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: x

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 presented, 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 around 100GB of raw data, single system solutions like SQL begin to struggle as not all of the 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 perform 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.

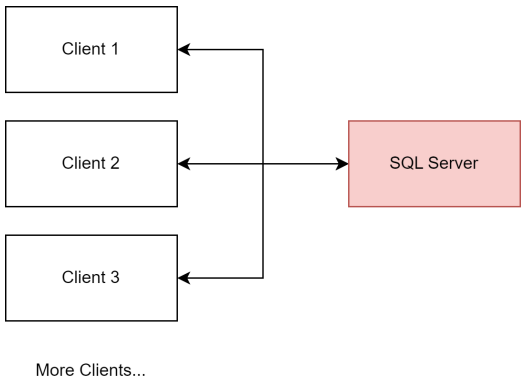


Figure 1.0.1: Single System Solution

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 some external source, processed in some relatively simple 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. This is usually a unit item within the dataset, for example 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; larger datasets can be accommodated by simply increasing the number of nodes in the cluster. Similarly, to an extent a query can always be processed faster by increasing the number of nodes, or the performance of each node.

1.1 Prior Work

Distributed Data Processing has existed conceptually since as early as the 1970s. A key paper by Philip Enslow Jr. from this period sets out characteristics across three 'dimensions' of decentralisation: hardware (the number of machines involved in the computation), control (the management of the cluster), and database (decentralisation of storage) [**enslow1978distributed**]. Enslow argued that these dimensions defined a distributed system, while also acknowledging that the technology of the period was not equipped to fulfil the goals he laid out. Further Rrsearch into distributed data processing has generally resulted in two kinds of [**yaqoob2016big**]:

- **Batch processing:** where data is gathered, processed and output all at the same time. This includes solutions like MapReduce and Spark [**dean2008mapreduce**, **zaharia2016spark**]. Batch processing works best for data that can be considered 'complete' at some stage.
- **Stream processing:** where data is processed and output as it arrives. This includes solutions like Apache Flink, Storm, and Spark Streaming [**carbone2015flink**, **toshniwal2014storm**, **armbrust2018sparkstreaming**]. Stream processing works best for data that is being constantly generated, and needs to be analysed as it arrives.

MapReduce, a framework introduced by Google in the mid 2000s, could be considered the breakthrough framework for performing massively scalable, parallelised data processing [**dean2008mapreduce**]. This framework later became one of the core modules for the Apache Hadoop suite of tools. It provided a simple API, where developers could describe a job as a *map* and a *reduce* step, and the framework would handle the specifics of managing the distributed system.

MapReduce was not without flaws, and many papers were published in the years following its initial release which performed performance benchmarks, and analysed its strengths and weaknesses [**lee2012parallel**]. Crucially, MapReduce appears to particularly struggle with iterative algorithms which are performed 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 the performance on iterative algorithms, like Twister and HaLoop both in 2010 [**ekanayake2010twister**, **bu2010haloop**].

MapReduce was challenging to use for developers used to more traditional data processing tools like SQL due to its imperative programming style, resulting in a number of tools being created to improve its usability and accessibility. Hive is one such tool, which features a SQL-like language called HiveQL to allow users to write declarative programs that compiled into MapReduce jobs [**thusoo2010hive**]. Pig Latin is similar, and features a mixed declarative and imperative language style that again compiles down into MapReduce jobs [**olston2008pig**]. Further tools in the wider areas of the field were introduced around 2010, including another project by Google named Pregel, specialised for performing distributed data processing on large-scale graphs [**malewicz2010pregel**].

In 2010, the first paper on Spark was released [**zaharia2010spark**]. Spark aims to improve upon MapReduce's weaknesses, by storing data in memory and providing fault tolerance by tracking the 'lineage' of data. For any set of data, Spark knows how the data was constructed from another persis-

tent 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 [**zaharia2012rdd**]. This approach was successful at resolving MapReduce's weakness for iterative algorithms, as keeping the data in-memory removes the read-write overhead that caused MapReduce's performance problems.

Spark quickly grew in popularity, with a number of extensions being added to improve its usability, including a SQL-style engine with a query optimiser, as well as an engine supporting stream processing [**armbrust2015sparksql**, **armbrust2018sparkstreaming**]. A second paper released in 2016 stated that Spark was in use in thousands of organisations, with the largest deployment running an 8,000 node cluster holding 100PB of data [**zaharia2016spark**]. Spark is designed to be agnostic of any particular storage mechanism, instead utilising existing Apache Hadoop connectors to retrieve data from various sources [**rddprogrammingguide**]. 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 data import speed through close integration with 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 the volumes of data, the fact that datasets can no longer ever be considered 'complete', along with demands for improved insight into the data means that streaming 'dataflow' models are the way forward [**akidau2015dataflow**]. Google publicly stated in their 2014 'Google I/O' Keynote that they were phasing out MapReduce in their internal systems [**googleio2014**].

While streaming solutions appear to be the direction of the wider industry, they are more suited for situations 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 will not be in scope for the core project, there is an opportunity for further investigation into this as an extension.

1.2 Project Aims

The objective of this project 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. This will allow it to be used for ETL workflows, where large data volumes often cause problems. Finally, as this will be designed as an all-in-one solution, the system should attempt to exploit the integration between the storage mechanism and nodes performing the computation to improve the execution speed. This appears to be an area where existing distributed data processing solutions struggle to perform as effectively. 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 solution. Chapter 4 provides full details of the tests.

1.3 Challenges

The project at this stage presents a few 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 and understand, but expressive

enough so the user can define any computation they need to. 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 aim of this process was to ensure that the limited development time for this project was spent developing the most effective features.

2.1 MoSCoW Requirements

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

F / NF	Priority	Requirement Description
Domain Specific Language - Expressions		
F	M	The language must support 5 data types: integers, floats, booleans, strings and date-time objects.
F	M	The language must 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 the user to write queries in Python.
F	M	The system must allow users to apply Select operations on datasets, applying custom expressions to the input data.
F	M	The system must allow users to apply Filter operations on datasets, applying custom comparisons to the input data.
F	M	The system must allow users to apply Group By operations on datasets, which take a number of expressions as unique keys, and a number of aggregate.
F	M	The Group By operation must allow users to apply Minimum, Maximum, Sum and Count aggregate functions to Group By operations.
F	C	The system could allow users to apply Distinct Count, String Aggregate, and Distinct String Aggregate aggregate functions to Group By operations.
F	S	The system should allow users to join two datasets together according to custom criteria.
NF	S	The complexities of the system should be hidden from the user; from their perspective the operation should be identical whether the user is running the code locally or over a cluster.
Cluster		
F	M	The system must allow the user to upload source data to a permanent data store.
F	M	The orchestrator node must split up the full query and delegate partial work to the worker nodes.
F	M	The orchestrator node must collect partial results from the cluster nodes to produce the overall result for the user.
F	M	The orchestrator node must handle worker node failures and other computation errors by reporting them to the user.
F	S	The orchestrator should perform load balancing to ensure work is evenly distributed among all nodes.
F	C	The orchestrator could handle worker node failures by redistributing work to active workers.
F	M	The worker nodes must accept partial work, compute and return results to the orchestrator.
F	M	The worker nodes must pull source data from the permanent data store.
F	M	The worker nodes must report any computation errors to the orchestrator.
F	S	The worker nodes should cache results for reuse in later queries.
F	S	The worker nodes should spill data to disk storage when available memory is low.

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 for the framework, because the requirements for the user interface suit an entirely different type of language to the requirements of the components performing the computation.

2.2.1 Frontend

The most important requirement for the front-end was to use SQL-like syntax. Pure SQL requires a parser in order to derive the actual query from the text, which would add a significant amount of development time to implement correctly, so a language which would be able to encode queries as classes and functions was preferred. Therefore, Python was selected as the language of choice for the frontend, as according to the 2022 StackOverflow Developer Survey, it is the fourth most popular language [**stackoverflowsurvey2022**]. Python also supports the second and third most popular non-web frameworks, NumPy and pandas, both designed for data processing and analysis [**reback2020pandas**, **harris2020array**]. This choice means that it is likely the widest range of users will have prior experience with Python, and there is the possibility of integrating with these frameworks to make the system even easier to use. Furthermore, Python is much more lightweight than many typical object-oriented languages, which would make developing and iterating upon the frontend much faster.

2.2.2 Orchestrator and Worker Nodes

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

The remaining options were a range of object-oriented and functional languages. Due to the nature of the project, much of the implementation would be CPU intensive, requiring iterating over large lists of items, so a language with strong support for parallelisation was preferable. This also ruled out languages like Python and JavaScript where parallelisation requires manual implementation by the developer [**pythonmultiprocessing**, **nodeworkerthreads**].

In the end, Scala was chosen [**scaladocs**]. It features a mix of both object-oriented and functional paradigms, and the ability to use both paradigms would allow for the best option to be selected for each task. Furthermore, Scala has built-in support for parallelised operations through operations like *map* and *reduce*, which is particularly helpful for iterating or combining the rows of a dataset. Furthermore, it is built on top of, and compiles into Java. This means that packages originally written for Java can be executed in Scala, which proved useful for later design decisions [**scalaforjavadevs**].

2.3 Runtime

2.3.1 Containerisation

With the nature of the project being to produce a distributed system, the clear choice for executing the code was within containers. Docker is one of the most popular options for creating and executing containers, but is not suitable for running and managing large numbers of containers [**orchestrationdockerdocs**]. For this, a container orchestration tool is required, and there are two main options: Docker Swarm, and Kubernetes [**dockerswarm**, **k8sapi**]. Docker Swarm is more closely integrated within the Docker ecosystem, being directly integrated into the Docker engine. However, many cloud services feature managed Kubernetes services which handle the complexity of creating and managing a cluster. For this reason, Kubernetes was chosen for the container orchestration tool.

2.3.2 Network Communication

A communication method had to be selected to allow the Python frontend, the orchestrator and the worker nodes to communicate. REST API frameworks initially appeared to be a suitable option, but upon further research, remote procedure call (RPC) frameworks would be more suited to the project's needs. This is because REST is designed to be stateless, providing a standard set of operations - create, read, update, delete (CRUD) [**masse2011rest**]. The proposed solution is more focused on stateful operations, so the CRUD model would not fit the requirements correctly.

In contrast, RPC frameworks are designed to allow function calls over a remote network [**srinivasan1995rpc**]. They are also typically not designed around a particular data model like REST, which would allow for the implementation of a custom query model.

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

2.4 Persistent Storage

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

- Single System SQL Databases *(Microsoft SQL Server, PostgreSQL, MySQL)*: while this option would be fastest to start using due to extensive usage in industry, the database would quickly become a bottleneck, as the rate at which data can be read from the server would determine how quickly computations could be performed.
- Distributed File Systems *(Hadoop Distributed File System)*: these provide a mechanism for storing files resiliently across a number of machines, which would reduce the bottleneck when reading data. However, they provide no straightforward way of querying the stored data, so this feature would have to be implemented manually.

