] was specifically useful with the implementation of the dollar sign, allowing specific extractions of columns without the use of select which would be required in SQLite[1]. Furthermore, the ability to pass results from previous queries into filters or aggregates proves unequivocally useful in contrast to SQL which would require a SELECT statement and therefore more computational resources (If the data has already been extracted once) before a filter can be applied.

Another benefit to using Dplyr[2] for data analysis (although not featured in Task 1) is that it utilises non-standard evaluation [6]. Non-standard evaluation allows expressions such as names, constants, calls and pairlists to be captured and saved for later, rather than evaluating expressions straight away at runtime. Hadly Wickham states the benefits of non-standard evaluation for data analysis which is "particularly useful for functions when doing interactive data analysis because it can dramatically reduce the amount of typing" [7]. Lazy, non-standard evaluation is possible using SQLite[1], however the innate, ease of use with the implementation in dplyr[2] is preferred. It is also noteworthy that with the introduction of non-standard evaluation hinderances do occur. In dplyr[2] references are not referentially transparent [6] meaning values cannot be replaced with another equivalent object which is elsewhere defined.

3. Computational Intensity

To assess the computational efficiency and speed of the dplyr[2] package and the SQLite[1] package the microbenchmark function [10] was used to repeat the selected operation "x" number of times and provide "Sub-Millisecond Accurate Timing Of Expression Evaluation"[10]

First and foremost, all the methods used to get the data to a position where analysis can be carried out were compared after 100 iterations. This includes creating the table or dataframe and adding the columns and indexing using the SS_ID. Although the code needed for SQLite[1] and dplyr[2] are different, the idea behind this test is to check the time required to get to a point where analysis can be carried out, regardless of the different amount of code.

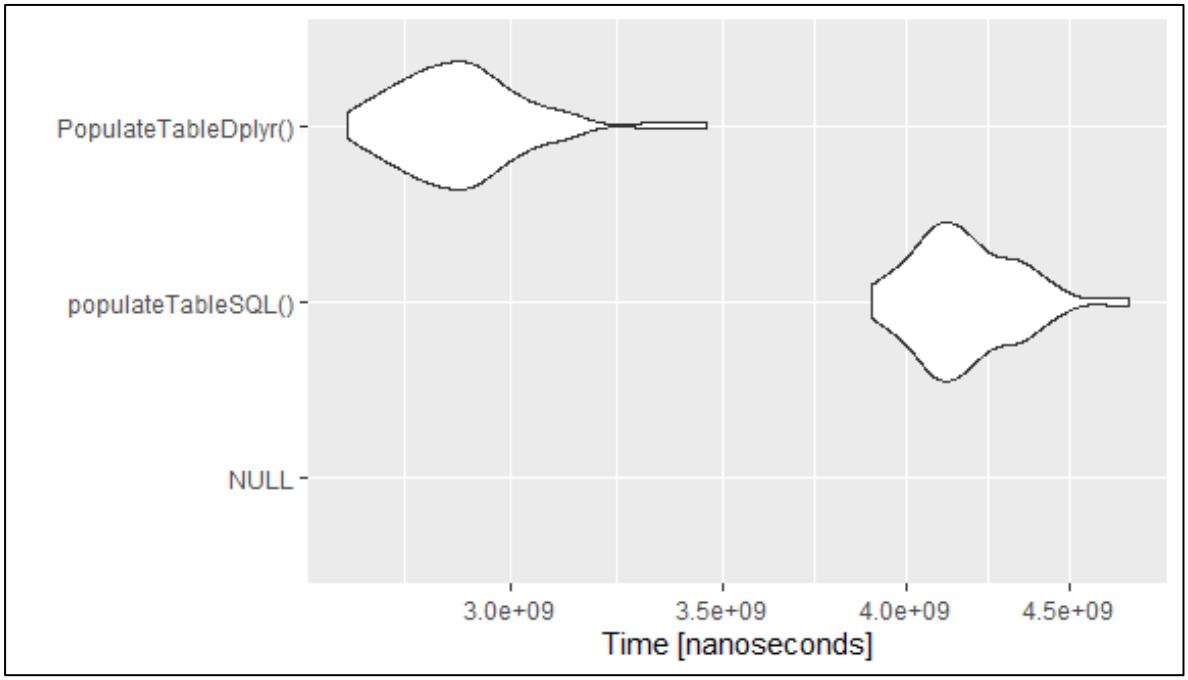

Figure 7: The difference in distribution and speed between the PopulateTableDplyr() method and the populateTableSQL() method.

