UNIVERSITY OF BIRMINGHAM

Parallelised Data Processing

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

Oliver Little

2011802

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

Supervisor: Vincent Rahli

School of Computer Science
College of Engineering and Physical Sciences
April 2023

Contents

List of Figures

Abstract

Chapter 1

Introduction

1.1 Background

1.2 Prior Work

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

Research into solutions for distributed data processing has generally resulted in two kinds of solutions [22]:

- **Batch processing:** where data is gathered, processed and output all at the same time. This includes solutions like MapReduce [6] and Spark [23]. Batch processing works best for data that can be considered 'complete' at some stage.
- **Stream processing:** where data is processed and output as it arrives. This includes solutions like Apache Flink [5], Storm [20], and Spark Streaming [3]. Stream processing works best for data that is being constantly generated, and needs to be analysed as it arrives.

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

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

MapReduce was not without flaws, and many papers were published in the years following its initial release which performed performance benchmarks, and analysed its strengths and weaknesses [14]. Crucially, MapReduce appears to particularly struggle with iterative algorithms, like the PageRank algorithm used by Google's own search engine. A number of popular extensions to MapReduce were introduced to improve the performance on iterative algorithms, like Twister [7] and HaLoop [4] both in 2010.

MapReduce's popularity also resulted in a number of tools being created to improve its usability and accessibility. Hive [19] is one such tool, which features a SQL-like language called HiveQL to allow users to write declarative programs that compiled into MapReduce jobs. Pig Latin [16] is similar, and features a mixed declarative and imperative language style that again compiles down into MapReduce jobs.

Further tools in the wider areas of the field were introduced around 2010, including another project by Google named Pregel [15], specialised for performing distributed data processing on large-scale graphs.

In 2010, the first paper on Spark [25] was released. Spark aims to improve upon MapReduce's weaknesses, by storing data in memory, and providing fault tolerance by tracking the 'lineage' of data. This means for any set of data, Spark knows how the data was constructed from another persistent, fault tolerant data source, and can use that to reconstruct any lost data in the event of failure. This in-memory storage, known as a resilient distributed dataset (RDD) [24] allows Spark to improve on MapReduce's performance for iterative jobs, whilst also allowing it to quickly perform ad-hoc queries. Effectively, Spark is strong at performing long batch jobs, as well as short interactive queries. This is something that I would like my solution to feature, as users of the framework will need to design long-running scripts to run on large amounts of data, as well as run ad-hoc queries to perform investigation.

Spark quickly grew in popularity, with a number of extensions being added to improve its usability, including a SQL-style engine with a query optimiser [2], as well as an engine to modify Spark to support stream processing [3]. A second paper released in 2016 [23] stated that Spark was in use in thousands of organisations, with the largest deployment running an 8,000 node cluster holding 100PB of data. One area where Spark struggles is with grouped data, as performing grouped operations requires shuffling the data between all nodes. I aim to improve upon this in my solution through the design of the system as a whole.

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

1.3 Project Aims

Chapter 2

Design

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

- Required Features
- Technologies and Frameworks
- Architecture

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

2.1 MoSCoW Requirements

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

F / NF	Priority	Requirement Description
		Domain Specific Language - Expressions
F	M	The language must support 5 data types: integers, floats, booleans, strings and date-time objects.
F	M	The language must 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.

2.2 Language

2.2.1 Frontend

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

Reference

Expand upon this choice

2.2.2 Orchestrator and Worker Nodes

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

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

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

Reference

Reference

Reference

Reference

Reference

Reference

2.3 Runtime

2.3.1 Containerisation

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

Reference

Reference

Reference

Reference

Reference

2.3.2 Network Communication

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

Reference

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

reference

Reference

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

Reference

reference

Reference

reference

2.4 Persistent Storage

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

- Single System SQL Databases *(Microsoft SQL Server, PostgreSQL, MySQL)*: while this option 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.

  Reference 3

- 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.

  Reference

- 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.

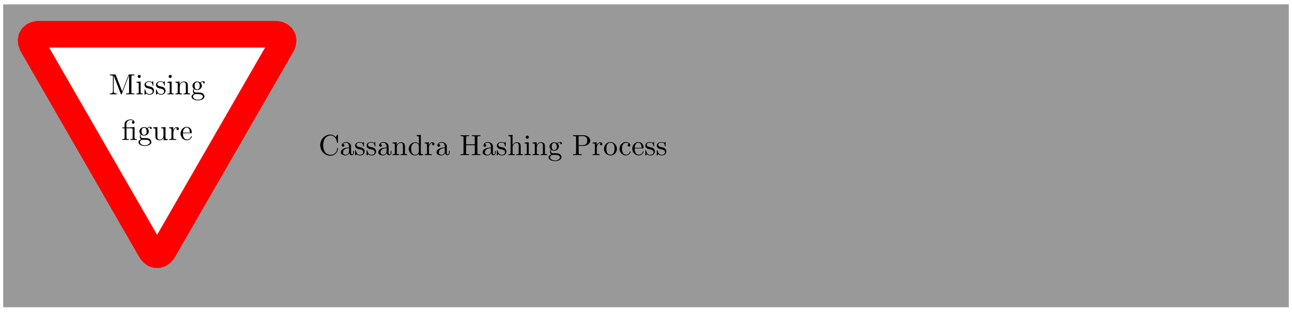
  Reference 2

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. 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.

Reference

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



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

Kubernetes Support Cassandra can be run on Kubernetes using K8ssandra [13]. This is a tool which can be used to initialise, configure and manage Cassandra clusters. This is particularly useful because the Cassandra cluster can be configured to run on the same machines as the worker nodes, enabling the source data to be co-located with the workers that will actually perform the computation, reducing network latency when transferring the source data.