- Distributed NoSQL Databases *(MongoDB, CouchDB)*: these are distributed, meaning the load of reading the data could be spread across a number of machines. However, the input data is tabular, meaning the features of a NoSQL architecture are not required. This is likely to result in added complexity when retrieving data from the database, and increased development time.

2.4.1 Apache Cassandra

In the end, Apache Cassandra was chosen for persistent storage. This has a number of benefits for the proposed solution [**lakshman2010cassandra**]. Firstly, the data model is tabular, and as such closely matches the format of the expected input data. Furthermore, Cassandra is a distributed database, so source data will be stored across a number of nodes, each on different computers. This will increase the effective read speed when retrieving data from the database, as the full load will be spread across all nodes.

Partitioning Cassandra's method of partitioning data is another key reason why it was selected. Each node in the database is assigned part of the full token range, which determines what records it holds. When data is inserted into the database, Cassandra hashes the primary key of each record, producing a 64-bit token that maps it to a node. Section 5.1 of the Cassandra paper walks through the partitioning process in detail [**lakshman2010cassandra**].

Figure 2.4.1 demonstrates how the full range of tokens is distributed among a number of Cassandra nodes.

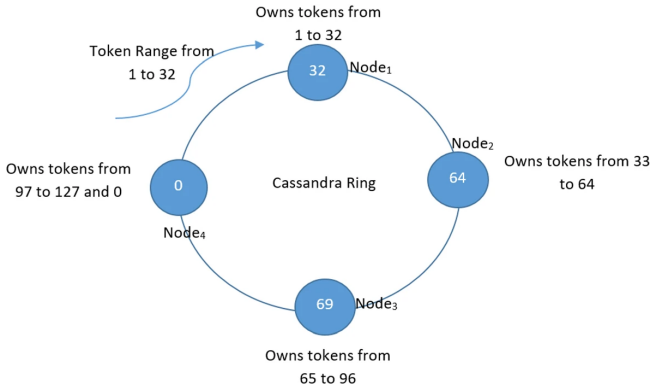


Figure 2.4.1: Cassandra Token Ring Distribution [**khatibi2019dynamic**]

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

Data Co-Location Cassandra can be run on Kubernetes using K8ssandra [**k8ssandra**], a tool which can be used to initialise, configure and manage Cassandra clusters. It is particularly useful because the Cassandra cluster can be configured to run on the same machines as the worker nodes, enabling the source data to be co-located with the workers that will actually perform the computation. This decision meets the requirement to exploit the integration between the storage mechanism and the rest of the system, as data co-location will reduce the network latency when importing the source data, and increase transfer speed as the data never leaves the same physical machine.

Language-Specific Drivers In terms of interfacing with the rest of the system, there are drivers for both Python and Java [**datastaxjavadriver**, **datastaxpythondriver**]. The Java driver has a core module which provides basic functionality for making queries and receiving results from the database.

There are optional modules for these drivers, as well as Scala-specific frameworks with more complex functionality, but these were not included in the solution as the extra functionality was not required, and the added complexity had the potential to cause problems.

2.5 Architecture

Based on the above design choices, a high level diagram was produced, detailing each of the components of the system and the interaction between them, 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 completed, the workers will return the result data to the orchestrator, which will then return the collated result to the frontend.

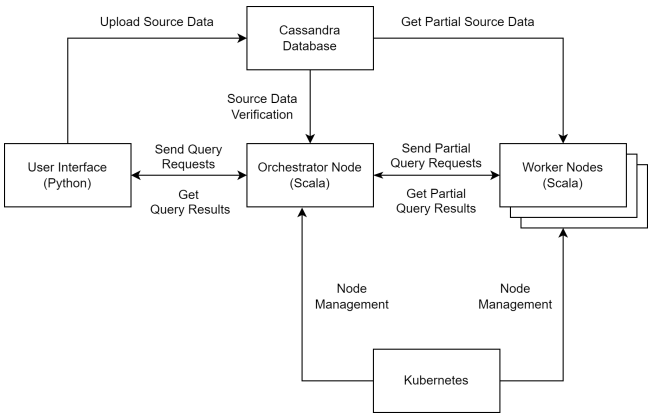


Figure 2.5.1: Proposed Solution - Overall Architecture

Chapter 3

Implementation

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

3.1 Overall Solution

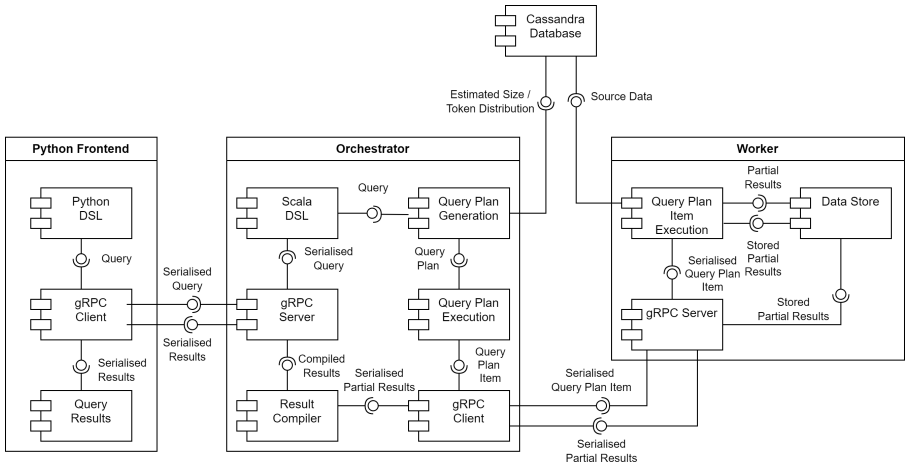


Figure 3.1.1: Solution Component Diagram

Figure 3.1.1 shows a component diagram for the core components, and their interactions. As described in Section 2.5, the frontend is the user's entrypoint to the system. It acts as 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 acts as the central state management for the system. A number of components are used to get from query to result. First, 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 a number of chunks of work (partitions) before being delegated to the workers. The workers are responsible for accepting these partitions, and performing the computation. Finally, when all query plan steps have been executed, the orchestrator fetches partial results from all workers, collates them into a final result, and returns this to the frontend. gRPC is used anywhere where network communication is required between the frontend, orchestrator and worker nodes.

The system supports three types of queries: Select, Filter and Group By. Both Select and Filter are row-level operations, meaning the workers do not have to pass data to one another during computation.

However, Group By does need the workers to cross-communicate, which means they also require a temporary store 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 the 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 types 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 is shown, using a container class. The container holds 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 class 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 entirely straightforward to implement. The container class could use a generic type parameter which stores the type information of the value in the container, but generic type information is erased at runtime [**ghosh2004generics**]. Instead, it could be defined as an interface, with each supported type providing an implementation of that interface, but this is not much better than the raw data solution, as the runtime type check simply becomes a pattern match on the class instance. A solution that exploits some kind of polymorphism is preferred.

Scala provides a feature known as *ClassTags* [**scalaclasstags**]. These save the erased type information and also permit equality checks between *ClassTag* instances, meaning the framework can use them to compare the type of a value to an expected type. The container class is therefore defined as an interface *ValueType*, which holds a *ClassTag* instance. This interface is implemented by *TableField*, which holds field name and 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.

There were two main goals when selecting these primitive types. Firstly, every type should be able to be represented without data loss in all parts of the system. Secondly, it should be possible to represent the types in protobuf format, as this would allow for easy serialisation of result data. All types except DateTime can be converted to protobuf natively, and DateTimes are supported by serialising as an ISO8601 formatted string [**iso˙8601**].

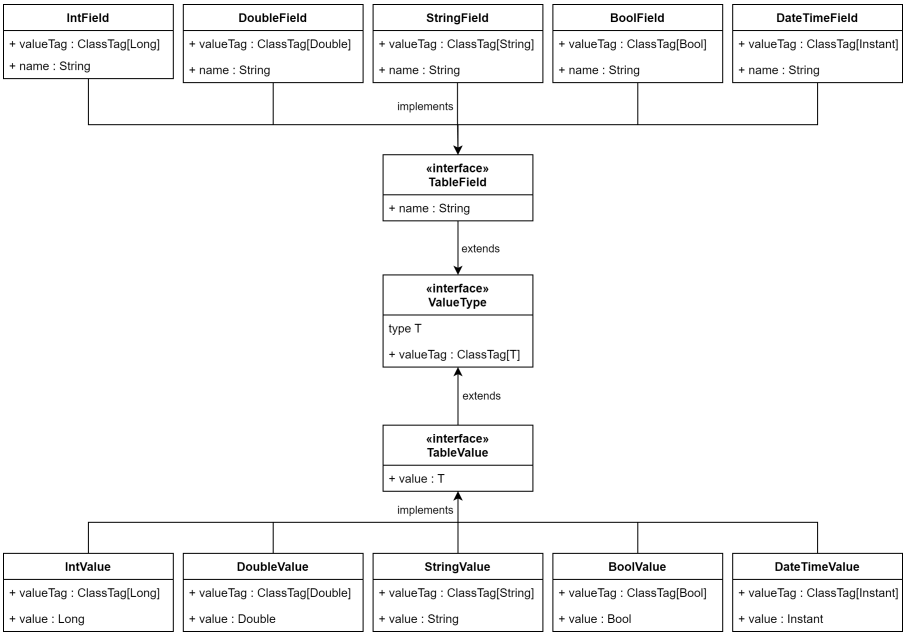


Figure 3.2.2: Custom Type System Hierarchy

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

3.2.2 Result Model

The hierarchy of classes and supported types provide everything needed to define the result of a computation, known as a *TableResult*. Headers are defined as a sequence of *TableFields* and result rows are stored as a two-dimensional array of `Option[TableValue]`. As defined in the requirements, null values must be supported, but the use of nulls in Scala is discouraged. 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 `Nothing`. Figure 3.2.4 shows what an example result looks like using this definition.

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: Example Table Result

3.3 Domain Specific Language

The user's interaction with the framework is driven entirely by the Domain Specific Language (DSL), which is modelled with SQL-like syntax. The language allows users to define expressions, then use these in computations like Select, Filter and Group By. Figure 3.3.1 provides a sample annotated DSL query, with links to the sections where each part is discussed.

3.3.1 FieldExpressions

FieldExpressions are a key part of the DSL, allowing the user to define arbitrary row-level calculations to be used as part of more complex operations. They are defined as an interface, with three subclasses:

- Values: define literal values which never change across all rows
- Fields: get the value from the given field name in the current row.
- FunctionCalls: perform arbitrary function calls using further FieldExpressions as arguments.

