

**The Department of Computer Science**

**Design of Complex Computer System**

CIS3157 CW2 – Complex Systems

Students:

Peter Gallant 23691077

Niall Davis 23546468

Luke Hilditch 23727764

Sean Weights 23746271

Table of Contents

[Section 1 Introduction 2](#_Toc39061862)

[Section 2 Complex System Design 3](#_Toc39061863)

[2.1 Explanation of System Design 4](#_Toc39061864)

[2.1.1 Hadoop Distributed File System (HDFS) 4](#_Toc39061865)

[2.1.2 MapReduce 4](#_Toc39061866)

[2.1.3 Spark 4](#_Toc39061867)

[2.1.4 SparkSQL 5](#_Toc39061868)

[2.1.5 Scala 5](#_Toc39061869)

[2.1.6 Command Line Interface 5](#_Toc39061870)

[Section 3 Complex System Implementation 6](#_Toc39061871)

[3.1 Command Line Interface Main Menu 6](#_Toc39061872)

[3.2 Functionality One 6](#_Toc39061873)

[3.3 Functionality Two 7](#_Toc39061874)

[3.4 Functionality Three 8](#_Toc39061875)

[3.5 Functionality Four 8](#_Toc39061876)

[3.6 Functionality Five 9](#_Toc39061877)

[3.7 Functionality Six 9](#_Toc39061878)

[3.8 Functionality Seven 10](#_Toc39061879)

[3.9 Functionality Eight 10](#_Toc39061880)

[3.10 Functionality Nine 11](#_Toc39061881)

[3.11 Functionality Ten 11](#_Toc39061882)

[3.12 Functionality Eleven 12](#_Toc39061883)

[3.13 Functionality Twelve 13](#_Toc39061884)

[3.14 Functionality Thirteen 14](#_Toc39061885)

[3.15 Functionality Fourteen 15](#_Toc39061886)

[3.16 Functionality Fifteen 15](#_Toc39061887)

[Section 4 Critical Analysis and Discussion of The Designed System 16](#_Toc39061888)

[Section 5 Conclusion 20](#_Toc39061889)

[References 21](#_Toc39061890)

[Appendix 24](#_Toc39061891)

[Appendix i Table of Contribution 24](#_Toc39061892)

Table of Figures

[Figure 1 High Level design of Complex System 3](#_Toc39061729)

[Figure 2 CLI Main Menu 6](#_Toc39061730)

[Figure 3 Number of reviews sorted by rating 6](#_Toc39061731)

[Figure 4 Price range of places 7](#_Toc39061732)

[Figure 5 Users by job title 8](#_Toc39061733)

[Figure 6 Frequency of three letter phrases 8](#_Toc39061734)

[Figure 7 Number of reviews per place 9](#_Toc39061735)

[Figure 8 Reviews made by job role 9](#_Toc39061736)

[Figure 9 Places which are closed or not closed 10](#_Toc39061737)

[Figure 10 Results of Inverted Index 10](#_Toc39061738)

[Figure 11 Average rating of each category 11](#_Toc39061739)

[Figure 12 Top five categories sorted by average rating 11](#_Toc39061740)

[Figure 13 Price range of closed places 12](#_Toc39061741)

[Figure 14 Average rating of reviews by price range 13](#_Toc39061742)

[Figure 15 Number of reviews by price range 14](#_Toc39061743)

[Figure 16 Average character length of reviews by rating 15](#_Toc39061744)

[Figure 17 Place with the most reviews 15](#_Toc39061745)

[Figure 18 Scala Code showing Scala integrating with Spark SQL 16](#_Toc39061746)

[Figure 19 Java code showing how to connect and use SQL with Java (Shah, 2020) 17](#_Toc39061747)

[Figure 20 Table showing computing time in minutes of each function 18](#_Toc39061748)

# Section 1 Introduction

The purpose of this report is to demonstrate a range of techniques in designing, developing and evaluating a real-world complex system in the context of a Google local data engineering team. Google Local requires an end to end complex system to efficiently store, process and analyse data of three given data sets. The Hadoop components that will be examined in this report will cover the Hadoop Distributed File System (HDFS), Spark and Spark SQL.

HDFS is an open-source file system that allows for distributed storage of large data sets which can then be used for large scale distributed data processing. Data is stored in 64mb blocks using the master-slave architecture, which involves a name node which controls what the data nodes do, store and retrieve (Babu et al., 2013).

Spark is an in-memory processing framework (Gurusamy et al., 2017). It can be used for either batch-processing or real-time analytics. Following similar principles to another Hadoop module called MapReduce, but one hundred times faster, this is due to in-memory operations (Apache, 2020).

Spark SQL is the batch processing module that is used to execute SQL queries that allow for data analytics and processing to be completed. Spark SQL uses MapReduce to complete the breakdown of the dataset so that it may be processed in batches that allow for the data to be processed quickly and efficiently (Armbrust et al., 2015).

The system should allow for storing, querying and automatically analysing the three datasets and be operated using a command-line interface which allows for the execution of these different functions. As detailed above, each Hadoop component of HDFS, Spark and Spark SQL have been used in the complex system.