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

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

2.5 Architecture

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

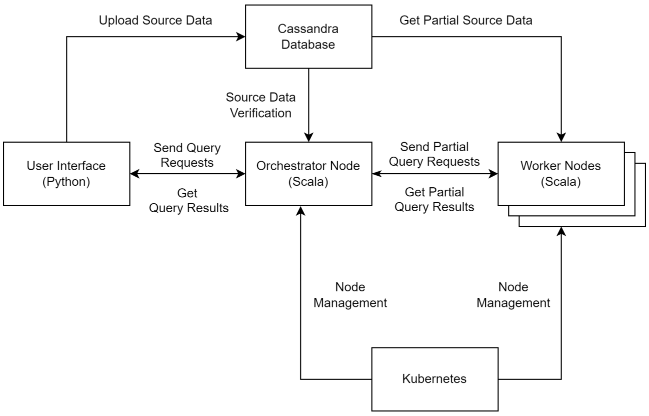


Figure 2.1: Overall Architecture Diagram for the Proposed Solution

Chapter 3

Implementation

3.1 Type System

The type system for the DSL is built using a subset of Scala, Python, and Cassandra's type systems. These types are shown in Figure 3.1.

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 3.1: Primitive Types

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 are can be converted to protobuf natively, and DateTimes are supported by serialising as an ISO8601 formatted string [12].

Designing the actual interfaces to represent these values presents a challenge. To be able to read and manipulate these types in Scala at runtime, the raw primitive types cannot be used. This is because the only common supertype of all primitive types is *Any*, as shown in Figure 3.2. There is effectively no information shared between all supported types in the system. To discover which type a given value is (or if the type is even supported), the system would have to perform runtime type checks against all possible supported types. These checks would add a significant amount of overhead, and as a result this approach was not selected.

Another possible approach to solving this problem is to create a lightweight container class, which holds the value, and the type information about the value at runtime. This means that the system only needs to perform the runtime type check once in order to create the correct class instance. Due to limitations of Java, this conceptual solution is not entirely straightforward to implement. The container class would use a generic type parameter which stores the type information of the value inside the container. A given row of data would need to support multiple types of data stored together, and this is not possible using generics as the type information of the generic is erased at runtime .

reference

The container class could instead be defined as an interface, and each supported type provides an

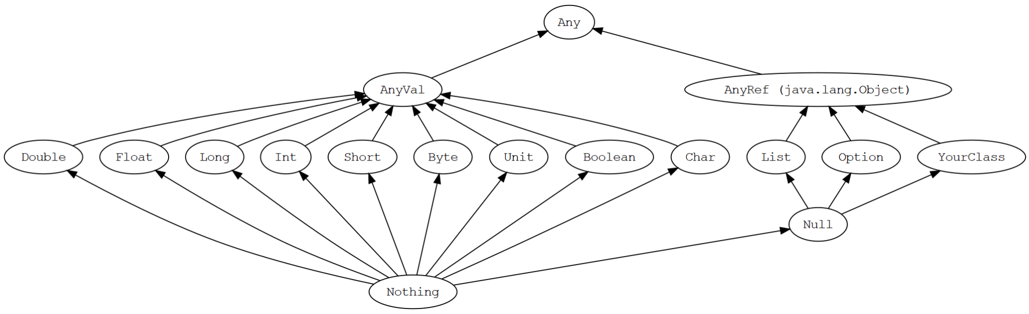


Figure 3.2: Scala Unified Types [21]

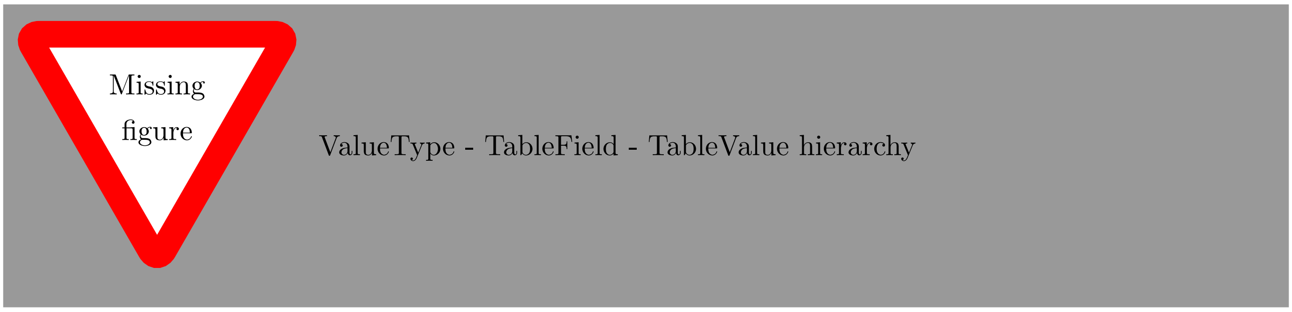
implementation of that interface, but this is not much better than the primitive type solution, as the runtime type check simply becomes a runtime pattern match on the class instance. A solution that exploits some kind of polymorphism is preferred.

Scala provides an feature known as ClassTags . These allow the erased type information to be recorded, and also allow equality checks to be performed between ClassTags. By storing a ClassTag instance, we can implement type equality checks between values in the system, to determine if they are of the same type. Furthermore, we can use this type information later to determine what types are accepted and returned by FieldExpressions.

reference

This is defined in the system using a base interface *ValueType*, which captures the ClassTag requirement. This interface is used as part of both *TableField*, which captures field information (name and type), and *TableValue*, which captures value information (value and type). Implementations for all the supported types are then provided for both *TableField* and *TableValue*. shows the hierarchy of these types.

insert figure ref



3.1.1 Result Model

This hierarchy of classes provides everything needed to define computation results in the system. Headers are defined as a sequence of TableFields, and results are defined 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`. shows what an example result looks like using this definition.

insert figure ref

3.2 Domain Specific Language

The user's interaction with the framework is driven entirely by the Domain Specific Language (DSL). The language allows the user to define expressions, comparisons, aggregates, and then use these to define computations like Select, Filter and Group By. As per the requirements, the DSL has been modelled with SQL-like syntax.