As seen in figure 7, after 100 iterations dplyr[2] was seen significantly faster at populating dataframes when compared to SQLite[1] populating tables. This could be due to numerous factors such as the simplicity of dataframes due to the lack of strict types, relational features of the SQLite table or simple due to the code efficiency of the packages. Nonetheless dplyr[2] outperforms SQLite[1] for this test; while there is not a large difference for the dataset used in task 1, problems

with speed could occur with bigger datasets meaning packages need to be chosen very carefully before analysis.

For the next test, queries from question 6 were compared and contrasted. The results shown in figure 8 once again demonstrated dplyr's superior speed when compared to SQLite[1]. An even greater differential was displayed than that in the populateTable() test highlighting the efficiency of dplyr[2] queries compared to SQL queries.

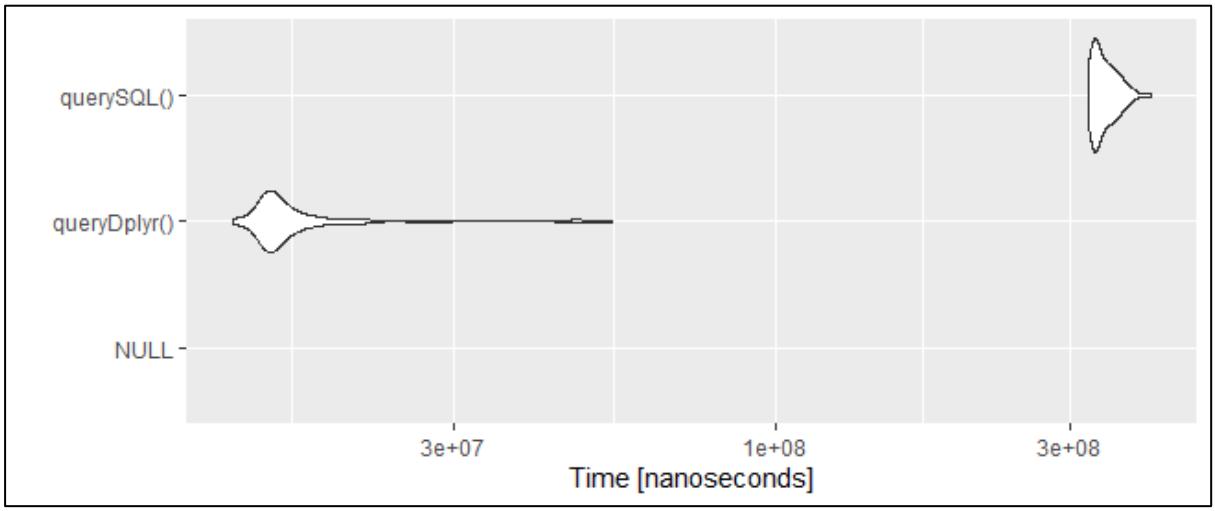

Figure 8: Benchmark results for question 6. ii.

The previous query investigated was then broken down to further provide further introspection into the differential in efficiency between the two packages and their methods. To start, the select method from each package was isolated. To ensure a fair test, the join statement was also included for dplyr[2], although it is not required in the query itself.