The rest of this report is prepared as follows: Section 2, will show a high-level architectural design, showing the main components used in the complex system. It will detail the interaction between the components and how the designed complex system has been implemented to solve the given problem. Section 3 will list the requirements and show the results of these requirements performed using the complex system, along with a description of how each requirement has been completed. Section 4 provides a comparative analysis of the time each requirement needs to be computed in the complex system, along with a critical analysis of the components used, when compared against other Hadoop components. Finally, Section 5 will conclude and summarise the findings of this report.

# Section 2 Complex System Design

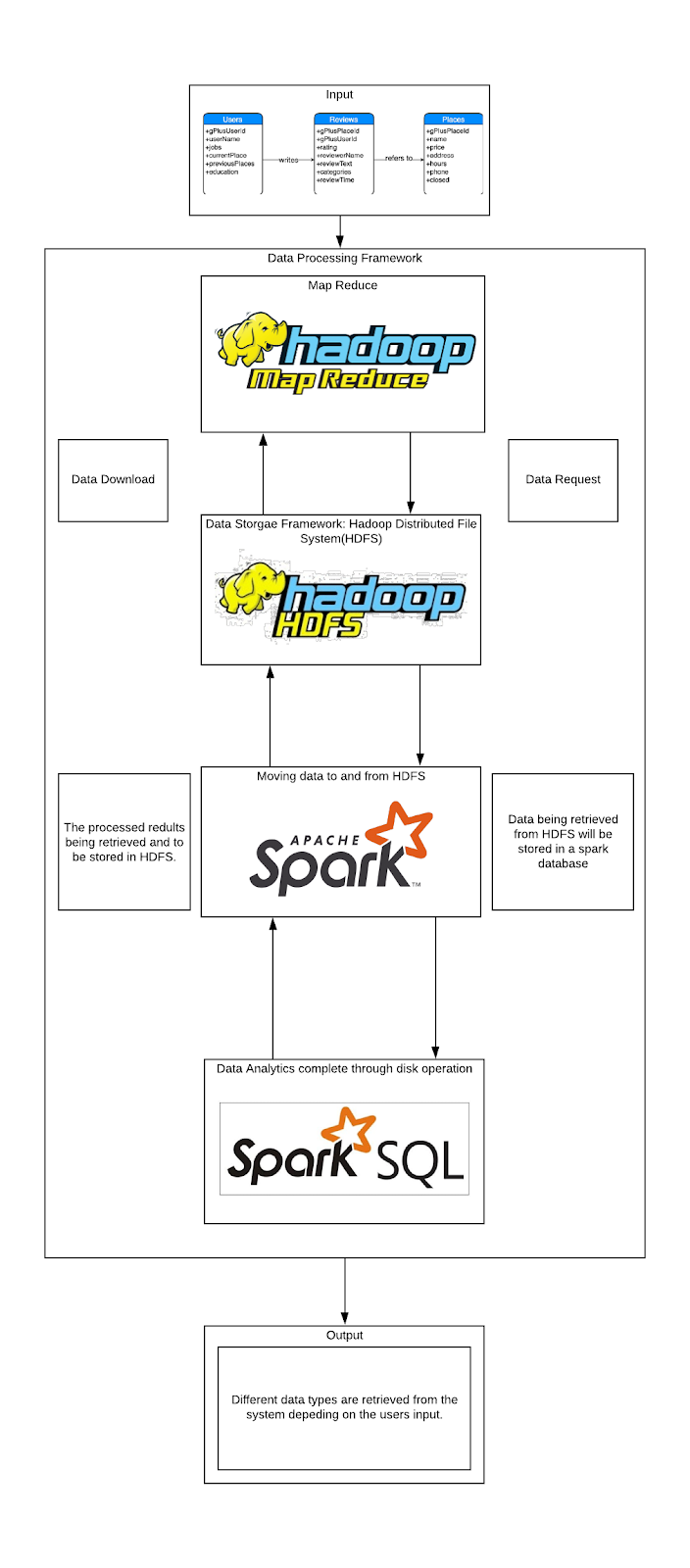


Figure 1 High Level design of Complex System

## 2.1 Explanation of System Design

Hadoop was created specifically for big data operations (Agoulmine et al., 2019; Chandakanna, 2018; Tao et al., 2019), widely used in a variety of clusters for "large-scale, high-performance systems" (Babu et al., 2013: 1). The design for this task, shown in figure 1, utilises the Hadoop framework. Also using components that make up its architecture, like the Hadoop Distributed Files System (DeRoos, 2014).

### 2.1.1 Hadoop Distributed File System (HDFS)

HDFS is an open-source file system (Agoulmine et al., 2019) is used to control how data is stored and retrieved (Butterfield et al., 2016). It has been primarily chosen for this build because not only is it built for running on commodity-based clusters (Babu et al., 2013), but it has a high fault-tolerance to battle data corruption (Borthakor, 2008). High fault-tolerance is especially important in big data operations because of the amount of data that will be processed by the complex system. In the event of a machine shutting down and losing data, HDFS will have copies of the data on another computer within the cluster, hence high fault-tolerance. HDFS will talk to other components in the system by receiving data requests from components like Spark and then upload the necessary data. Any new data will be stored within the HDFS.