3.2.1 FieldExpressions

FieldExpressions are the key building block of the DSL. They allow the user to define arbitrary row-level calculations to be used as part of more complex operations. FieldExpressions are defined as an interface, which provides a standard set of methods, and there are three subclasses that provide implementations:

- Values: define literal values which never change across all rows
- Fields: when iterating over the rows of a result, gets the value from the named field in the current row.
- FunctionCalls: perform arbitrary function calls using further FieldExpressions as arguments.

Examples of FieldExpressions are shown in Figure 3.3.

```
Fields: F("field_name")
Values: V("a"), V(1), V(1.5), V(True), V(datetime.date(2021, 12, 31))
Functions: Function.Left(Function.ToString(F("string_field")), 10)
           F("int_field") * V(2)
```

Figure 3.3: Examples of FieldExpressions

Many basic functions have been implemented, including arithmetic, string and cast operations. However, the function system is designed to be extensible. A number of helper classes 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 constraints 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 the header of the input result, and verifies that the FieldExpression is well typed with regards to that result. This is necessary for Fields, which may be valid for one result but invalid for another, for example if the field name is missing from the result. The evaluation step performs the computation on a row from that result without any type checking.

This two-step process has a number of benefits. The resolution step enables a form of polymorphism on some functions like arithmetic operations. At the resolution step, these functions determine what types are returned by their sub-expressions, and resolve to the correct version for evaluation. For example, the add function can resolve to AddInt, AddDouble, or Concat for strings. Also, there is reduced overhead at runtime as type checking does not need to be performed for each row - unchecked casts are used here instead.

Named Expressions When performing a Select operation, the output fields are all expected to be named. This allows the user to repeatedly chain operations by referencing fields from the input. Therefore, FieldExpressions have a method to allow them to be named. When this method is called, the FieldExpression is wrapped as a tuple with the name into a NamedFieldExpression. Field references are able to reuse their previous name automatically to reduce the need for repeated naming calls.

3.2.2 FieldComparisons

FieldComparisons are another key building block of the DSL. They allow the user to define arbitrary row-level comparisons. FieldComparisons use a two-step resolution-evaluation process. This is in place to accommodate the two-step process that already exists for FieldExpressions. They are defined as an interface, and there are four subclasses that provide implementations:

- Null Checks: takes a single FieldExpression as input, and filters out rows where it is null/not null.
- Equality Checks: performs an equal/not equal check on two FieldExpressions.
- Ordering Checks: applies an ordering comparator to two FieldExpressions.
  - Supports <, <=, >, >=.
- String Checks: applies a string comparator to two FieldExpressions
  - Supports contains, starts with and ends with (both case sensitive and insensitive versions).

Examples of FieldComparisons are shown in Figure 3.4.

```
F("int_field") > V(2)
F("string_field").contains("hello") | (F("double_field") <= 1.5))
F("incomplete_field").is_not_null() & (F("string_field2") == "goodbye")
```

Figure 3.4: Examples of FieldComparisons

Combined Comparisons The user is able to combine multiple FieldComparisons using AND/OR operators. This is a lightweight wrapper around Scala's own AND (&&) and OR (||) boolean operators, meaning optimisations like short circuiting operate as normal with no extra work.

3.2.3 Aggregate Expressions

Aggregate Expressions 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. Aggregate Expressions take a NamedFieldExpression as an argument, and compute a single row output from any number of input result rows.

The system supports Min, Max, Sum, Average, Count, and String Concatenation operations. These aggregates are polymorphic where possible. For example, minimum and maximum handle numeric types numerically and strings lexicographically.

expand + examples

3.2.4 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.

3.2.5 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. These are known as double underscore *(dunder)* methods. The Python implementation of FieldExpression overrides the arithmetic operators, as well as comparison operators to allow the user to automatically generate function calls and FieldComparisons, without having to write the full, verbose definition.

Furthermore, query results can be converted from their protobuf definition as received from the server to a pandas DataFrame [18]. This decision was made because pandas is one of the most commonly used frameworks for data analysis in Python. In the 2022 Stack Overflow Developer Survey, it was the third most popular non-web framework [17]. Therefore, it is likely that the intended users of the system will have prior experience performing data processing using DataFrames.

3.3 Minimum Viable Product

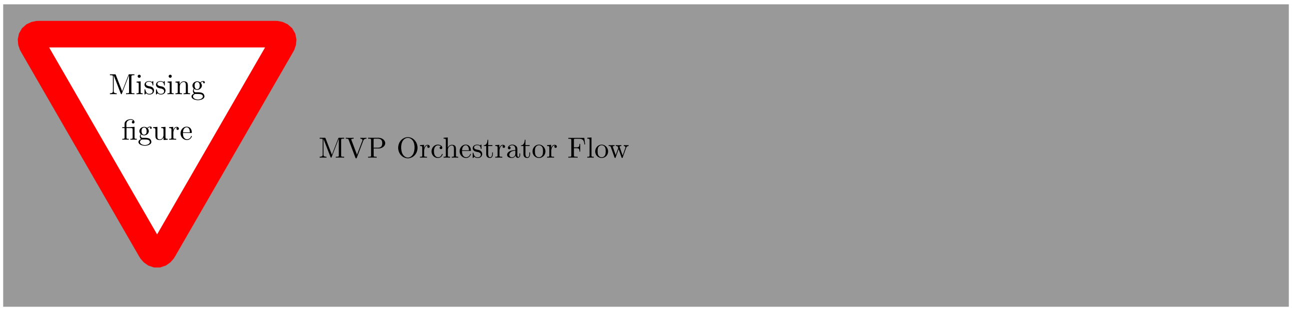
To break development into more manageable chunks, a minimum viable product (MVP) was implemented first. This required implementing partitioning of Cassandra Source Data, and the Select and Filter operations. By taking this approach, any issues in the design of the system could be discovered and rectified earlier. Also, this would ensure that modules built for the MVP were reusable, as they would be designed with the purpose of integrating into the full solution.