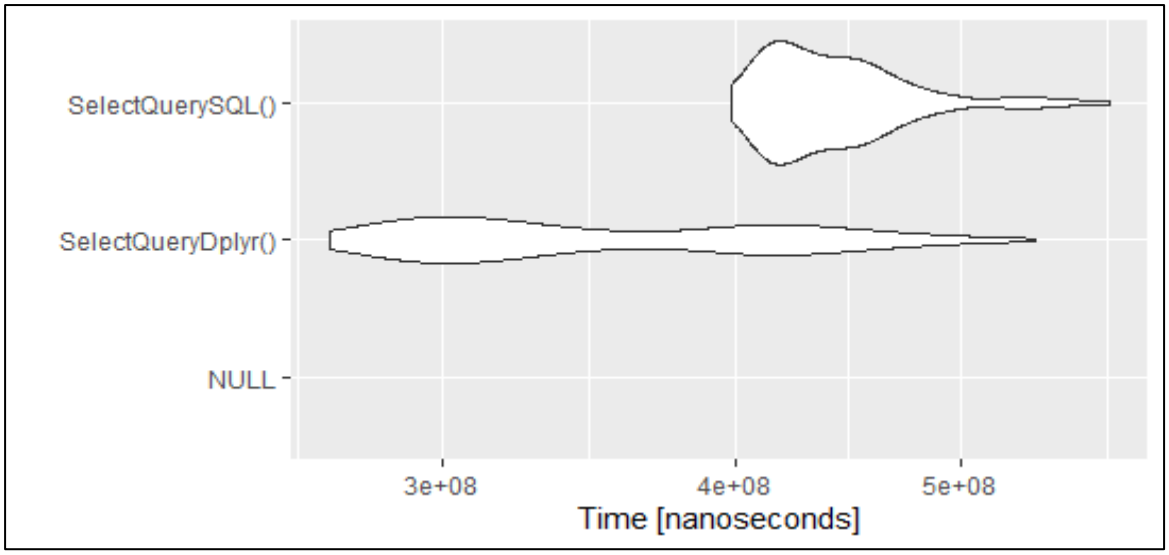

Figure 9: Benchmark results for selecting the data from question 6.ii., including the join statement for both packages.

Interestingly when interpreting the results seen in figure 9, the distribution between SQLite[1] and dplyr[2] differs greatly. On average dplyr[2] preforms the fastest, however has greater variability

in execution time. After the addition of filtering the select data the difference in speed lowered, suggesting SQLite[1] is efficient for filtering data but not as efficient at selecting data.

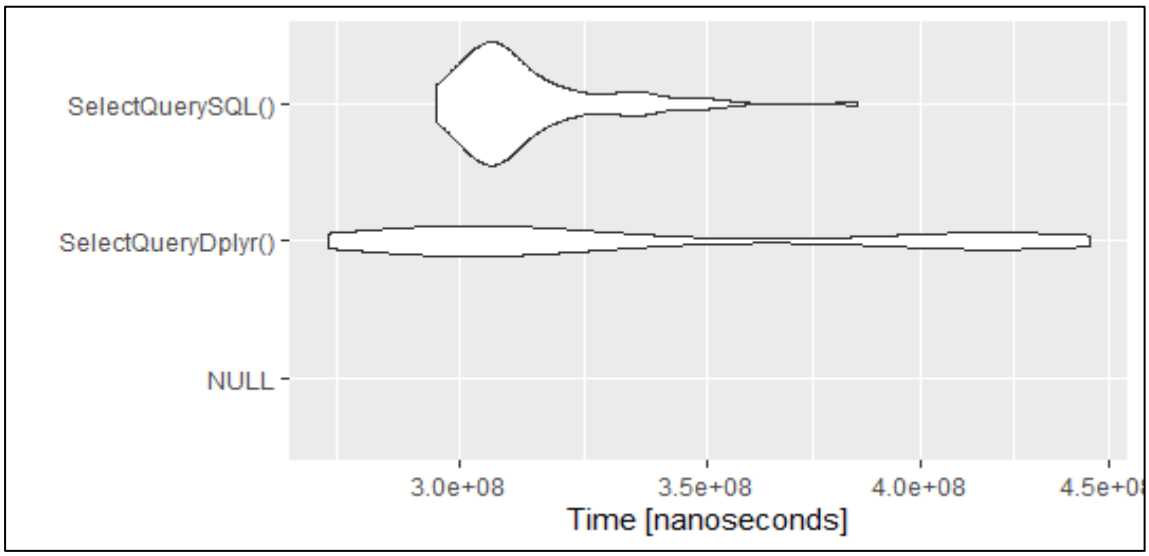

Figure 10: Benchmark results including the filter and select statement from question 6. ii.

Finally, the last test conducted (figure 11) was a general efficiency test investigating the speed of execution for the full code of task 1 for each package. Dplyr completed all the tasks on average faster than SQLite