### 2.1.2 MapReduce

MapReduce is a Hadoop component built for batch processing (Shahrivari, 2014). It is used in this task to process and manage large-scale datasets (Li et al., 2014) flowing to and from the HDFS. Allowing for other components to download/upload data to/from the HDFS. MapReduce works by mapping and reducing, hence 'MapReduce'.

Mapping will perform filtering and sorting operations by splitting the data into chunks and spreading the tasks out to multiple machines (DataNodes) within the cluster. Reducing is where the DataNodes will solve the task by aggregating solutions into sub-tasks on the rest of the available computers (DeRoos, 2014).

### 2.1.3 Spark

Spark is a processing framework that can be used to perform in-memory processes (Gurusamy et al., 2017), like real-time analytics and batch-processes. In this instance, Spark will perform batch-processing as the provided dataset isn't suitable for real-time analytics. Therefore, processing will be performed on disk.

The primary use for Spark in this task is to store data in a database (Gurusamy et al., 2017) which allows for higher speeds. Data stored within the Spark database will be accessed by another component called Spark SQL to run SQL scripts on the data.

### 2.1.4 SparkSQL

Spark SQL is an additional module added to Apache Spark that integrates relational processing with functional programming. This allows for Spark programmers to gain the benefits of relational processing which is one of the main reasons it was used in this system as it allows for data to be retrieved using well-known SQL statements (Armbrust et al., 2015). Spark is easier to use for querying data as errors are thrown to the user in the system while writing queries instead of causing errors when the code is executed.

### 2.1.5 Scala

Scala is also good at avoiding bugs in complex applications. It is an "object-oriented and functional programming" (Scala, 2020) language combined into one. Using its vast ecosystem of libraries, Scala and its data frames will be used to store the results set from a given query which is then used to display the output to the user (Dayananda, 2020). This will be used in our system to run queries that will retrieve data and show the results back to the user through the command-line interface.

### 2.1.6 Command Line Interface

Bash will be used as the main command-line interface. It will allow the user to interact with the operating system through commands in a bash script (Milligan, 2018). When the script is run, the commands are executed. By default, bash is available on Linux and macOS operating systems. It is used in this scenario to execute Spark SQL queries on the data stored within the system.

# Section 3 Complex System Implementation

In this task, the complex system must be able to complete ten out of the fifteen given requirements. These requirements are shown in the main menu of the Command Line Interface (CLI). The GoogleLocal datasets are loaded into a Spark database, and the data will be executed on by various Spark SQL queries to extract information pertaining to the different functionalities of the system. The user interacts with the Complex System using the CLI that can be seen in figure 2.

## 3.1 Command Line Interface Main Menu

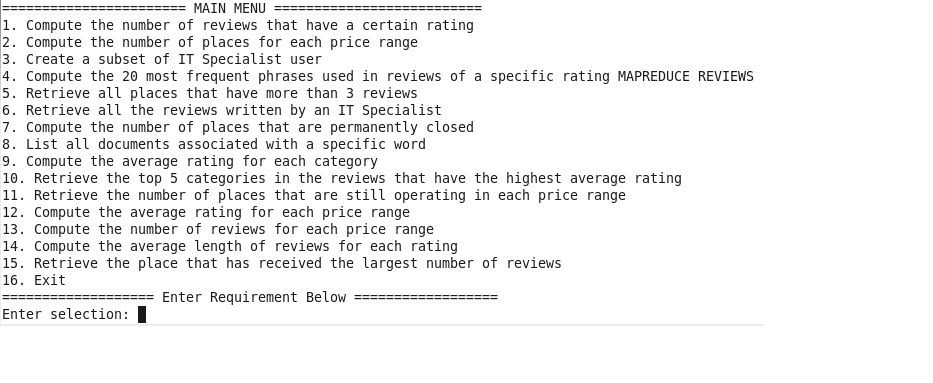


Figure 2 CLI Main Menu

## 3.2 Functionality One

Functionality one runs a Spark SQL query to grab the number of reviews with each rating (1, 2, 3, 4, 5) and stores the results in a file (one file for each rating). The user will input which rating they want to view, the system then grabs the number from the file that it is stored in and outputs it see figure 3.



Figure 3 Number of reviews sorted by rating

## 3.3 Functionality Two

Functionality two executes a Spark SQL query that counts the number of places in the *places* dataset, using the GROUP BY parameter to group the number of places by each price range. The query will then get all the prices which have $, $$ and $$$, the results are placed in a data frame and outputted seen in figure 3.

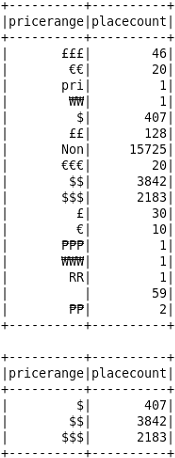


Figure 4 Price range of places

## 3.4 Functionality Three

Functionality three executes a Spark SQL query that selects all the rows in the user's dataset where their job is equal to "IT Specialist". The query creates a data frame containing the information and outputs the resulting rows to the user, shown in figure 5.