Group By partitioning would not be implemented for the MVP, meaning all operations available to the user are row-level, and only one set of partitions need to be generated for a query. This removes the need for state in the workers, greatly simplifying their design at this stage.

The below diagrams are provided to give a high-level picture of how the MVP system operates. The Orchestrator manages calculating partitions, delegating them to workers, and collating final results.

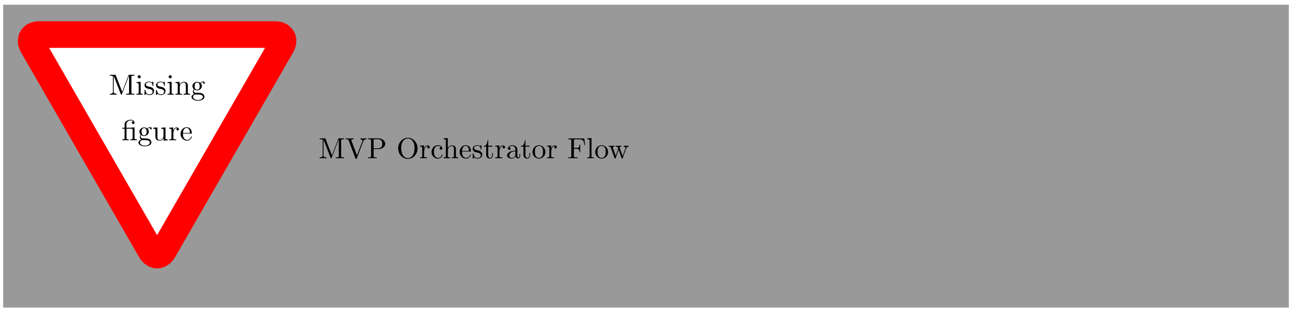
insert figure here

shows the data flow for the Orchestrator.



insert figure here

shows the data flow for all Workers.



3.3.1 Cassandra Partitioning

The goal of the partitioning module is to split up the source data into roughly equal chunks of a manageable size. 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.

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 .

Reference

From a size estimate of the full table, estimates can be calculated for any given token range using the equation in Figure 3.5. The fraction calculates the percentage of all tokens that the given token range represents.

$$\frac{\text{Number of Tokens in Token Range}}{\text{Total Number of Tokens: } ((2^{64}-1)-(-2^{64}))} \times \text{Estimated Table Size}$$

Figure 3.5: Token Range Size Estimation Equation

It is the orchestrator's responsibility to generate an allocate the partitions to each worker. 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 flowchart in Figure shows the full process for generating the output partitions.

insert missing figure

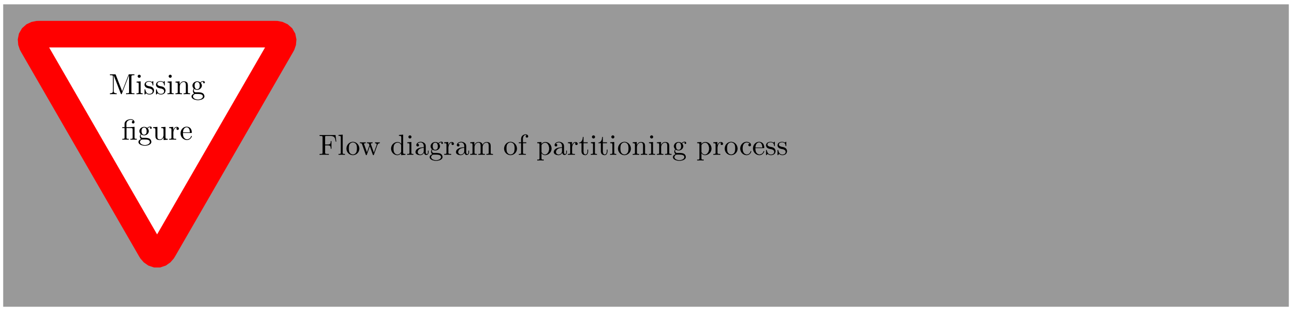


Figure demonstrates how the splitting process works. The system calculates how much times larger the token range currently is than the goal partition size, then splits the

