POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Distributed Arrow-based Shuffle Operation using Arrow Flight RPC

Supervisors

Prof. Paolo GARZA

Ing. Andrea FONTI

Candidate

Andrea FERRETTI

07 2023

Summary

The primary objective of this thesis work is to explore the potential of Arrow and Arrow Flight RPC in order to assess the feasibility of distributing the shuffle step in DataFusion. The thesis work begins with an overview of the shuffle operation and its significance in distributed data processing frameworks, implementing a prototype in Python that follows the behaviour of the MapReduce programming model. The prototype demonstrates the basic mechanics of leveraging Arrow Flight RPC to distribute the shuffle process, showcasing its potential for enhancing the efficiency of data transformations in distributed environments.

Starting from a naive version of the prototype, it is enhanced and extended in the following sections, where the design and implementation choices are outlined. Furthermore, a comprehensive evaluation of the prototype is conducted, comparing its performance against a non-distributed shuffle approach, measuring the execution time. By taking into account different scenarios, the obtained results analyze the strengths and limitations of the prototype implementation and offer insights into the potential scalability and performance gains achievable with a fully distributed shuffle step in DataFusion using Arrow Flight RPC.

Acknowledgements

Before proceding with the treatment, I would like to dedicate some space to everyone that helped me in this path towards personal and professional growth.

First of all, I want to thank my supervisors Garza and Fonti for their availability and promptness for every need I had regarding this work. Thank you for following me during these months.

Without the moral support of my girlfriend Giada and my parents, I probably would have never gotten this far. A heartfelt hank you for always being there when I needed it and for making possible an incredible experience in Denmark when I have done the internship there.

I would also like to thanks the parents of Giada, that gave me a good support in this hard journey. Thank you for being available and caring towards me.

Finally, I would like to thank my friends that helped reduce the stress in this intense period, by being there when I needed to.

Thanks to all of you, it would have been a lot harder without you.

Table of Contents

List of Tables vii

List of Figures viii

Glossary x

1 Introduction 1

2 Related Work 3

2.1 MapReduce . . . 3
2.2 Joins . . . 4
2.2.1 Hash join . . . 4
2.2.2 Sort-merge join . . . 4
2.3 Arrow . . . 5
2.4 Arrow Flight . . . 8
2.4.1 Downloading data . . . 8
2.4.2 Uploading data . . . 9
2.4.3 Exchanging data . . . 10
2.5 Datafusion . . . 11

3 Problem specification and system design 13

3.1 MapReduce implementation in Python . . . 14
3.2 Arrow local . . . 18
3.3 Arrow distributed . . . 21
3.3.1 Driver-Shuffler communication with the servers . . . 22
3.3.2 Driver-Reducer communication with the servers . . . 25

4 Experimental evaluation 29

4.1 MapReduce implementation in Python . . . 30
4.2 Arrow local . . . 33
4.3 Arrow distributed . . . 38

5 Conclusions 43

Bibliography 44

List of Tables

4.1 MapReduce performance with 10 unique keys 31
4.2 MapReduce performance with 15 million unique keys 31
4.3 Performance with 10 unique keys using Arrow 33
4.4 Performance with 15 million unique keys using Arrow 34
4.5 Distributed performance with 10 unique keys using Arrow Flight RPC 39
4.6 Distributed performance with 15 million unique keys using Arrow Flight RPC 39
4.7 Performance comparison between the local and distributed versions with 10 unique keys 41
4.8 Performance comparison between the local and distributed versions with 15 million unique keys 41

List of Figures

2.1 Row data format vs columnar data format, from [1] 5
2.2 Arrow array's data structure, from [3] 7
2.3 Retrieving data via `DoGet`, from Arrow Flight documentation [6] . . 9
2.4 Uploading data via `DoPut`, from Arrow Flight documentation [6] . . 10
2.5 Exchanging data via `DoExchange`, from Arrow Flight documentation [6] 11

3.1 MapReduce example 15
3.2 Partitioning keys using md5 16
3.3 Data formats between the different steps 18
3.4 Sequence diagram of the driver-shufflers communication with the servers 24
3.5 Sequence diagram of the driver-reducers communication with the servers 27

4.1 Time taken for each step in 100 iterations, with 20 partitions 35
4.2 Time taken for each step, per partition, in the two tests 36
4.3 Total time taken per partition in the two tests 37
4.4 Time taken for each step in the two distributed tests 40
4.5 Total time taken for each step in the two distributed tests 40
4.6 Time taken for each step varying the number of input files in the distributed version 42

Glossary

hashing Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a hash value or simply, a hash.

Chapter 1

Introduction

In the era of big data, the ability to efficiently process and analyze massive datasets is crucial for businesses and organizations across various domains. To meet the demands of processing such voluminous data, distributed data processing frameworks have emerged as powerful tools. These frameworks leverage the parallelism of distributed computing clusters to execute complex computations in a scalable and efficient manner. One key operation in these frameworks is the shuffle step, which involves redistributing and grouping data across nodes to enable subsequent data transformations and aggregations.

Apache Arrow, an open-source in-memory data format and associated libraries, has gained significant attention in the data processing community due to its columnar representation and efficient memory utilization. It provides a standardized way to represent and manipulate data structures across different programming languages. Leveraging the capabilities of Arrow, several data processing frameworks, such as DataFusion, have been developed to harness the benefits of vectorized processing and enhance query performance.

DataFusion, an open-source query engine built on Apache Arrow, has gained significant attention in the field of distributed data processing due to its performance optimizations and compatibility with the Arrow data format. By leveraging the benefits of vectorized processing and memory-efficient data structures, DataFusion offers substantial performance improvements over traditional query engines. However, despite its remarkable capabilities, DataFusion still relies on a centralized shuffle step, which can become a performance bottleneck in certain scenarios.

To address this limitation, this thesis work wants to explore the feasibility of implementing a distributed shuffle step in DataFusion using **Arrow Flight RPC**, thereby enabling enhanced scalability. Arrow Flight RPC, also based on Apache Arrow, is a high-performance remote procedure call framework designed for efficient data transfer between different processes and machines.