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

Many basic functions have been implemented, including arithmetic, string and cast operations. ver, the function system is designed to be extensible. A number of helper classes are defined to allow the creation of basic unary, binary and ternary functions, but *FunctionCall* is itself an interface which can be given completely custom implementations if required. The main constraint on the functions that can be defined are that only the 5 primitive input types are supported.

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

There are two main situations where the resolution step catches errors. 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.

Initialise cluster connection and select a Cassandra source table:

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

Select query, 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, 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, uses *AggregateExpressions* (Section 3.3.3):

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

Figure 3.3.1: Example DSL Query

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

```
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

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

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, resolving to the correct version for evaluation. For example, the add 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 are all expected to be named. This allows the user to chain operations by referencing fields from the previous input. Figure 3.3.6 shows the two ways of naming a field: *FieldExpressions* can be assigned a user-defined name, or *F()* expressions will also keep their name automatically.

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 are another key building block of the DSL, allowing the user to define arbitrary row-level comparisons. *FieldComparisons* use a two-step resolution-evaluation process to accommodate the same process as *FieldExpressions*. They are defined as an interface, with a number of comparison types implemented already, including null, equality, numerical and string comparisons. Examples of each are shown in Figure 3.3.7.

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

Combined Comparisons The user is able to combine multiple FieldComparisons using AND/OR operators, shown in Figure 3.3.8. This is a lightweight wrapper around Scala's own AND (`&&`) and OR (`||`) boolean operators, meaning optimisations like short circuiting operate as normal.

```
(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

3.3.3 Aggregate Expressions

AggregateExpressions are the final part of the DSL. These are used only as part of Group Bys, and allow the user to define methods of aggregating all rows of a result. They take a *NamedFieldExpression* as an argument, and compute a single row output from any number of input result rows.

The supported operations are Minimum, Maximum, Sum, Average, Count, and String Concatenation, and they are polymorphic where possible. For example, minimum and maximum handle numeric types by ordering numerically and string types lexicographically. Figure 3.3.9 shows examples of *AggregateExpressions*.

```
Max(Function.ToString(F("date")).as_name("date_string"))
Max(F("duration"))
Sum(F("amount"))
Count(F("id"))
```

Figure 3.3.9: *AggregateExpression* examples

3.3.4 Protocol Buffer Serialisation

All components of the DSL have been designed to be serialised to protobuf format. This allows any queries written by the user to be passed around the system using gRPC, and if required the query can also be serialised to a file. The results of a query 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 results are split up by row and streamed individually.

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, including arithmetic and comparison, 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 data analysis in Python [**reback2020pandas**]. The frontend is able to convert query results from their protobuf definition to a pandas DataFrame to allow further analysis to be performed immediately.

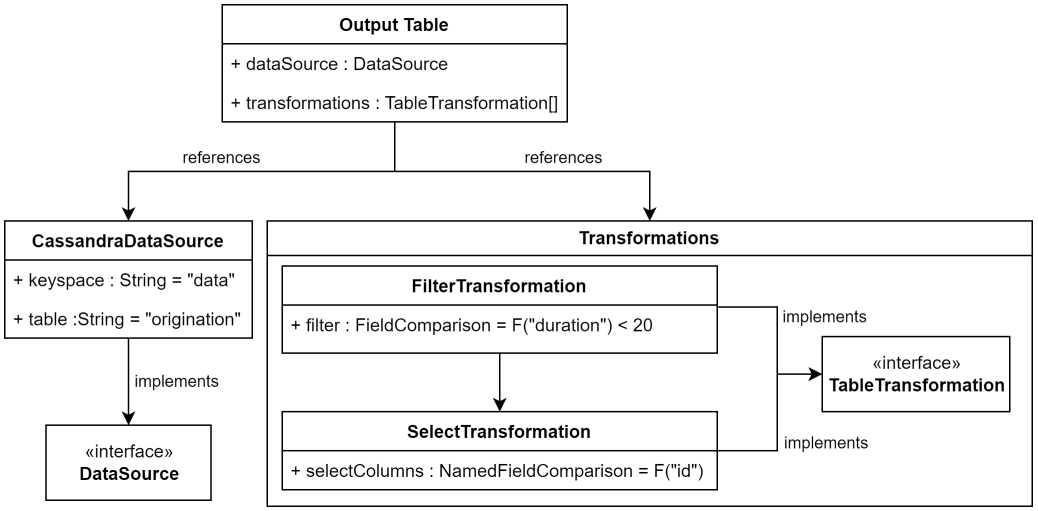
3.4 Data Model

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 any part of the query containing purely row-level computations, including Select and Filter operations. It is composed of a *DataSource* and a list of row-level computations known as *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 fully computed before it can be generated. A Cassandra *DataSource* will always act as the terminal component for a query, as it has no dependencies.

For demonstration purposes, 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 placed in the Table component, as no new partitions are required to produce the result.

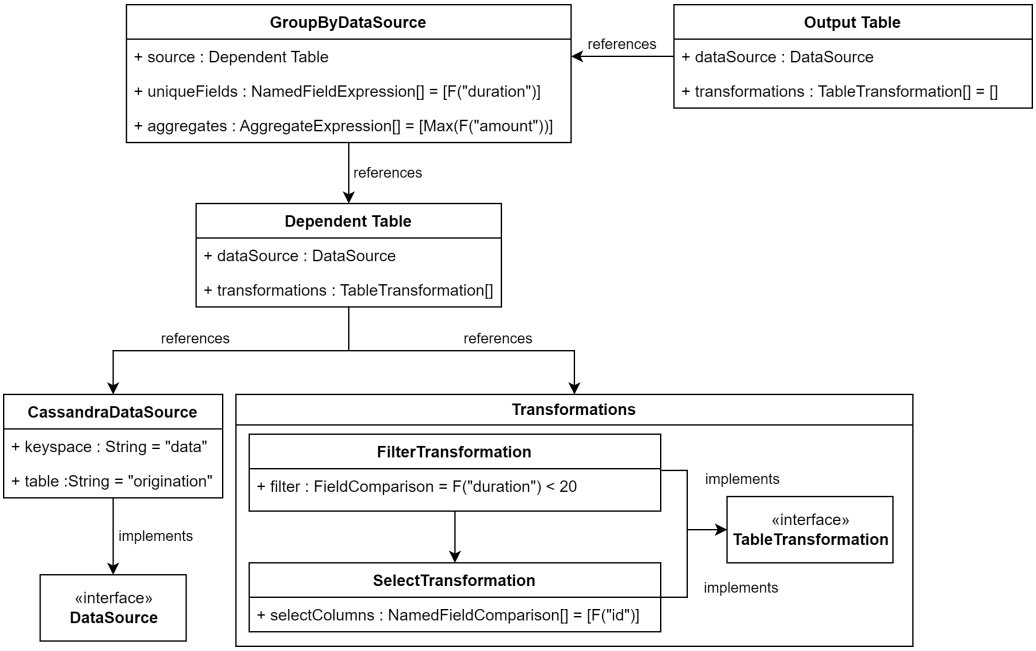



```
ClusterManager("orchestrator-service")
.cassandra_table("data", "origination")
.filter(F("duration") < 20)
.select(F("id"))
.evaluate()
```