insert missing figure

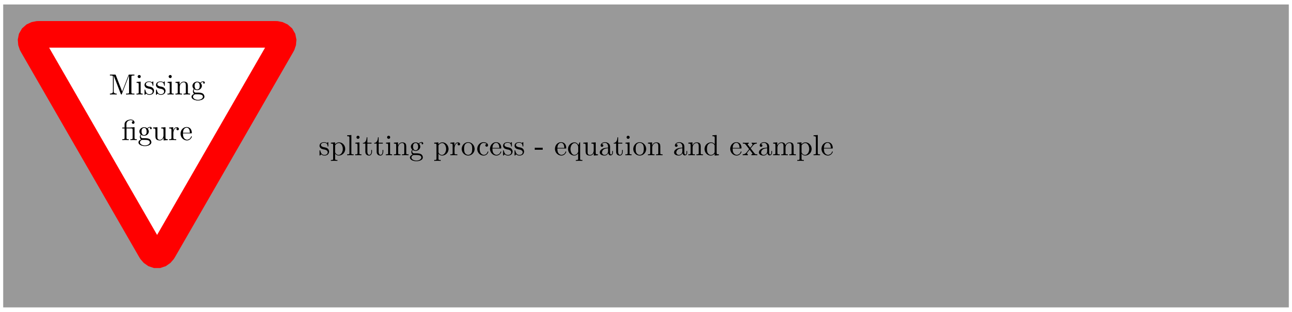
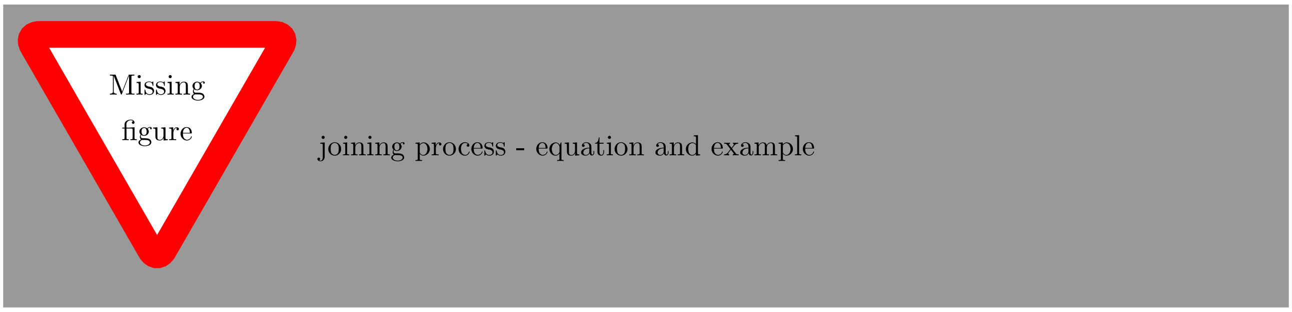


Figure demonstrates how the joining process works. Given a sorted list of token ranges, the system repeatedly adds sequential elements to a partition until it is larger than the goal partition size. Then, the partition is marked as completed. The list is sorted by size ascending to ensure the smallest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

insert missing figure



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.

3.3.2 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. Figure shows this

insert figure here

process.

Missing figure

worker latency calculation diagram

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's possible for more than one worker to be matched to the same set of partition, and it's 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 shows an example optimal assignment after this process is complete.

insert figure here

Missing figure

worker optimal assignment to partitions diagram

3.3.3 Work Assignment

Given an optimal assignment between workers and partitions, the orchestrator manages the process of sending requests to all workers to compute the partitions. A simple solution to this problem would be to iterate through the optimal assignments in order for each worker, delegating work when a request finishes and stopping when the 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 first computes all of its own optimal partitions 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 be computed.

However, this presents concurrency challenges, including preventing race conditions like the same partition being delegated twice to different workers. The final approach taken to solve this uses the actor model, first introduced in 1973 by Carl Hewitt [11]. Specifically, the Akka Actors framework was chosen, as it is one of the most widely supported implementations of this model for Scala . 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. The framework provides a guarantee that an actor will only ever process one message at a time.

reference docs

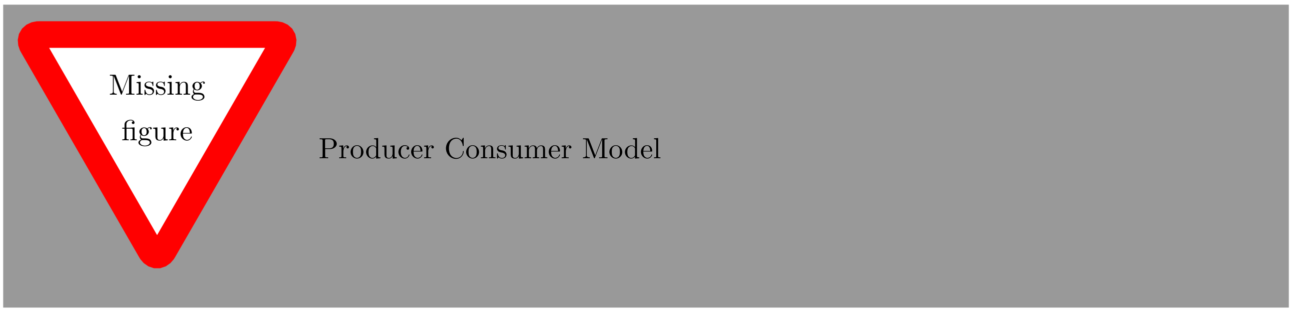
The solution models the optimal partitions as *producer* actors, and the workers as *consumers*. Producers will respond to requests for work with partitions, and will shut down when all partitions have been given out. Consumers are provided with an ordered list of producers, then repeatedly request and compute work from each in order. When all assigned producers for a consumer are empty, the consumer will shut down. Each consumer is given a list with a different order, with the producer of that consumer's optimal partitions placed first in the list.

This solution also handles the case where no worker is co-located with a set of partitions, as this producer will simply be placed at the end of the list of producers for each consumer. As a result, these partitions will eventually be processed.

The final part of this solution is an *assembler*, which compiles the results from each consumer into a complete result. When all results are received, the assembler sends the result data back to the user.

Figure provides a visual representation of the model.

insert figure here



Producer Consumer Model

3.3.4 Result Computation

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.

Select This operation takes any number of NamedFieldExpressions. To compute a result, apply each NamedFieldExpression 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.

Figure shows how a Select is computed.