The primary objective of this thesis work is to develop a prototype implementation that demonstrates the potential of utilizing Arrow Flight RPC for distributed shuffling. Although achieving a fully distributed shuffle step in DataFusion within the scope of this work may be ambitious, the prototype implementation will serve as a foundation for evaluating the performance gains, limitations, and challenges associated with the distributed shuffle approach using Arrow Flight RPC.

This thesis work begins by providing a comprehensive overview of the shuffle operation and its significance in distributed data processing frameworks. The fundamental concepts and challenges of distributed shuffling are discussed, highlighting the need for efficient and scalable solutions. The subsequent sections delve into the capabilities and features of Arrow and Arrow Flight RPC.

Following that exploration, the thesis focuses on the design and implementation of the prototype for the distributed shuffle step using Arrow Flight RPC in a simplified program. To evaluate the effectiveness of the implemented prototype, a comprehensive performance analysis is conducted and the results are compared against a non-distributed shuffle approach.

Chapter 2

Related Work

2.1 MapReduce

MapReduce is a programming model and implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. The important contributions of the MapReduce framework are the scalability and fault-tolerance, leading to an advantage in performance when using multi-processor hardware containing multi-threaded implementations. Using this model is beneficial only when the distributed shuffle operation is optimized and are present the tolerance features of the MapReduce framework. Therefore, an important part is the optimization of the communication cost.

A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation. The major advantage of the MapReduce model is its scalability, as it enables easy parallelization and distribution of large data sets across a cluster of computers (nodes)

MapReduce is usually composed of three operations, which are controlled by a master node:

1. **Map**: each worker node applies the map function to the local data, writing the output to a temporary storage.
2. **Shuffle**: worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
3. **Reduce**: worker nodes now process each group of output data, per key, in parallel.

In this work, we implemented a simple version in Python to start with, so that the single operations can be studied and visualized.

2.2 Joins

A join is a fundamental operation in relational databases that allows to combine data from multiple tables based on a related column between them. A join is performed by matching each row from one table with matching rows from other tables, creating a new result set that combines data from all the tables involved.

In this work, we will consider two types of join: **hash join** and **sort-merge join**, which are involved in the shuffling operation that is performed in our work (hash join) and in datafusion 2.5.

2.2.1 Hash join

A hash join is a popular technique used in database systems to efficiently join large tables together based on a common attribute. In a hash join, the tables to be joined are first partitioned into smaller subsets based on the values of the common attribute. Each subset is then hashed using a hash function, and the resulting hash table is stored in memory. The hash tables for each subset are then probed to find matching records, and the matching records are combined to form the final result set.

Hash joins are commonly used in data warehouses and other large-scale database applications because they can be highly parallelized, allowing them to take advantage of the processing power of modern multi-core processors. They are also highly efficient, requiring only a single pass over each input table, and can handle tables with millions or even billions of records.

2.2.2 Sort-merge join

A sort-merge join is another common technique used in database systems to efficiently join large tables together based on a common attribute. In a sort-merge join, both input tables are first sorted on the common attribute. The sorted tables are then merged together using a two-pass algorithm that reads both tables sequentially and merges matching records.

The first pass of the algorithm reads both tables and produces runs of matching records that are written to temporary files on disk. The temporary files are then merged together in the second pass to produce the final result set.

Sort-merge joins are also highly efficient and can handle large tables with millions or even billions of records. However, they can be slower than hash joins because they require sorting both input tables, which can be expensive for large datasets.

2.3 Arrow

Apache Arrow, developed by the Apache Software Foundation, is an open-source **columnar in-memory** data format that aims to provide a standardized and efficient way to represent structured and semi-structured data for high performance processing across different systems and programming languages. It removes the serialization costs providing zero-copy streaming messaging, an Interprocess Comunication (IPC) format and it guarantees a constant-time when randomly accessing the data.

Traditional data formats (such as CSV or JSON) are **row-based**, storing data row by row, which can be suboptimal for many analytical workloads, but efficient for OLTP (online transactional processing). Row-stores are known to perform well for a single transaction like inserting, updating, or deleting small amounts of data, since in these cases it is better considering the record as a whole rather than a subset of it. However, if only a field is required, all the record has to be read, leading to a higher computational cost than what is required. A columnar data format, instead, stores data in a contiguous column oriented way, which is better for OLAP (online analytical processing). When it is required to aggregate the data, considering only a subset of the values, it is more efficient to read only the required columns rather than all the rows and then only using the columns of interest. Therefore, for partial reads column-stores are preferred since a smaller volume of data is loaded. In figure 2.1 is shown an example to better understand the difference.

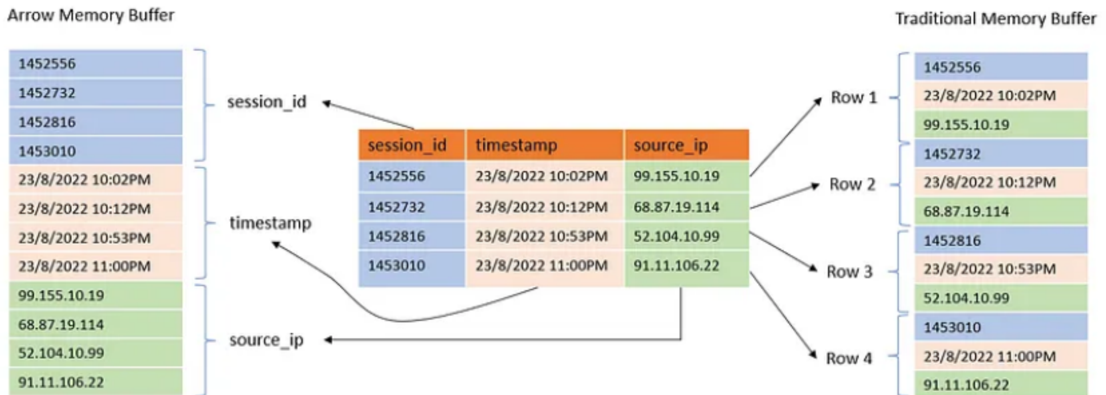

Figure 2.1: Row data format vs columnar data format, from [1]

