UNIVERSITY OF BIRMINGHAM

Using distributed systems to increase data processing performance for larger than memory datasets

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

Word Count: 9991

Contents

List of Figures

Abstract

The explosion in data volumes in the past 15 years has resulted in increasing demands for solutions to process and analyse this data. Existing single-system solutions like SQL struggle to cope with extremely large volumes of data, suffering from performance slowdowns and long execution times.

While there is an upper limit to the performance of a single system, distributed systems do not face the same issues and can scale to as many nodes as required, therefore presenting an excellent alternative solution to this problem.

This report presents a distributed systems solution for performing various types of data processing tasks, designed around splitting the source data into manageable partitions which can be computed by any node in the cluster. By focusing on closely integrating the persistent storage and computation nodes, the solution is able to assign partitions of the source data to the closest node, thereby reducing the effect of network latency.

As part of the solution, a domain specific language (DSL) is also developed, enabling users to succinctly describe complex data manipulations, including Select, Filter and Group By operations.

The solution is evaluated in a number of ways. Firstly, raw performance testing is conducted against SQL server, a typical single-system approach, identifying that further optimisations are required for the solution to truly compete with existing options. Testing is also performed surrounding the optimal level of parallelisation, showing that this depends on the application and data volumes. Finally, the solution has the potential to automatically adjust the number of nodes in the cluster based on demand to provide cost benefits in a public cloud environment; the results of this testing show that at small data volumes there is minimal performance impact when the number of nodes are reduced.

Chapter 1

Introduction

Data processing is required by every modern business in some form. Existing solutions like SQL fit the requirements of the majority of use cases, as the volume of data they process can be contained within a single system. However, as the volumes of data begin to increase, in particular over 100GB of raw data, single-system solutions like SQL begin to struggle as not all data can be loaded into memory at once.

Figure 1.0.1 shows the network model when using a single system to perform data processing on a SQL server. A number of clients are all connected to a single server, and the server can become overloaded if a large number of clients are making intensive queries, or if some of the queries operate on a large amount of data. This results in significantly reduced query performance, or even outright failure, because the system must spend a large amount of time moving data in and out of memory to complete the computation. It is possible to make extremely powerful servers, but ultimately these solutions are constrained to a single machine, meaning there is an upper limit to the compute performance based on the current technology available.

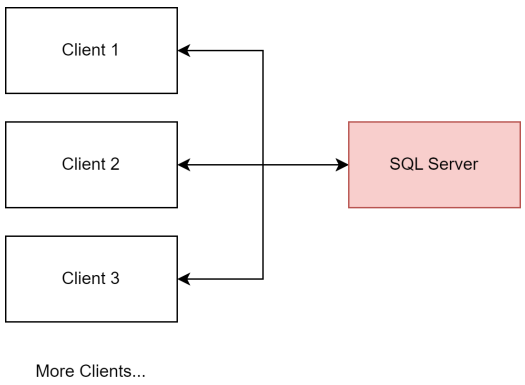


Figure 1.0.1: Single System Solution

This type of processing is generally used in *extract-transform-load* (ETL) workflows, which describe the type of workflow where data must be imported in bulk from an external source, processed in some way, then exported elsewhere for further analysis. Often, the processing to be performed acts on small groups of rows in the original dataset at once, usually a unit item like a loan, customer, or product. As a result, even if the overall dataset is extremely large, the processing can be easily parallelised, as only a small number of rows are needed at any one time to produce the required output.

Distributed systems therefore present an excellent opportunity for solving this problem. If a system

can be designed to automatically split a dataset up and perform the processing over a number of nodes in a cluster, this system could potentially be able to process data of any size, accommodating larger datasets by simply increasing the number of nodes. Similarly, performance can be scaled to an extent by increasing the number of nodes.

1.1 Prior Work

Distributed Data Processing has existed conceptually since as early as the 1970s. A paper by Philip Enslow Jr. defines a distributed system using three 'dimensions' of decentralisation: hardware (the number of machines), control (cluster management), and database (decentralisation of storage) [14]. While the technology of Enslow's time was not equipped to fulfil these goals, more recent research appears to prioritise hardware and database decentralisation to accelerate data processing speeds. Studies identify two major categories within distributed data processing [50]:

- **Batch processing:** data is gathered, processed and output all at the same time. This includes solutions like MapReduce and Spark [12, 51]. Batch processing works best for data that can be considered 'complete' at some stage.
- **Stream processing:** data is processed and output as it arrives. This includes solutions like Apache Flink, Storm, and Spark Streaming [8, 48, 4]. Stream processing works best for data that is being constantly generated, and needs to be analysed in real-time.

MapReduce, a framework introduced by Google in the mid 2000s, could be considered the breakthrough framework for performing massively scalable, parallelised data processing [12]. It later became one of the core modules for the Apache Hadoop suite of tools. MapReduce enables developers to describe jobs as *map* and *reduce* steps, and the framework handles cluster management. While MapReduce was Google's offering, other large technology companies released similar solutions, including Microsoft with DryadLINQ in 2009 [15].

MapReduce was not without flaws, and papers were published in the years following its release which completed performance benchmarks, analysing its strengths and weaknesses [28]. Crucially, MapReduce appears to struggle with iterative algorithms which are executed over a number of steps, as it relies on reading and writing to a persistent storage format after each step. A number of popular extensions to MapReduce were introduced to improve iterative algorithm performance, like Twister and HaLoop, in 2010 [13, 7].

MapReduce was challenging to use for developers familiar with more traditional data processing tools like SQL due to its imperative programming style, resulting in the introduction of many tools to improve its usability. Hive is one example, featuring a SQL-like language called HiveQL that allowed users to write declarative programs, compiling into MapReduce jobs [47]. Pig Latin is similar, with a mixed declarative and imperative style [32].

In 2010, the first Spark paper was released [53]. Spark aims to improve upon MapReduce's weaknesses, by storing data in memory and providing fault tolerance by tracking data 'lineage'. For any set of data, Spark knows how it was constructed from another persistent data source, and can use that to reconstruct lost data in failure scenarios. This in-memory storage, known as a resilient distributed dataset (RDD) allows Spark to improve on MapReduce's performance for iterative jobs, whilst also allowing it to quickly perform ad-hoc queries for interactive usage [52].

Spark quickly grew in popularity, with a number of extensions being added to improve its usability, including a SQL-style engine featuring a query optimiser, and an engine supporting stream processing [3, 4]. A second paper released in 2016 stated that Spark was used by thousands of organisations,

with the largest deployment running an 8,000 node cluster containing 100PB of data [51]. Spark is designed to be agnostic of any particular storage mechanism, instead utilising existing Apache Hadoop connectors to retrieve data from various sources [36]. This provides the advantage that Spark can interface with a large number of data sources, but it is not specialised for any of them, presenting an opportunity for a new solution to improve on importing data through close integration with the persistent storage.

More recent research indicates that the field is moving away from batch processing towards stream processing. A 2015 paper by Google argues that increasing data volumes, the fact that datasets can no longer ever be considered 'complete', along with demands for improved data analytics means that streaming 'dataflow' models are the way forward [2]. Google publicly stated in their 2014 'Google I/O' Keynote that they were phasing out MapReduce in their internal systems [18]. Streaming solutions appear to be the direction of the wider industry, but they are more suited for datasets where data is produced and must be processed at a constant rate. This project is specifically aimed at implementing a generic framework for bulk processing large, complete datasets. Therefore, while a streaming solution is not in scope for the core project, there is an opportunity for further investigation into this as an extension.

1.2 Project Aims

The main objective is to design a query processing engine to perform data processing over a distributed cluster of nodes. To ensure accessibility for users of existing tools, the system should feature a SQL-like interface implemented in a widely-used language, which will allow it to be used for ETL workflows where large data volumes often cause problems. Finally, the system should attempt to exploit the integration between the storage mechanism and cluster nodes to improve execution speed. This appears to be an area where existing distributed data processing solutions could be improved. Details of the design and implementation of the solution are described in Chapters 2 and 3.

Once complete, the implementation should be assessed in a number of ways. Testing against a single-system solution like SQL Server should be conducted, along with further tests to determine the impact of varying the level of parallelisation in the cluster. Chapter 4 provides full details of the tests.

1.3 Challenges

The project presents a number of key challenges that must be solved. The component which splits the dataset up will need to cope with small and large data volumes effectively, and the component which delegates parts of the full dataset to nodes in the cluster will have to handle clusters with any number of nodes correctly. The user's interface must be simple to use, but expressive enough so the user can define a wide range of computations. Finally, the persistent storage mechanism must be carefully selected to provide performance benefits when loading source data into the cluster.

Chapter 2

Design

This chapter will cover the design of the solution, in particular focusing on the following areas:

- Required Features
- Languages, Technologies and Frameworks
- Architecture

The design phase aims to ensure that the limited development time available prioritises the most effective features.

2.1 MoSCoW Requirements

Before considering specific technologies and frameworks, a list of MoSCoW requirements is produced, shown in the table below. The first column represents whether the requirement is **F**unctional or **N**on-**F**unctional, and the second column represents requirements that **M**ust, **S**hould or **C**ould 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 be able to tolerate null values in the results of a dataset.
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 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 Concatenation, and Distinct String Concatenation aggregate functions to Group By operations.
F	C	The system could 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.

NF	S	The permanent data store should be chosen to provide performance benefits when importing source data.

2.2 Language

The first key design decision was to determine what languages would be used to implement the system. A decision was made to use more than one language, because the requirements of the user interface suit a different type of language to the other components.

2.2.1 Frontend

The most important requirement for the frontend was to use SQL-like syntax. Pure SQL requires a text parser to generate queries, which would add a significant amount of development time to implement, so a language which would be able to encode queries as classes and functions was preferred. Therefore, Python was selected for the frontend, as according to the 2022 StackOverflow Developer Survey, it is the fourth most popular programming language [44]. Python also supports some of the most popular data processing frameworks, NumPy and pandas [46, 20]. Due to Python's popularity, it is likely the widest range of users will have prior experience with it, and there is the possibility of integrating with these frameworks to further improve the system's usability.

2.2.2 Orchestrator and Worker Nodes

It was decided that the orchestrator and worker nodes will use the same language, as feature overlap between the two is likely. A language with either automatic or manual garbage collection (GC) had to be chosen. Choosing a manually GC language would theoretically allow for higher performance due to more granular control over memory allocation. However, this could slow 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. Much of the project would be CPU intensive iteration over large lists of items, so a language with strong support for parallelisation was preferable. This ruled out languages like Python and JavaScript where parallelisation requires manual implementation by the developer [30, 49].

In the end, Scala was chosen [38]. It features a mix of both object-oriented and functional paradigms, which would allow the best option to be selected for each task. Furthermore, Scala has built-in support for parallelisation through operations like *map* and *reduce*, which is helpful for iterating over or combining a dataset. It compiles into Java, meaning that Java packages can be executed in Scala, which proved useful for later design decisions [39].

2.3 Runtime

2.3.1 Containerisation

With this being a distributed systems project, the clear choice for executing the code was within containers. Docker is the best option for creating and executing containers, but is not suitable for running and managing large numbers of containers [33]. For this, a container orchestration tool is required, where there are two main options: Docker Swarm, and Kubernetes [45, 26]. Docker Swarm is

more closely integrated within the Docker ecosystem, but many cloud providers feature fully managed Kubernetes services, so Kubernetes was chosen.

2.3.2 Network Communication

A framework had to be selected to allow the frontend, orchestrator and worker nodes to communicate. RPC frameworks are designed to implement custom function calls over a remote network, making them an excellent option to implement a custom query model [43]. The chosen framework was gRPC, designed and maintained by Google [19]. There are well-maintained implementations for both Python and Scala, and gRPC uses protocol buffers (protobuf) as its message system, which are designed to be more space efficient than XML or JSON [40, 35]. This would be useful for transmitting large amounts of data.

2.4 Persistent Storage

There are a wide range of options for persistent data storage. The first key decision 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, at the cost of increased development time. A decision was made not to create a custom solution due to time constraints, so a number of types of existing file systems and databases were instead considered, but were also not selected:

- Single System SQL Databases *(Microsoft SQL Server, PostgreSQL, MySQL)*: while this option would be fastest to start using due to extensive industry support, the database would quickly become a bottleneck, as all nodes would import data from the same location.
- Distributed File Systems *(Hadoop Distributed File System)*: these provide a mechanism for storing files resiliently across a number of machines, removing the data import bottleneck. However, they provide no way of querying the stored data, so this feature would have to be implemented manually.
- Distributed NoSQL Databases *(MongoDB, CouchDB)*: these spread the load of reading the data across a number of machines. However, the expected input data is tabular, meaning the NoSQL architecture is likely to result in added complexity when importing data.

2.4.1 Apache Cassandra

Apache Cassandra was chosen as the persistent storage mechanism for a number of reasons [27]. Firstly, the data model is tabular, as is the expected input data. Furthermore, Cassandra is a distributed database, so the source data will be stored across a number of nodes. This will increase the effective read speed when retrieving data, as the full load will be spread between the nodes.

Partitioning Cassandra's method of partitioning data is another key reason why it was selected. When a record is inserted into the database, Cassandra hashes the primary key, producing a 64-bit token. Each node in the database is assigned part of the full token range, which determines what records it holds [27]. Figure 2.4.1 demonstrates how the full range of tokens is distributed among a number of Cassandra nodes.

Cassandra allows token ranges to be provided as filters in queries, meaning the worker nodes can split up and control the data they read from the database, to read smaller chunks.

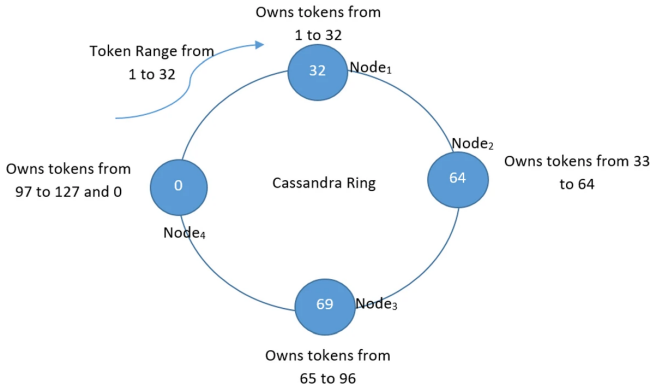


Figure 2.4.1: Cassandra Token Ring Distribution [25]

Data Co-Location Cassandra can be run on Kubernetes using K8ssandra [24]. 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 performing the computation. This decision meets the requirement to exploit the integration between the storage mechanism and cluster nodes, as data co-location will reduce the network latency when importing the source data, and increase transfer speed in some cases if the data never leaves the physical machine.

