Parallelised Data Processing

An investigation into the development of a tool for processing queries across a cluster of nodes.

Oliver Little

2011802

D.A. B.Sc Computer Science with Digital Technology Partnership (PwC)

Supervisor: Vincent Rahli

School of Computer Science

College of Engineering and Physical Sciences

April 2023

Contents

List of Figures

Abstract

Chapter 1

Introduction

1.1 Background

1.2 Prior Work

Distributed Data Processing has existed conceptually since as early as the 1970s. A key paper by Philip Enslow Jr. from this period [8] sets out characteristics across three 'dimensions' of decentralisation - hardware, control and database. Enslow argued that these dimensions defined a distributed system, while also acknowledging that the technology of the period was not equipped to fulfil the goals he laid out.

Research into solutions for distributed data processing has generally resulted in two kinds of solutions [17]:

- **Batch processing:** where data is gathered, processed and output all at the same time. This includes solutions like MapReduce [6] and Spark [18]. Batch processing works best for data that can be considered 'complete' at some stage.
- **Stream processing:** where data is processed and output as it arrives. This includes solutions like Apache Flink [5], Storm [16], and Spark Streaming [3]. Stream processing works best for data that is being constantly generated, and needs to be analysed as it arrives.

MapReduce [6], a framework introduced by Google in the mid 2000s, could be considered the breakthrough framework for performing massively scalable, parallelised data processing. This framework later became one of the core modules for the Apache Hadoop suite of tools. It provided a simple API, where developers could describe a job as a *map* and a *reduce* step, and the framework would handle the specifics of managing the distributed system.

While MapReduce was Google's offering, other large technology companies had similar solutions, including Microsoft, who created DryadLINQ in 2009 [9]. However, due to the massive success of MapReduce, Microsoft discontinued DryadLINQ in 2011.

MapReduce was not without flaws, and many papers were published in the years following its initial release which performed performance benchmarks, and analysed its strengths and weaknesses [12]. Crucially, MapReduce appears to particularly struggle with iterative algorithms, like the PageRank algorithm used by Google's own search engine. A number of popular extensions to MapReduce were introduced to improve the performance on iterative algorithms, like Twister [7] and HaLoop [4] both in 2010.

MapReduce's popularity also resulted in a number of tools being created to improve its usability and accessibility. Hive [15] is one such tool, which features a SQL-like language called HiveQL to allow users to write declarative programs that compiled into MapReduce jobs. Pig Latin [14] is similar, and features a mixed declarative and imperative language style that again compiles down into MapReduce jobs.

Further tools in the wider areas of the field were introduced around 2010, including another project by Google named Pregel [13], specialised for performing distributed data processing on large-scale graphs.

In 2010, the first paper on Spark [20] was released. Spark aims to improve upon MapReduce's weaknesses, by storing data in memory, and providing fault tolerance by tracking the 'lineage' of data. This means for any set of data, Spark knows how the data was constructed from another persistent, fault tolerant data source, and can use that to reconstruct any lost data in the event of failure. This in-memory storage, known as a resilient distributed dataset (RDD) [19] allows Spark to improve on MapReduce's performance for iterative jobs, whilst also allowing it to quickly perform ad-hoc queries. Effectively, Spark is strong at performing long batch jobs, as well as short interactive queries. This is something that I would like my solution to feature, as users of the framework will need to design long-running scripts to run on large amounts of data, as well as run ad-hoc queries to perform investigation.

Spark quickly grew in popularity, with a number of extensions being added to improve its usability, including a SQL-style engine with a query optimiser [2], as well as an engine to modify Spark to support stream processing [3]. A second paper released in 2016 [18] stated that Spark was in use in thousands of organisations, with the largest deployment running an 8,000 node cluster holding 100PB of data. One area where Spark struggles is with grouped data, as performing grouped operations requires shuffling the data between all nodes. I aim to improve upon this in my solution through the design of the system as a whole.