The aim of Arrow is to reduce the overload when communicating with different services, removing the need of serializing / de-serializing the data. The in-memory data representation in Arrow is equal to the representation of the data when it is

stored, which can then be used in any programming language without the need to convert it in a suitable format, but can be just read as-is. Arrow is a **memory-mapped** format, which means that the data can be used directly from the disk without having to load all of it in memory. This is thanks to Arrow serialization design, which provides a data header describing where the memory buffers for all the columns in a table are located and their size. In [2] the performance of these data formats for operations on a dataset is analyzed, and is useful to see how Arrow with memory mapping performs far better than using the dataset stored in csv or even Parquet, which is a columnar data format for storing data:

- when using Parquet, only the required columns can be read, which is faster than having to read the entire dataset as in csv. Arrow is faster than Parquet because it removes the need of decompressing the data, which is an important part of Parquet that reduces the size of the dataset on the disk but requires some time to be serialised / de-serialised.
- If the dataset does not contain null values, Arrow can use it directly from disk with zero-copying, which obtains the best performance.

The key data structures of Arrow are **table** and **record batch**. A table is a list of chunked arrays (vectors of values and their data type) with a schema, while a record batch is a list of arrays (vector of values and its data type) with a schema.

In an **array**, data is stored in one or more buffers, where a buffer is a contiguous block of memory with a given length. The array has also some metadata to retrieve the length of the array and the number of null values. The number of buffers associated with an array depends on the type of data being stored. For an integer array there is a validity bitmap buffer and a data value buffer. In figure 2.2 is shown how the array data structure would be in memory.

The validity bitmap buffer is binary-valued and must be large enough to have at least 1 bit for each array slot. It has a 1 if the corresponding slot in the array is a valid value, 0 otherwise. In this case, byte 0 would contain all 5 values, and the other 63 bytes would be just composed by zeros.

The data value buffer has a similar logic: it is padded out to a length of 64 bytes and each slot in this case is composed by 4 bytes (since the values data type is an integer of 32 bits). An important thing to mention about an Arrow array is that it is an immutable object, which reduces the need to create copies. This is a huge advantage for large datasets, but it comes with some limitations: when a new batch of data arrives, the only solution would be to create a new array, which would split the data in two.

Chunked arrays solve the problem that immutable arrays have: a chunked array is a wrapper around a list of arrays, allowing to index their contents as if they were a single array. Each array is one chunk which is stored in separate places in memory, but with its abstraction they look like they are all one thing.

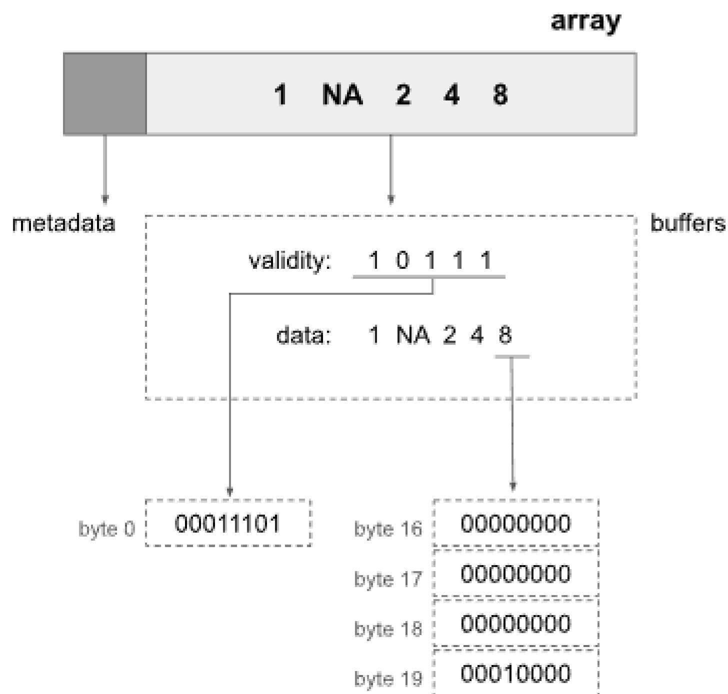

Figure 2.2: Arrow array's data structure, from [3]

A **record batch** is a data structure similar to a chunked array, since it is a sequence of arrays, with the difference that the arrays can be of different types but must all be the same length. Each array is referred to as one of the "fields" or "columns" of the record batch. In memory, the record batch simply contains pointers to the arrays and its own validity bitmap.

When we want to communicate with different services, normally the data would be serialised so that it can be written to disk or a stream. This leads to more efforts to store and read the data between different processes. To get around this, the Interprocess Communication (IPC) serialisation format specified by Arrow is designed to ensure that Arrow data objects can be transmitted (and saved) efficiently, ensuring that the structure of the serialised record batch is essentially identical to its physical layout in-memory. To store a record batch, the IPC format collects the relevant metadata into a "data header", and then lays out the buffers one after the other.

A **table**, instead of storing each column as an array, stores it as a chunked array. For this reason, tables can be concatenated without creating a new one every time there is new data. Since tables are much more flexible than record batches, arrow functions tend to return tables. However, in this work we want to use record

batches, since we are studying the behaviour of the basic data structure used for communication.

2.4 Arrow Flight

Arrow Flight is a RPC (Remote procedure call) framework based on Arrow data and built on top of gRPC [4] and the IPC format. In Flight, streams of Arrow record batches are downloaded from or uploaded to another service, with a set of metadata methods describing the streams. Methods and message wire formats are defined by Protobuf [5], which helps with interoperability with clients that may support gRPC and Arrow, but not Flight. However, Flight implementations have optimized the usage of Protobuf by avoiding excessive memory copies and, in general, overhead usage.

Flight define a set of RPC methods to operate on data streams that a client can call, while a service implements some subset of these methods. Each Data stream is identified by a descriptor, which is a path or a command. A Flight client can connect to any service and perform basic operations, where a Flight service is expected to support some common request patterns: downloading data, uploading data, exchanging data.

2.4.1 Downloading data

In order to download the data from a Flight service (figure 2.3), a client would need to:

1. Have a `FlightDescriptor` for the dataset to be downloaded.

   This can be already known and therefore constructed by the client or unknown and acquired by a method that gives information about the data stream.