Figure 3.4.1: Example Filter and Select Query

Figure 3.4.2 shows a more complex 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 *GroupByDataSource*. The final output is then generated in 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. Both *PartialDataSource* and *PartialTable* always hold a reference to the complete version of themselves. To demonstrate how to produce a final result from a set of partial results, Figure 3.4.3 shows a high level example of one possible way of partitioning and computing the previous Filter and Select query.




```
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

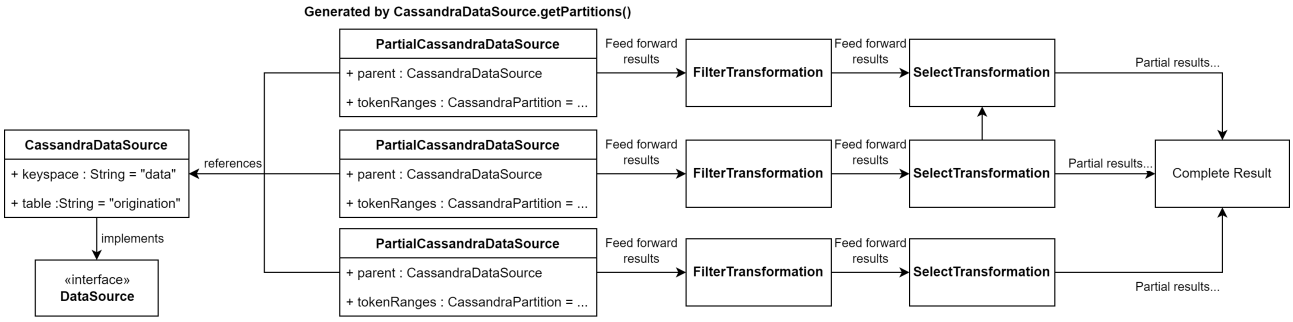


Figure 3.4.3: Example Filter and Select Query

3.5 Data Store

The data store is the most important component for the worker implementation. It 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 worker will be processing more than one request at the same time, which presents issues with handling concurrency and synchronisation.

The approach taken to solve this uses the actor model, first introduced in 1973 by Carl Hewitt [**hewitt1973session**]. Specifically, the Akka Actors framework was chosen as an implementation of the actor model in Scala [**akkaactors**]. The actor model abstracts away the complexity of synchronisation and thread management. Instead, components of the system become actors. Each actor defines a set of messages that it accepts, and the response to each message, and the framework provides a guarantee that an actor will only ever process one message at a time.

The data store is modelled as 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 in the process of computing a Group By partition.

For each supported type of data, the data store 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 of *Tables* and *DataSources* always contain a reference to the full version, but not the other way around. Therefore, this decision does not increase the time taken to insert new data significantly, but it is particularly useful when fetching the results for a *Table*, or removing a *Table* or *DataSource*. Without this approach, completing these operations would require searching the entire Map to find any matches, turning the O(1) lookup time into O(n).

3.5.1 Spill to Memory

In situations with very large datasets, the amount of memory available to the workers will be less than the amount of data the system attempts to load. In this case, it is likely that the JVM will run out of heap space, causing a crash when it attempts to allocate more memory to store data. Therefore, the system features a module which allows it to move cached data onto disk to free up heap space. This module is part of the data store, and functions transparently to the rest of the system - data store queries are the same whether the result is held in-memory or on-disk.

Storage Interface To implement the spill process, an interface *StoredTableResult* is defined. This interface holds a key which corresponds to that result, and a *get* operation to retrieve the result

data. There are two main subclasses that implement this interface: *InMemoryTableResult* and *ProtobufTableResult*.

InMemoryTableResult is a simple wrapper for the interface, which simply contains the result and holds it in memory. It also features a *spillToDisk* method which moves the data onto disk by creating a file under a randomised folder name for that execution, with the name set to the hashcode of the key.

ProtobufTableResult only holds a pointer to the data on disk, and reads the data from there when the *get* operation is called. It also features a cleanup method which removes the stored data from disk.

Spill Process The data store is responsible for managing in-memory and on-disk data. Before almost every operation on the data store, it makes a check for the current memory utilisation, which is calculated using a set of methods on the *Runtime* class [**javaruntimeclass**]. If the memory utilisation is over a given threshold, the data store attempts to spill at least the amount of bytes over the utilisation threshold.

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

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

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 After the process is completed, the data store forces the JVM to perform Garbage Collection to immediately free the relevant amount of memory.

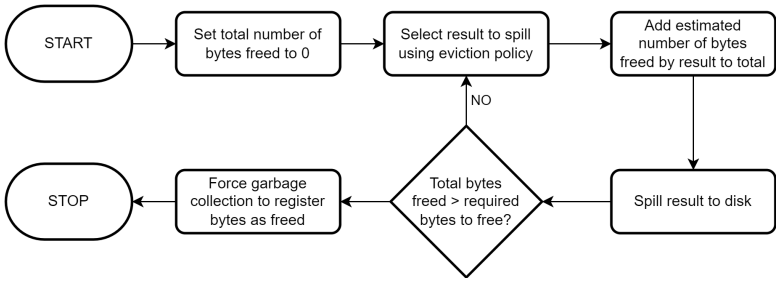


Figure 3.5.2: Spill to Disk Decision Tree

This process is not without flaws. It relies on no other class in the current JVM instance holding references to any of the results being spilled. In the controlled worker environment, it is possible to ensure this is the case, meaning the spill works reliably, but this approach would not work more generally. Also, this approach is reliant on size estimates, meaning the actual amount of memory freed will not be the same as the estimated memory freed. With a suitably low threshold for spilling (60-70% of maximum memory) and regular checks of memory utilisation, this risk does not become a real problem.

Eviction Policy Finally, the policy for determining results to spill is important. A policy that does not fit the way the results are being used could result in large performance impacts, as it could cause

antipatterns like the data store spilling a result, then immediately reading it back to memory.

The data store makes use of a least-recently-used (LRU) policy to determine the next result to spill to disk, implemented using an ordered list containing all in-memory results. When results are inserted into the data store, they are added to the end of the list. When results are read from the data store, they are moved to the end of the list. Then, when a result must be selected for spill, the item at the head of the list is chosen and removed.

3.6 Partitioning

One of the most important jobs of the orchestrator during a computation is to calculate partitions. There are two situations where partitions need to be calculated: when pulling source data from Cassandra, and when computing a Group By. The overall goal is to split the dataset into roughly equal chunks of a manageable size.

3.6.1 Cassandra

As described in the Design Chapter, Cassandra was selected as the persistent storage module because its storage model closely matches the type of partitioning the system requires. The Cassandra *DataSource* getPartitions method investigates the source data in the database, and generates an appropriate number of partitions. This process is described below.

When data is stored in Cassandra, the primary key of each row already has a 64-bit token assigned to it. Cassandra allows queries to filter on ranges of these tokens, meaning the data can be split up and read from the database in chunks. To be able to ensure the partitions are a manageable, defined size, an estimate of the size of the full source data is required. Cassandra provides this information in the `system.size_estimates` table. This table provides an estimate of the size of each table in the database, and these are generated automatically every 5 minutes.

From a size estimate of the full table, estimates can be calculated for any given token range using the equation in Figure 3.6.1. The fraction calculates the percentage of all tokens 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, it first gathers the token ranges which each node is responsible for storing. Then, it performs a joining and splitting process over each node, depending on the size of the token ranges. The set of token ranges produced after this process are the partitions used during the computation. The flowchart in Figure 3.6.2 shows the full process for generating the output partitions,

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