More recent research indicates that the future of the field is moving away from batch processing, and towards stream processing for data that is constantly being generated. A 2015 paper by Google [1] argues that the volumes of data, the fact that datasets can no longer ever be considered 'complete', along with demands for improved insight into the data means that streaming 'dataflow' models are the way forward. Google publicly stated in their 2014 'Google I/O' Keynote [10] that they were phasing out MapReduce in their internal systems. The data I will be using is not being received at this constant rate, and as such designing for a streaming solution is not required in this case.

1.3 Project Aims

Chapter 2

Design

After conducting my review of previous work, an analysis of the high-level design of the solution was conducted, in particular focusing on the following areas:

- Required Features
- Technologies and Frameworks
- Architecture

The aim of this process was to ensure that the limited development time for this project was spent developing the most effective features.

2.1 MoSCoW Requirements

Before considering specific technologies and frameworks, a list of MoSCoW requirements is produced. Each requirement in the table below has two extra columns. The first column represents whether the requirement is Functional (F) or Non-Functional (NF), and the second column represents requirements that Must (M), Should (S) or Could (C) be completed.

F / NF	Priority	Requirement Description
		Domain Specific Language - Expressions
F	M	The language must support 5 data types: integers, floats, booleans, strings and date-time objects.
F	M	The language must allow users to reference a field in the current dataset.
F	M	The language must allow users to reference a constant value, which can take one of the data types defined above.
F	M	The language must support arithmetic operations like add, subtract, multiply, division and modulo.
F	M	The language must support string slicing and concatenation.
F	S	The language should utilise polymorphism in add operations to apply string concatenation, or arithmetic addition depending on the data types of the arguments.
F	C	The language could be designed in such a way to allow the user to define their own functions.
NF	S	The language should be intuitive to use, with SQL-like syntax.
		Domain Specific Language - Comparisons

F	M	The user must be able to provide expressions as inputs to comparison operators.
F	M	The language must support equals, and not equals comparisons
F	M	The language must support inequalities, using numerical ordering for number types, and lexicographic ordering for strings.
F	M	The language must support null and not null checks.
F	S	The language should support string comparisons, including case sensitive and insensitive versions of contains, starts with, and ends with.
F	S	The language should allow the user to combine multiple comparison criteria using AND and OR operators.
Data Processing		
F	M	The system must allow the user to write queries in Python.
F	M	The system must allow users to apply Select operations on datasets, applying custom expressions to the input data.
F	M	The system must allow users to apply Filter operations on datasets, applying custom comparisons to the input data.
F	M	The system must allow users to apply Group By operations on datasets, which take a number of expressions as unique keys, and a number of aggregate.
F	M	The Group By operation must allow users to apply Minimum, Maximum, Sum and Count aggregate functions to Group By operations.
F	C	The system could allow users to apply Distinct Count, String Aggregate, and Distinct String Aggregate aggregate functions to Group By operations.
F	S	The system should allow users to join two datasets together according to custom criteria.
NF	S	The complexities of the system should be hidden from the user; from their perspective the operation should be identical whether the user is running the code locally or over a cluster.
Cluster		
F	M	The system must allow the user to upload source data to a permanent data store.
F	M	The orchestrator node must split up the full query and delegate partial work to the worker nodes.
F	M	The orchestrator node must collect partial results from the cluster nodes to produce the overall result for the user.
F	M	The orchestrator node must handle worker node failures and other computation errors by reporting them to the user.
F	S	The orchestrator should perform load balancing to ensure work is evenly distributed among all nodes.
F	C	The orchestrator could handle worker node failures by redistributing work to active workers.
F	M	The worker nodes must accept partial work, compute and return results to the orchestrator.
F	M	The worker nodes must pull source data from the permanent data store.
F	M	The worker nodes must report any computation errors to the orchestrator.
F	S	The worker nodes should cache results for reuse in later queries.
F	S	The worker nodes should spill data to disk storage when available memory is low.