2. Use the descriptor to call `GetFlightInfo(FlightDescriptor)` and obtain a `FlightInfo` message.

   Since in Flight data may be on different servers, the metadata server could just return details on where the data is located. This is encoded as a series of `FlightEndpoint` messages inside `FlightInfo`, where each endpoint represents some location with a subset of the data. More in detail, an endpoint contains a list of locations where the data can be retrieved from, and a `Ticket`, which is an opaque binary token used by the server to identify the requested data.

3. Consume each endpoint returned by the server.

   This is done by connecting to one of the locations in the endpoint, then call `DoGet(Ticket)` with the ticket in the endpoint. This method will return the

client a stream of Arrow record batches. The endpoints represent data that is partitioned or otherwise distributed and, to retrieve the complete dataset, the client must consume all endpoints, which can be done in parallel.

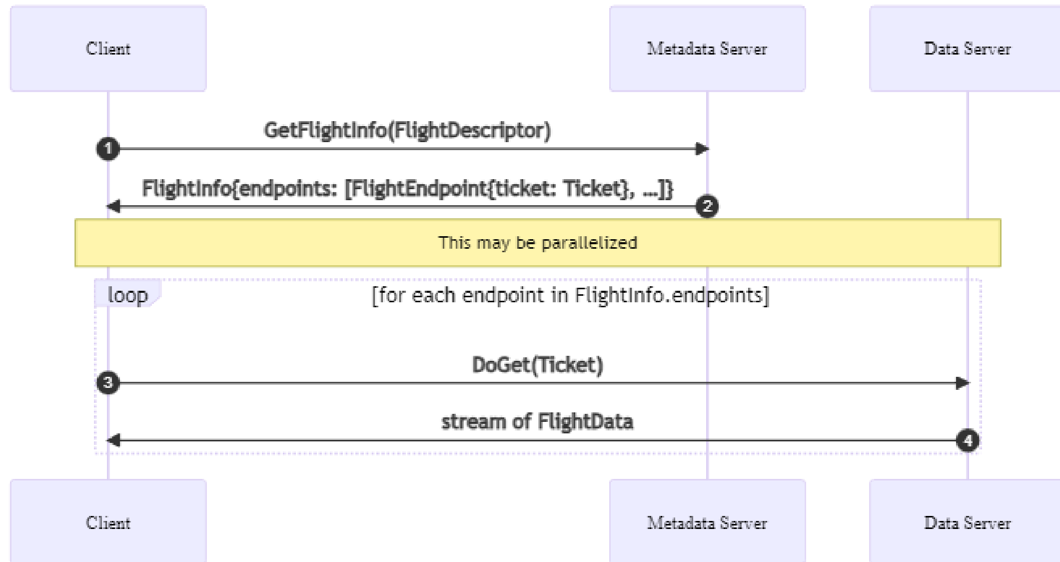

Figure 2.3: Retrieving data via `DoGet`, from Arrow Flight documentation [6]

2.4.2 Uploading data

To upload a data, in figure 2.4 is shown what a client needs to do:

1. Construct or acquire a `FlightDescriptor`, like before.
2. Upload a stream of Arrow record batches by calling `DoPut(FlightData)`, where `FlightData` contains the `FlightDescriptor` so that the server can identify the dataset.

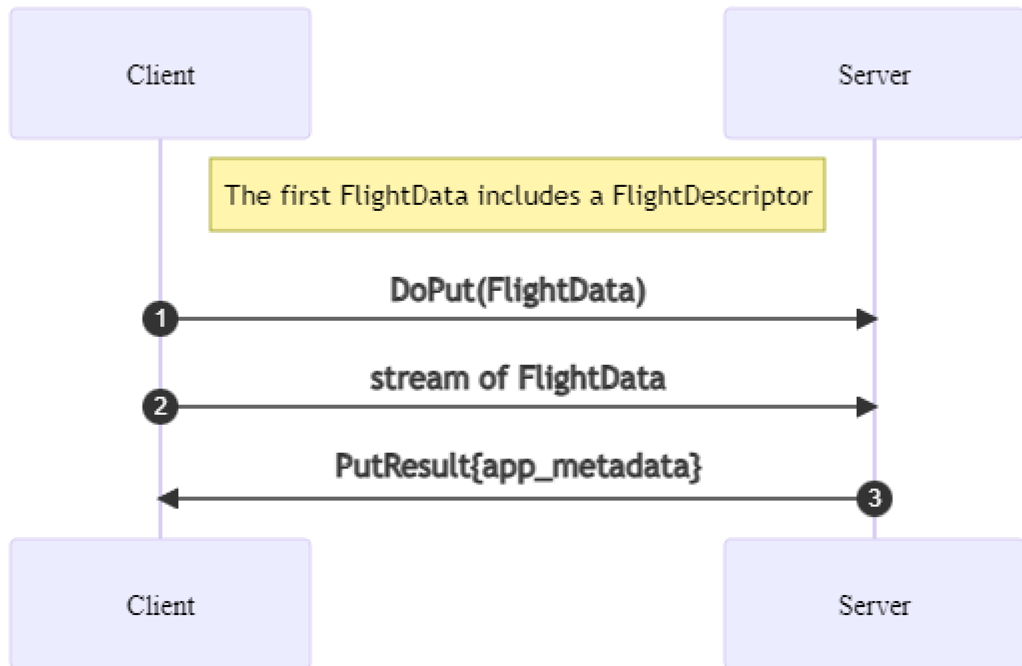

Figure 2.4: Uploading data via `DoPut`, from Arrow Flight documentation [6]

2.4.3 Exchanging data

In some cases it might be required to upload and download data within a single call (for example, the client may upload data and wants the server to respond with a transformation of that data), instead of being stateful which could be difficult. To exchange the data, a client would (figure 2.5):

1. Construct or acquire a `FlightDescriptor`.
2. Call `DoExchange(FlightData)`, where the `FlightDescriptor` is included with the first message. After that, both the client and the server can simultaneously stream data to the other side.