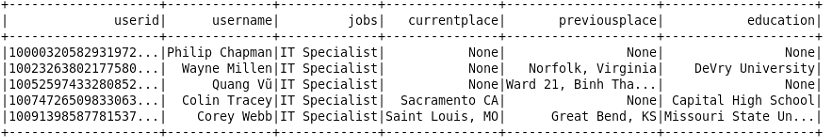


Figure 5 Users by job title

## 3.5 Functionality Four

Functionality four calculates the frequency of each phrase in every review, where a phrase is equal to a string of three words. The java program does this by splitting each review into an array of words; it loops through the array of words and for each word stores the phrase beginning with the selected word and made up of the two words after it in the array. It does this for each set of three words, creating a new record if it has not appeared before, or adding to the count of the existing file if it already exists. This results in an index of each phrase against the number of times it occurs in the entire set of reviews. The system grabs the dataset for phrases in reviews of a user inputted rating. It then displays the results in descending order for the user; figure 6 shows the results for rating 1 and 5.

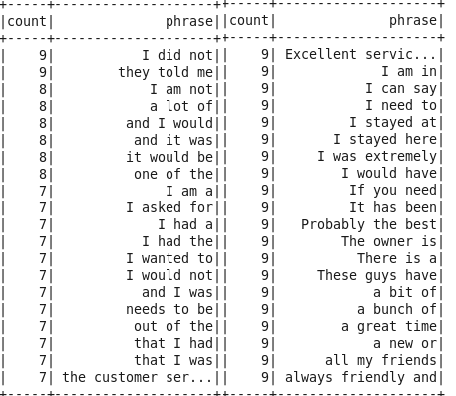


Figure Frequency of three letter phrases

## 3.6 Functionality Five

Functionality five executes a Spark SQL query on the reviews dataset that retrieves the number of rows for a specific place by grouping the results by the unique *placeid*. The results are ordered by descending number of reviews and placed in a data frame to be outputted to the user, see figure 7.

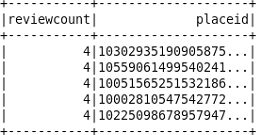


Figure Number of reviews per place

## 3.7 Functionality Six

Functionality six executes a Spark SQL query on both the reviews and users dataset using an inner join to connect the two tables where the attribute *userid* is the same in both tables. From the tables, the system gets the ID of the user that made the review, the review itself and the profession of the user that wrote the review. The query filters the results only to retrieve reviews made by users with the job "IT Specialist". The results are stored in a data frame, and it outputs the result to the user, see figure 8.

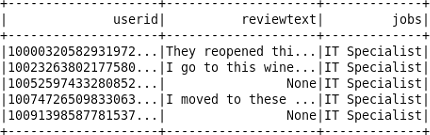


Figure Reviews made by job role

## 3.8 Functionality Seven

Functionality seven checks if the results have been calculated yet, if not the system runs a Spark SQL query that selects the number of places from the *places* dataset that are permanently closed and the number of places that are open, which depends on the *closed* attribute in the dataset. The results are saved in the HDFS and are grabbed by the system and displayed to the user, shown in figure 9.



Figure Places which are closed or not closed

## 3.9 Functionality Eight

Functionality eight runs a MapReduce program on the reviews dataset to create an inverted index. The functionality maps each word alongside the *reviewid* and *userid* attributes of the reviews it appears in, resulting in a new word identifier. The user is then able to give the system a word which will then search for the word in the results document. The system will retrieve the line which contains the given word and all the identifiers of the reviews it appears in; below is an example of a keyword search through the inverted index for the word 'quantities'.



Figure Results of Inverted Index

## 3.10 Functionality Nine

Functionality nine runs a Spark SQL query on the *reviews* dataset that selects each category and the average rating of each of the reviews for that category. The results are placed into a data frame and outputted to the user, shown in figure 11.

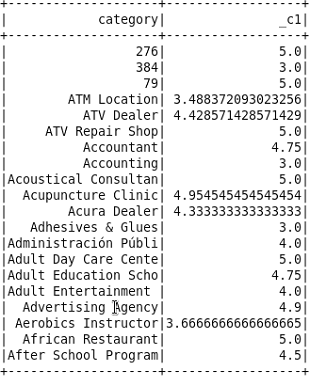


Figure Average rating of each category

## 3.11 Functionality Ten

Functionality ten executes a Spark SQL query that grabs the average rating for each category, shown in figure 12, but orders them by descending average and only outputs the top 5 results.

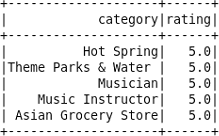


Figure Top five categories sorted by average rating

## 3.12 Functionality Eleven

Functionality eleven executes a Spark SQL query that grabs the number of places in each price range that are permanently closed. The query filters the rows by the *closed* attribute and groups them by price range, stores the results in a dataset. Figure 13 shows the output to the user.

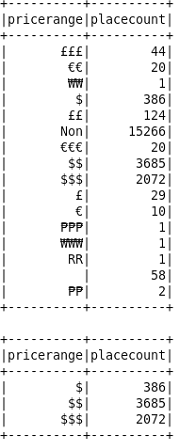


Figure Price range of closed places

## 3.13 Functionality Twelve

Functionality twelve executes a Spark SQL query that selects the average rating of reviews in each price range. The query selects these from the places and reviews datasets, inner joining the two tables where the *placeid* attribute is the same. The system places these results into a data frame and outputs it to the user shown in figure 14.