Language-Specific Drivers Drivers for both Python and Java are available, which provide basic functionality for querying the database [10, 11]. There are optional modules for these drivers, and Scala-specific frameworks with more complex features, but these were not used as the extra functionality was not required, and the added complexity had the potential to cause problems.

2.5 Architecture

Based on the above design choices, a high level diagram was produced, detailing each of the system's components and the interaction between them, shown in Figure 2.5.1. The user will be able to define queries using the Python frontend, which are then sent using gRPC to the orchestrator. It will break the full query up into partial queries, which will be computed on the worker nodes. When the computation is complete, the workers will return results to the orchestrator, which will combine the result and return it to the frontend.

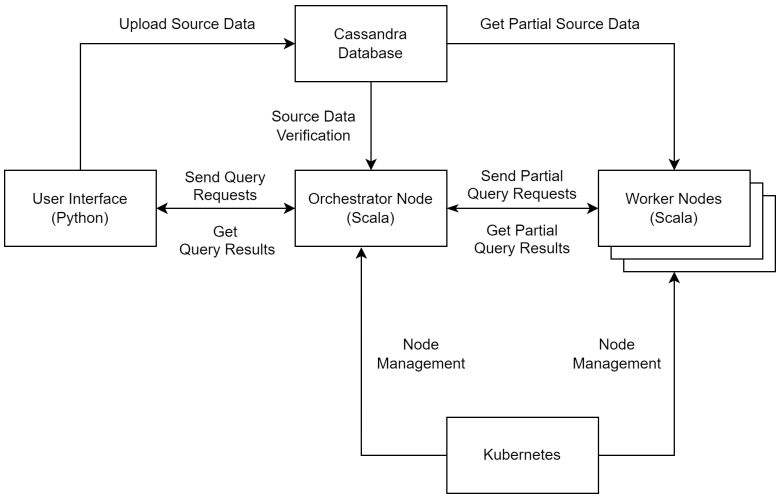


Figure 2.5.1: Proposed Solution - Overall Architecture

Chapter 3

Implementation

A number of components were created to implement the proposed architecture. This chapter will provide a high level overview of the solution, then examine each component in further detail.

3.1 Overall Solution

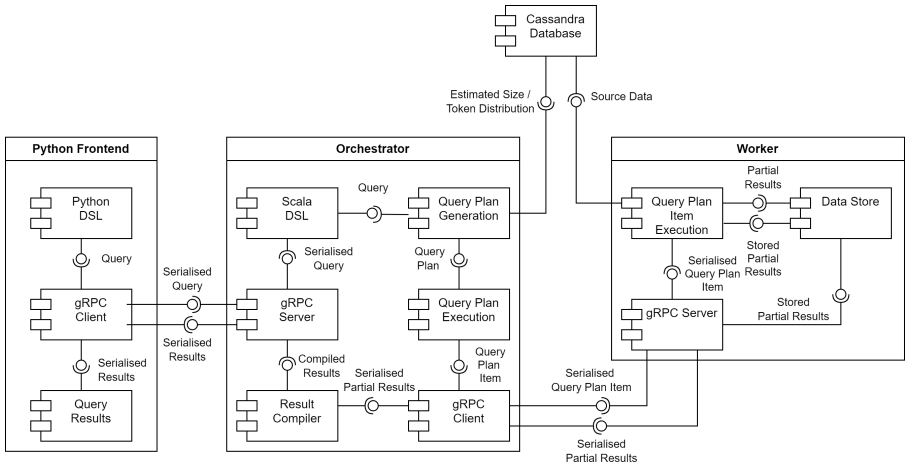


Figure 3.1.1: Solution Component Diagram

Figure 3.1.1 is a component diagram describing the interactions of the core components. As described in Section 2.5, the user accesses the system through the frontend. This is a terminal, allowing the user to define queries using the Python Domain Specific Language (DSL), and receive results. Queries are sent to the orchestrator, which controls the state of the system. To execute a query, it generates a query plan which describes the steps for computing a result. Then, the query plan is executed step-by-step, with the data used in each step being split up into chunks of work (partitions) before being delegated to the workers, which are responsible for actually computing partitions. When all query plan steps have been executed, the orchestrator collates the partial results from all workers, and returns the final result to the frontend. gRPC is used for network communication between the frontend, orchestrator and worker nodes.

The system supports three query types: Select, Filter and Group By. Select and Filter are row-level operations, meaning the workers do not have to share data during computation. However, Group By does need the workers to cross-communicate, so a temporary store is needed to cache partial results.

3.2 Type System

The type system's role is to provide a representation for every type of value the user can store within a table. However, designing the interfaces to represent these values presents a challenge. Figure 3.2.1 demonstrates the problem.

On the left, an example is shown where a table is a 2D list of raw values. As multiple types are stored in the same list, there is no shared type information between the values, so they cannot be used to determine the table's contents. To discover a value's type, the system would have to perform runtime type checks against all supported types, adding a significant amount of overhead.

On the right, a conceptual model using a container class is shown. The container stores a raw value with the type information about the value, and the table is a 2D list of containers. The system only needs to perform one runtime type check in order to instantiate the correct container instance.

Header: [id, name]	**Header:** [(id, int), (name, string)]
Row 1: [1, "Alice"]	**Row 1:** [(1, int), ("Alice", string)]
Row 2: [2, "Bob"]	**Row 2:** [(2, int), ("Bob", string)]

(a) Raw Data - No Type Information Available

(b) Container Class - Value and Type Information Stored Together

Figure 3.2.1: Type System - Motivating Example

Due to Java limitations, this conceptual solution is not straightforward to implement. The container class could use a generic type parameter which stores the type information of the value, but generic type information is erased at runtime [16]. Instead, it could be defined as an interface, with subclass implementations for each supported type, but this is not much better than the raw data solution, as the runtime type check simply becomes a pattern match on the class type. A solution that exploits some kind of polymorphism is preferred.

Scala provides a feature known as *ClassTags* [9]. These save the erased type information and support equality checks between *ClassTag* instances, meaning the framework can use them to compare the type of a value to an expected type. The conceptual container class is therefore defined as an interface *ValueType*, which holds a *ClassTag* instance. This interface is implemented by *TableField*, which holds a field name and its type, and *TableValue*, which holds a value and its type. Concrete implementations for the 5 supported types described in Section 3.2.1 are provided for both subclasses. Figure 3.2.2 shows the class hierarchy.

3.2.1 Supported Types

The type system supports a subset of Scala, Python, and Cassandra's types, shown in Figure 3.2.3. These types were selected to ensure they could be represented in all parts of the system, including when serialising to protobuf format.

3.2.2 Result Model

The hierarchy of classes and supported types provide everything needed to define a table of result data, known as a *TableResult*. Headers are a sequence of *TableFields* and result rows are a two-dimensional array of `Option[TableValue]`. As defined in the requirements, null values must be supported, but this is discouraged in Scala. Instead, Option is preferred as it is supported by all the typical functional methods. In this model, values are represented by `Some(TableValue())`, and null values are represented by `None`. Figure 3.2.4 shows an example *TableResult*.

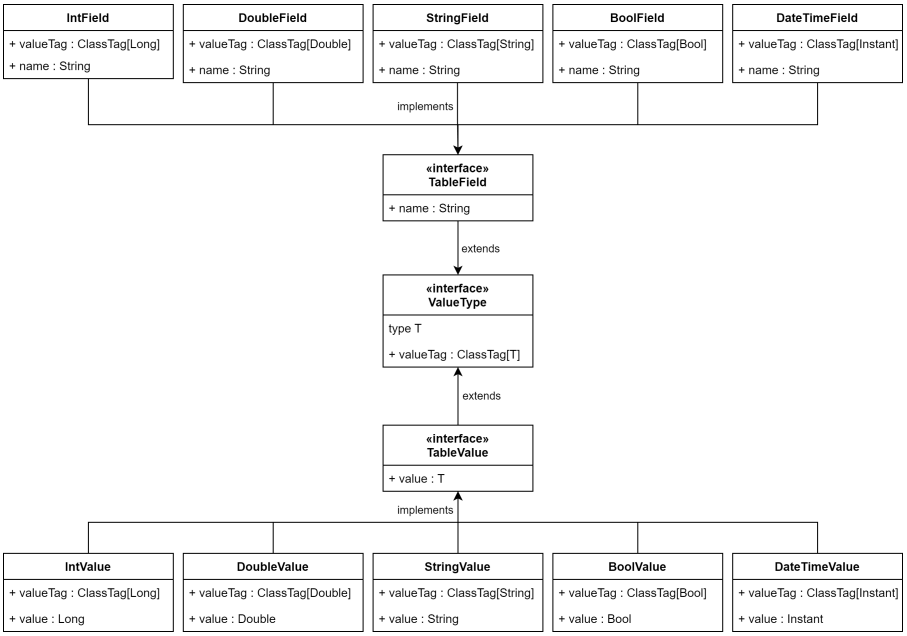


Figure 3.2.2: Custom Type System Hierarchy

Base Type	**Python**	**Scala**	**Cassandra**
Integer	int	Long	bigint
Float	float	Double	double
String	string	String	text
Boolean	boolean	Boolean	boolean
DateTime	datetime	Instant	timestamp

Figure 3.2.3: Primitive Types

Header: [	IntField("ID"),	StringField("Name"),	BoolField("Passed")	]
Row 1: [[	Some(IntValue(1)),	Some(StringValue("Alice")),	Some(BoolValue(true))	],
Row 2: [	Some(IntValue(2)),	None,	None	],
Row 3: [	Some(IntValue(3)),	Some(StringValue("Bob")),	None	]]

Figure 3.2.4: *TableResult* example

3.3 Domain Specific Language

The user interacts with the framework through the Domain Specific Language (DSL), which is modelled with SQL-like syntax. Users can define expressions, then use these in computations like Select, Filter and Group By. Figure 3.3.1 shows an annotated DSL query, with links to the sections where each part is discussed further.

Initialise cluster connection and select a Cassandra source table (Section 3.6.1):

```
ClusterManager("orchestrator-url")
  .cassandra_table("example", "table")
```

Select query (Section 3.7.1), uses *FieldExpressions* (Section 3.3.1) and Python Operators (Section 3.3.5):

```
  .select(
    F("id"),
    (F("duration") * 2).as_name("duration2"),
    Function.Left(Function.ToString(F("date")), 8)
      .as_name("yyyy-mm")
  )
```

Filter query (Section 3.7.2), uses *FieldComparisons* (Section 3.3.2) and Python Operators (Section 3.3.5):

```
  .filter(
    (F("duration2") > 40) &&
        (F("yyyy-mm").contains("2021")
  )
```

Group By query (Section 3.6.3), uses *AggregateExpressions* (Section 3.3.3):

```
  .group_by(
    [F("duration2")],
    [
      Max(F("id")),
      Count(F("yyyy-mm"))
    ]
  )
```

Figure 3.3.1: Example DSL Query

3.3.1 FieldExpressions

FieldExpressions allow the user to define arbitrary row-level calculations. They are defined as an interface, with three subclasses:

- Values: define literal values which never change across all rows

- Fields: get the value at the current row for the given field name.
- FunctionCalls: perform arbitrary function calls using *FieldExpressions* as arguments.

Figure 3.3.2 provides examples for each type of *FieldExpression*.

Values (from left to right): string "a", integer 1, double 1.5, boolean True, date 31/12/2021.

```
V("a") , V(1), V(1.5), V(True), V(datetime.date(2021, 12, 31))
```

Fields (from left to right): fieldName, duration, id, creationDate.

```
F("fieldName"), F("duration"), F("id"), F("creationDate")
```

Top Function: convert 'field1' to a string, then take the left 10 characters of each row.
Bottom Function: multiply 'field2' by 2, then divide by "field3"

```
Function.Left(Function.ToString(F("field1")), 10)
(F("field2") * 2) / F("field3")
```

Figure 3.3.2: *FieldExpression* implementations and examples

Many basic functions have been implemented, including arithmetic, string and cast operations. However, the function system is designed to be extensible. A number of helper classes and interfaces are provided to do this, and the main constraint is that functions can only take the 5 primitive types as arguments.

Type Resolution Type Resolution is performed in two stages: resolution, and evaluation. The resolution step takes in type information from the input result header, and verifies that the *FieldExpression* is well-typed with regards to that result by comparing *ClassTags*; see Section 3.2 for details. The evaluation step performs the computation on a *TableResult* row without any type checking. Unchecked casts are used here instead, demonstrated for a binary function in Figure 3.3.3.

```
val resolvedLeft = left.resolve(header)
val resolvedRight = right.resolve(header)
return ResolvedFunctionCall((row) =>
    resolvedLeft.evaluate(row).flatMap(l =>
        resolvedRight.evaluate(row).map(r =>
            // Extract inner values from left and right arguments, and perform unchecked cast
            // to convert to runtime type (type checking already performed by .resolve)
            function(l.value.asInstanceOf[LeftArgType], r.value.asInstanceOf[RightArgType])
        )
    )
)
```

Figure 3.3.3: Runtime Evaluation of Functions

The resolution step catches two key types of invalid expression. Firstly, a Field reference is invalid if the field name is not present in the table header, shown in Figure 3.3.4. Secondly, a Function call is invalid if an argument returns an invalid type, shown in Figure 3.3.5.

A two-step process has a number of benefits. The resolution step enables a form of polymorphism on some functions like arithmetic operations. These determine what types are returned by their sub-expressions during resolution, and change their behaviour for evaluation. For example, the add

For the expression F("field"):

Header: [IntField("field")]

(a) Valid - "field" is present in the header

Header: [StringField("differentField")]

(b) Invalid - "field" not present in the header

Figure 3.3.4: Type Resolution of Fields

Valid - both arguments to *Concat* are strings:

```
Function.Concat(V("hello"), V("Alice"))
```

Invalid - *V(1)* is not a string:

```
Function.Concat(V("hello"), V(1))
```

Figure 3.3.5: Type Resolution of Functions

function resolves to *AddInt*, *AddDouble*, or *Concat*. It also reduces the overhead at runtime as type checking does not need to be performed for each row.

Named Expressions When performing a Select operation, the output fields must be named to so operations can reference fields from the previous input. Figure 3.3.6 shows the two ways of naming a field: *FieldExpressions* can be assigned a name, and single Field references will keep their previous name.

Top Expression Name: 'twice_duration'
Bottom Expression Name: 'creation_date'

```
(F("duration") * 2).as_name("twice_duration")
F("creation_date")
```

Figure 3.3.6: *NamedFieldExpression* examples

3.3.2 FieldComparisons

FieldComparisons allow the user to define arbitrary row-level comparisons. They are defined as an interface, supporting a number of comparison types including null, equality, numerical and string comparisons. Examples of each are shown in Figure 3.3.7.

Combined Comparisons The user can combine multiple *FieldComparisons* using AND/OR operators, shown in Figure 3.3.8. This is implemented around Scala's own boolean operators, meaning optimisations like short circuiting operate as normal.