Figure 3.6.4 provides an example how the joining process works. Given a sorted list of token ranges, the system repeatedly adds sequential elements to a partition until it is larger than the goal partition size. Then, the partition is marked as completed. The list is sorted by size ascending to ensure the

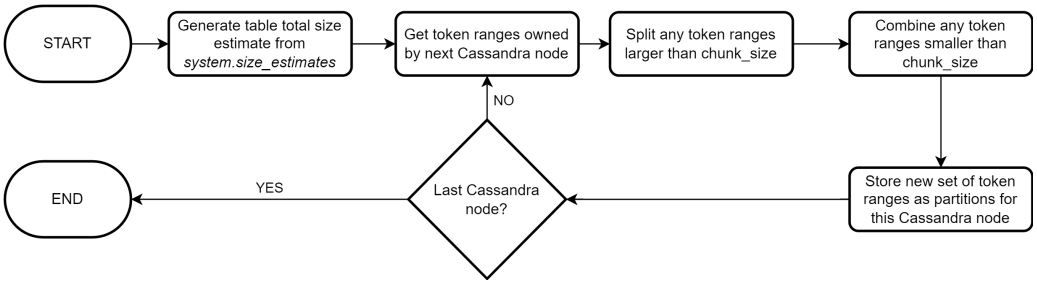
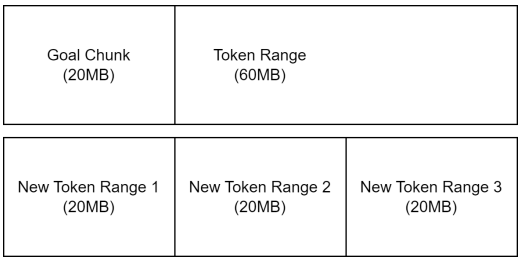


Figure 3.6.2: Cassandra Partitioning Process

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

(a) Equation



(b) Example

Figure 3.6.3: Token Range Splitting

smallest number of partitions are created - if there are a large number of very small token ranges, these will be combined into a single large partition together.

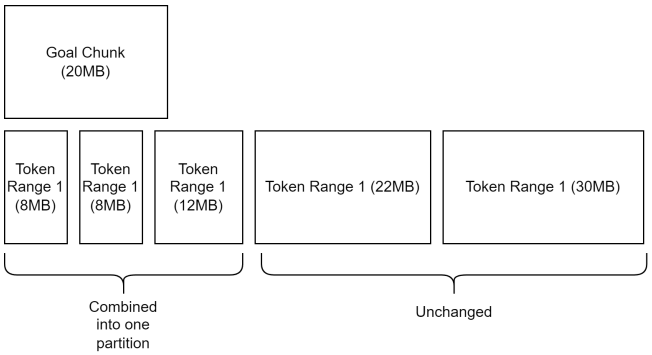


Figure 3.6.4: Token Range Joining Example

The Cassandra Java Driver provides a wide range of helper functions for performing the joining and splitting of token ranges accurately to produce new token ranges. The system simply calculates how much joining or splitting is required based on the size of the token ranges and the table size estimate. It then uses the driver to perform the calculations.

3.6.2 Cassandra Data Co-Location

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

To do this, each worker first calculates its closest Cassandra node. This is done by opening a TCP

connection multiple times with each Cassandra node, and averaging the time taken over multiple attempts. The node with the lowest average latency is selected as the closest node. The orchestrator uses this information to match each worker to a Cassandra node and its corresponding list of partitions. The output is an optimal assignment between workers and partitions. It is possible for more than one worker to be matched to the same set of partitions, and it is also possible for a set of partitions to have no co-located worker nodes. In this case, they are kept as unassigned, and the work assignment algorithm handles their allocation. Figure 3.6.5 shows an example cluster setup with three physical nodes, and the optimal assignment after this process is complete.

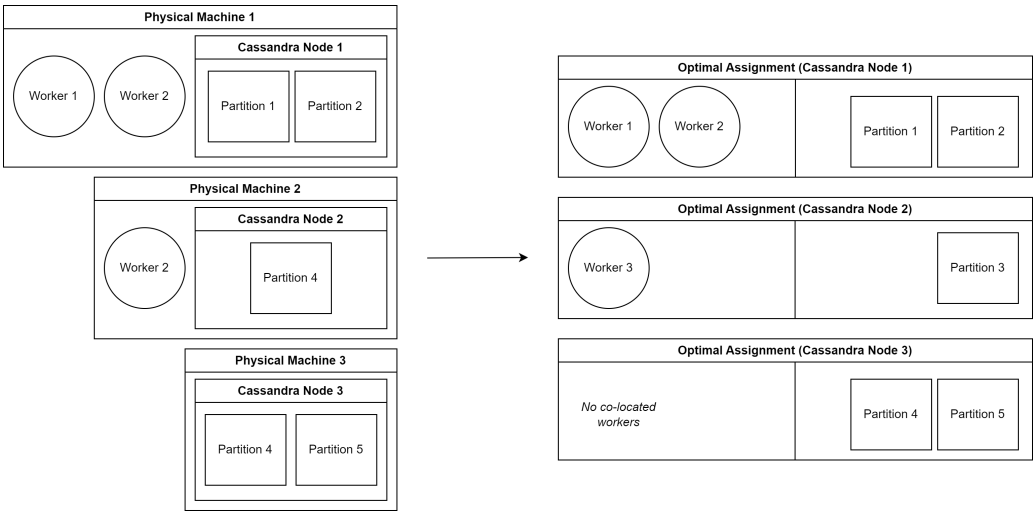


Figure 3.6.5: Optimal Assignment Example

3.6.3 Group By

The Group By operation takes any number of *NamedFieldExpressions* to act as unique keys, as well as any number of *AggregateExpressions* which will be computed for each combination of unique keys. As with pulling source data from Cassandra, the goal when computing a Group By is for the partitions generated to be roughly equal chunks of a manageable size. To do this, two things are needed: an estimate of the full size of the dataset, and a way of splitting the dataset up to keep unique keys together.

The process of calculating partitions and hashes, shuffling data, computing the result and deleting the original hashes is all controlled by the orchestrator during Query Plan Execution.

Partitions The first step is to compute the number of partitions that the Group By will be made up of, which is based on the size of the full dataset. To estimate this, an existing class, *SizeEstimator*, was reused from Apache Spark, with some changes for compatibility with Scala 3 [**zaharia2016spark**]. This class provides a static method *estimate* which accepts any Scala object and produces an estimate, in bytes, of the size of the object. This can be called on all partial results across all workers, and the results 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.

Hashing A unique partition is then defined as the tuple of the total number of partitions, and the partition number of this partition. Hashing, combined with the modulo operation, is used to determine which partition each row should be assigned to. In particular, Murmur3Hash is used as the hashing algorithm, which is used in Cassandra and is provided natively by Scala [**murmur3hash**]. Figure

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

Figure 3.6.6: Group By - Total Partitions

3.6.7 shows the high level equation for assigning rows to partitions. This process ensures that the new partitions will be roughly equal in size, and all rows with the same values for the unique keys will be mapped to the same partition.

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

Figure 3.6.7: Group By - Row Partition Assignment

Computation After the hashes are computed and a worker is assigned a particular partition, it must communicate with all other workers in the cluster to get any data that relates to that partition.This ensures that the partition data is complete, and it is much the same as when the orchestrator requests query result data from the workers. The worker makes a request to all other workers, and they stream the header and rows of their partial data back to the worker that made the request. When a Group By is being computed, it's likely that a worker will 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 data relating to a particular partition, it must compute the Group By result. Scala features a built-in Group By function, which this operation uses. The values for the unique keys of the Group By are calculated for each row, and the rows are placed into groups based on those values. Then, the aggregate functions are computed for each of the groups, resulting in one output row for each combination of unique keys.

Deletion Finally, the last step of computing a Group By is to remove the hashed partition data stored on each worker, as it is no longer needed. This is handled by the orchestrator automatically during the process of computing the Group By *DataSource*.

3.7 Row-Level Computations

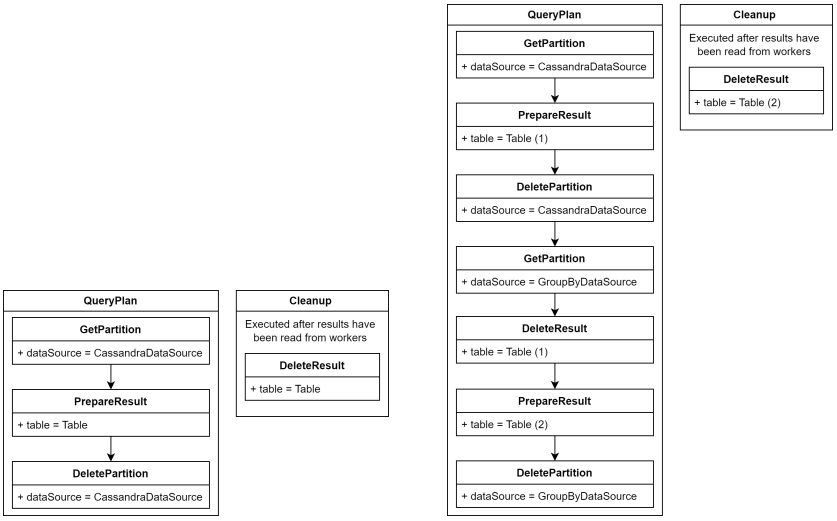
Once partitions have been delegated to the workers, performing the computation is relatively straightforward. Included below are descriptions of how the Select and Filter operations are performed.

3.7.1 Select

This operation takes any number of *NamedFieldExpressions*. To compute a result, each *NamedFieldExpression* is applied to each row of the input result. The output result has the same number of rows as the input, and fields equal to the number of *NamedFieldExpressions*.

3.7.2 Filter

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



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

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

any rows where the comparison returns `false`.

3.8 Query Plan

Query plans are the process through which the orchestrator can calculate how to get from a query written by the user, to a result which it can return. A Query Plan is made up of a sequence of *QueryPlanItems*, which is an interface defining one part of a query plan. Each *QueryPlanItem* has an *execute* method, which will make some change to the state of all workers in the cluster when called. There are four main QueryPlanItem implementations, which allow *DataSources* and *Tables* to be calculated and deleted.

Both *DataSource* and *Table* have a function that generates the full Query Plan to compute their output from scratch, as well as a second Query Plan to clean any result in the data store.

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

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.

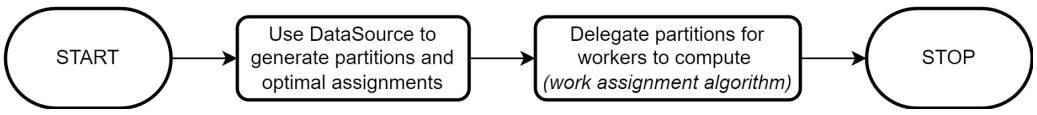


Figure 3.8.2: Get Partition Execution - Without Dependencies

If the *DataSource* has dependencies, then Figure 3.8.3 shows the process for this item. 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*. These steps are the same as when calculating a Group By (Section 3.6.3), which is an implementation of *DataSource*.

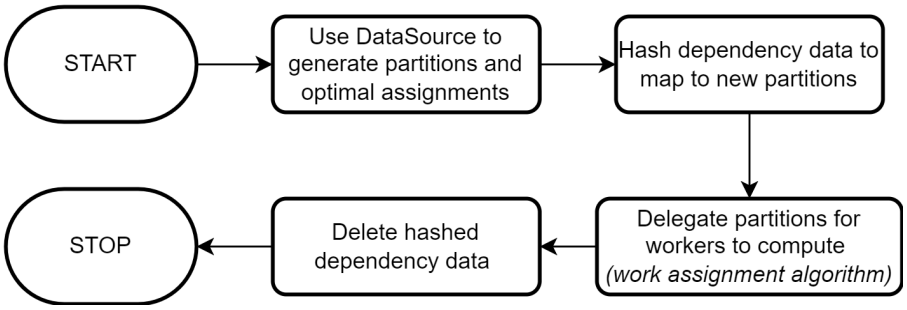


Figure 3.8.3: Get Partition Execution - With Dependencies

Work Assignment Algorithm The details of how the partitions are actually computed are abstracted behind the *DataSource* interface as they are implementation specific, but GetPartition always needs to manage the process of sending requests to the workers to compute the partitions. This is done using the Work Assignment Algorithm.