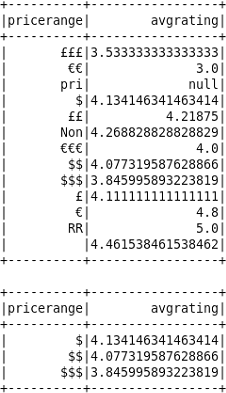


Figure Average rating of reviews by price range

## 3.14 Functionality Thirteen

Functionality thirteen executes a Spark SQL query on the places and reviews datasets, joining the two tables where the *placeid* attribute is the same in both. The query selects the number of reviews for each price range by getting every review and grouping them by their price range. The results are stored in a data frame and outputted to the user, shown in figure 15.

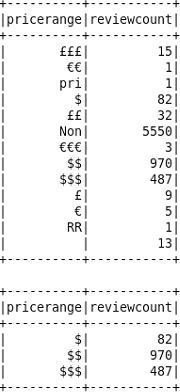


Figure Number of reviews by price range

## 3.15 Functionality Fourteen

Functionality fourteen executes a Spark SQL query on the reviews dataset that selects the average length of each review in each rating. For each rating, the query gets the average of each length of the *reviewtext* attribute that holds the review, and groups the results by each rating. The results are stored in a data frame and outputted to the user, seen in figure 16.

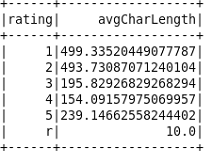


Figure Average character length of reviews by rating

## 3.16 Functionality Fifteen

Finally, functionality fifteen executes a Spark SQL query on the reviews and the places dataset; it retrieves the number of reviews for a specific place by grouping the results by the unique placeid, while inner joining the places dataset to get the name of the place as well. The results are ordered by descending number of reviews and placed into a data frame, which will have the top row outputted to the user. This means the user sees the place with the most amount of reviews, shown in figure 17.

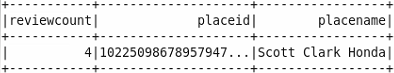


Figure Place with the most reviews

# Section 4 Critical Analysis and Discussion of The Designed System

The task was to create a complex system to enable the GoogleLocal team to store, process and analyse reviews of locations. The solution designed for this task was created to give the Google Locals team a system that was generally fast and accurate yet could also handle a vast dataset.  As has been shown in sections two and three, the complex system is predominantly based upon a Spark core framework utilising Spark SQL, and its inbuilt Resilient Distributed Dataset. Using Spark SQL has allowed the data frames created to be reused again and again (Samadi, Zbakh and Tadonki, 2017). It has also used MapReduce to create an inverted index of words from the reviews uploaded by the users**.**

The reviews are uploaded by users and include a rating system using numbers between one and five. The system uploads and stores the data using the Spark SQL framework; it can do this in HDFS as Spark is built on top of Hadoop architecture (Omar and Jumaa, 2019). By using a BigSQL framework, the data was able to be stored in its raw format and processed according to accurate entries. However, when data is stored in this way, there will inevitably be errors, such as grammatical, type case and unknown formats. So, even though the data is stored in its raw format, it should be cleaned and prepared so that analysis tasks can be performed upon it. For all the requirements, baring functionality eight, data preparation is not a problem for this task.

To carry out the requirements using the complex system, the user must utilise the CLI. As shown in section three, figure 2, the user is greeted with a menu asking for input. This menu, along with the code for the functionalities, was written using the Scala programming language.

Scala was specifically chosen as the Apache Spark framework is written in this language. Scala is a Java Virtual Machine (JVM) based language (Poslavsky, 2019), which means when the code is compiled, it is compiled into Java bytecode. Because of this property, the complex system can be accessed by any client machine that has a JVM program.

One of the main benefits of using Scala is that the requirements were able to access parts of the Spark framework which are unavailable in the SparkAPI used by other languages such as Java and Python (Omar and Jumaa, 2019).  A further benefit to using Scala was the amount of coding needed. With Spark being created in Scala, the amount of code necessary to create a program is significantly less than using another language (Omar and Jumaa, 2019) as seen in figures 18 and 19.