3.3.3 Aggregate Expressions

AggregateExpressions are only used in Group Bys, taking a *NamedFieldExpression* as an argument, and aggregating any number of input result rows into a single result using *reduce*. Figure 3.3.9 demonstrates all supported operations.

Null checks: verify whether an expression is null or not null.

```
F("duration").is_null()
F("duration").is_not_null()
```

Equality checks: verify whether two *FieldExpressions* are equal or not equal.

```
F("duration") == 20
F("duration") != F("other_duration")
```

Ordering checks: apply ordered comparators between two *FieldExpressions*.

```
F("duration") < 20
F("duration") <= 19
F("duration") > 20
F("duration") >= 21
```

String checks: apply contains, starts with and ends with operators (case insensitive versions also available).

```
F("name").contains("Alice")
F("name").starts_with("Bob")
F("name").ends_with("Smith")
```

Figure 3.3.7: *FieldComparison* examples

```
(F("duration") < 20) && (F("name").starts_with("Bob"))
(F("duration") < 20) || (F("name").starts_with("Bob"))
(F("duration") < 20) && (
  (F("name").contains("Bob")) || (F("name").contains("Alice"))
)
```

Figure 3.3.8: *FieldComparison* AND/OR Combinations

Minimum/Maximum - supports integers, floats and dates (ordered numerically), strings (ordered lexicographically):

```
Min(F("duration"))
Max(Function.ToString(F("date")).as_name("date_string"))
```

Sum - supports integers and floats:

```
Sum(F("amount"))
```

Average - supports integers, floats and dates:

```
Avg(F("date"))
```

Count (and distinct) - supports all data types:

```
Count(F("id"))
DistinctCount(F("date"))
```

String Concatenation (and distinct) - combines all strings in a field with a given string delimiter.

```
StringConcat(F("name"), ", ")
DistinctStringConcat(F("address"), ", ")
```

Figure 3.3.9: *AggregateExpression* examples

3.3.4 Protocol Buffer Serialisation

Any DSL query can be serialised to protobuf format, allowing it to be sent using gRPC. *TableResults* are also serialisable, to allow the system to return query results to the user. gRPC has a size limit of 4MB for individual messages, so *TableResults* are streamed row-by-row.

3.3.5 Python Implementation

The Python frontend is designed to be straightforward to use, hiding the complexities of the computation being performed in the backend. A number of Python-specific features were used to help with this.

Python allows developers to override common operators with custom definitions. The Python implementation of *FieldExpression* overrides the arithmetic operators, as well as comparison operators to allow the user to automatically generate functions and *FieldComparisons*, without having to write the full definition.

Furthermore, as discussed in 2.2.1, pandas is widely used for Python data analysis [46]. The frontend is able to convert query results from protobuf to a pandas DataFrame to aid further analysis.

3.4 Data Model

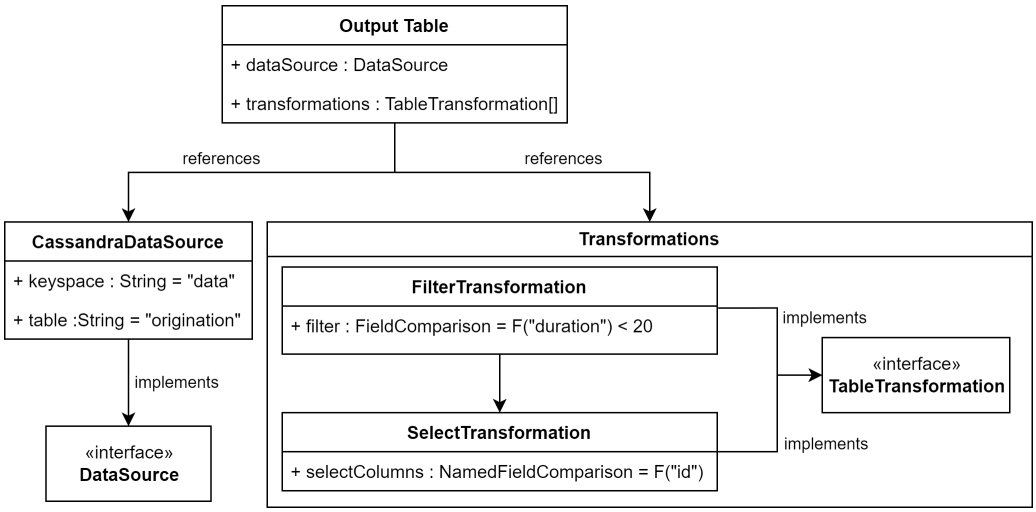
The data model is split into two key components: *DataSource* and *Table*.

DataSource is an interface, representing any part of the query where data must be rearranged into new partitions including Cassandra source data, and Group By operations. *Table* is a class, representing the computation of any row-level operations like Select and Filter, which must implement the *TableTransformation* interface. It is composed of a *DataSource* and a list of *TableTransformations*. To compute its output, the *DataSource* is computed first, then all transformations are applied

sequentially.

Optionally, *DataSources* can have dependent *Tables* which must be calculated first. For example, a Group By *DataSource* requires a single *Table* to be computed before it can be generated. A Cassandra *DataSource* will always act as the terminal component for a query, as it has no dependencies.

Figure 3.4.1 shows a Filter and Select query in the DSL, and the data model. Note that any number of Select and Filter operations can be added to the output table, as these operations do not require generating new partitions.




```
ClusterManager("orchestrator-service")
.cassandra_table("data", "origination")
.filter(F("duration") < 20)
.select(F("id"))
.evaluate()
```

Figure 3.4.1: Example Filter and Select Query

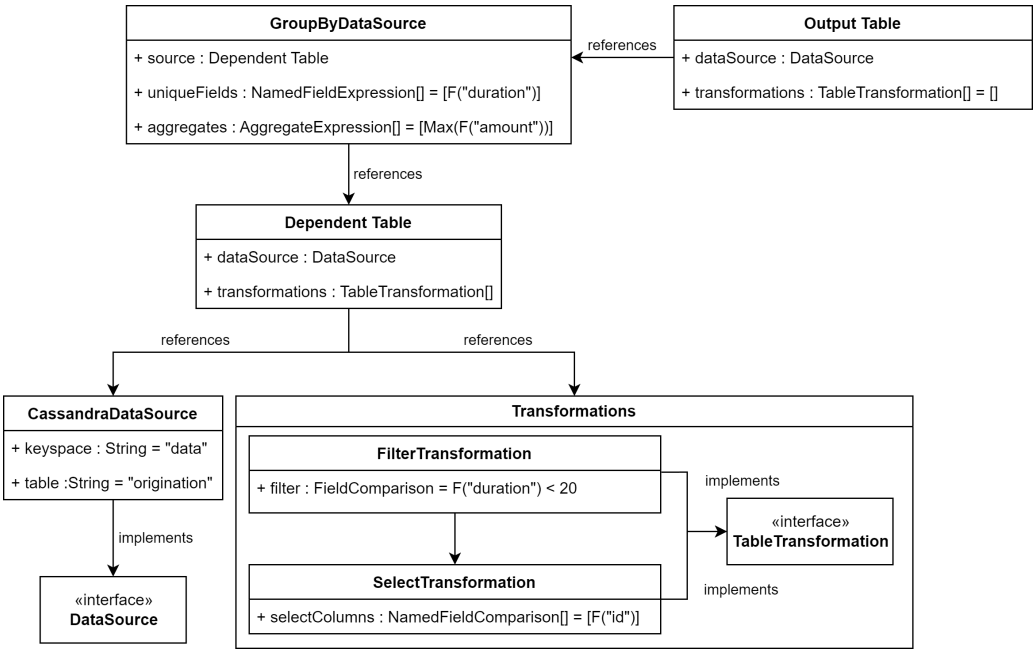
Figure 3.4.2 shows a query containing a Group By in the DSL, and the data model. The dependent table (bottom left) is calculated first, and its output is used to compute the Group By, in *GroupBy-DataSource*. The final output is then generated by the output table.

A *Table* or *DataSource* cannot be computed directly, but must first be split into partitions, which are provided by the *DataSource*. These partitions are represented by the interface *PartialDataSource*, with the partitioning method being specific to each implementation. The *Table* class has a similar partial form, *PartialTable*, which references a *PartialDataSource* and can be computed directly. To demonstrate how to produce a final result from a set of partitions, Figure 3.4.3 shows a high level example of one possible way of partitioning and computing the previous Filter and Select query.

3.5 Data Store

The data store allows the workers to store partially computed data, which can be reused in later parts of the query execution. In particular when workers are communicating with one another, it is likely that the data store will receive multiple simultaneous requests, presenting issues with handling concurrency and synchronisation.

The approach taken to solve this uses the actor model, first introduced in 1973 by Carl Hewitt [21].




```
ClusterManager("orchestrator-service")
.cassandra_table("data", "origination")
.filter(F("duration") < 20)
.select(F("id"))
.group_by([F("duration")], [Max(F("amount"))])
.evaluate()
```

Figure 3.4.2: Example Group By Query

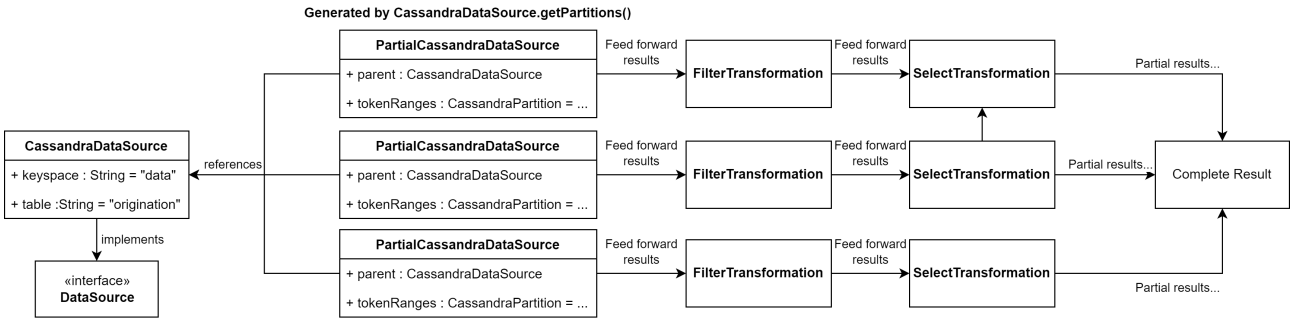


Figure 3.4.3: Example Filter and Select Query

Specifically, the Akka Actors framework was chosen for a Scala actor model implementation [1]. It abstracts away the complexity of synchronisation and thread management, modelling components of the system as actors. Each actor defines a set of accepted messages, and the response to each. The framework then guarantees that an actor will only ever process one message at a time.

The data store is an actor which stores results as key-value pairs. Three kinds of data can be used as keys for storage: *Table* computation results, *DataSource* computation results and hashed data results, which are used when computing Group Bys. It uses a two-stage lookup, internally implemented using nested HashMaps. First, the full version of the data (*Table* or *DataSource*) is looked up, then the partial version (*PartialTable* or *PartialDataSource*). Partial forms always contain a reference to the full version, but not the other way around. Therefore, this design does not increase the insert time significantly, but it is particularly useful when reading or deleting a *Table* or *DataSource*. Otherwise, these operations would require searching the entire HashMap, turning the O(1) lookup time into O(n).

3.5.1 Spill to Memory

When operating on very large datasets, the dataset will be larger than the available memory of the workers. In this case, the JVM will run out of heap space, causing a crash when it tries to allocate more memory. Therefore, the data store has a module designed to move in-memory data onto disk to free up heap space, operating transparently from the perspective of other worker components.

Storage Interface To implement the spill process, an interface, *StoredTableResult*, is defined. This interface holds a key which corresponds to a result, and a *get* operation to retrieve the result data. There are two subclasses with implementations: *InMemoryTableResult* and *ProtobufTableResult*. *InMemoryTableResult* holds the result in-memory, and has a *spillToDisk* method which moves the data to disk. *ProtobufTableResult* holds a pointer to the data on-disk, reading the data from there when the *get* operation is called.

Spill Process The data store is responsible for managing in-memory and on-disk data. Before almost every operation, it checks current memory utilisation, which is calculated using Java's *Runtime* class [37]. If the utilisation is over a given threshold, the data store attempts to spill data to get below the threshold.

Figure 3.5.1 shows how the amount of bytes over a percentage threshold is calculated. The division calculates the current memory utilisation percentage, then the threshold is subtracted to get the percentage amount over the threshold. This is multiplied by the total number of bytes to get the result.

$$\left(\frac{\text{Bytes in Use}}{\text{Total Bytes Available}} - \text{Threshold}\right) * \text{Total Bytes Available}$$

Figure 3.5.1: Number of Bytes Over Memory Threshold

To perform the spill, the data store will follow the decision tree shown in Figure 3.5.2.

This process is not without flaws. It relies on no other classes in the current JVM instance holding references to any spilled results. In the controlled worker environment, this can be guaranteed, meaning the spill works reliably.

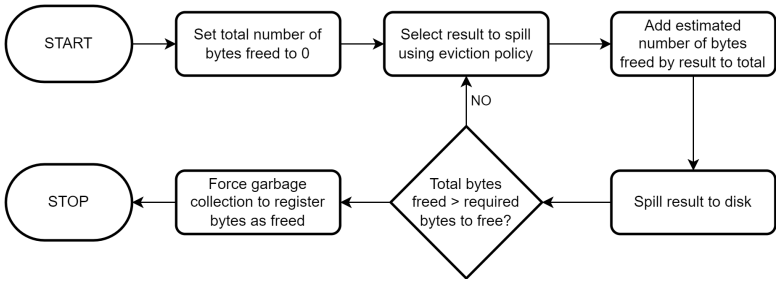


Figure 3.5.2: Spill to Disk Decision Tree

Eviction Policy Finally, the policy for selecting spill results is important. A policy that does not fit the data store's usage could cause anti-patterns like the data store spilling a result, then immediately reading it back to memory. The data store uses a least-recently-used policy to select a result to spill, implemented using an ordered list.

3.6 Partitioning

One of the most important roles of the orchestrator during a computation is to calculate partitions. There are two situations where this is required: pulling source data from Cassandra, and computing a Group By. The data model uses the *DataSource* interface to represent computations that require new partitions.

The goal of partitioning is to split a dataset into roughly equal chunks of a manageable size. Two things are required to do this: an estimate of the full size of a dataset, and a way of splitting the dataset to keep unique keys together. These are referred to as the partitioning requirements.

3.6.1 Cassandra

Cassandra was selected for persistent storage because its features meet the partitioning requirements. As discussed in Section 2.4.1, Cassandra's token range system natively provides a way of splitting the source dataset. Furthermore, Cassandra provides size estimates for any table automatically in the `system.size_estimates` table.