2.2 Language

2.2.1 Frontend

Python was selected as the language of choice for the frontend. This is because the intended users of my solution are most experienced with Python and SQL, which should make adopting the solution faster and easier.

2.2.2 Orchestrator and Worker Nodes

Both the orchestrator and worker nodes would use the same language, which would reduce overhead as the same codebase could be used for both parts of the system. When selecting a language, a decision had to be made to use a language with either automatic or manual garbage collection (GC). Choosing a manually GC language would theoretically allow for higher performance due to more granular control over memory allocation and release. However, this could result in slower development, as time would have to be spent writing code to perform this process. Therefore, manually GC languages were ruled out.

The remaining options were a range of object-oriented and functional languages. Due to the nature of the project, much of the implementation would be CPU intensive, requiring iterating over large lists of items, so a language with strong support for parallelisation was preferable. Functional languages are strong at this because of their use of operations like *map* and *reduce*, which can be easily parallelised. This also ruled out languages like Python and JavaScript which are largely single-threaded ; both support some form of parallelisation, but much more manual intervention by the developer is required.

In the end, Scala was chosen. It features a mix of both object-oriented, and functional paradigms, The mix of programming paradigms would allow for the best option to be selected for each task to be completed. Scala has built-in support for parallelised operations and threading through the use of asynchronous operations like Futures and Promises . Furthermore, it is built on top of, and compiles into Java. This means that packages originally written for Java can be executed in Scala , which proved useful for later design decisions.

2.3 Runtime

2.3.1 Containerisation

With the nature of the project being to produce a distributed system, the clear choice for executing the code was within containers. Docker is by far the most popular option for creating images to run as containers, but is not suitable for running and managing large numbers of containers . For this, a container orchestration tool is required, and there are two main options: Docker Swarm , and Kubernetes . Docker Swarm is more closely integrated within the Docker ecosystem, and can be managed directly from Docker Desktop. However, Kubernetes is more widely used in industry and many cloud services also feature managed Kubernetes services which handle the complexity of creating and managing a cluster. For this reason, Kubernetes was chosen for the container orchestration tool.

2.3.2 Network Communication

A communication method had to be selected to allow the Python frontend, the orchestrator and the worker nodes to communicate. REST API frameworks initially appeared to be a suitable option, but upon further research, remote procedure call (RPC) frameworks would be more suited to the project's needs. This is because REST is resource-centric, providing a standard set of operations - create, read, update, delete (CRUD) . The proposed solution is more focused on operations than acting on [Reference] resources, so the CRUD model wouldn't fit the requirements correctly.

In contrast, RPC frameworks are designed to allow function calls over a remote network, while hiding the complexity of the communication from the developer . They are also typically not designed around [reference] a particular data model like REST, which would allow custom function calls to be implemented . [Reference]

The chosen framework was gRPC , which is designed and maintained by Google. Firstly, there are [Reference] well-maintained implementations for both Python and Scala, which would make using it in all parts of the solution straightforward . Secondly, it uses protocol buffers as its message system, which is a [reference] serialisation format also maintained by Google. Protocol buffers are designed to be extremely space [Reference] efficient, reducing network overhead compared to a solution that used something like XML or JSON . As protocol buffers are also a serialisation format, APIs are provided to store messages on disk. [reference]