A simple solution would be to use a round-robin process to match optimal assignments to their workers, delegating work when a request finishes and stopping when the assignment list is 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 takes unexpectedly long to complete their work.

The ideal solution would ensure a worker computes all of its own optimal partitions first and, once these are complete, computes the partitions that were originally assigned to other workers. This implements a form of dynamic load balancing, because a faster-running worker is able to take on more requests than a slower worker, and no worker will be idle unless there are no partitions left to compute. However, this presents concurrency challenges, including preventing race conditions like the same partition being delegated twice to different workers.

The actor model, used previously in the data store, is ideal for this situation. 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 provided with an ordered list of producers, then repeatedly request and compute work from each in order. Each consumer is given a differently ordered list, with the producer of that consumer's optimal partitions placed first in the list. When all assigned producers for a consumer are empty, the consumer will shut down. The final part of the system is a counter, which 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 the case where no worker is co-located with a set of partitions; this producer will simply be placed last in the list of producers for each consumer, resulting in these partitions eventually being processed.

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

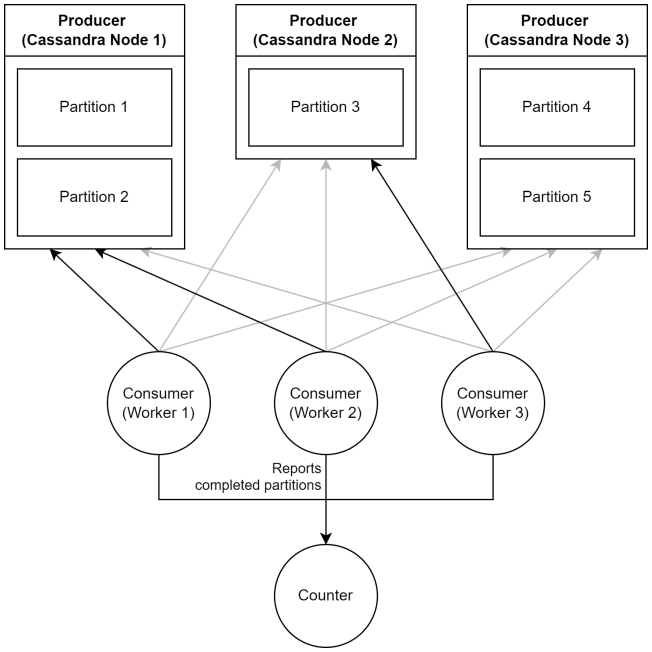


Figure 3.8.4: Producer Consumer Model Example

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 partitions generated by *GetPartition* as an argument. The implementation uses a modified version of *GetPartition*'s actor system, which iterates through all partitions on each worker, sending a request for each partition to perform the *Table* computation.

3.8.3 DeleteResult and DeletePartition

DeletePartition and *DeleteResult* are *QueryPlanItems* for removing the results of a *GetPartition* and *PrepareResult* operation, respectively. These will send a single request to each worker, and they 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 over all partial results in the data store, streaming the data back to the orchestrator. An actor system is used to pipe the concurrent responses from all workers into a single thread, which combines the results and streams them to the frontend.

The request is initiated by the orchestrator, rather than the workers pushing data to 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 much also be the case for result collation.

3.9 Deployment

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

As previously described in 3.6.2, worker nodes will 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, and the following two scheduling rules will provide this:

1. Workers should be placed on the same Kubernetes node as a Cassandra node if possible.
2. Workers should not be placed on the same node as other worker nodes.

The scheduling rules are expressed as preferences rather than requirements, 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 of the components of the system, specifically using GitHub Actions [**githubactions**]. 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 high level aim of this system is to improve the speed of data processing for large datasets. Therefore, it is important 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.

Firstly, raw performance testing is conducted against a typical data processing solution, Microsoft SQL Server. Then, a number of alternative approaches are assessed for determining how well the cluster leverages the parallelisation of running the computation over multiple nodes.

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 [**scalatestuserguide**]. 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 [**scalatestplusmockito**, **datastaxjavadriver**]. Using these tools and frameworks, unit tests have been written for the majority of core code; this includes the DSL, query model and partitioning code among other classes. 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 the chosen test data is loan origination (creation) data. 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.

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

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 in this testing.

Test Plan The 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 of the query will be written and tested. For the cluster processor, the tests are run tables containing loan origination data with the following number of rows: 1000, 10000, 100,000, 1 million and 10 million rows. For SQL, tables with the same number of rows are generated, as well as two further tables: 50 and 100 million rows. These are included to provide further context of how SQL scales at larger data volumes; the cluster processor was unable to test these tables due to time and cost constraints.

To reduce the effect of random error, each test is run 5 times, and the results averaged across all tests. This is particularly important when running on a cloud environment, as there is less control over the hardware running the tests. In particular, there is no control over what other work that hardware is doing alongside the testing, so averaging a number of results should reduce any impact this has.

In all of these tests, Microsoft SQL Server was running on an instance of Azure SQL Database [**azuresqldatabase**]. The Cluster Processor was running on Azure Kubernetes Service, with a pool of three nodes available, each node having 4 vCores, and 16GB memory [**azurekubernetesservice**]. The actual CPU and memory available to each pod in the cluster is controlled by Kubernetes.

4.3.1 Controls

A number of variables must be considered which could have an impact on the results of this test. The testing attempts to mitigate the effects of these variables in order to make the results as comparable as possible.

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 [**k8sapi**]. 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.

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 [**azurecloudshell**]. 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.

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.

4.3.2 Select Query

The first query is a pure select, which essentially tests how fast both solutions can send results over the network. Figure A.1 shows the SQL query, and Figure A.2 shows the Cluster Processor query. The second select query is a more complex select, with some conversion operations to test if this has an impact on the computation time. Figures A.3 and A.4 show the queries.

Results The results for this query are shown in Figure 4.3.1. 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, which is analysed further in Chapter 5. As a result, the graph is filtered to exclude the SQL results for 50 and 100 million rows.

However, the data that is available shows that the Cluster Processor is more than 10x slower at performing Select queries than SQL. This is most likely related to the format used to send the result data, which requires serialising both the type and value of each cell in the data for transmission. For both SQL and the Cluster Processor, there is no significant difference between the raw Select, and the Select with operations. The largest difference is at 100k rows for Cluster Processor, where there is a 0.3s increase when computing the operations compared to a raw Select.

4.3.3 Filter Query

The first query is a simple filter, with no AND/OR combinations. Figures A.5 and A.6 show the queries. The second filter is more complex, testing both boolean operators. Figures A.7 and A.8 show the queries. The results for this query are shown in Figure 4.3.2.