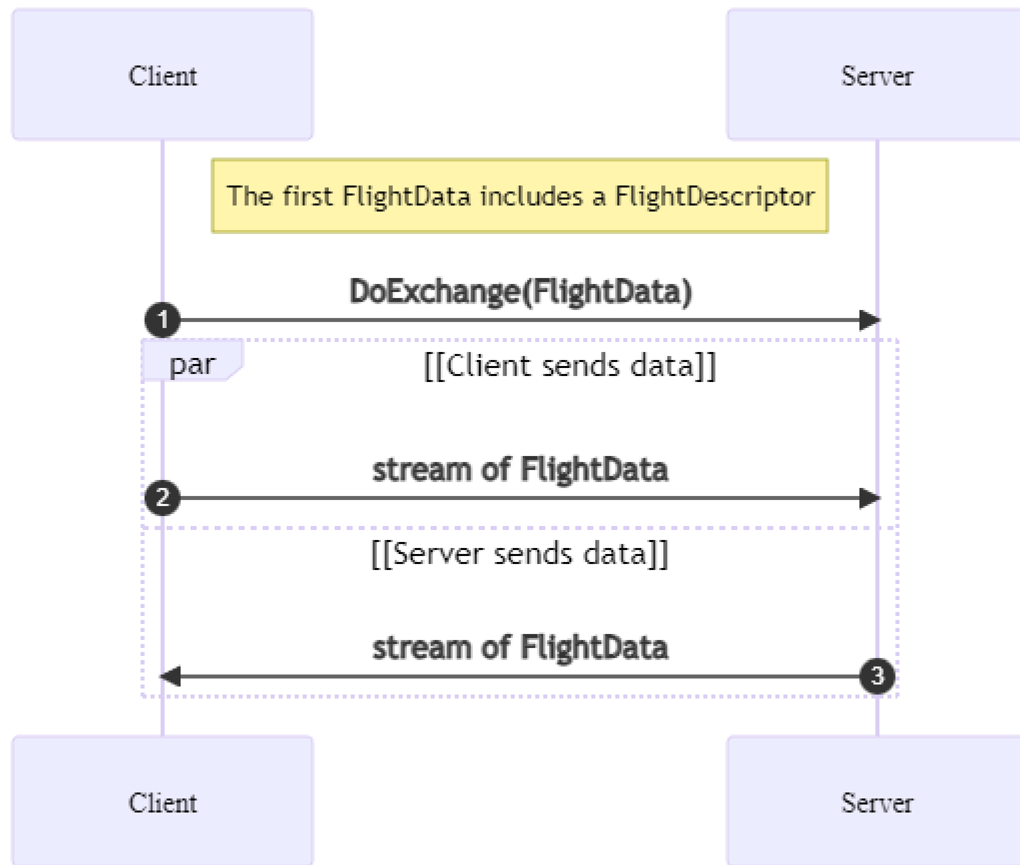

Figure 2.5: Exchanging data via `DoExchange`, from Arrow Flight documentation [6]

2.5 Datafusion

Arrow Datafusion is an open-source data processing framework that leverages Apache Arrow's in-memory columnar data format to accelerate data processing across a wide range of data sources and processing engines. Arrow Datafusion provides a unified interface for developers to build data processing pipelines that can run on a variety of compute platforms, including local machines, clusters, and cloud environments. It supports SQL and other high-level languages, and can be used to build both batch and streaming data processing applications, quickly running complex queries using a sophisticated query planner, a columnar, multi-threaded, vectorized execution engine, and partitioned data sources. It is designed for easy customization and is implemented in **Rust**, which is a fast and memory-efficient programming language, with no runtime or garbage collector, that leverages on the

concept of ownership to guarantee memory-safety and thread-safety.

Datafusion can run complex SQL and DataFrame queries using a query planner, a columnar, multi-threaded, vectorized execution engine, and partitioned data sources. It can be extended in many points with user-defined functions, custom optimizer rules, personal schemas, table lists or datasources.

After collecting and filtering all the data, by executing a specific query logical and execution plans are created. Both plans are also characterised by an optimization process. The logical plan defines what data is needed and in what order, the optimization process can speed up execution times significantly, with 14 built-in optimization passes and the ability to add custom ones. The execution plan then goes more into detail on how to execute the query and where the data is. Some optimization rules may be projection and filter push down, which minimizes the amount of data that needs to be read by reading only the desired columns from the input files.

The optimization and execution plan, when executing a Rust query engine using Datafusion, work by coalescing batches of data to make them big enough for efficient processing and introducing extra parallelism with the `repartition exec` function, that splits data into multiple partitions.

Chapter 3

Problem specification and system design

The focus of our proposed work is to explore the feasibility of extending DataFusion 2.5 in a distributed way so that, when the data is partitioned, each partition can be sent to a worker node which will perform the required operation, sending back the result. In order to do so, the first step is to create a prototype in Python that should represent the behaviour of the application in an easier way to visualize and to study, which will be useful to implement a working distributed environment while studying Arrow Flight 2.4 This will lead to the only requirement of converting the interested parts to Rust with a solid foundation already implemented. The prototype is a MapReduce 2.1 program, with the focus on the shuffle part.

The proposed work is divided in three phases, which constitute three versions of the prototype, and help looking into different functionalities one at a time instead of developing everything together at once:

1. First of all, we implement the MapReduce in Python, in order to have an in-depth analysis over its functioning. This helps acquiring a solid understanding of the MapReduce paradigm, its components and how they work together to process large-scale data, ensuring that the program correctly performs data processing and handles data serialization and deserialization.

2. After that it is important to adapt the program to use Arrow's record batches as the inter-process communication format, therefore extending our MapReduce work using Python's module for Arrow (`pyarrow`). This step is incorporating Arrow's in-memory format, which by utilizing Arrow's data structures and APIs it improves data processing efficiency, reduce memory usage and enhance overall performance, also taking advtange of its capabilities for efficient serialization and deserialization of data.

3. Finally, the program is extended to a distributed version using Arrow Flight as the RPC framework for handling communications between different nodes exchanging Arrow data, which implements exchange mechanisms to facilitate efficient and scalable distributed shuffling.