2.4 Persistent Storage

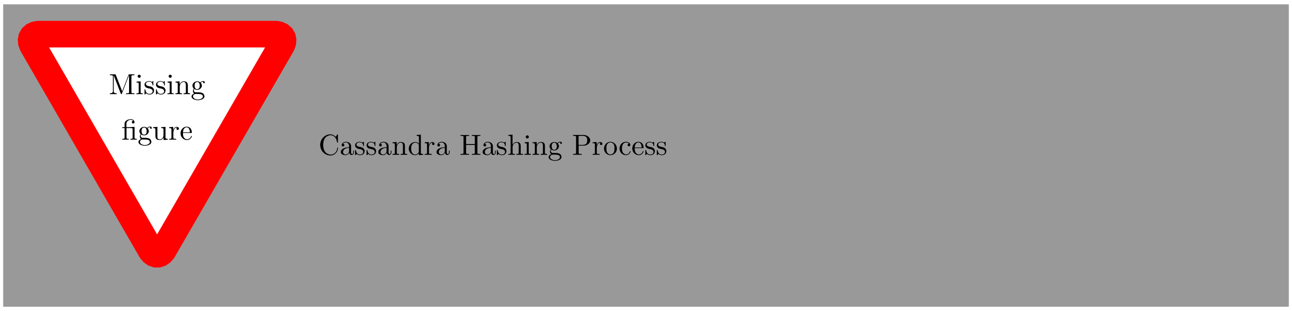
There are a wide range of options for persistent data storage. The first key decision in this area was whether to design a custom solution, or use an existing solution. A custom implementation would come with the benefit of being more closely integrated with the rest of the system, but at the cost of increased development time. A decision was made not to create a custom solution due to time constraints, and the amount of work required to achieve this. A number of types of existing file systems and databases were considered, that were not selected for use in the system:

- Single System SQL Databases *(Microsoft SQL Server, PostgreSQL, MySQL)*: while this option [Reference 3] would be fastest to start using due to extensive usage in industry, the database would quickly become a bottleneck, as the rate at which data can be read from the server would determine how quickly computations could be performed.
- Distributed File Systems *(Hadoop Distributed File System)*: these provide a mechanism for [Reference] storing files resiliently across a number of machines, which would reduce the bottleneck when reading data. However, they provide no straightforward way of querying the stored data, so this feature would have to be implemented manually.
- Distributed NoSQL Databases *(MongoDB, CouchDB)*: these are distributed, meaning the load [Reference 2] of reading the data could be spread across a number of machines. However, the input data is tabular, meaning the features of a NoSQL architecture are not required. This is likely to result in added complexity when retrieving data from the database, and increased development time.

2.4.1 Apache Cassandra

In the end, Apache Cassandra was chosen for persistent storage. This has a number of benefits for [Reference] the proposed solution. Firstly, the data model is tabular, and as such closely matches the format of the expected input data. Furthermore, Cassandra is a distributed database, so source data will be stored across a number of nodes, each on different computers. This will increase the effective read speed when retrieving data from the database, as the full load will be spread across all nodes.

Partitioning Cassandra's method of partitioning data is another key reason why it was selected. Each node in the database is assigned a token range, which determines what records it holds. When data is inserted into the database, Cassandra hashes the primary key of each record, producing a 64-bit token that maps it to a node. A diagram showing this process is included below:



Cassandra allows token ranges to be provided as filters in queries, which will allow the worker nodes to control what data is retrieved in each query.

Kubernetes Support Cassandra can be run on Kubernetes using K8ssandra [11]. This is a tool which can be used to initialise, configure and manage Cassandra clusters. This is particularly useful because the Cassandra cluster can be configured to run on the same machines as the worker nodes, enabling the source data to be co-located with the workers that will actually perform the computation, reducing network latency when transferring the source data.

Language-Specific Drivers In terms of interfacing with the rest of the system, there are drivers for both Python and Java , which are maintained by one of the largest contributors to Apache Cassandra. Reference The Java driver has a core module which provides basic functionality for making queries and receiving results from the database. There are optional modules for these drivers with more complex functionality including query builders, but these were not included in the solution as the extra functionality was not required, and the added complexity had the potential to cause problems.

There are also some Scala specific frameworks for executing Cassandra queries, including Phantom and Quill . For the same reason as the optional modules above, these were not chosen for the system. Reference

2.5 Architecture

Based on the above design choices, a high level diagram was produced, detailing each of the components of the system and the interaction between them.