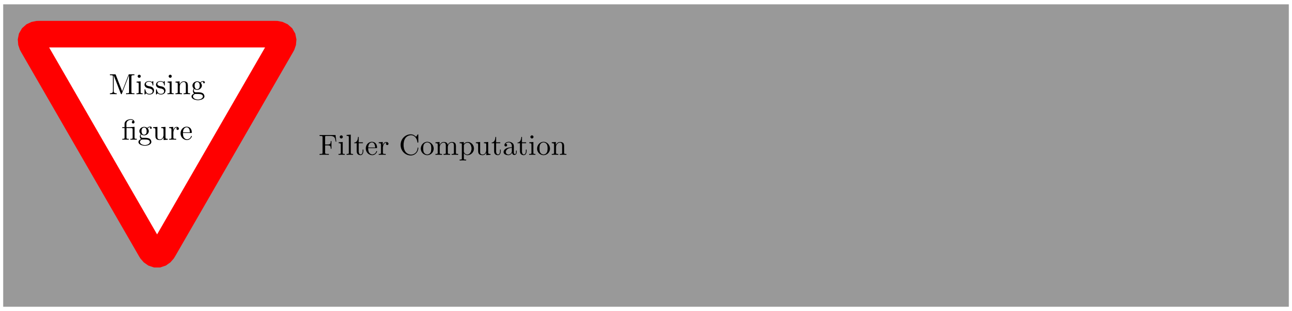
insert figure here

Select Computation

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

Figure shows how a Filter is computed.

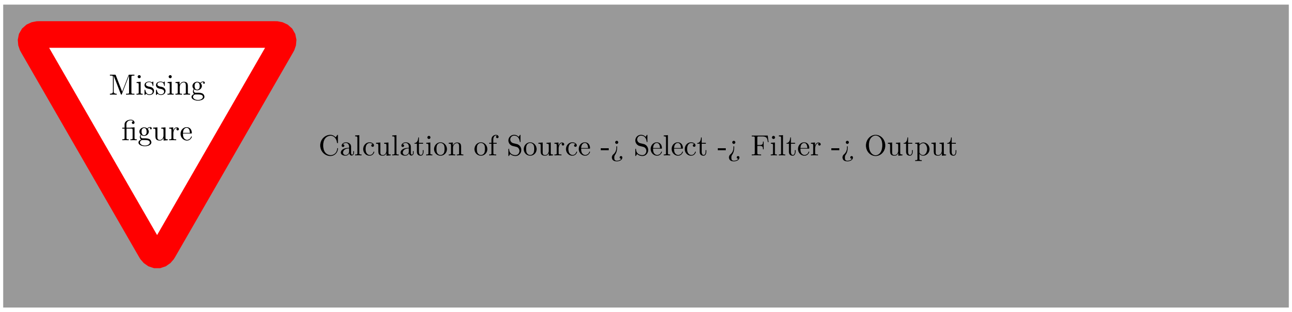
insert figure here



Filter Computation

Selects and Filters can be applied repeatedly, in any order from the source dataset to design a query. Figure shows an example query calculation using a series of Select and Filter operations. This process is repeated for each partition.

insert figure here



Calculation of Source -¿ Select -¿ Filter -¿ Output

3.4 Overall Solution

The main change from the MVP to the final solution is to add Group By operations. However, this is the first operation that cannot be computed row-by-row, which introduces a significant amount of complexity, as the worker nodes will now need to be stateful. In particular, the workers will have to cache partial results, and communicate with one another to generate new partitions for the Group By operation. This change also introduces further complexity for the Orchestrator, as it will have to manage the states of each worker, and ensure the worker state is cleared after a query completes.

3.4.1 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 new partitioning is required to continue, including Cassandra Source Data, and Group By operations. Table is a class, representing any part of the query that contains purely row-level computations, including Select and Filter operations. Tables depend on a specific DataSource.

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.

Furthermore, DataSources and Tables also have a partial form, which represents part of the full source dataset. *PartialDataSource* is also provided as an interface, as the implementation is different depending on how the dataset is partitioned. *PartialTable*, is essentially a duplicate of Table, but references a PartialDataSource rather than a DataSource.

Any query the user writes is made up of a mix of DataSource and Table components. Figure shows the outline for a Filter and Select query.

insert figure here

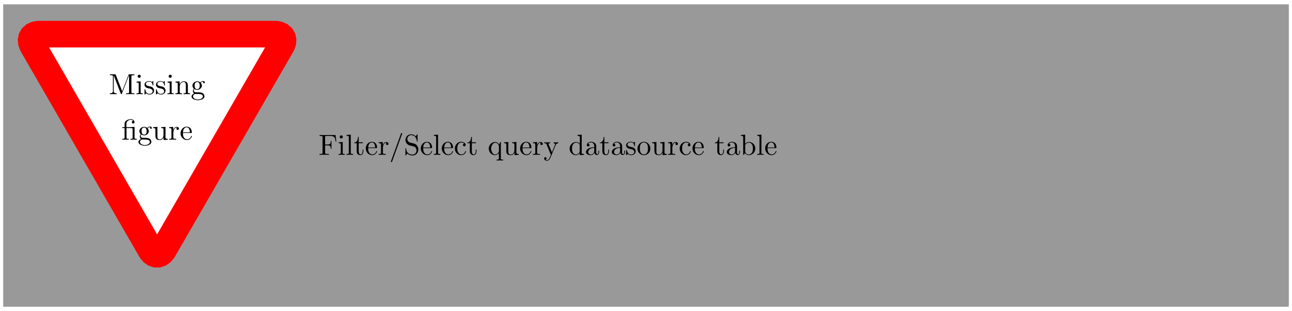


Figure shows the outline for a more complex query including a Group By. Note that any number of Select and Filter operations can be placed in the Table component, as no new partitions are required to produce the result.