The final result is a MapReduce program with the shuffle step distributed, so that the data is stored in multiple data servers that are not on the same cluster as the machine performing the steps, so that the reducer step can get the data from those data servers. This is to show how Arrow and Arrow Flight works to enhance the capabilities of a MapReduce application and to prove that the shuffle step in DataFusion can be extended by adapting this work to collaborate with it.

3.1 MapReduce implementation in Python

Suppose we want to know how many clicks a website is having, giving each click a value based on it being directly from a person (*User*, which is valued 2) or redirected by another website (*Robot*, which is valued 1). In this scenario, each record of the data would have the website's URL and the associated value for each specific click. In the MapReduce fashion, the keys would be each unique website's URL and the value would be the associated click value. In order to achieve our goal, we design a MapReduce program in Python that has to:

1. Apply a mapping function to every input file that maps the input values to be as requested, so that the total sum can be computed later.
2. Group the keys so that every value referred to the same URL, coming from different files, has to be collocated in the same group.
3. Sum up all the values for each key, returning as result the total number of clicks, with the right "weight", per URL.

Figure 3.1 shows an example of MapReduce program, where in each step the desired output is indicated. In the example we consider a dataset that is split among 3 nodes, having 2 lines per node (1 in the last). The data divided in 3 worker nodes can also be referred to as input partitions where ideally, for each partition, there is one Mapper that performs the map operation. Then, we can decide the number of output partitions, based on the characteristics of the worker nodes that we have, since dividing the process in multiple nodes requires time and memory to communicate between those nodes. This is limited by the size of the dataset: for example, if the dataset is 1TB and the worker nodes have 16GB of memory, it would be very slow for them to handle all the data, while having 64 nodes (1024GB / 16GB) may improve the time by making each one of them

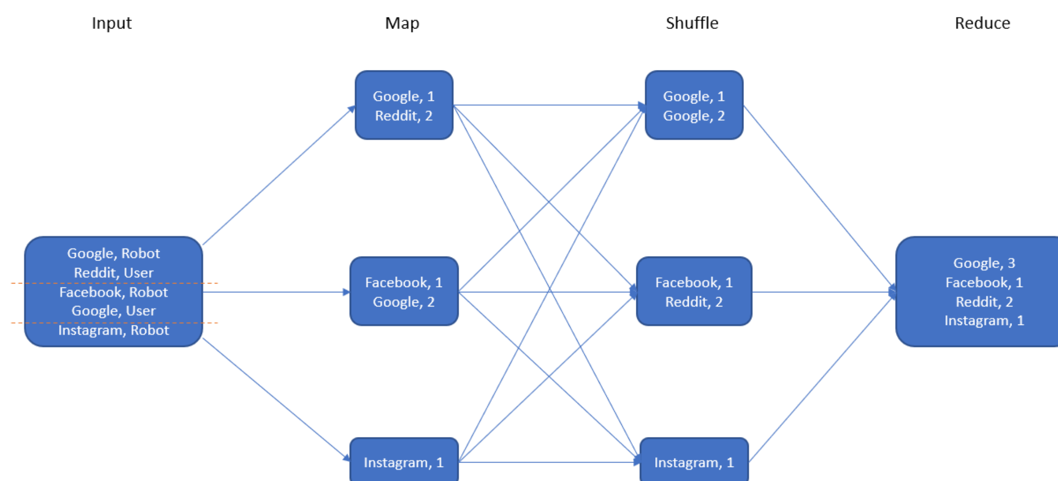

Figure 3.1: MapReduce example

go at full capacity. In this case there are 3 output partitions. The program uses Python's module `joblib`, which makes it possible to create a desired number of python processes so that we can simulate the behaviour of separate worker nodes, and is organized in different scripts. The *main.py* acts like the master node which has the role of deciding the number of processes running for each of the 3 steps and is the driver handling the calls of each step with the right input parameters and at the right time. The steps are:

- **Map**: the *mapper* takes all of the input partitions and processes them, replacing the values associated to each key with 1 if the value is "Robot", 2 if "User". It then temporarily stores the new partitions.

- **Shuffle**: the *shuffler* is the most memory consuming step since for each input partition, it stores the desired number of output partitions. In this case, we have 3 input partitions and want 3 output partitions, so the *shuffler* is creating 9 temporary files. The organization of the output partitions is the most important part and the *shuffler* has the role of storing the same keys in the same partitions. This is a crucial assumption that the *reducer* will be doing and can be performed using a **hashing** function that can return the same value given the same key. In this case, we are using `md5` to get the hashed value of a key, which is then divided by the total number of output partitions. The row with that key is stored in the partition number equal to the resulting remainder of the division (see Figure 3.2).

- **Reduce**: finally, the *reducer* has the job of reading all the temporary files created by the *shuffler* for each partition and apply the desired computations

on its rows, returning one file per partition. In this case, since we want to know the total number of clicks, the *reducer* is summing up all the values per key.