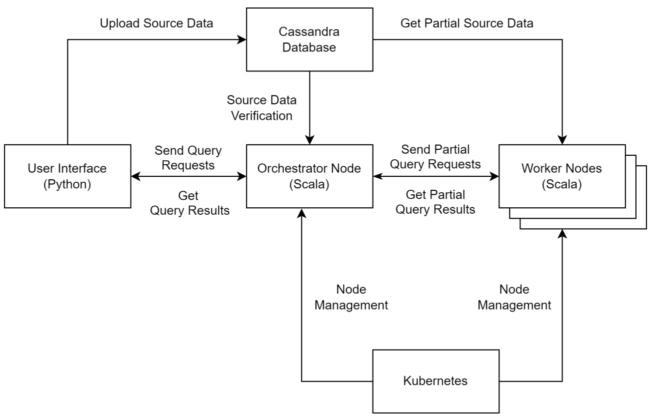


Figure 2.1: Overall Architecture Diagram for the Proposed Solution

Chapter 3

Implementation

Chapter 4

Testing

Chapter 5

Evaluation

Todo list

Bibliography

[1] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: (2015). URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43864.pdf.

[2] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[3] Michael Armbrust et al. "Structured streaming: A declarative api for real-time applications in apache spark". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 601–613. DOI: 10.1145/3183713.3190664. URL: https://doi.org/10.1145/3183713.3190664.

[4] Yingyi Bu et al. "HaLoop: Efficient iterative data processing on large clusters". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 285–296. DOI: 10.14778/1920841.1920881. URL: https://doi.org/10.14778/1920841.1920881.

[5] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[6] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[7] Jaliya Ekanayake et al. "Twister: a runtime for iterative mapreduce". In: *Proceedings of the 19th ACM international symposium on high performance distributed computing*. 2010, pp. 810–818. DOI: 10.1145/1851476.1851593. URL: https://doi.org/10.1145/1851476.1851593.

[8] Philip Harrison Enslow. "What is a" distributed" data processing system?" In: *Computer* 11.1 (1978), pp. 13–21. DOI: 10.1109/c-m.1978.217901. URL: https://doi.org/10.1109/c-m.1978.217901.

[9] Yuan Yu Michael Isard Dennis Fetterly et al. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language". In: *Proc. LSDS-IR* 8 (2009).

[10] *Google I/O Keynote*. 2014. URL: https://youtu.be/biSpvXBGpE0?t=7668 (visited on 02/09/2023).

[11] *K8ssandra*. Available at: https://web.archive.org/web/20230317160100/https://k8ssandra.io/. URL: https://k8ssandra.io/ (visited on 03/17/2023).

[12] Kyong-Ha Lee et al. "Parallel data processing with MapReduce: a survey". In: *AcM sIGMoD record* 40.4 (2012), pp. 11–20. DOI: 10.1145/2094114.2094118. URL: https://doi.org/10.1145/2094114.2094118.

[13] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 135–146. DOI: 10.1145/1807167.1807184. URL: https://doi.org/10.1145/1807167.1807184.

[14] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1099–1110. DOI: 10.1145/1376616.1376726. URL: https://doi.org/10.1145/1376616.1376726.

[15] Ashish Thusoo et al. "Hive-a petabyte scale data warehouse using hadoop". In: *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE. 2010, pp. 996–1005.

[16] Ankit Toshniwal et al. "Storm @twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 147–156. DOI: 10.1145/2588555.2595641. URL: https://doi.org/10.1145/2588555.2595641.

[17] Ibrar Yaqoob et al. "Big data: From beginning to future". In: *International Journal of Information Management* 36.6 (2016), pp. 1231–1247. DOI: 10.1016/j.ijinfomgt.2016.07.009. URL: https://doi.org/10.1016/j.ijinfomgt.2016.07.009.

[18] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[19] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28.

[20] Matei Zaharia et al. "Spark: Cluster computing with working sets". In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. 2010. URL: https://www.usenix.org/event/hotcloud10/tech/full_papers/Zaharia.pdf.