A table size estimate can be used to derive a token range size estimate using the equation in Figure 3.6.1. The division calculates the percentage of the full token range that the given token range represents.

$$\frac{\text{Number of Tokens in Token Range}}{\text{Total Number of Tokens: } ((2^{63}-1)-(-2^{63}))} \times \text{Estimated Table Size}$$

Figure 3.6.1: Token Range Size Estimation Equation

Using this equation, the token ranges which each node is responsible for storing are collected. Then, the orchestrator performs a joining and splitting process over each node, depending on the size of the token ranges. The set of token ranges produced by this process are the partitions used during the computation. Figure 3.6.2 shows the full process for generating the output partitions.

Figure 3.6.3 demonstrates the token splitting process. The system calculates how many times larger the token range is than the goal partition size, then splits the token range evenly by that amount.

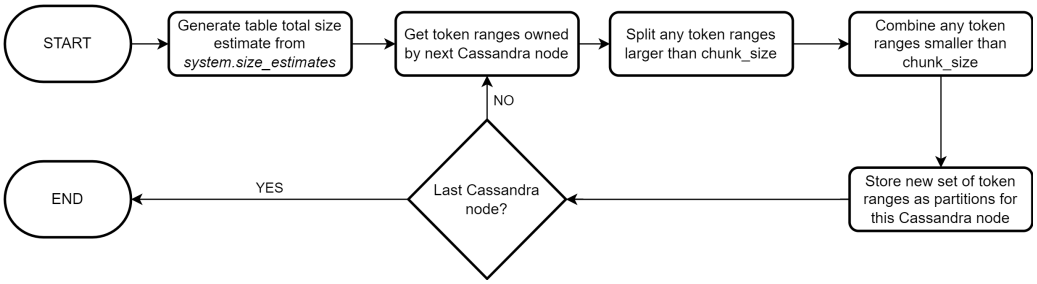
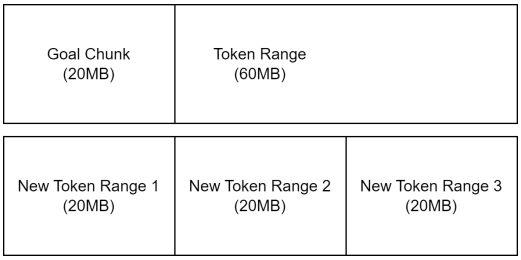


Figure 3.6.2: Cassandra Partitioning Process

$$\text{Number of Splits} = \frac{\text{Token Range Size}}{\text{Chunk Size}}$$

(a) Equation



(b) Example

Figure 3.6.3: Token Range Splitting

Figure 3.6.4 provides an example of the joining process. Given a list of token ranges, the orchestrator combines sequential elements until they are larger than the goal partition size, then it marks this as a new partition. The list is sorted by size ascending to combine small token ranges together, producing fewer partitions.

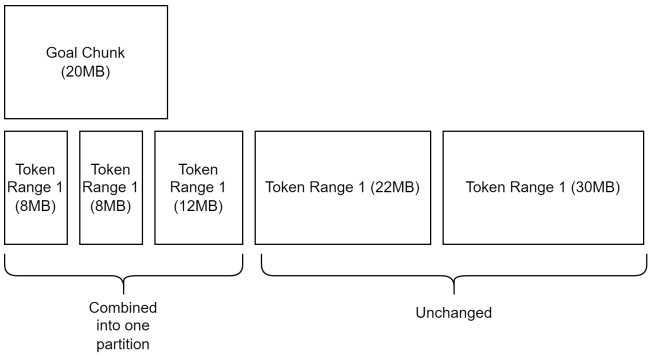


Figure 3.6.4: Token Range Joining Example

3.6.2 Cassandra Data Co-Location

From a list of partitions for each Cassandra node, the system attempts to co-locate workers to Cassandra nodes. The goal of this process is to produce an *optimal assignment*, where each partition is matched to one or more workers to minimise network latency when importing data from Cassandra.

To do this, each worker first calculates its closest Cassandra node by opening a TCP connection with each Cassandra node, averaging the latency over multiple attempts, and selecting the node with the lowest latency. The orchestrator uses this information to match each worker to a Cassandra node and its corresponding list of partitions, producing an optimal assignment between workers and partitions. Partitions can have no co-located worker nodes. In this case, the partitions are unassigned, and the

work assignment algorithm handles their allocation; see Section 3.8.1 for details. Figure 3.6.5 shows an example cluster with three physical nodes, and the corresponding optimal assignment.

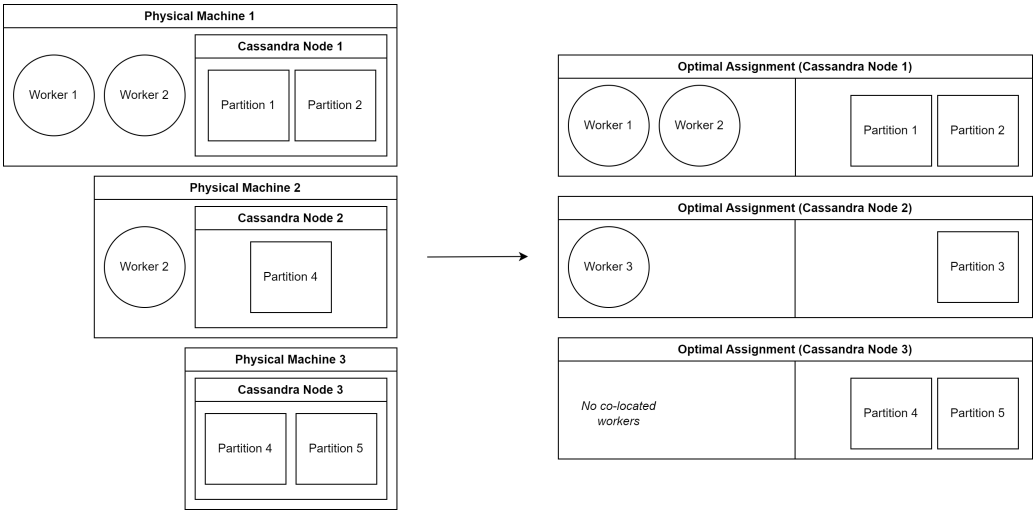


Figure 3.6.5: Optimal Assignment Example

3.6.3 Group By

The Group By operation takes any number of *NamedFieldExpressions* as unique keys, and any number of *AggregateExpressions* to calculate for each combination of unique keys. This operation is not reliant on external components, so custom implementations must be used to meet the partitioning requirements.

The process of calculating a Group By is described below. These steps are controlled by the orchestrator during the execution of *GetPartition*; see Section 3.8.1 for details.

Partitions The first step is to compute the number of output partitions for the Group By, which is based on a size estimate of the dependent *Table*. To generate this, a class from Apache Spark was reused, with some changes for compatibility with Scala 3 [51]. This class provides a static method *estimate* which produces a size estimate, in bytes, for any Scala object. The orchestrator calls this on all partial results across all workers, and the estimates are totalled to calculate the total size of all data for a *Table*. Figure 3.6.6 shows how the size estimate can be used to derive the total number of partitions to generate for a *Table*.

$$\frac{\text{Table Size Estimate}}{\text{Goal Partition Size}}$$

Figure 3.6.6: Group By - Total Partitions

Hashing A unique partition is defined as a tuple, shown in Figure 3.6.7. Hashing, combined with the modulo operation, is used to assign rows of data to partitions. In particular, Murmur3Hash is used as the hashing algorithm, used in Cassandra and provided natively by Scala [31]. Figure 3.6.8 shows the high level equation for assigning rows to partitions. This equation maps all rows with the same combination of unique keys to the same partition.

$$(\text{Total Number of Partitions}, \text{Partition Number for this Partition})$$

Figure 3.6.7: Group By - Unique Partition Definition

$$Murmur3Hash(\text{Unique Key Data}) \ \% \ \text{Total Number of Partitions}$$

Figure 3.6.8: Group By - Row Partition Assignment

Computation After the hashes are computed and a worker is assigned a particular partition, it must cross-communicate with all other workers to fetch any data relating to that partition to ensure that the partition data is complete. To do this, the worker makes requests to all other workers, and they stream the header and rows of their partial data back to the worker. When a Group By is being computed, a worker will likely be simultaneously receiving data from another worker, and sending a different set of data to it. This makes the actor system driving the data store in each worker particularly valuable, as it provides thread-safe concurrent access to the data store.

Once a worker has collected all partition data, the groups are computed using a built-in Scala operation, and the *AggregateExpressions* are evaluated for each group.

Deletion The last step of computing a Group By is to remove the hashed partition data stored on each worker, which is handled by the orchestrator automatically.

3.7 Row-Level Computations

Once partitions have been delegated to the workers, performing the computation is straightforward.

3.7.1 Select

This operation takes any number of *NamedFieldExpressions*. To compute a result, each *NamedFieldExpression* is mapped over each input result row.

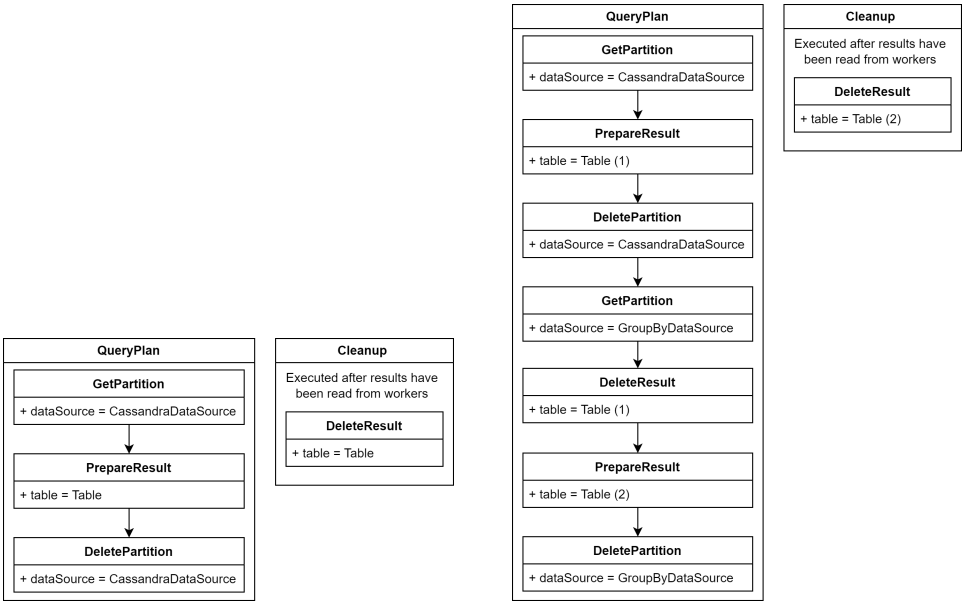
3.7.2 Filter

This operation takes a single *FieldComparison*, or *FieldComparisons* combined using boolean operators. To compute, each comparison is applied to each row of the input result, removing any rows where the comparison returns `false`.

3.8 Query Plan

Query Plans are the process through which the orchestrator can compute a user-defined query, to get a result. They are a sequence of *QueryPlanItems*, which define a single step. Each *QueryPlanItem* has an *execute* method, which will make some change to the state of all workers in the cluster when called. Both *DataSource* and *Table* have a function that generates the full Query Plan to compute their output, and a second Query Plan to clear the data store of their output.

Figure 3.8.1 shows the query plans for the Filter and Select query (3.4.1) and Group By Query (3.4.2).



(a) Filter and Select Query (b) Group By Query

Figure 3.8.1: Query Plans - Filter-Select and Group By Query

3.8.1 GetPartition

This is the most complex *QueryPlanItem*, encapsulating a number of steps in order to compute and store the partitions of a *DataSource*. There are two main flows depending on if the *DataSource* has dependent Tables.

If the *DataSource* has no dependencies, for example when pulling data from Cassandra, then Figure 3.8.2 shows the process for this item.

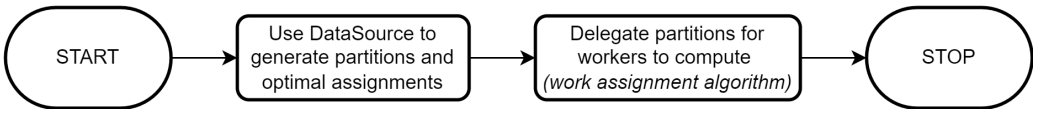


Figure 3.8.2: Get Partition Execution - Without Dependencies

If the *DataSource* has dependencies, then Figure 3.8.3 shows the process for this item. These steps are the same as when calculating a Group By (Section 3.6.3). First, the partitions and optimal assignments are generated, then the dependency data is hashed based on the number of partitions to generate. Both of these steps are implementation-specific, and are therefore abstracted behind the *DataSource* interface.

The work assignment algorithm is then run to delegate partitions to the workers, followed by deleting the hashed dependency data. These steps do not change based on the *DataSource*, so are handled exclusively by *GetPartition*.

Work Assignment Algorithm *GetPartition* manages the process of delegating partitions to workers for computation using the Work Assignment Algorithm. A simple solution would use a round-robin process to match optimal assignments to their co-located workers, delegating work when a request finishes and stopping when the assignments are empty. However, this can result in idle workers, for example if one worker's list of optimal assignments is shorter than the others, or if one worker is

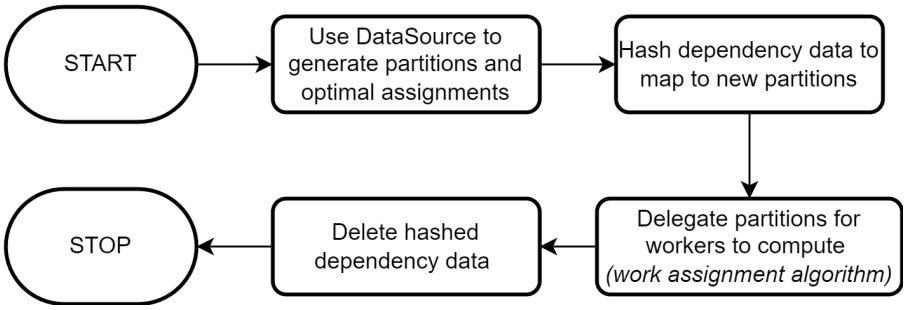


Figure 3.8.3: Get Partition Execution - With Dependencies

unexpectedly slow.

Ideally, workers would compute all of their own optimal partitions first, then compute the partitions that were originally assigned to other workers. This implements dynamic load balancing: a faster-running worker can take on proportionally more requests, and no worker will be idle unless all partitions are computed. However, race conditions need to be avoided, like delegating same partition twice to different workers.