Results The results for this query are shown in Figure 4.3.2, where testing was performed for the full range of datasets in both SQL and the Cluster Processor. The results show that SQL is again significantly faster in this test case, producing results around 15-20x faster. This is likely to be partially caused by the transmission format, as with the Select queries. However, it is also likely that SQL is

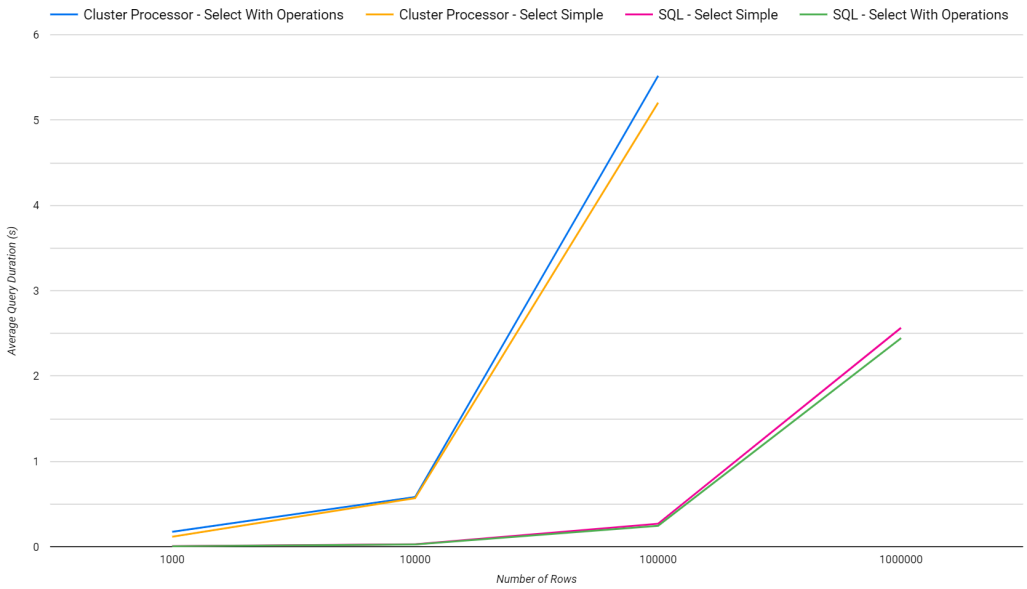


Figure 4.3.1: SQL vs Cluster Processor - Select Query Results

able to exploit caching more extensively over repeated tests when compared to the Cluster Processor, particularly with the smaller tables which can be held in memory permanently. This is because the Cluster Processor fetches fresh data from Cassandra every time the query is called.

Another relevant insight from this data is that the complex filter reliably executes faster than the simple filter across all results. For the Cluster Processor, it is nearly 2 seconds faster on average at 10 million rows, and for SQL it is almost 6 seconds faster at 100 million rows. This is to be expected, since the complex filter is more restrictive in the results that it returns, so there is less data to transfer over the network.

4.3.4 Group By Query

The first query is a simple group by, essentially acting as a `DISTINCT` check. Figures A.9 and A.10 show the queries. The second group by is more complex, featuring a number of aggregations. Figures A.11 and A.12 show the queries.

Results The results for this query are shown in Figure 4.3.3, where again testing was performed for the full range of datasets in both solutions. 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 so much 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 which resulted in slower computation. Another round of testing would need to be performed to determine if this was the case.

Despite this unexpected difference in computation times between the queries at 10 million rows, there is still a significant performance drop-off when compared to the results at 1 million records. Analysing the outputs from the workers at this data volume, 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

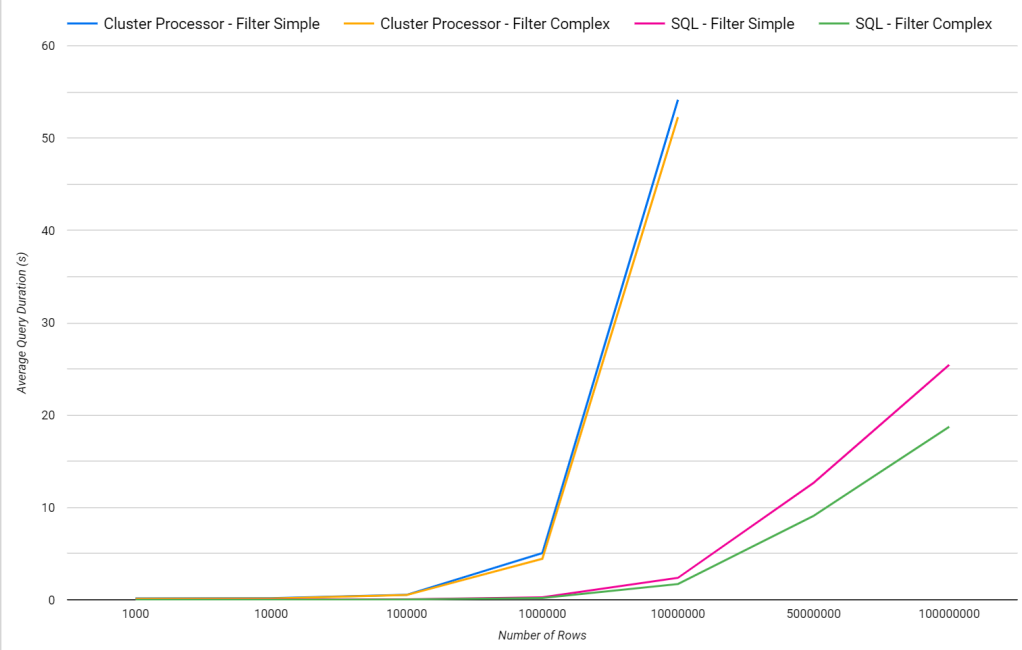


Figure 4.3.2: SQL vs Cluster Processor - Filter Query Results

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.

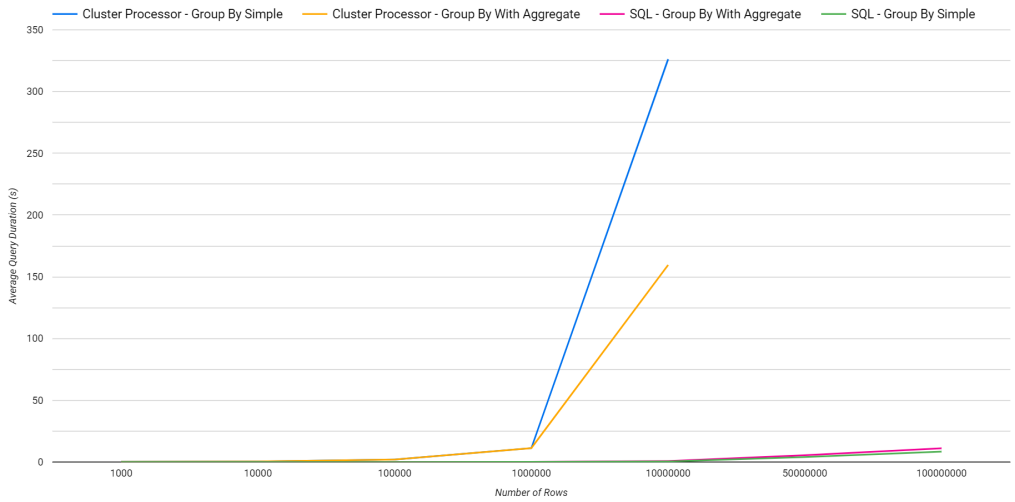


Figure 4.3.3: SQL vs Cluster Processor - Group By Query Results

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. In particular, the system appears to be weakest at transmitting raw data quickly across the network, as well as computing Group Bys quickly.

One way the transmission format for results could be optimised is to reduce the size of the serialised

row data. Currently, every value within a row also stores type information, but this information has already been sent as part of the header. Therefore, the system could stop sending type information with the value data, then use the header to perform a conversion when the row is received.

The main inefficiency that causes the poor Group By performance is the workers holding onto more data than necessary. Currently for simplicity, the workers hash a copy of all rows in the data before cross-communication occurs. It should be possible to partially perform the Group By operation for each partition on each worker, then finalise those partial results on the worker that actually holds the partition, significantly reducing the amount of stored and transferred data. As testing has already established that the raw data transmission is suboptimal, by sending more data than is required, the network inefficiencies have a larger impact on performance. Furthermore, when computing Group Bys at 1 million rows, the workers ran low on free memory, and began spilling data to disk. This is significantly slower than only in-memory storage, so by reducing the amount of stored data, spilling would occur at larger data volumes, and performance would be improved.

4.4 Level of Parallelisation

This section will compare the performance of the Cluster Processor when the number of workers is varied, but the overall performance in terms of 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 the Select, Filter and Group By queries were executed, with a different range of rows depending on the test. See Appendix A for details of the queries that were executed. 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.

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

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. Ultimately, this test is checking how fast each layout can pull data from Cassandra, and send it to the Orchestrator. It is likely that the extra overhead introduced by the higher levels of parallelisation slowed 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 slow 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 also performed slower than the 3 node cluster. This is likely because this layout has one less worker node than Cassandra node, meaning latency is added to retrieve data from the Cassandra node without a co-located worker.

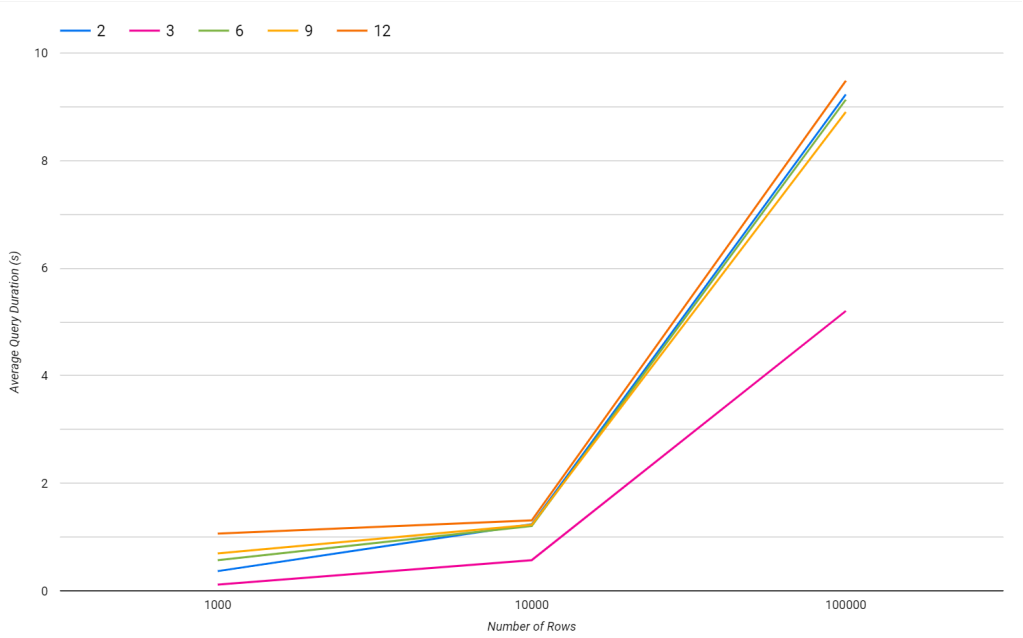


Figure 4.4.2: Parallelisation - Select Query Results

4.4.2 Filter Query

The results of this test are shown in Figure 4.4.3. At small numbers of rows, there is very little difference between each of the cluster layouts. However, an interesting trend appears as the number of rows increases above 100,000. The 3 node cluster is fastest until this point, but is then overtaken by the 9 node cluster at 1 million.

To investigate this trend further, the test was also run for all cluster layouts at 10 million rows, shown in Figure 4.4.4. At this number of rows, the layouts with 6, 9 and 12 nodes are all faster than the 3 node cluster, with 9 nodes appearing to be the optimal level of parallelisation. This shows that as the amount of work increases, the increased level of parallelisation becomes a benefit, resulting in faster computation times. The fact that 12 nodes is slower than 9 nodes at 10 million rows suggests that there is an optimal point in terms of maximising parallelisation, without introducing too much overhead from the number of nodes.

4.4.3 Group By Query

The results of this test are shown in Figure 4.4.5. These results again suggest that there is a balancing point for the level of parallelisation. The 3 node cluster is consistently faster than all other cluster layouts. The 6, 9 and 12 cluster layouts are likely to be slower because increasing the level of parallelisation will result in higher amounts of cross-communication, and network transfer is one of the slowest areas, as identified in the SQL test.

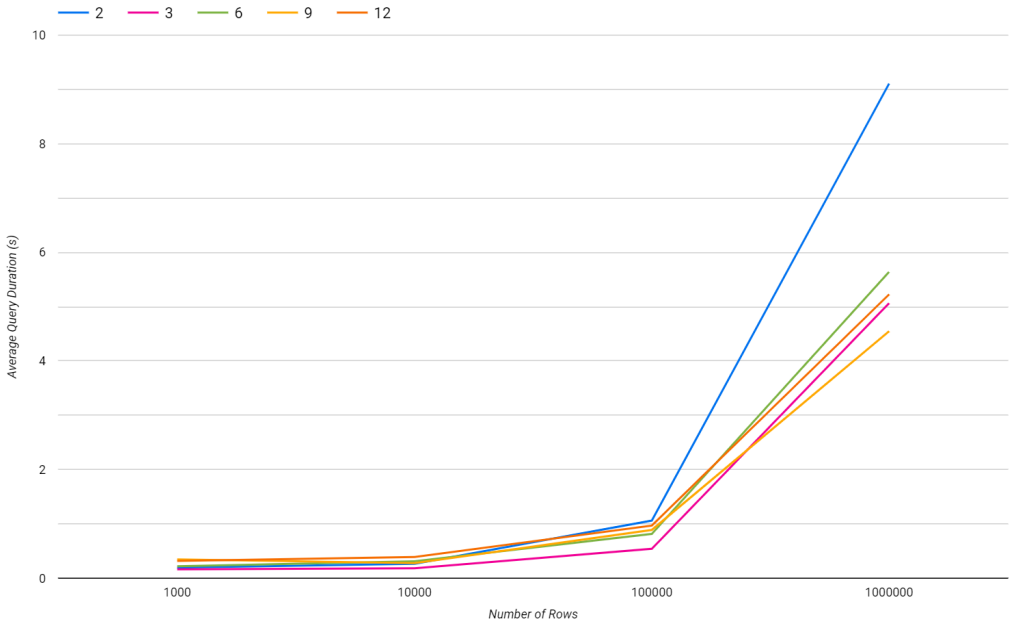


Figure 4.4.3: Parallelisation - Filter Query Results

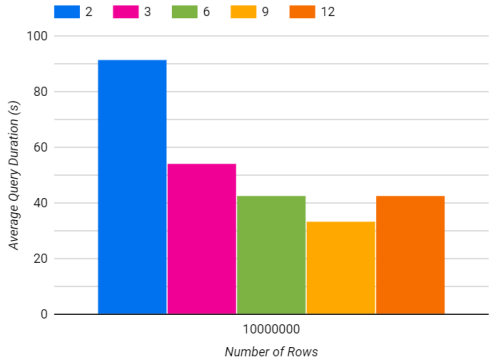


Figure 4.4.4: Parallelisation - Filter Query Results, 10 Million Rows

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 node caused by having fewer worker nodes resulted in some data being spilled to disk, which results in reduced query performance.

4.4.4 Analysis

Overall, the outcomes of this test show that there is no clear solution to the number of nodes in the cluster. As a general rule, matching the number of workers to the number of Cassandra nodes in the cluster will result in good performance, but the Filter test also shows that increasing the level of parallelisation can result in better performance for the same resources at larger query sizes. Solutions to this problem depend entirely on the queries being run, and the data volumes the queries are applied to. There is an opportunity for further research designing a system that can analyse the queries being executed, and adjust the cluster layout to match the requirements of those queries.

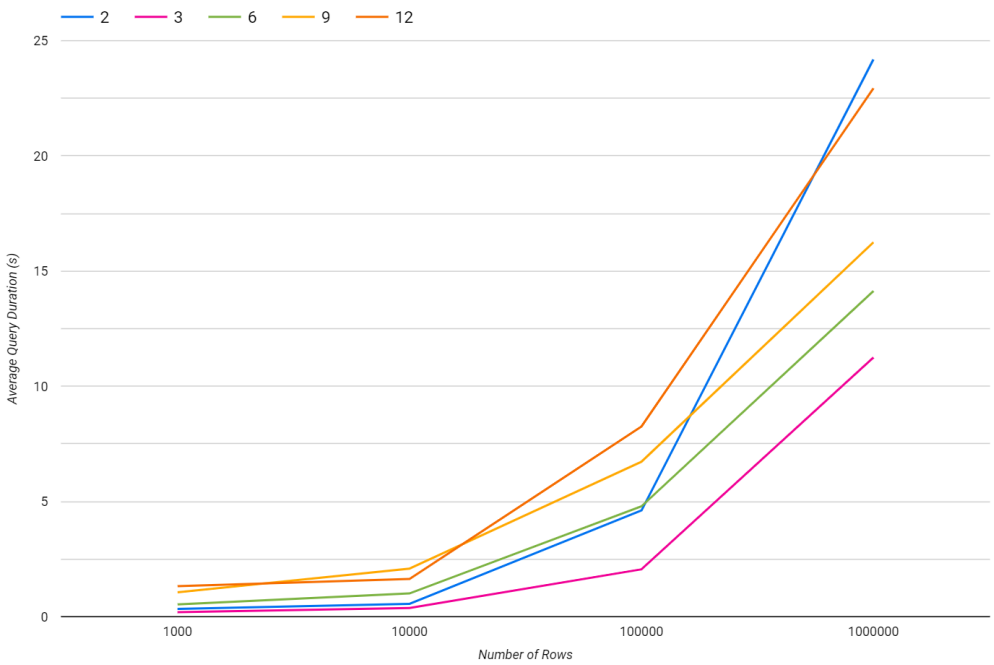


Figure 4.4.5: Parallelisation - Group By Query Results