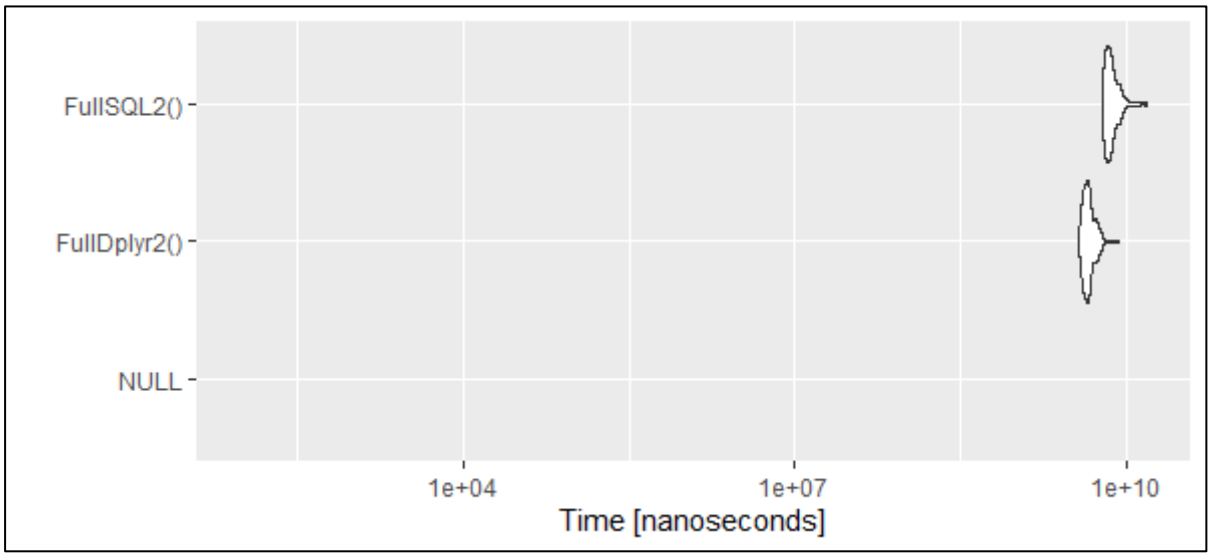

Figure 11: Benchmark results for the full analysis of task 1.

The results above mostly support dplyr's claim for lighting fast performance with SQLite[1] being outperformed in all tests featured above. Based on this, a simplistic conclusion can be drawn, in regards to the data and operations provided in task 1 dplyr[2] provides faster data processing and analysis. However, It is important to consider the inconsistency of execution time on dplyr's [2] side and the rudimentary basis of the tests conducted. In order to further support this decision additional tests and investigations would be required, including a greater number of iterations for each method or function as individuals.

4. Discussion

Both SQLite[1] and dplyr[2] have their place in data analytics, however the difference between them is substantial. Traditionally with SQL being developed to query large, relational databases and not specifically for data analytics, often the rigid syntax and query structure provide an obstacle for more complex analysis tasks. Dplyr's[2] implementation of the forward pipe operator and the utilisation of flexible dataframes and vectors prove superior during both programming and execution, especially for one-off complex analysis. SQLite[1] truly shines when periodic multi-user queries and analysis are required. Furthering this, the mostly simplistic syntax and predefined arithmetic operators provide a quick start up for elementary queries.

Based on the functionality required for task one, if the data fits into memory and the benefits of the SQLite[1] discussed above are not required, dplyr[2] would be preferred to SQLite[1]. The preferential difference drawn from within this report is attributable to dplyrs[2] merits in speed, flexibility and its efficacy for powerful analysis as seen during the investigation of task 1.

5. References

[1] Wickham, H., James, D. A. & Falcon, S. (2014). RSQLite[1]: SQLite[1] Interface for R. URL http://CRAN.R-project.org/package=RSQLite[1]. R package version 1.0.0.

[2] Wickham H, Romain F. (2014) dplyr[2]: a grammar of data manipulation. R package version 0.2. Available: http://cran.r-project.org/web/packages/dplyr[2].

[3] Allaire, J. J., Eddelbuettel, D., and François, R. (2017), Rcpp Attributes, vignette included in R package Rcpp, Vienna, Austria: CRAN. Available: www.rproject.org/pub/R/web/packages/Rcpp/vignettes/Rcpp-attributes.pdf

[4] C L Blake, C J Merz. UCI repository of machine learning databases University of California, Irvine, Department of Information and Computer Sciences. 1998. Available : http://rexa.info/paper/A3DDA1F6F088B349EC0530194F768E08DC86BEF7

[5] Rdocumentation.org. (2019). *microbenchmark function | R Documentation*. [online] Available at: https://www.rdocumentation.org/packages/microbenchmark/versions/1.4-7/topics/microbenchmark

[6] Dplyr.tidyverse.org. (2019). *Programming with dplyr*. [online] Available at: https://dplyr.tidyverse.org/articles/programming.html [Accessed 19 Dec. 2019].

[7] Wickham, H. (2014). Advanced R. CRC Press.