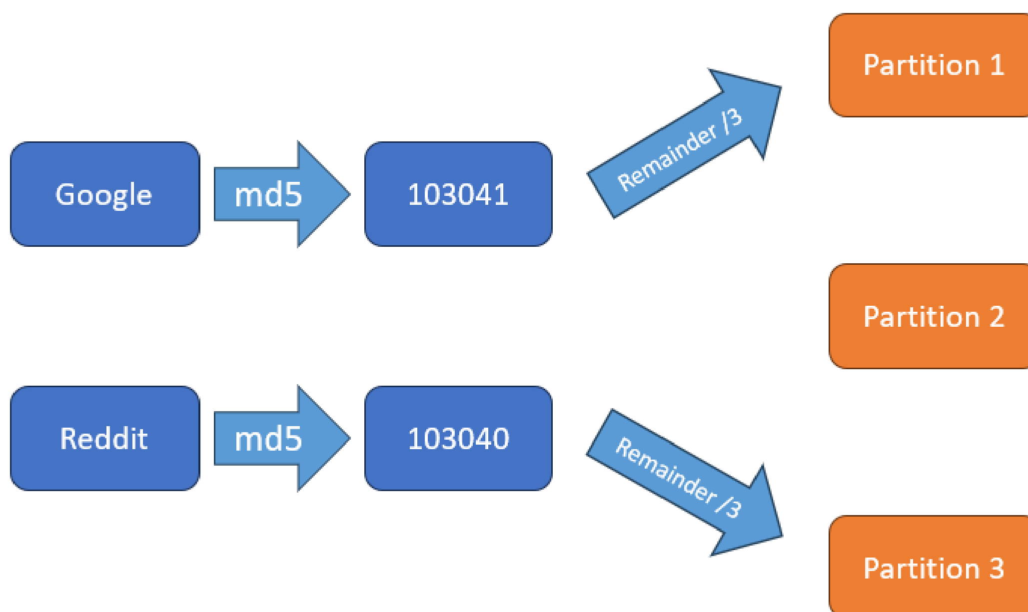

Figure 3.2: Partitioning keys using md5

It is important to notice that the behaviour of the hashing function on the data distribution depends on the specific algorithm used for hashing and the distribution of the data itself. In general, a good hashing function will evenly distribute the data across the nodes in the system, so that each node has a roughly equal amount of data to store. However, there are some factors that can affect the evenness of the data distribution, such as the hashing function being not well-designed, or the input data being not evenly distributed, which would result in some nodes having a higher load than others.

Generally, when the input data size is increasing there is less probability that the output partitions will be unbalanced, since with more keys, even if not evenly distributed, the hashing function will tend to even the distribution in output. Moreover, the *mapper* and *shuffler* use a number of parallel processes equal to the number of input files, since they can process one file each, while the *reducer* number of processes is equal to the desired number of output partitions.

An important part of the designing is deciding the number of partitions, which determines the granularity at which the data is divided and processed in parallel.

The number of partitions has an impact on many factors:

- **Data distribution**: If there are more partitions than the available reducers, some reducers may end up with minimal or no data to process, which results in utilizing less of the available resources. Oppositely, with fewer partitions than reducers, some reducers may become bottlenecks since they would have to handle large amounts of data.
- **Task granularity**: More partitions allows better load balancing and potentially reduces the overall execution time. This is true but with a limit, since having too many small partitions can introduce overhead when coordinating the different processes.
- **Shuffle overhead**: By increasing the number of partitions, the amount of data transferred during the shuffle phase would increase as well, impacting network bandwidth and overall performance. This is because every shuffle is performed on a single file, which results in a number of output temporary files equal to the number of partitions. More partitions means more files, which increase the overhead and the communication costs.
- **Reducer efficiency**: More partitions means less data to be handled by a single reducer, which can improve the processing times for each one of them. However, too many partitions would lead to each partition being too small, having an overhead for setting up and managing each reducer impacting much on the processing time.

Therefore, choosing the number of partitions in very important, and it will be done by testing different cases in Section 4. Specifically, for this first version, the results will be shown in 4.1, where the best number of partitions will be found.

3.2 Arrow local

Now that the first version has been studied, we want to change it in order to have Arrow as the data format for storing the temporary files in each step. The reason for this choice is relative to the aim of extending the application to be distributed using Arrow Flight, which handles Arrow data as the inter-process communication format.

To do so, the first step is to change the application to make use of Arrow Record Batches, storing all the temporary data in between in Arrow IPC format. Therefore, the input partitions are in csv, but the output of the map and shuffle operations are in Arrow IPC, while the final output of the reduce is again in csv (Figure 3.3).

Figure 3.3: Data formats between the different steps

Arrow has a python module called `pyarrow` which is imported to add its functionalities. The objective here is to change the code which made use of Python's objects, into a program that could benefit from Arrow's qualities, whenever possible, maintaining a good execution time as before. The data will not be considered one record at a time anymore, but one Record Batch at a time, where a Record Batch is an Arrow data structure which contains arrays of equal-length but different data types, which are specified in the schema of each batch.

The first version of the MapReduce using Arrow is as follows:

- **Map**: the mapper reads the csv file of the input partition and stores each row in two separate Python arrays, one for the keys and one for the values. For the values array, the value appended is 1 for "Robot" and 2 for "User". When the number of rows read is equal to the desired batch size (which is specified in the configuration file), a record batch is created containing both arrays, which are the two columns, and written to the relative output file. The schema for the record batch is specified in here and will be used the same in every step: string for *keys* and int32 for *values*.

- **Shuffle**: the shuffler is now reading an Arrow IPC file composed by multiple record batches of the same size (except the last one), which are read one by one. Here, the memory consuming part is storing, for each output partition, the records that will be used to populate the record batch when they are enough. For this purpose, a Python list is used, in which every index of the

list corresponds to the output partition number. Every element of this list is a dictionary, where the key is the batch number that is being read and the value is a list containing the indices where the records are, for the desired output partition, inside that specific record batch. The list of partition indices for each record batch is obtained by applying the hash function to each of the keys inside of it and then taking the remainder of the division on the number of output partitions. Therefore, for every record batch read, we iterate through the list of partition indices and store inside the list, at the index number relative to the partition number being considered at the iteration, the dictionary relative to this record batch's number containing the list of indices where the records in this batch that should go in that partition are. Every time one of the elements of the list reaches the desired batch size, a record batch is created taking the records at the right index in each input record batch, and it is then written in the output Arrow IPC file.

- **Reduce**: the reducer receives as input a series of Arrow IPC files that has to reduce to one final csv file for its output partition. To do so, a Python dictionary is again used, storing the click value for each key. Whenever a value for the same key is found, it is summed with the current value in the dictionary. This is done partially (for every record batch that is being read) and then again after having stored all the data in memory, which is possible because the sum is an associative function. The values to be stored must be converted from Arrow to Python objects, since Arrow integer scalars (which are the input data types for the clicker column) can't be summed up.

An important thing that can immediately be noticed is the fact that Python's objects are still being used, which limits the potential of Arrow. This is related to the immutability of Arrow objects, so that it is not possible to modify an array or a dictionary once created. This version is tested and it shows a performance similar to before in terms of execution time, which tends to increase more when increasing the number of output partitions. In section 4.2 the program is tested with different parameters and the results are studied.

What matters here is that the memory used by the program is less, since only Python's objects are being allocated in memory while Arrow makes use of its memory mapping functionalities (section 2.3). Also, now that we're considering one record batch at a time, it can easily be extended to a distributed scenario, where it will be easier to parallelize the execution by sending a batch every time it is written, without having to wait that all the file has been read and processed.