The actor model is ideal for this situation [21]. Sets of optimal partitions are modelled as *producer* actors, and the workers as *consumers*. Producers respond to requests for work with partitions to be computed. Consumers are given an ordered list of producers, then repeatedly request and compute work from each in order. Each consumer has the producer holding that consumer's optimal partitions first in the list. A counter actor tracks the number of completed partitions, sending a signal when all partitions have been computed, or an error if any worker fails.

This solution also handles unassigned partitions with no co-located workers; this producer is placed last in the list of producers for each consumer, where they will eventually be processed.

Figure 3.8.4 provides the initial state of this model, using the example optimal assignment from Figure 3.6.5. Dark arrows represent the producer which the consumer will empty first, containing its optimal assignments, and light grey arrows represent other producers which the consumer can access. Producer 3 is not first for any worker, but will eventually be checked by the workers when all others are exhausted.

3.8.2 PrepareResult

This *QueryPlanItem* computes a *Table* from the partitions of a *DataSource* that are already stored on the workers. Therefore, it will always be called after *GetPartition*, with the new partitions as an argument. A modified version of *GetPartition*'s actor system is used to iterate through all partitions on each worker, sending a request to perform the *Table* computation for each.

3.8.3 DeleteResult and DeletePartition

DeletePartition and *DeleteResult* are *QueryPlanItems* for removing the results of a *GetPartition* and *PrepareResult* operation, respectively. They send a single request to each worker, which will remove all results that relate to a *Table* or *DataSource*, and respond with a confirmation.

3.8.4 Result Collation

After all the steps of a query are completed, the final results are stored across all the workers. The orchestrator makes a request to all workers to return the computed results, and each worker iterates

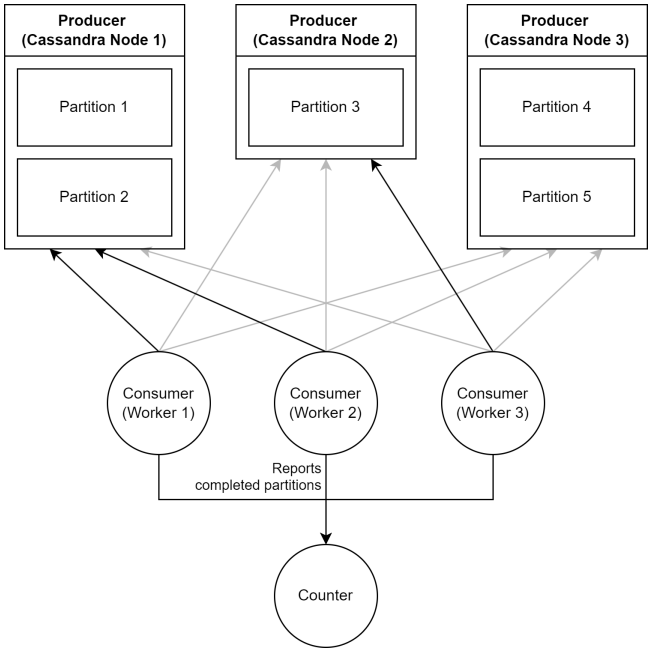


Figure 3.8.4: Producer Consumer Model Example

over all partial results in the data store, streaming the data back to the orchestrator. An actor system pipes the concurrent responses into a single thread, which combines the results and streams them to the frontend. The request is initiated by the orchestrator because gRPC requires that one system act as a server, and another as a client. For the query plan model, it makes most sense for the workers to be servers, so this model is also used for result collation.

3.9 Deployment

As discussed in Chapter 2, Kubernetes was chosen to manage all nodes in the cluster. One feature that makes it useful for the system is scheduling rules, which provide Kubernetes with information about how containers should be assigned to physical nodes.

As previously described in 3.6.2, workers determine their closest Cassandra node automatically based on latency. To make the best use of the cluster, workers should be distributed evenly across all nodes that have a Cassandra node. This is implemented by these scheduling rules:

1. If possible, workers should be placed on the same Kubernetes node as a Cassandra node.
2. Workers should not be placed on the same node as other workers.

The scheduling rules are preferences rather than requirements, meaning Kubernetes is still able to schedule the nodes if there are more workers than Cassandra nodes.

3.9.1 CI/CD

To aid in deploying to Kubernetes, continuous integration/continuous deployment (CI/CD) pipelines are used for each major system component, specifically using GitHub Actions [17]. Each pipeline runs all unit tests for the component, and provided they succeed, builds a docker container and pushes it to a container registry, ready for use in the Kubernetes cluster.

Chapter 4

Testing

As discussed in Chapter 1, the main objective is to improve the speed of data processing for large datasets, making it essential to conduct performance testing of the overall system. This section aims to cover a number of methods in which the performance of the completed solution is evaluated.

4.1 Unit Tests

Thorough unit testing is important for any software engineering focused project. By writing tests throughout development, the expected behaviour of individual components in the system can be validated. Through the use of continuous integration/continuous deployment (CI/CD) pipelines, this behaviour can continue to be validated as other parts of the system are improved, to ensure changes do not break the behaviour of the component.

Scalatest was chosen as the unit test framework [42]. Furthermore, both gRPC and Akka Actors provide classes for writing unit tests around the frameworks, and this is combined with Mockito to mock any dependencies that cannot be run during testing like the Cassandra Driver [41, 10]. Using these tools and frameworks, unit tests have been written for the majority of core code, including the DSL, query model and partitioning code. In total, more than 350 individual tests were written for this project, split across the Python frontend, core code, orchestrator and worker code.

4.2 Test Data

To aid performance testing, fake data is created to test the different operations of the system, produced based on discussions with potential users of the system. The intended users have a financial background, as they work within Audit at PwC, so loan origination (creation) data was selected for testing. A short Python script generates this data by randomising a number of fields between specified bounds. Figure 4.2.1 shows ten example records of the data.

4.3 SQL vs Cluster Solution

This section will compare the performance of an instance of Microsoft SQL Server, against the completed solution, referred to as the Cluster Processor.

Test Plan Tests are conducted for the three types of query that the Cluster Processor supports: Select, Filter and Group By. Within each type of query, a simple, and a complex version is tested.

Loan ID	Amount	Interest Rate	Duration (Yrs)	Origination Date
0	590,418	0.041139	24	2021-04-23 18:13:00
1	697,824	0.095023	20	2021-10-06 20:07:00
2	271,853	0.029358	23	2021-03-08 05:12:00
3	329,950	0.038111	23	2021-01-18 21:05:00
4	1,381,994	0.055411	30	2021-05-13 15:54:00
5	1,365,793	0.0093872	29	2021-05-04 03:18:00
6	1,143,926	0.078929	21	2021-07-11 19:10:00
7	461,215	0.082520	23	2021-05-04 17:50:00
8	287,307	0.040382	21	2021-05-20 06:08:00
9	191,668	0.061314	25	2021-09-03 16:21:00

Figure 4.2.1: Example Loan Origination Data

See Appendix A.1 for the full SQL and Cluster Processor queries.

Figure 4.3.1 shows the data volumes of the tables the queries will be executed on. SQL has two larger tables to provide further context of how it scales at larger data volumes; the Cluster Processor was unable to test these tables due to time and cost constraints.

SQL	Cluster Processor
1,000	1,000
10,000	10,000
100,000	100,000
1,000,000	1,000,000
10,000,000	10,000,000
50,000,000	
100,000,000	

Figure 4.3.1: SQL and Cluster Processor Testing - Number of Rows

To reduce the effect of random error, each test is run 5 times, and the results are averaged. This is particularly important for a cloud environment, as there is less control over the hardware running the tests.

Microsoft SQL Server was running on an instance of Azure SQL Database for these tests [6]. The Cluster Processor was running on Azure Kubernetes Service with a pool of three nodes, each having 4 vCores, and 16GB memory available [22]. The CPU and memory available to each worker is controlled by Kubernetes.

4.3.1 Controls

A number of variables must be considered which can impact the test results, including available CPU and memory, network latency and database warm-up periods. These variables are controlled so they are comparable between SQL and the Cluster Processor. Full details are provided in Appendix A.2.

4.3.2 Select Query

The first query is a pure select, essentially testing how fast both solutions can send results over the network. The second select query is more complex, with conversion operations to determine if this has an impact on the computation time.

Results The results for this query are shown in Figure 4.3.2. Data is only available for Cluster Processor up to 100,000 rows because of a memory issue when passing data from the workers to the orchestrator. This is analysed further in Chapter 5. Therefore, the graph is filtered to exclude the SQL results for 50 and 100 million rows.

The results show that the Cluster Processor is more than 10x slower than SQL. This is most likely related to the format used to send the result data. See Section 4.3.5 for further analysis.

For both SQL and the Cluster Processor, there is no significant difference between the simple and complex Select, which shows that the overhead for the operations in this query is insignificant.

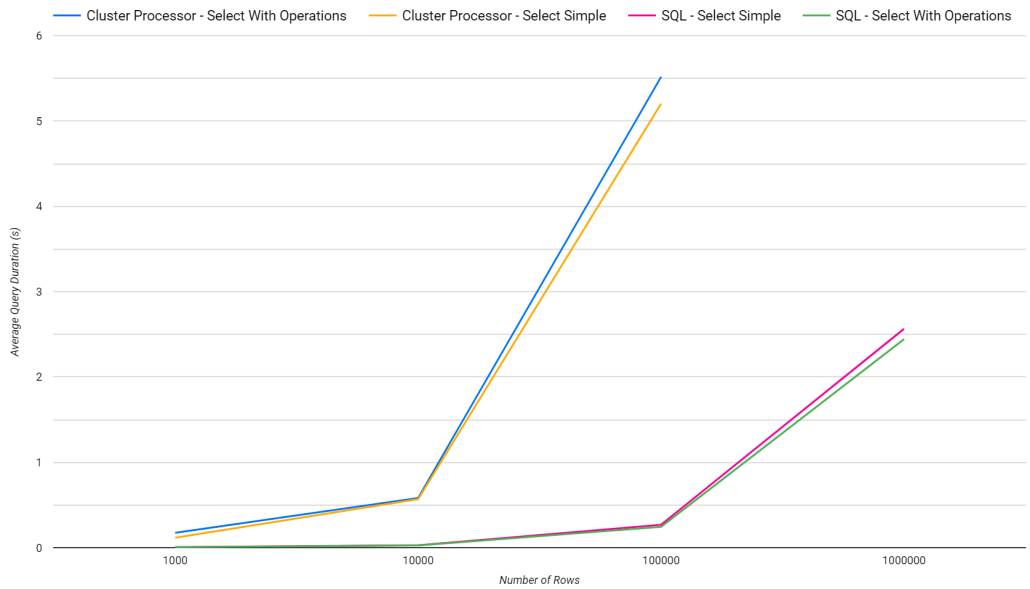


Figure 4.3.2: SQL vs Cluster Processor - Select Query Results

4.3.3 Filter Query

The first query is a simple filter, with no AND/OR combinations. The second filter is more complex, testing both boolean operators.

Results The results for this query are shown in Figure 4.3.3. SQL is around 15-20x faster in this test case, which is likely to be partially caused by the transmission format, as with the Select test. However, SQL is likely able to exploit caching more extensively over repeated tests when compared to the Cluster Processor, particularly with smaller tables that can be held in-memory permanently. Conversely, the Cluster Processor fetches fresh data from Cassandra every time the query is called.

The complex Filter reliably executes faster than the simple Filter across all results in both environments. This is expected, since the complex filter is more restrictive in the results that it returns, resulting in less data transferred over the network.

4.3.4 Group By Query

The first query is a simple group by, essentially performing a `DISTINCT` operation. The second group by is more complex, featuring a number of aggregations.

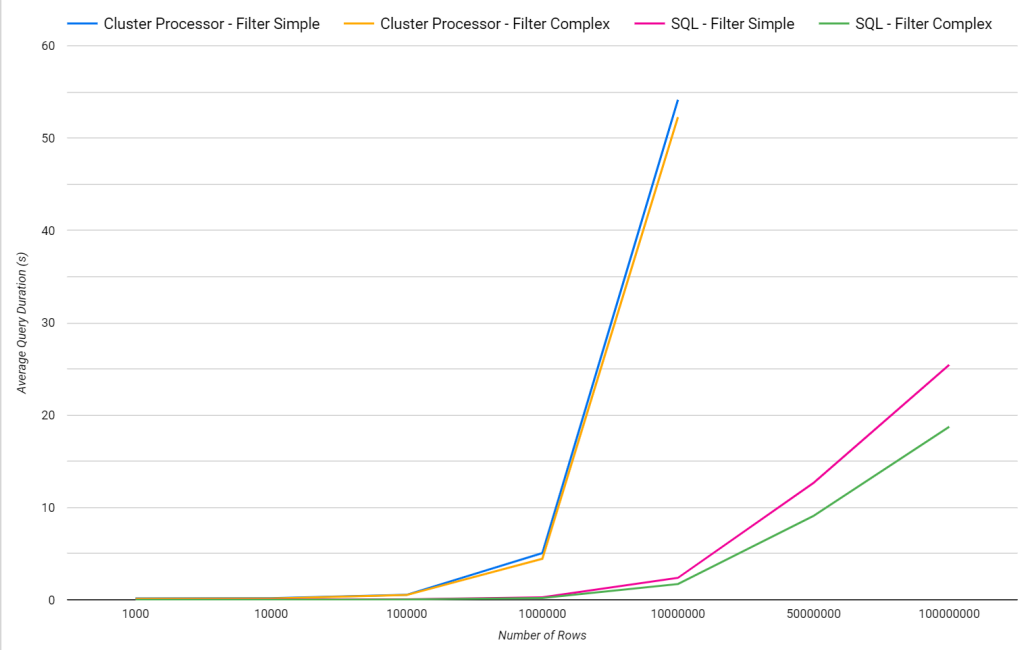


Figure 4.3.3: SQL vs Cluster Processor - Filter Query Results

Results The results for this query are shown in Figure 4.3.4. SQL is significantly faster, and scales much better as the data sizes increase. For the Cluster Processor, the aggregate group by is over twice as fast on average than the simple version at 10 million records. The testing was performed sequentially, with no break between tests, so it is unclear why this result is faster. Furthermore, the same difference in computation time is not present at smaller data volumes. This could be caused by the less controlled cloud environment, meaning perhaps some unexpected load was present during the simple test, resulting in slower computation. Another round of testing would need to be performed to determine if this was the case.

Despite this unexpected difference, there is still a significant performance drop-off compared to the results at 1 million records. Analysing the outputs from the workers, the results could not be stored entirely in-memory, resulting in a large amount of computation time being spent swapping partial results to and from disk. Group By operations suffer from memory shortages worse than other types of queries, as around twice the normal data volume is stored at one time: the source data for the Group By, data for the new hashes, and the newly computed group by partitions are all kept in the data store at the same time.