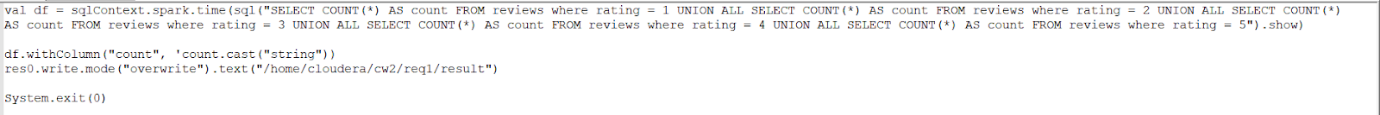
**

Figure Scala Code showing Scala integrating with Spark SQL

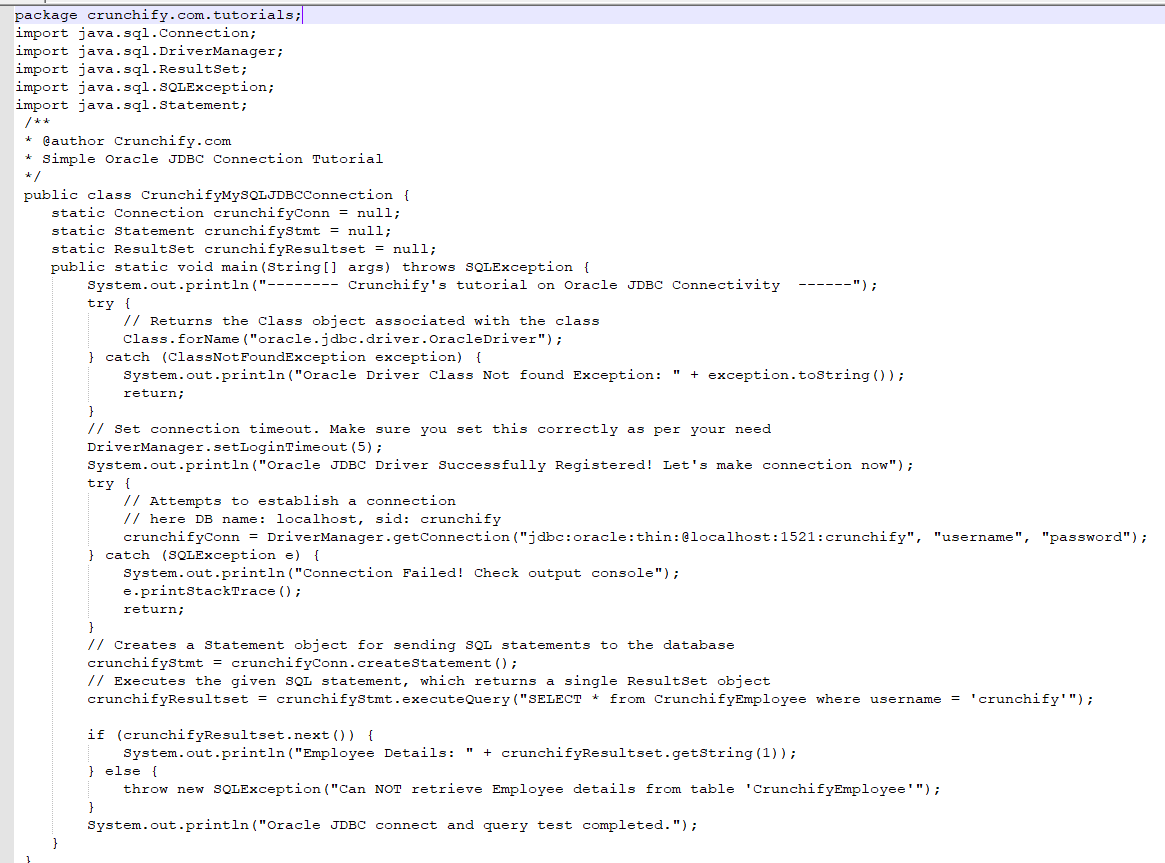


Figure Java code showing how to connect and use SQL with Java (Shah, 2020)

By analysing both code blocks above, it can be seen that Scala was the correct choice both for time and ease of use. By interfacing with the SparkSQL framework directly rather than through an external connection, as would be the case using the Java Database Connection, the user will spend less time interacting with the system.

|  |  |
| --- | --- |
| **Functionality** | **Time in Minutes and Seconds** |
| Functionality 1 | 0:0001.0 |
| Functionality 2 | 0:26.1 |
| Functionality 3 | 0:27.0 |
| Functionality 4 | 0:17.84 |
| Functionality 5 | 0:32.6 |
| Functionality 6 | 0:26.0 |
| Functionality 7 | 0:0001.0 |
| Functionality 8 | 173:27.0 |
| Functionality 9 | 0:27.3 |
| Functionality 10 | 0:28.6 |
| Functionality 11 | 0:29.0 |
| Functionality 12 | 0:28.4 |
| Functionality 13 | 0:28.2 |
| Functionality 14 | 0:27.6 |
| Functionality 15 | 0:31.0 |

Figure Table showing computing time in minutes of each function

By examining the data in figure 20, it can be seen, all the requirements utilising Scala are, on average, completed in twenty-three seconds. Even though it has previously been identified that SparkSQL and Scala are fast, see figure 20, and easy to use in the designed system, there are faults. The main fault is that SparkSQL suffers from a reliability issue (Ji. et al., 2020). During a single session, the more data that is read and written can adversely affect the time it takes for these tasks to be completed, thus increasing the computing power needed for a simple job.

MapReduce has been chosen to complete the requirement of building an inverted word index. This Hadoop framework was selected as it has been proven to be reliable and can handle extensive datasets, which is why it is used by social media companies (Liu et al., 2016). Once the data had been cleansed the results, as shown in figure 10, are incredibly accurate. This has been demonstrated that MapReduce is still an excellent system to use.

However, to confound the Spark SQL reliability issue, MapReduce also has a massive problem with time. The MapReduce requirement requires just under three hours to complete. For an extensive data system which is defined by the three V's, Volume, Variety and Velocity (Niebel, Rasel and Viete, 2018), the time this requirement takes is too long.