insert figure here

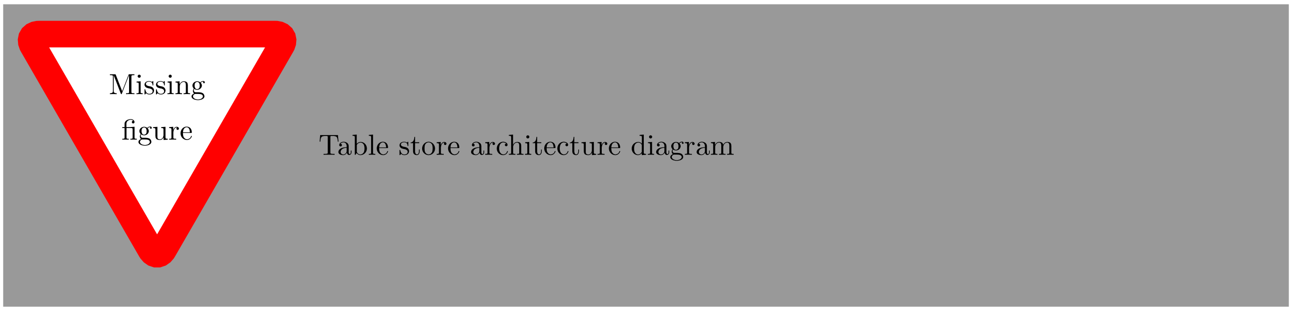
3.4.2 Data Store

The data store is a core worker component in the overall solution. It allows the workers to store partially computed data, which can be referenced again 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. Therefore, Akka Actors is used again in this component to simplify the implementation of the data store, by ensuring it only processes one request at a time.

The data store is modelled as an actor which contains three kinds of data: results, partitions and hashes. Results contain a set of outputs for a Table computation, partitions contain a set of outputs for a DataSource computation, and hashes are used in the process of computing a Group By partition.

The data store provides a set of create, read, and delete operations for each of these types of data, as well as helper functions for managing the data store state. Figure shows the architecture of the data store.

insert figure here



The data store is designed using two lookups at each stage. First, the full version of the data is looked up, then the partial version. 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 does improve the speed of get and deletes, which would otherwise require searching the entire data store to find the items to get or delete.

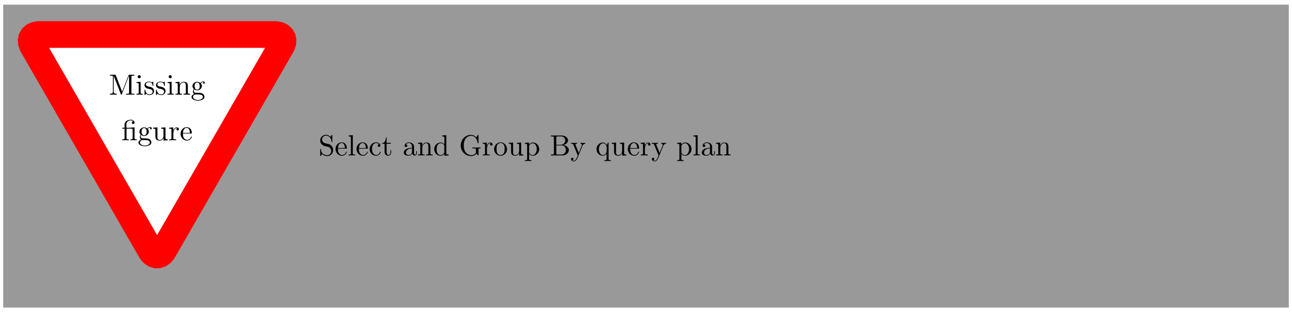
3.4.3 Query Plan

By separating Tables and DataSources, a generic framework for executing queries can be designed. This is known as a Query Plan. The Orchestrator is responsible for generating and managing the execution of query plans when the user makes a request.

A Query Plan is made up of a sequence of *QueryPlanItems*, which is an interface defining some 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 QueryPlanItems, which allow DataSources and Tables to be calculated and deleted, and both DataSource and Table have a function that generates the full Query Plan to compute their output from scratch.

Figure shows the query plan generated for the Filter and Select query and the Group By Query shown in 3.4.1.

insert figures here



Select and Group By query plan

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 shows the process for this item.

insert figure here

Flow with no dependencies

If the DataSource has dependencies, for example when computing a Group By operation, then Figure shows the process for this item. The details of the hashing process are abstracted behind the *DataSource* interface from the perspective of GetPartition, but further details of how this is implemented for Group By can be found in 3.4.4.

insert figure here

In both cases, GetPartition needs to manage the process of sending the requests to the workers to actually compute the partitions.

PrepareResult The aim of this QueryPlanItem is to compute a Table from 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. It uses a modified version of GetPartition's actor system to iterate through all partitions stored on each worker, sending a request for each to perform the Table computation for that partition.

DeleteResult and DeletePartition There are also delete operations to remove a result when it is no longer required, *DeletePartition* and *DeleteResult*. These will send a single request to each worker to delete a specific Table or DataSource. The workers will remove all results that relate to that Table or DataSource, and respond with a confirmation.

3.4.4 Group By

The Group By operation takes any number of NamedFieldExpressions to act as unique keys, as well as any number of aggregate expressions 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. As before, 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 four-step process of calculating hashes, shuffling data, computing the result and deleting the original hashes is all controlled by the query plan execution in the orchestrator

Hashing To estimate the size of the full dataset, an existing class, *SizeEstimator*, from Apache Spark was used, with a small number of changes for compatibility with Scala 3 . 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 shows how the size estimate can be used to derive the total number of partitions to generate for a Table.