4.3.5 Analysis

As the raw performance testing results show, a significant amount of optimisation would be required for the Cluster Processor solution to truly compete with SQL with regards to computation speed. The system appears to be weakest at transmitting raw data quickly across the network, and computing Group Bys.

One way to optimise the transmission format for results is to reduce the size of the serialised row data. Currently, every serialised table cell stores value and type information, but types are already sent within the header. Therefore, the system could reduce the serialised size by only sending values, and using the header to perform type casts.

The main inefficiency causing the poor Group By performance is the workers holding onto more

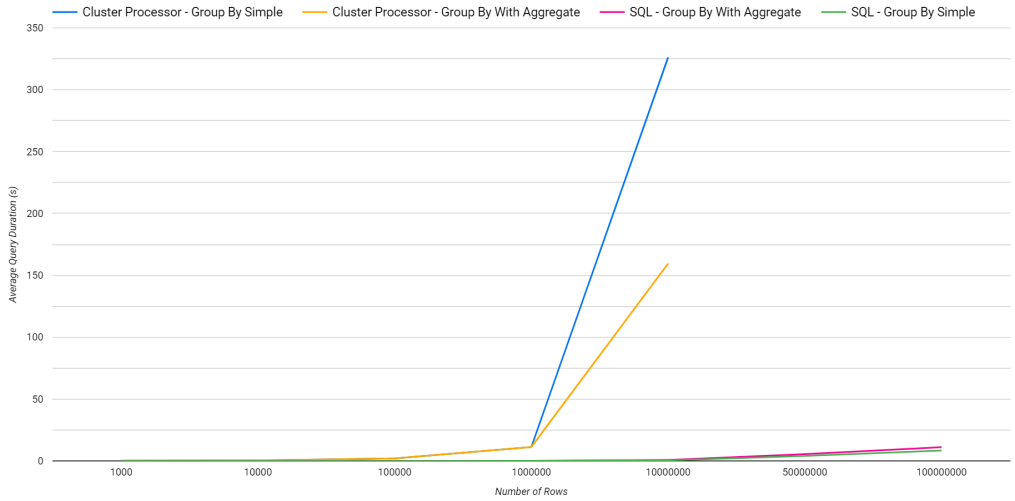


Figure 4.3.4: SQL vs Cluster Processor - Group By Query Results

data than necessary. Currently, the hashing process stores a copy of the entire result before cross-communication occurs. It should be possible to partially perform the Group By operation for each partition on each worker, then finalise the partial results when the partition is computed, significantly reducing the amount of stored and transferred data. As testing has already established that the transmission format is suboptimal, by sending more data than is required, the network inefficiencies have an increased impact on performance.

Furthermore, when computing Group Bys at 10 million rows, the workers ran low on memory, and began spilling data to disk, which is significantly slower than in-memory storage. By storing less data, spilling would occur at larger data volumes, improving the overall performance.

4.4 Level of Parallelisation

This section will compare the performance of the Cluster Processor when the number of workers is changed, but the overall available resources is the same. The aim is to determine how changing the level of parallelisation in the cluster impacts the computation speed.

In these tests the simple versions of each query type were executed, with different data volumes depending on the test. See Appendix A.1 for query details.

Figure 4.4.1 shows the cluster layouts for each test case; the number of workers, and the resources available to each worker. As shown, the overall number of vCores and GB of memory available to the cluster is the same in each case.

Throughout all of these tests, the number of Cassandra nodes in the cluster remained consistent: 3 nodes, one placed on each Kubernetes node.

4.4.1 Select Query

The results of this test are shown in Figure 4.4.2. Due to the same memory issue as in the SQL test, only 1000 to 100000 row tables were tested. The cluster layout with 3 nodes appears to execute around twice as fast across all data volumes compared to the other layouts, which all have similar execution times on average. It is likely that the extra overhead introduced by the increased parallelisation slowed

Workers	vCores	Worker Memory	Total vCores	Total Memory
2	3	9GB	6	18GB
3	2	6GB	6	18GB
6	1	3GB	6	18GB
9	0.666	2GB	6	18GB
12	0.5	1.5GB	6	18GB

Figure 4.4.1: Parallelisation - Number of Workers and Resources

down data transfer, and there was no computation to perform which would benefit the increased number of nodes.

However, at 100,000 rows the 9 node cluster ran marginally faster than the other slower layouts. These are still tightly clustered with all results within 0.4s of one another, suggesting this is likely caused by random variation, rather than the 9 node cluster being specifically faster. Further testing, particularly at higher data volumes, would be required to confirm any trends here.

Interestingly, the 2 node cluster performed slower than the 3 node cluster, likely because this layout has less workers than Cassandra nodes. This adds latency when importing data from the Cassandra node without a co-located worker.

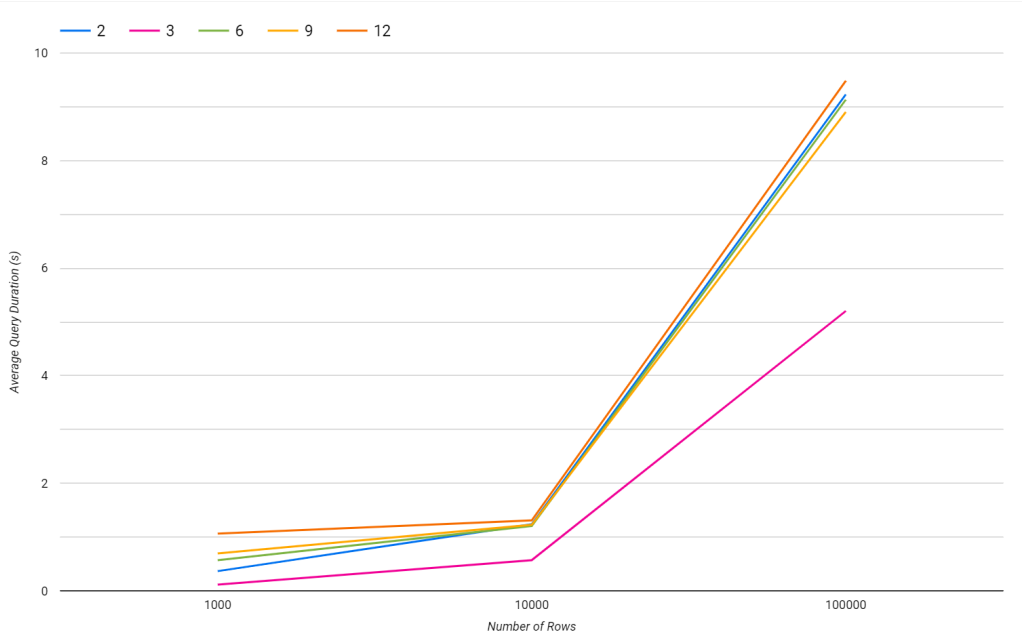


Figure 4.4.2: Parallelisation - Select Query Results

4.4.2 Filter Query

The results of this test are shown in Figure 4.4.3. The 3 node cluster is fastest until 100,000 rows, but is then overtaken by the 9 node cluster, showing an interesting trend.

To investigate this trend further, the test was also run for all cluster layouts at 10 million rows, shown in Figure 4.4.4. The larger clusters (6, 9 and 12 nodes) are all faster than the 3 node cluster, with 9 nodes executing quickest. This shows that as the amount of work increases, the increased level of parallelisation becomes a benefit. The fact that 12 nodes is slower than 9 suggests there is an

optimal point that maximises parallelisation without introducing too much overhead from the number of nodes.

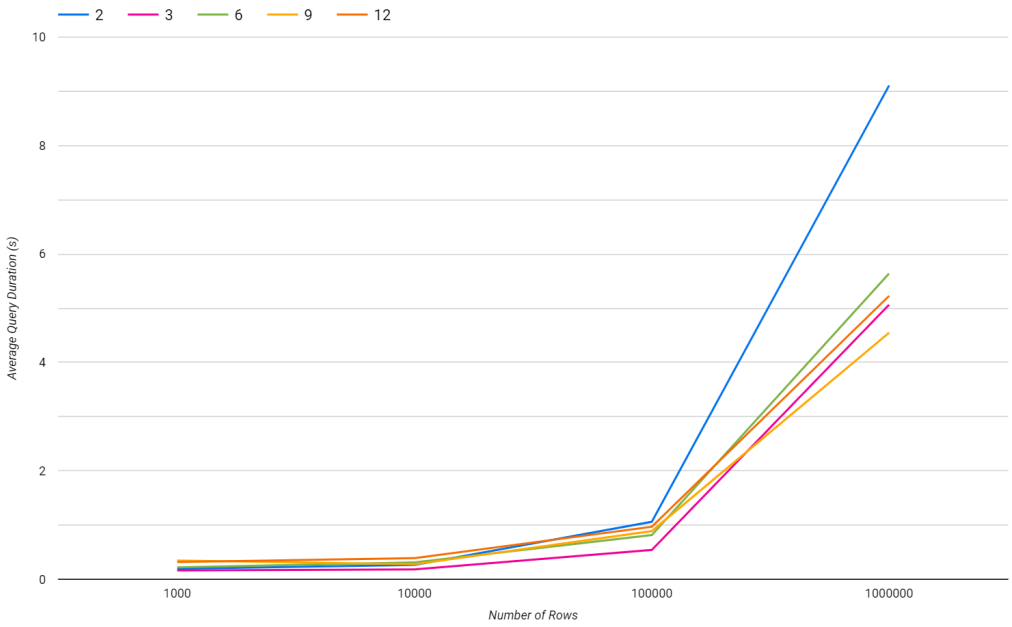


Figure 4.4.3: Parallelisation - Filter Query Results

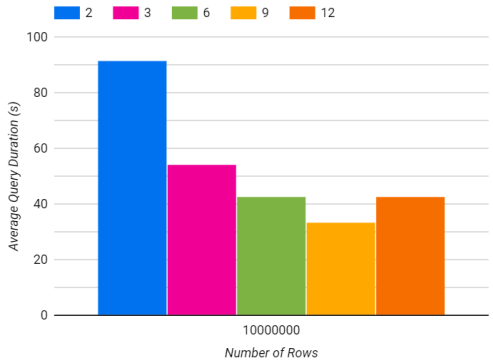


Figure 4.4.4: Parallelisation - Filter Query Results, 10 Million Rows

4.4.3 Group By Query

The results of this test are shown in Figure 4.4.5. They again suggest that there is a balancing point for the level of parallelisation. The 3 node cluster is consistently faster than all other layouts. The larger layouts may be slower because the increased parallelisation introduces more cross-communication, which is slower than local execution.

Interestingly, the 2 node cluster is second fastest until 1 million rows, when its performance significantly reduces. This is likely to be because the increased memory demands on each worker resulted in some results being spilled.

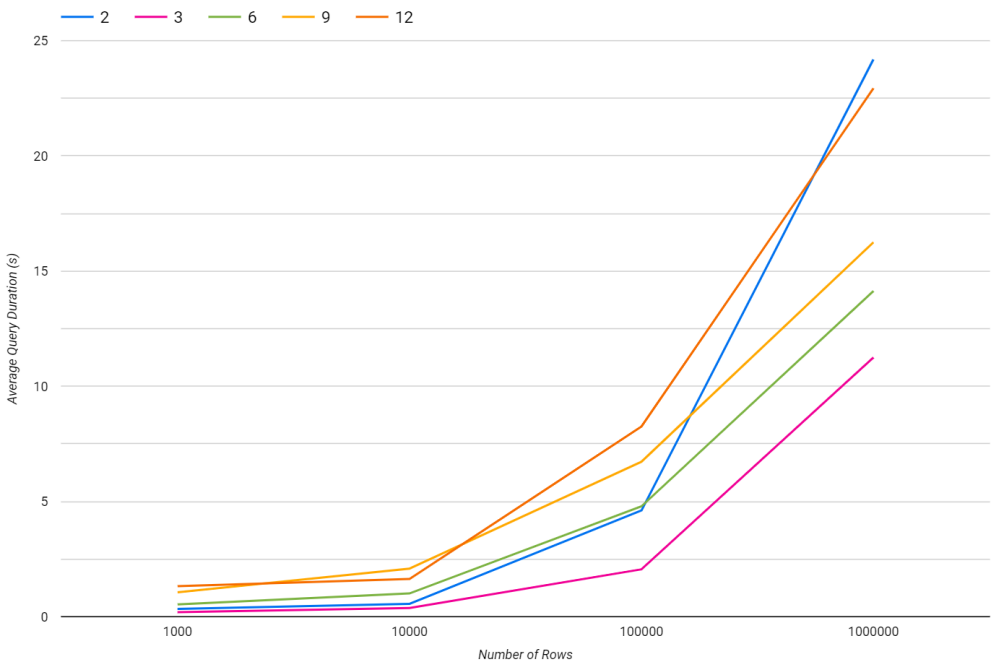


Figure 4.4.5: Parallelisation - Group By Query Results

4.4.4 Analysis

The outcomes of this test show that there is no clear solution to the level of parallelisation. As a general rule, having the same number of workers as Cassandra nodes will result in good performance, but the Filter test also shows that increasing parallelisation can result in better performance for the same resources at larger query sizes.

The best solution depends on the queries and data volumes being calculated. This presents an opportunity for further research designing a system that analyses the queries being executed, automatically adjusting the cluster layout to match the requirements of those queries.

4.5 Autoscaling

This section will compare computation speed of the Cluster Processor when reducing the overall performance, motivated by the autoscaling feature present in many cloud services, including Azure Kubernetes Service. It continually analyses the load of the Kubernetes cluster, changing the number of physical nodes based on current demand [5]. By doing this, applications with fluctating demand can save costs by reducing the number of machines they pay for when demand is low.

While the Cluster Processor would not currently support autoscaling, this test aims to identify the effectiveness of autoscaling on this solution. In these tests the simple versions of each query type were executed, with different data volumes depending on the test. See Appendix A.1 for query details.

Figure 4.5.1 shows the cluster layouts for each test case. Each layout has one less worker, and 33% less resources.

Throughout all of these tests, the number of Cassandra nodes in the cluster remained consistent: 3 nodes, one placed on each Kubernetes node.

Workers	vCores	Worker Memory	Total vCores	Total Memory
3	2	6GB	6	18GB
2	2	6GB	4	12GB
1	2	6GB	2	6GB

Figure 4.5.1: Autoscaling - Number of Workers and Resources

4.5.1 Select Query

The results of this test are shown in Figure 4.5.2. As expected, the query performance decreases as the number of workers decreases. However, at 1000 rows the difference between 3 workers and 1 worker is around 0.3s, and at 10000 rows it is 0.9s.