The identified time issue with MapReduce is well known. It is caused as MapReduce only runs over the data once and then provides results. As has been defined, MapReduce is accurate over large datasets. However, with requirement eight of this task, when new reviews are added, and this requirement is rerun, the time will only increase.

Hadoop Hive and Impala are other systems that could have been utilised. However, as Identified in Chang, Tsai and Lee (2018), each of these systems also has their faults. Again, the flaws with these frameworks are time as they also use the MapReduce framework to conduct the data analysis. Even though both Hive and Impala can support SQL queries like Spark, they are all computed on disk, thus increasing computational time, in this task that could have considerably increased the computational time. Whereas, Spark uses in-memory computation, thus providing faster speeds.

# Section 5 Conclusion

In conclusion, weighing up the advantages and disadvantages of this complex system, an HDFS was decided on for use in this project., mainly because it can be applied to a cluster and perform large scale distributed processing. The system will also use Spark to store data in memory which will allow for faster data processing. By using the data frames available in the SparkSQL, the system can use its in-memory capabilities, rather than if the data that was to be processed using on-disk operations, in a database, as can be seen in requirement eight. MapReduce will also be used within this system to break down the tasks to be stored and processed across the cluster.

As shown in the implementation, the solution designed for this task is successful and has met all fifteen requirements. Although there were some problems faced. Requirement eight, of which required MapReduce, produced time constraint issues. Taking just under three hours to deliver a solution meant that applying this to larger datasets would require more computational power to get more efficient results, even though MapReduce is built for large datasets. Similarly, Spark SQL had reliability issues that originate from the lack of computational power. These problems could be resolved through the use of Yarn which would allow more in memory and MapReduce to be carried out, which could lead to a more efficient method to be found. Another solution could be an increase in the size of the cluster using vertical scalability and increase the hardware capabilities of the cluster to allow for more processing power to be given to the task of MapReduce.

In summary, all fifteen requirements have been met. The system is not optimal but works as it should. Implementing future ideas and knowledge would allow for more efficient and practical solutions.

# References

AGOULMINE, N., EL-MAWASS, N., MALLAT, I. and TAHER, N., 2019. An IoT Based Solution for Real-Time and Batch Processing of Big Data: Application in Healthcare. In: *3rd International Conference on Bio-engineering for Smart Technologies (BioSMART)*, 24 – 26 April 2019, Paris. USA: Institute of Electrical and Electronics Engineers. [online]. pp. 1-8. Available from: <https://doi.org/10.1109/BIOSMART.2019.8734185> [Accessed 06 April 2020].

APACHE SPARK, 2020. *Apache* Spark*™ - Unified Analytics Engine For Big Data* [online]. Available from: <https://spark.apache.org/>  [Accessed 06 April 2020].

ARMBRUST, M., BRADLEY, J., FRANKLIN, M., GHODSI, A., HUAI, Y., KAFTAN, T., LIAN, C., LIU, D., MENG, X., XIN, R. and ZAHARIA, M., 2015. Spark SQL: Relational Data Processing in Spark. In. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 31 May - 04 June 2015. Melbourne. New York: Association for Computing Machinery. [online]. pp. 1383-1394. Available from: <https://dl.acm.org/doi/pdf/10.1145/2723372.2742797> [Accessed 06 April 2020].

BABU, C., CHANDRASEKAR, S., DAKSHINAMURTHY, R., PRABAVATHY, B. and SESHAKUMAR, P., 2013. A novel indexing scheme for efficient handling of small files in Hadoop Distributed File System. In: *International Conference on Computer Communication and Informatics*, 4 – 6 January 2013. Coimbatore. USA: Institute of Electrical and Electronics Engineers. [online]. pp. 1-8. Available from: <https://doi.org/10.1109/ICCCI.2013.6466147> [Accessed 06 April 2020].

BAKER, J. and MILLIGAN, I., 2014. *Introduction To The Bash Command Line* [online]. Available from: <https://programminghistorian.org/en/lessons/intro-to-bash>  [Accessed 06 April 2020].

BUTTERFIELD, A., KERR, A. and NGONDI, G., 2016. *A Dictionary of Computer Science.* [eBook] 7th ed. Oxford: Oxford University Press. Available from: <https://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-6361> [Accessed 06 April 2020].

BORTHAKUR, D. 2008. *The hadoop distributed file system: Architecture and design* [eBook]. USA: Apache Software Foundation. Available from:<http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.16.1/docs/hdfs_design.pdf> [Accessed 06 April 2020].

BOUTABA, R., GONG, Z., LIU, Y., LIU, Z. and ZHANG, Q. 2016. ROUTE: run-time robust reducer workload estimation for MapReduce. *International Journal of Network Management* [online]. 26 (3), pp. 224-244. Available from: <https://doi.org/10.1002/nem.1928> [Accessed 22 April 2020].

BROWN, B., COSS, R., DEROOS, D., MELNYK, R. and ZIKOPOULOS, P., 2014, *Hadoop for dummies* [eBook]*.* Hoboken: John Wiley & Sons. Available from: [https://ebookcentral.proquest.com/lib/edgehill/reader.action?docID=1658809#](https://ebookcentral.proquest.com/lib/edgehill/reader.action?docID=1658809) [Accessed 06 April 2020].