4.5 Autoscaling

This section will compare computation speed of the Cluster Processor when the overall performance is reduced. The motivation for the test is the autoscaling feature present in many managed Kubernetes services, including Azure Kubernetes Service, which continually analyses the load of the Kubernetes cluster, and increases or decreases the number of physical nodes based on current demand [**aksautoscaling**]. By doing this, applications with fluctating levels of demand are able to save costs by reducing the number of machines they pay for in periods of low demand.

While the Cluster Processor is not currently implemented in a way that supports autoscaling, this test is performed to identify how effective autoscaling would be on this solution. In these tests the simple versions of the Select, Filter and Group By queries were executed, with a different range of rows depending on the test. Figure 4.5.1 shows the cluster layouts for each test case; the number of workers, and the resources available to each worker. Each cluster layout has one less worker, and 33% less resources available.

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

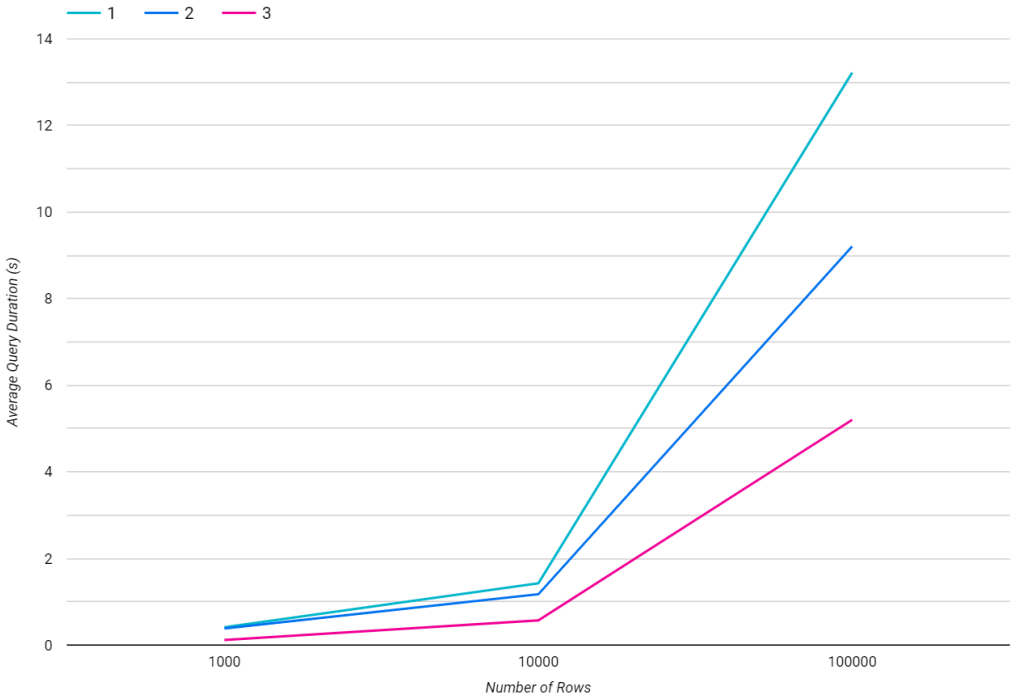


Figure 4.5.2: Autoscaling - Select Query Results

4.5.2 Filter Query

The results of this test are shown in Figure 4.5.3. As before, the query performance decreases as the number of workers decreases. However, at small data volumes the difference between 1 and 3 workers is even smaller than in the Select test. The difference between 1 and 3 workers is 0.08s at 1000 rows, 0.2s at 10000 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 1000 and 10000 rows, there is effectively no difference between the three cluster layouts, and at 100000 rows the 1 node cluster is the fastest by 0.5s on average. Once the data volume increases to 1 million rows, the 3 node cluster is significantly faster than the other two layouts. However, these results suggest that, at very small data volumes, it is faster to perform all of the computation on a single node. This is because it prevents the need for the workers to cross-communicate, and if all data can be kept in memory, the computation will be performed significantly faster.

4.5.4 Analysis

The outcomes of this test show that it is effective to reduce the cluster size if the data volumes are small. The time difference between the smallest and largest cluster is typically less than a second when operating on less than 1 million rows. This is insignificant for most use cases, and if a time-critical

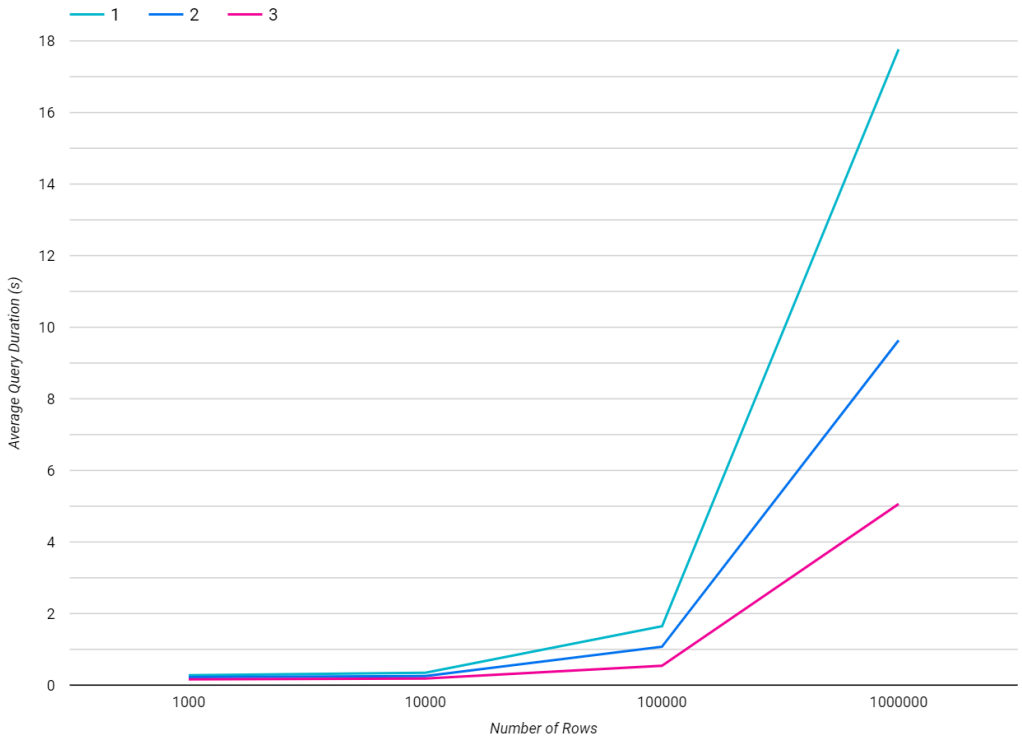


Figure 4.5.3: Autoscaling - Filter Query Results

system was reliant on querying this small amount of data, a more typical solution like SQL would be better suited to solve the problem regardless.

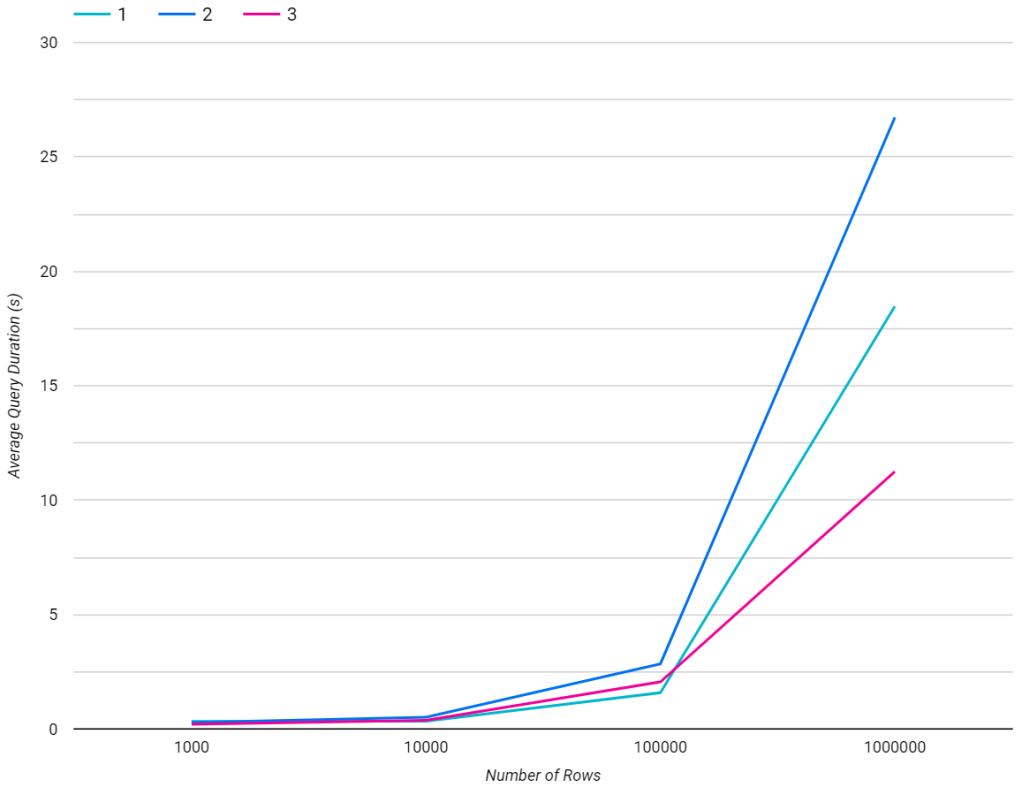


Figure 4.5.4: Autoscaling - Group By Query Results

Chapter 5

Evaluation

Following the performance testing of the completed solution, this section will discuss a high level evaluation of the solution, and the project as a whole.

5.1 Limitations

The features of the solution have a number of limitations which could be improved upon in future. As discussed in Chapter 4, the implementation of 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. This is because each cell in a result dataset is wrapped in a class containing its type, as well as the value. This decision results in even small datasets taking up large amounts of memory. For example, a 100MB source data file can use up to 800MB of memory once loaded into the storage format. As discussed in the Implementation chapter, this approach is essential to the functionality of the DSL, as the type information is not accessible at runtime otherwise. Further investigation would be required to determine an improvement to this limitation.

Due to time constraints, the system's security is limited. There is no authentication to access the orchestrator, and Cassandra only has basic username and password requirements for data inserts. This is something that would need to be developed for the system to be integrated in a production environment.

Finally, as discovered in Chapter 4, the result collation algorithm cannot cope with returning extremely large amounts of results, typically more than 1 million rows. All workers send results to the orchestrator, which sends them to the frontend as fast as it can. However, as there are more workers than the single orchestrator, data arrives into the orchestrator faster than it leaves. This means that with a large enough dataset, the orchestrator will run out of memory and crash. Alternative solutions need to be considered to fix this, but due to time constraints this limitation was not able to be fixed.

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. Queries with less computation, or that run on smaller amounts of data are better applied to smaller clusters, while increasing the level of parallelisation is better when the query is larger or more complex. Therefore, 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 in the computation of queries by temporarily storing partial result data. However, its design means that it could also be used to store results between queries. This would improve the computation time of repeated queries to the same dataset, as the steps to reach the stored result would not have to be repeated each time. Join operations would also be able to use this improvement to the data store. They are currently not implemented in the solution, but are essential for many types of queries, particularly when relating two different datasets by a common key.

The error handling in the system is designed to send an error message to the Python frontend if anything goes wrong during computation. For some errors, like if the Cassandra database is unexpectedly not responsive, this is acceptable. For other errors, like if a worker node suddenly goes down, it should be possible to handle this error by delegating the failed worker's partition to others, without alerting the user of any failure in the first place.

Finally, the Cassandra database is currently only used as a permanent data store, and for splitting up the source data into partitions. However, it is a database in its own right, and some computations could be performed on Cassandra before sending any data at all, improving query times by reducing the amount of network transfer. In particular, any filter operation on the source dataset, and group by operations on the primary key are perfect candidates for this optimisation, as it could be implemented with minimal work.

5.3 Conclusion

The objective as stated in Chapter 1 was to design a query processing engine for a distributed cluster of nodes, to increase the speed of data processing for large datasets. The types of queries possible in the system are numerous, and *DataSource* and *Table* 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 should these optimisations be made. 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 the *FieldExpression* and *FieldComparison* system allowing complex data manipulations to be defined with relative ease. The implementation of the Function system permits new functions to be added easily within the bounds of the type system. The operation of the frontend is seamless to the user, as the background operation of the framework is entirely hidden on the other nodes. Furthermore, the pandas integration means that users can immediately start manipulating result data using tools already familiar to them.

Todo list

Appendix A

Testing Figures

Included in this appendix 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 Complex

```
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 Complex

```
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 Complex

```
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 Complex