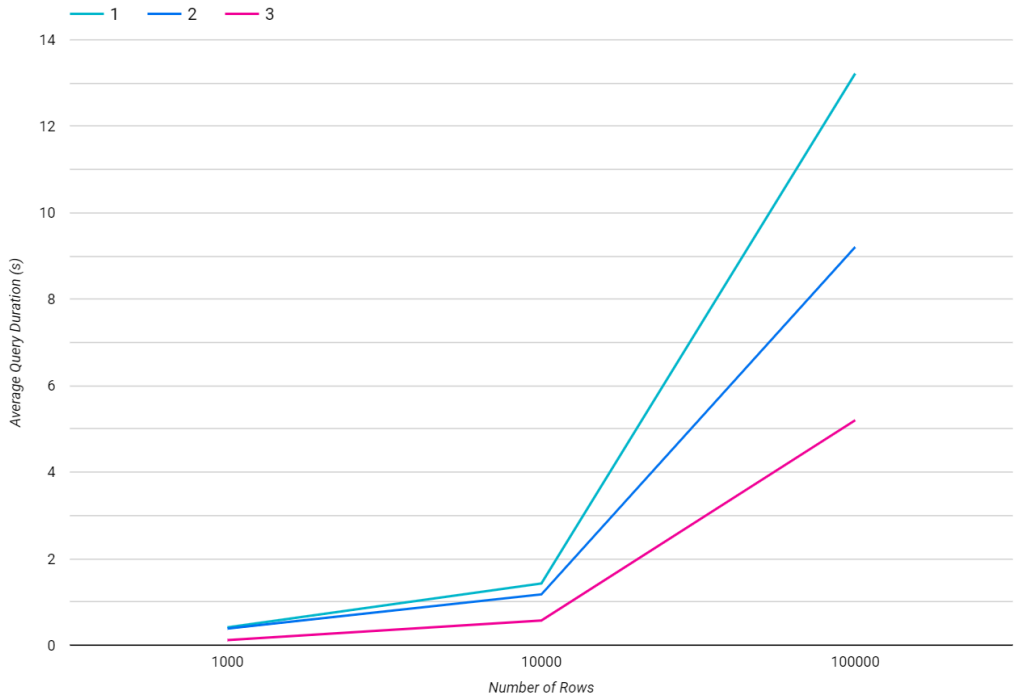


Figure 4.5.2: Autoscaling - Select Query Results

4.5.2 Filter Query

The results of this test are shown in Figure 4.5.3. As before, the query performance decreases with the number of workers. The difference between 1 and 3 workers is 0.08s at 1,000 rows, 0.2s at 10,000 rows, and 1.1s at 100,000 rows.

4.5.3 Group By Query

The results of this test are shown in Figure 4.5.4. At the two smallest volumes, there is almost no difference between the three layouts, and at 100,000 rows the 1 node cluster is fastest. This suggests that, at very small data volumes, it is faster to perform all of the computation on a single node, as it prevents the need for worker cross-communication.

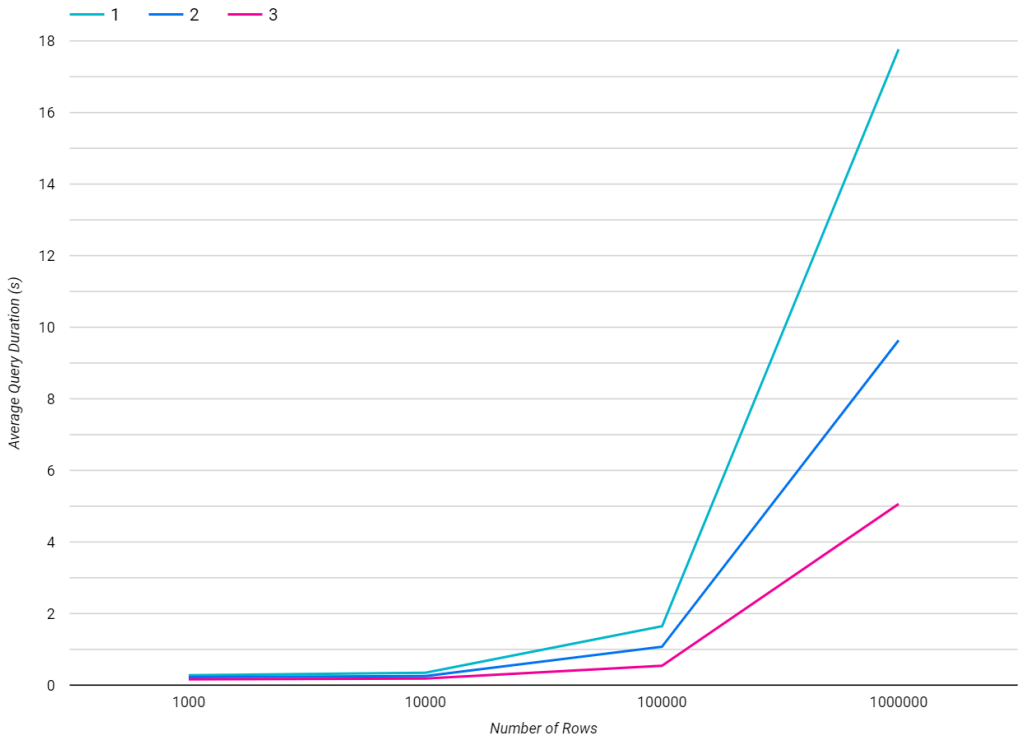


Figure 4.5.3: Autoscaling - Filter Query Results

4.5.4 Analysis

The outcomes of this test show that reducing the cluster size is effective at small data volumes. The time difference between the smallest and largest cluster is typically less than a second at less than 1 million rows, which is insignificant for most use cases. If a time-critical system was reliant on querying this amount of data, a single-system solution like SQL would be better suited regardless.

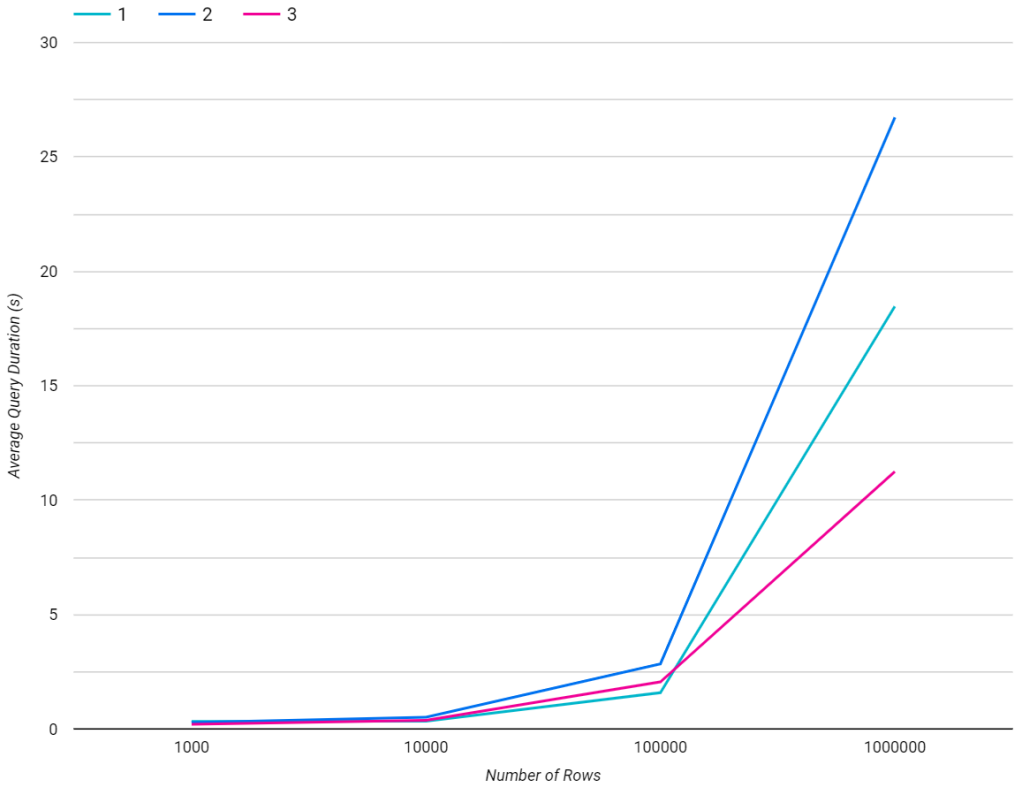


Figure 4.5.4: Autoscaling - Group By Query Results

Chapter 5

Evaluation

This section will discuss a high level evaluation of the solution, and the project as a whole.

5.1 Limitations

The solution has limitations which could not be fixed due to time constraints, but further investigation into optimisations or alternative solutions may be able to improve or fix them. These are listed below.

As discussed in Chapter 4, the Group By operation, and the method of transferring data around the network could both be further optimised.

Result data uses an extremely large amount of space when resident in memory. For example, a 100MB source data file can use up to 800MB of memory once stored. This is because of the container class described in Section 3.2. However, this class is a core component of the DSL, as it ensures the type information is accessible at runtime.

The system's security is limited, which prevents its use in a production environment. The orchestrator has no authentication, and Cassandra only has basic username and password authentication.

Finally, as discovered in Chapter 4, the result collation algorithm cannot return large amounts of results, typically more than 1 million rows. All workers send results to the orchestrator, which forwards them to the frontend. However, as there are more workers than the single orchestrator, data enters the orchestrator faster than it leaves, meaning that with a large enough dataset, the orchestrator will run out of memory and crash.

5.2 Further Work

The nature of this project means that there is a large scope for future work and improvements. As discussed in Section 4.4, different cluster layouts are more optimised for different kinds of queries. A module that runs at the Kubernetes level, monitoring the utilisation of the cluster and the types of queries being executed may be able to improve computation times by adjusting the cluster layout.

The data store is currently used to assist computations by temporarily storing partial result data. However, the design would allow it to store results between queries, improving the computation time of repeated queries to the same dataset. Join operations are also not currently implemented, but would benefit from this improvement to the data store.

The current error handling is designed to forward any errors to the frontend. In some situations, like if the Cassandra database is unresponsive, this is acceptable. For other errors, like if one worker is unresponsive, this can be handled by delegating the failed worker's partitions to others, without alerting the user at all.

Finally, Cassandra is currently only used for storage and partitioning. However, it is also a query engine, meaning some computations could be performed on Cassandra directly, improving query times by reducing the amount of data transferred. In particular, Filters on the source dataset, and Group Bys on the primary key are perfect candidates for this optimisation.

5.3 Conclusion

The objective as stated in Chapter 1 was to design a query processing engine for a distributed cluster of nodes. The types of queries possible in the system are numerous, and the data model allows easy implementation of new query types. While performance testing results showed that the solution requires further optimisation to truly compete with existing frameworks, they also showed that there is promise in the scalability of the solution. Furthermore, testing revealed interesting findings regarding the number of workers in the cluster, and raised the possibility of a stand-alone module for performing node management.

A secondary goal was to design the frontend to be easy-to-use, with SQL-like syntax. The DSL meets this goal, and is one of the defining features of the tool, with *FieldExpressions* and *FieldComparisons* allowing complex data manipulations to be defined with relative ease. The implementation of Functions permits easy extensions within the type system's bounds. The frontend operates seamlessly for the user, hiding the background operation of the framework entirely. Furthermore, the pandas integration means that users can immediately start manipulating result data using tools already familiar to them.

Bibliography

[1] *Actors — Akka Documentation.* URL: https://doc.akka.io/docs/akka/2.8.0/typed/index.html (visited on 04/03/2023).

[2] 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.

[3] 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.

[4] 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.

[5] *Automatically scale a cluster to meet application demands on Azure Kubernetes Service (AKS).* Available at: https://web.archive.org/web/20230403130017/https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler. URL: https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler (visited on 04/03/2023).

[6] *Azure SQL Databse Documentation.* Available at: https://web.archive.org/web/20230403130833/https://learn.microsoft.com/en-us/azure/azure-sql/database/?view=azuresql. URL: https://learn.microsoft.com/en-us/azure/azure-sql/database/?view=azuresql (visited on 04/03/2023).

[7] 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.

[8] 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).

[9] *ClassTag — Scala Documentation.* Available at: https://web.archive.org/web/20230403124654/https://scala-lang.org/api/3.2.2/scala/reflect/ClassTag.html. URL: https://scala-lang.org/api/3.2.2/scala/reflect/ClassTag.html (visited on 04/03/2023).

[10] *DataStax Java Driver Documentation.* Available at: https://web.archive.org/web/20230403123814/https://docs.datastax.com/en/developer/java-driver/4.14/. URL: https://docs.datastax.com/en/developer/java-driver/4.14/ (visited on 04/03/2023).

[11] *DataStax Python Driver Documentation.* Available at: https://web.archive.org/web/20230403123932/https://docs.datastax.com/en/developer/python-driver/3.26/. URL: https://docs.datastax.com/en/developer/python-driver/3.26/ (visited on 04/03/2023).

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] Debasish Ghosh. "Generics in Java and C++ a comparative model". In: *ACM SIGPLAN Notices* 39.5 (2004), pp. 40–47.

[17] *GitHub Actions Documentation*. URL: https://docs.github.com/en/actions (visited on 04/03/2023).

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

[19] *gRPC Documentation*. Available at: https://web.archive.org/web/20230403121905/https://grpc.io/docs/. URL: https://grpc.io/docs/ (visited on 04/03/2023).

[20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute Menlo Park, CA. 1973, p. 235.

[22] *Introduction to Azure Kubernetes Serice*. Available at: https://web.archive.org/web/20230403130550/https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes. URL: https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes (visited on 04/03/2023).

[23] ISO Central Secretary. *Date and time — Representations for information interchange — Part 1: Basic rules*. en. Standard ISO/8601-1:2019. Geneva, CH: International Organization for Standardization, 2019. URL: https://www.iso.org/standard/70907.html (visited on 03/27/2023).

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

[25] Elahe Khatibi and Seyedeh Leili Mirtaheri. "A dynamic data dissemination mechanism for Cassandra NoSQL data store". In: *The Journal of Supercomputing* 75 (2019), pp. 7479–7496. DOI: 10.1007/s11227-019-02959-7. URL: https://doi.org/10.1007/s11227-019-02959-7.

[26] *Kubernetes API Documentation*. Available at: https://web.archive.org/web/20230403121002/https://kubernetes.io/docs/reference/kubernetes-api/. URL: https://kubernetes.io/docs/reference/kubernetes-api/ (visited on 04/03/2023).

[27] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS operating systems review* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[28] 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.

[29] Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces.* ” O’Reilly Media, Inc.”, 2011.

[30] *multiprocessing - Process-based-parallelism.* Available at: https://web.archive.org/web/20230403123637/https://docs.python.org/3.10/library/multiprocessing.html. URL: https://docs.python.org/3.10/library/multiprocessing.html (visited on 04/03/2023).

[31] *Murmur Hash GitHub Repository.* URL: https://github.com/aappleby/smhasher (visited on 04/04/2023).

[32] 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.