reference

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

Figure 3.6: Group By - Total Partitions

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 the same as in

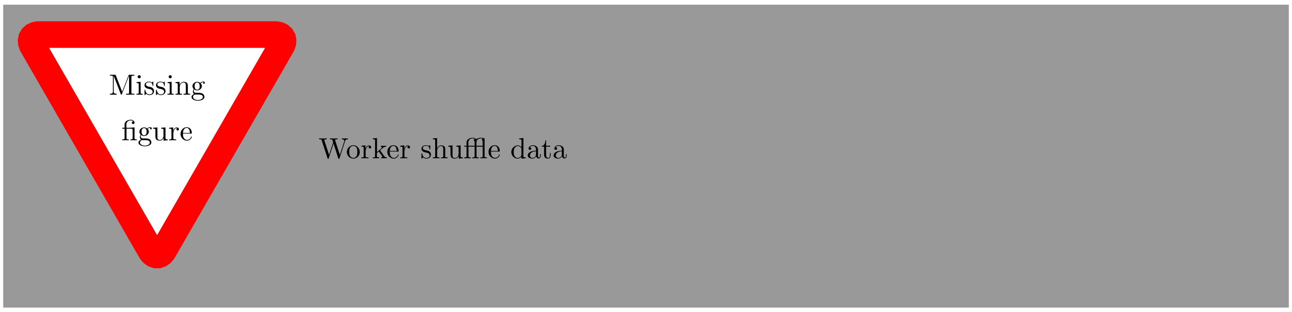
Cassandra and is provided natively by Scala. Figure 3.7 shows the high level equation for assigning rows to partitions. This process ensures that the new partitions will be roughly equal in size, and all rows with the same values for the unique keys will be mapped to the same partition.

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

Figure 3.7: Group By - Row Partition Assignment

Shuffling 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. Figure shows how this works. 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.

insert figure here



Computation 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. Figure provides a visual representation of this process.

insert figure here

3.5 Spill to Memory

3.6 Kubernetes

As discussed in the Design chapter, Kubernetes is used to manage the cluster, including Cassandra, the Orchestrator and the Worker nodes.

3.6.1 K8ssandra

Creating a Cassandra cluster on Kubernetes using K8ssandra is simple, with the key decision being the number of nodes to create. K8ssandra automatically provides scheduling rules to ensure that no more than one Cassandra node is placed on each Kubernetes node (physical machine). Therefore, the clear choice is to have the same number of Cassandra nodes as Kubernetes nodes in the cluster.

3.6.2 Worker Scheduling

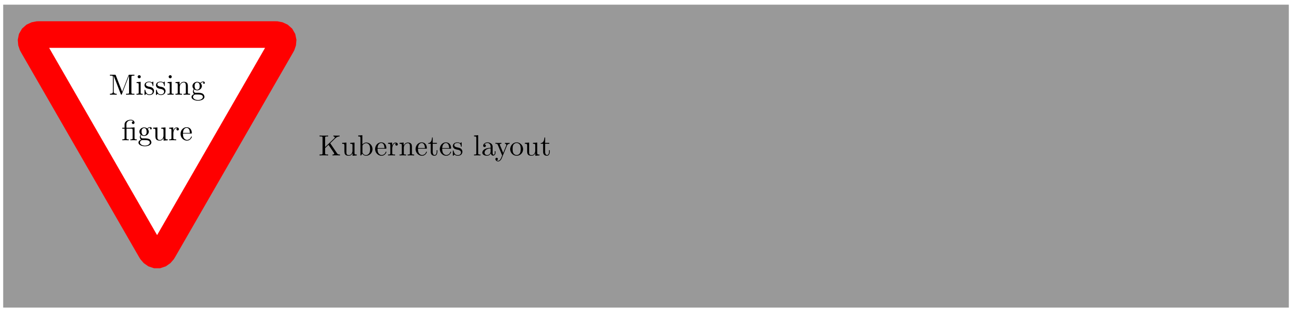
Scheduling rules can also be used to ensure optimal placement of worker nodes. As previously described in 3.3.2, worker nodes will determine their closest Cassandra node automatically based on latency. Therefore, only two scheduling rules are required to ensure optimal placement of workers.

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.

These scheduling rules ensure that the workers are spread evenly among the Kubernetes nodes, providing even coverage of all Cassandra nodes. If there are the same number of workers as Cassandra nodes, each worker corresponds to a single Cassandra nodes.

Figure shows the layout of the system with 3 Kubernetes, Cassandra and Worker nodes.

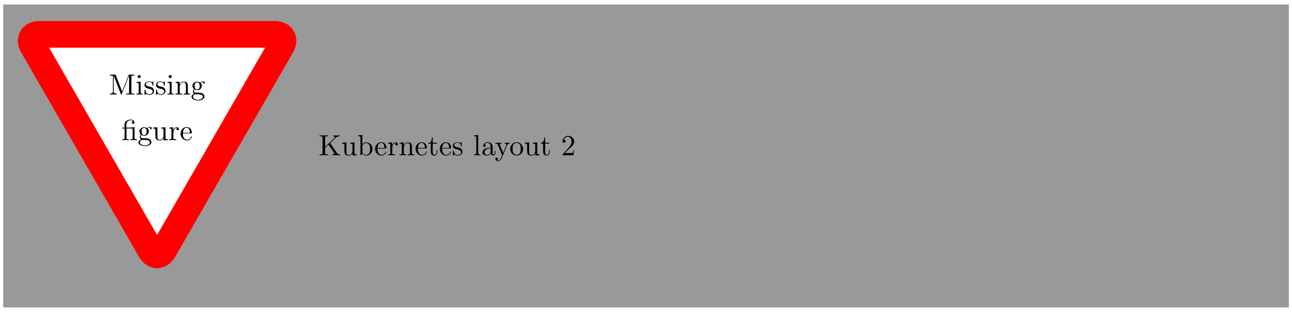
insert figure here



The scheduling rules are preferences rather than requirements, so the cluster is still able to handle the case where there are more workers than Kubernetes nodes.

Figure shows the layout of the system with 3 Kubernetes nodes, 3 Cassandra nodes and 5 Worker nodes.

insert figure here



Chapter 4

Testing

Chapter 5

Evaluation

Todo list

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] *Unified Types — Tour of Scala — Scala Documentation*. Available at: https://web.archive.org/web/20230327105547/https://docs.scala-lang.org/tour/unified-types.html. URL: https://docs.scala-lang.org/tour/unified-types.html (visited on 03/27/2023).

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

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

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

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