CHANDAKANNA, V., 2018. REHDFS: A random read/write enhanced HDFS. *Journal of Network and Computer Applications* [online]. 103, pp. 85 –100. Available from: <https://doi.org/10.1016/j.jnca.2017.11.017> [Accessed 06 April 2020].

CHANG, B., TSAI, H. and LEE., Y., 2018. Integrated High-Performance Platform for Fast Query Response in Big Data with Hive, Impala and SparkSQL: A Performance Evaluation. *Applied Sciences* [online]. 8 (9), pp. 1514. Available from: <https://doi.org/10.3390/app8091514> [Accessed 24 April 2020].

DAYANANDA, S., 2019. Spark SQL Tutorial - Understanding Spark SQL With Examples. *Apache Spark and Scala*. [Blog online]. 22 May. Available from: <https://www.edureka.co/blog/spark-sql-tutorial/> [Accessed 6 April 2020].

GURUSAMY, V., KANNAN, S. and NANDHINI, K., 2017. The Real Time Big Data Processing Framework: Advantages and Limitations. *International Journal of Computer Sciences and Engineering* [online]. 5 (12). pp. 305-312. Available from: <https://doi.org/10.26438/ijcse/v5i12.305312> [Accessed 06 April 2020].

JI, X., WU, Q., ZHAI, M. and ZHAO, M., 2020. Query Execution Optimisation in Spark SQL. *Scientific Programming* [online]. 2020 (2020), article ID 6364752. Available from: <https://doi.org/10.1155/2020/6364752> [Accessed 22 April 2020].

JUMAA, A. and OMAR, H., 2019. Big Data Analysis Using Apache Spark MLlib and Hadoop HDFS with Scala and Java. *Kurdistan Journal of Applied Research* [online]. 4 (1), pp. 7-14. Available from: <https://doi.org/10.24017/science.2019.1.2> [Accessed 22 April 2020].

LI, F., OOI, B., ÖZSU, M. and WU, S., 2014. Distributed data management using MapReduce. *ACM Computing Surveys (CSUR)* [online]. 46 (3), pp. 1-42. Available from:<https://dl.acm.org/doi/10.1145/2503009> [Accessed 06 April 2020].

NIEBEL, T., RASEL, F. and VIETE, S., 2018. BIG data - BIG gains? Understanding the link between big data analytics and innovation. *Journal of Economics of Innovation and New Technology* [online]. 28 (3), pp. 296-316. Available from: <https://doi.org/10.1080/10438599.2018.1493075> [Accessed 22 April 2020].

POSLAVSKY, S., 2019. RINGS: An efficient Java/Scala library for polynomial rings. *Computer Physics Communications* [online]. 235, pp. 400-413. Available from: <https://doi.org/10.1016/j.cpc.2018.09.005> [Accessed 22 April 2020].

SAMADI, Y., TADONKI, C. and ZBAKH, M., 2017. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. *Concurrency and Computation: Practice and Experience* [online]. 30 (2), pp. e4367. Available from: <https://doi.org/10.1002/cpe.4367> [Accessed 22 April 2020].

SCALA, 2020. *The Scala Programming Language* [online]. Available from: <https://www.scala-lang.org/>  [Accessed 06 April 2020].

SHAH, A., 2020. Simple Oracle Database JDBC Connect and ExecuteQuery Example in Java. *Java and J2EE Tutorials*. [Blog Online]. 16 March. Available from: <https://crunchify.com/simple-oracle-database-jdbc-connect-and-executequery-example-in-java/> [Accessed 22 April 2020].

SHAHRIVARI, S., 2014. Beyond batch processing: towards real-time and streaming big data. *Computers* [online]. *3* (4), pp.117-129. Available from:<https://doi.org/10.3390/computers3040117> [Accessed 06 April 2020].

TAO, W., TCHAYE-KONDI, J. and ZHAI, Y., 2019. LHF: A New Archive Based Approach to Accelerate Massive Small Files Access Performance in HDFS. In: *IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService)*, 4 – 9 April 2019, Newark. USA: IEEE [online]. pp. 40-48. Available from: <https://doi.org/10.1109/BigDataService.2019.00012> [Accessed 06 April 2020].

# Appendix

## Appendix i Table of Contribution

|  |  |
| --- | --- |
| **Section** | **Student Contribution** |
| Introduction | Sean - wrote a general introduction to the task including what aspect of a complex system would be included in the report and the content of the report. |
| Complex System Design | Luke completed sections: 2.1, 2.11, 2.12.  Niall completed sections: 2.15, 2.16.  Sean drew the high level architecture design.  Peter completed sections: 2.13, 2.14. |
| Complex System Implementation | Luke completed functionalities 2, 5, 9, 14.  Niall completed functionalities 3, 8, 12, 15.  Sean completed functionalities 1, 7, 11.  Peter completed functionalities 4, 6, 10, 13. |
| Complex System Evaluation | All members contributed as part of group discussion |
| Conclusions | Sean - wrote a general conclusion summarizing the report with evaluating points made in the report. |