[33] *Orchestration — Docker Documentation.* Available at: https://web.archive.org/web/20230403120602/https://docs.docker.com/get-started/orchestration/. URL: https://docs.docker.com/get-started/orchestration/ (visited on 04/03/2023).

[34] *Overview of Azure Cloud Shell.* Available at: https://web.archive.org/web/20230403130921/https://learn.microsoft.com/en-us/azure/cloud-shell/overview. URL: https://learn.microsoft.com/en-us/azure/cloud-shell/overview (visited on 04/03/2023).

[35] *Protocol Buffers Documentation.* Available at: https://web.archive.org/web/20230403122839/https://protobuf.dev/. URL: https://protobuf.dev/ (visited on 04/03/2023).

[36] *RDD Programming Guide.* Available at: https://web.archive.org/web/20230406124102/https://spark.apache.org/docs/latest/rdd-programming-guide.html. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html (visited on 04/06/2023).

[37] *Runtime — Java 8 Documentation.* URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html (visited on 04/03/2023).

[38] *Scala 3 Documentation.* Available at: https://web.archive.org/web/20230403115618/https://www.scala-lang.org/api/3.2.2/. URL: https://www.scala-lang.org/api/3.2.2/ (visited on 04/03/2023).

[39] *Scala for Java Developers — Scala 3 - Book — Scala Documentation.* Available at: https://web.archive.org/web/20230403120751/https://docs.scala-lang.org/scala3/book/scala-for-java-devs.html. URL: https://docs.scala-lang.org/scala3/book/scala-for-java-devs.html (visited on 04/03/2023).

[40] *ScalaPB: Protocol Buffer Compiler for Scala.* Available at: https://web.archive.org/web/20230403122538/https://scalapb.github.io/. URL: https://scalapb.github.io/ (visited on 04/03/2023).

[41] *ScalaTest Mockito User Guide.* Available at: https://web.archive.org/web/20230403125722/https://www.scalatest.org/plus/mockito. URL: https://www.scalatest.org/plus/mockito (visited on 04/03/2023).

[42] *ScalaTest User Guide.* Available at: https://web.archive.org/web/20230403125549/https://www.scalatest.org/user_guide. URL: https://www.scalatest.org/user_guide (visited on 04/03/2023).

[43] Raj Srinivasan. *RPC: Remote procedure call protocol specification version 2.* RFC 1831. 1995. DOI: 10.17487/RFC1831. URL: https://www.rfc-editor.org/rfc/rfc1831.

[44] *Stack Overflow Developer Survey 2022*. Available at: https://web.archive.org/web/20230327142640/https://survey.stackoverflow.co/2022/. URL: https://survey.stackoverflow.co/2022 (visited on 03/27/2023).

[45] *Swarm Mode Overview*. Available at: https://web.archive.org/web/20230403120710/https://docs.docker.com/engine/swarm/. URL: https://docs.docker.com/get-started/orchestration/ (visited on 04/03/2023).

[46] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[47] 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.

[48] 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.

[49] *Worker Threads — Node.js v19.8.1 Documentation*. Available at: https://web.archive.org/web/20230403123623/https://nodejs.org/api/worker_threads.html. URL: https://nodejs.org/api/worker_threads.html (visited on 04/03/2023).

[50] 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.

[51] 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.

[52] 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.

[53] 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.

Appendix A

Testing Queries and Controls

This appendix provides details of the queries used for performance testing, and the controls used in the SQL test.

A.1 Performance Testing Queries

Included below are figures with the queries used for performance testing.

```
SELECT * FROM data.origination_1000
```

Figure A.1: SQL - Select Simple

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000").evaluate()
```

Figure A.2: Cluster Processor - Select Simple

```
SELECT
Loan_ID + 1 as Loan_ID_Inc,
interest_rate + 1 as Interest_rate_Inc,
power(duration, 2) as Duration_Pow,
substring(cast(origination_date as nvarchar(300)), 0, 11) as
   origination_date_str
FROM data.origination_1000
```

Figure A.3: SQL - Select With Operations

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000").select(
(F("loan_id") + 1).as_name("loan_id_inc"),
(F("interest_rate") + 1).as_name("interest_rate_inc"),
Function.Pow(Function.ToDouble(F("duration")),
   2.0).as_name("duration_pow"),
Function.Substring(Function.ToString(F("origination_date")), 0,
   10).as_name("origination_date_str")
).evaluate()
```

Figure A.4: Cluster Processor - Select With Operations

```
SELECT *
FROM data.origination_1000
WHERE duration = 30
```

Figure A.5: SQL - Filter Simple

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000")
.filter(F("duration") == 30)
.evaluate()
```

Figure A.6: Cluster Processor - Filter Simple

```
SELECT *
FROM data.origination_1000
WHERE
(duration = 30 AND amount > 500000)
OR loan_id = 1
```

Figure A.7: SQL - Filter Complex

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000")
.filter(
((F("duration") == 30) & (F("amount") > 500000.0))
| (F("loan_ID") == 1)
).evaluate()
```

Figure A.8: Cluster Processor - Filter Complex

```
    SELECT duration
    FROM data.origination_1000
    GROUP BY duration
```

Figure A.9: SQL - Group By Simple

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000")
.group_by([F("duration")])
.evaluate()
```

Figure A.10: Cluster Processor - Group By Simple

```
SELECT
duration,
MAX(origination_date) as Max_origination_date,
AVG(interest_rate) as Avg_interest_rate,
Min(amount) as Min_amount
FROM data.origination_1000
GROUP BY duration
```

Figure A.11: SQL - Group By With Aggregates

```
manager = ClusterManager("orchestrator-service")
manager.cassandra_table("data", "origination_1000")
.group_by(
[F("duration")],
[       Max(F("origination_date")),
Avg(F("interest_rate")),
Min(F("amount"))
]
).evaluate()
```

Figure A.12: Cluster Processor - Group By With Aggregates

A.2 Test Controls

Included below are full details of the controls used in the SQL performance test.

A.2.1 CPU and Memory

At larger data volumes, CPU and Memory is the biggest contributing factor that will affect how quickly the computation is performed, both on SQL and the Cluster Processor. Ensuring these are comparable is essential for producing reliable test results. For Azure SQL Database, a slider can be used to set the maximum number of vCores available, and a set amount of memory is assigned based on the number of cores. In this case, a maximum of 6 vCores were used, which results in of 18GB memory accessible to the database.

As the Cluster Processor is running on Kubernetes, granular control over the number of vCores and amount of memory available to each node is possible using resource limits [26]. Workers were configured to have a maximum of 2 vCores, and 6GB memory available to each, with 3 workers in total. As a result, the cluster as a whole has 6 vCores and 18GB memory available, the same as the SQL database.

A.2.2 Network Latency

Controlling network latency is particularly important for small data volumes which resolve quickly. For a request that takes 0.5s to complete, 100ms of latency will make the completion 20% slower. Testing was always performed on an instance of Azure Cloud Shell, which is a terminal running inside the same Azure datacenter as the test environment [34]. This ensures the network latency is minimised and comparable between environments, with 16ms average round-trip time for a TCP ping to both SQL server, and the Cluster Processor.

A.2.3 Warm-Up

Both SQL and Cluster Processor have a warm-up periods when they are first started. Azure SQL Database is run using a serverless computation style, which means the server is scaled to 0 resources when it is unused. This has the disadvantage that when a query is first run, there is a short delay while the resources are provisioned again. Cluster Processor has a similar warm-up when it is first started, because the gRPC connections between the orchestrator and workers are not actually created until the first request is made. To overcome both of these warm-up periods, a number of queries are run just before testing begins, and the time taken to run these is not tracked.

Appendix B

Project Folder Structure and Execution Instructions

This appendix describes the folder structure of the Git repository, and also includes instructions for executing the Cluster Processor.

B.1 Folder Structure

There are a number of folders in the root of the project. Details of each folder, as well as execution instructions, are included below:

- `kubernetes`: this folder contains .yaml files required for initialising the Kubernetes cluster and creating resources like the orchestrator, workers and Cassandra cluster within it.
- `protos`: this folder contains protobuf definition files, which are used by both the python client, and the orchestrator and worker nodes.
- `python_client`: this folder contains all code related to the Python frontend, and performance testing code.
- `report`: this folder contains the full project report, along with all source LaTeX files and images used for generating the report.
- `server`: this folder contains three submodules.
  - `core`: this folder contains a Scala project with the core code shared by the worker and the orchestrator modules.
  - `orchestrator`: this folder contains a Scala project with orchestrator-specific code, but it depends on the core module.
  - `worker`: this folder contains a Scala project with worker-specific code, but it depends on the core module.

B.2 Execution Instructions

There are two main ways of executing this project:

- Kubernetes: this is best for use in a production environment.

- Local: this is best for use in testing, but is restricted to execution on a single machine.

These instructions are also included in the `README` files of the project directory, with direct file links to any files referenced in the instructions.

B.2.1 Kubernetes Execution

Testing on Kubernetes was performed using Azure Kubernetes Service [22], and as such all yaml files in the repository are provided with this in mind. Changes will need to be made for alternative cloud providers.

1. Create an instance of Azure Kubernetes Service.
   - NOTE: each node should have at least 2 vCores and 8GB RAM available.
2. Run create_azure_cluster.sh. The number of Cassandra nodes in the cluster can be customised in demo-cluster.yaml, line 24.
   - These files can be found in `/kubernetes/azure/`.
   - NOTE: there should be as many Kubernetes nodes as the number of Cassandra nodes, and each node should have 4GB RAM free for Cassandra.
3. Build docker images for the python_client, and the orchestrator and worker nodes.
   - The python_client folder contains a docker file, which can be built normally.
   - The orchestrator and worker folders must be built using `sbt docker:publishLocal`, which will build and publish to the local docker engine.
4. Store the built docker images in a container registry which the Kubernetes service can access. Azure Container Registry can be configured to allow access from the Kubernetes service.
5. Edit the yaml file for the worker (`kubernetes/configs/worker.yaml`):
   - Change the number of replicas (line 8) to the desired amount - 1 worker per node is the recommended.
   - Change the image (line 20) to point to the chosen container registry.
   - Change the resource requests and limits to the chosen amount - generally, the more CPU and memory available, the better.
   - *Required only if changes were made to demo-cluster.yaml.* Change the following variables to match the configuration of demo-cluster.yaml:
     - Datacenter Name (line 37)
     - Cassandra All Pods Service Base URL (line 43): `{cluster-name}-{datacenter-name}-all-pods-service`
     - Number of Cassandra Nodes (line 45)
     - Cassandra Node Name (line 47): `{cluster-name}-{datacenter-name}-default-sts`
     - Cassandra Authentication Secret (line 54): `{cluster-name}-superuser`
     - Pod Affinity Cluster (line 70): `{cluster-name}`
     - Pod Affinity Datacenter (line 75): `{datacenter-name}`

6. Edit the yaml file for the orchestrator (`kubernetes/configs/orchestrator.yaml`):
    - Change the image (line 19) to point to the chosen container registry.
    - Change the number of workers to the number of configured replicas (line 40)
    - *Required only if changes were made to demo-cluster.yaml.* Change the following variables to match the configuration of demo-cluster.yaml:
        - Cassandra Service URL (line 27): `{cluster-name}-{datacenter-name}-service`
        - Datacenter Name (line 29)
        - Cassandra Authentication Secret (line 44): `{cluster-name}-superuser`
7. Create the orchestrator and worker configurations using `kubectl`:
    - `kubectl apply -f orchestrator.yaml`
    - `kubectl apply -f worker.yaml`
8. Kubernetes will create the appropriate pods, and the orchestrator will be available at orchestrator-service inside the cluster.

The instructions below can be used create an interactive pod to execute Python DSL commands in. The following command will start bash in an interactive container containing the python code.

```
kubectl run python-shell --rm -i --tty --image oliverlittle.azurecr.io/python_client --bash
```

Replace `oliverlittle.azurecr.io` with your chosen container registry. From here, you can generate new data and import it into the server.

To run inserts, you will need access to the Cassandra username and password:

- This command will get the username, although it's likely that this will be `demo-superuser`:
    - `kubectl get secrets/demo-superuser --template="{{.data.username}}" | base64 -d`
- This command will get the password (randomised per cluster):
    - `kubectl get secrets/demo-superuser --template="{{.data.password}}" | base64 -d`

Python can be started from the entrypoint of the interactive container as follows:

- `python -i main.py`

From there, create a CassandraConnector:

- `connector = CassandraConnector("demo-dc1-service", 9042, username, password)`

Run an insert:

- `CassandraUploadHandler(connector).create_from_csv("/path/to/file.csv", "keyspace", "table", ["partition", "keys])`

Query the cluster:

- `ClusterManager("orchestrator-service").cassandra_table("keyspace", "table").evaluate()`

B.2.2 Local Execution

Local execution requires the following to be installed:

- Docker Desktop
- sbt, with Java 8 or 11.

Instructions to setup the cluster locally are included below:

1. Start Cassandra on docker, with port 9042 exposed to the local machine using the following command:
    - `docker run -d --name cassandra -p 9042:9042 cassandra`
2. Set the paths to each of the workers in line 10 of application.conf, and save the file. As this is running locally, each worker will be running on localhost, and each will have to bind to a different port.
    - application.conf can be found in `server/core/src/main/resources`.
3. Run the orchestrator node using `sbt run` from the orchestrator directory: `/server/orchestrator`.
4. Run each of the worker nodes in a separate terminal instance from the worker directory: `/server/worker`. Ensure the ports defined in application.conf match the ports the workers are created with.
    - For example, if a worker needs to be assigned to port 50051, use the command `sbt "run 50051"`
5. Finally, in a separate terminal instance, start an interactive Python shell using `python -i main.py` from the python_client directory: `/python_client/`.
    - Connect to the orchestrator using `ClusterManager("localhost")`
    - Create a CassandraConnector to insert data using `connector = CassandraConnector("localhost", 9042)`

Appendix C

Dependency Code

This appendix describes which parts of the code are executed using dependency code from other projects. Most dependencies are listed in the build.sbt file for scala, and requirements.txt for python. Therefore, the actual code for these dependencies is not in the project directory, but is available to import.

For Scala, this includes:

- DataStax Java Driver
- scalapb and its gRPC implementation
- Akka Actors
- scalatest
- slf4j

For Python, this includes:

- DataStax Python Driver
- pandas
- gRPC (Python implementation)
- pytest
- tqdm

The file `SizeEstimator.scala`, from the Spark project, is also external code. This file was taken from the Spark repository, which is a Scala 2 project, and minor changes were made to adapt it for Scala 3. A standalone Scala 2 adaptation was also referenced to aid these changes.

- The file in Spark repository can be found here.
- The standalone Scala 2 adaptation can be found here.
- The LICENSE from Spark for this file (Apache 2.0) can be found here.
- The NOTICE from Spark for this file can be found here.