After this first version, two modifications have been made to improve the performances:

1. The reducer has not changed much from the previous Python only version, hence some modifications could be tried. Instead of using a Python dictionary,

it is possible to obtain a dictionary encoded array from the Arrow array containing the keys (the first column of the record batch in input). By doing so, the only Python object left is a list containing the click counter for each key, reducing the memory consumed. Then, all the keys arrays (which are the unique keys obtained from the dictionary encode) are concatenated into a single one and the click counter is updated accordingly. This can be performed partially without having to wait that every value has been read, since the sum is an associative function and works even if the records are aggregated step by step. Therefore, for every input batch a reduce is performed and then all the partial results are joined together to do the final reduce, which will save the results in a csv. This version did not have a considerable impact on the reducer's execution time, but improved the memory allocations performed, so that now less memory is used to store the data processed.

2. Finally, a big improvement has been made inside the shuffler: instead of waiting for a batch to be the specified size, it is written to the temporary output files every time a input batch is read. Therefore, if the keys are uniformly spread, the batches will have roughly the same size, which would be the mean of the number of keys per batch over the total number of partitions. This improved the execution time and the memory consumption, by not having to store all the records inside Python objects before being able to write a batch of the proper size.

3.3 Arrow distributed

In order to extend the program to a distributed version, Arrow Flight RPC (2.4) is used. Thanks to this framework, the current local version based on Arrow data can easily be adapted to work in a distributed fashion, implementing the client / service communication methods.

What we want to do is manage the most consuming part of the data (the temporary files stored by the shuffle) in different remote machines, which would decrease the impact on the local working machine. This may also be extended to a complete distributed version where each step in the application is on a different machine rather than on a different process on the same machine, but it is not required unless the data is very large. However, once implemented the methods specified by Arrow Flight, it will not be difficult to further extend it.

Therefore, this final version of the prototype will focus on communicating with a server to store the data resulting from the shuffle operation in a distributed fashion, and then gather that data when it is needed in the reduce operation so that it can be processed without the need of storing it on the local machine. The only part that has not been changed is the mapper, that is processing the data and storing it locally like before.

Flight defines a set of rpc methods that has to be implemented by its services. In this case, what is required to do by the program is: uploading/downloading data, retrieving information about the available data streams and perform application-specific methods. The program behaviour can be divided in two parts: the communication of the shuffler with the servers to send the data and the communication of the reducer with the servers to get the data. The driver is the handler of the calls, the part of the program that gets the information from the server to where the data should be sent or retrieved and how to do it, and then runs all the MapReduce steps with those informations as well.

The service part of Arrow Flight is organized in a way that can better handle the data, and it is composed by:

- **Metadata Server**: the metadata server is responsible for holding the information about the data servers, such as which datasets are stored in this service and how they are partitioned among the data servers, but does not have the data. It is the server communicating with the client that will redirect it to the right data server for the desired operation. The only data that the metadata server has is the metadata relative to the data servers under its "domain". In this case, the metadata server is running all the data servers once it is started, so that it knows for sure where the servers are, while in a real life scenario the data servers could be running separately from it, and it should just know where the servers are and how to communicate with them.

- **Data Server**: there can be one or more data servers and they are responsible for storing the data relative to one or more partitions of the dataset. A data server is identified by the location where it is running (address and port) and it is designed to handle data storage and retrieval operations and ensure data integrity and availability. In the context of our application, the data server is responsible for storing and serving the data resulting from the shuffle operation.

The client side of the program is being handled by a driver, which is responsible for communicating with the Arrow Flight service and run the mapper, shuffler and reducer instances with the proper requirements. The mapper is the first step of this program that runs, and it is working locally, ideally with one mapper process per input file. Like before, it processes the input data and stores locally the results, in Arrow IPC format.

Now we will focus on the shuffler and reducer parts, with attention on how the communication with the servers is happening and what all the related parties are doing to make this possible.

3.3.1 Driver-Shuffler communication with the servers

In Figure 3.4 is represented a sequence diagram for the communication of the driver and the shufflers with the servers, where the data servers and shufflers are indicated with 1 and N meaning that there could be one or more of each. The communication starts with the driver connecting to the metadata server and calling the `do_action` method with the `put_shuffle` as action, telling the metadata server to set up the data servers to handle the data that will be sent, with the number of output partitions as the body request (Listing 3.1 lines 1-2). Upon receiving this request, the metadata server decides how to divide the partitions based on the data servers it has linked (for example, if there are 5 data servers and the driver wants 10 output partitions the metadata server decides to make the data servers handle 2 partitions each). It then returns the list of endpoints where each endpoint, which is relative to one partition, contains the location where the data server handling that partition can be found, and the descriptor that will be used by the shuffler to store the data (Listing 3.1 lines 4-16). The driver saves the list of endpoints which will be used later.

The first step that is being performed is the map, which is done locally. The driver instantiate the desired number of mapper processes and waits for them to finish writing the mapped files. Like before, the mapper is mapping the record values to be 1 if *Robot* or 2 if *User* and is storing the data in record batches of a given size, deciding here the schema that will be used for all record batches.

After the map is done, the driver creates the shuffler instances with also the list of endpoints received from the metadata server. Each shuffler now is acting as a

client that wants to upload data in the right data server. First of all, the shuffler opens a connection with every data server in the list of endpoints. Then, the shuffler calls the `do_put` method for each connection, with the upload descriptor that is basically the descriptor received from the metadata server, for that particular endpoint, with appended the name of the file that the shuffler is processing. This is a contract stipulated between the client and the server and is therefore known by both of them, it is a design choice that does not impact on the behaviour of the system. For each connection, the shuffler is receiving a writer that can be used to stream data to the endpoint. The shuffler then iterates through every record batch in the input mapped file, gets the list of partition indices for every record inside the record batch by applying the hashing function and streams one record batch per partition to the endpoint handling that partition. Every time a data server is receiving a stream containing a record batch, it is immediately storing it, which results in each data server having a number of files equal to the partitions that they can handle multiplied by the number of input files.

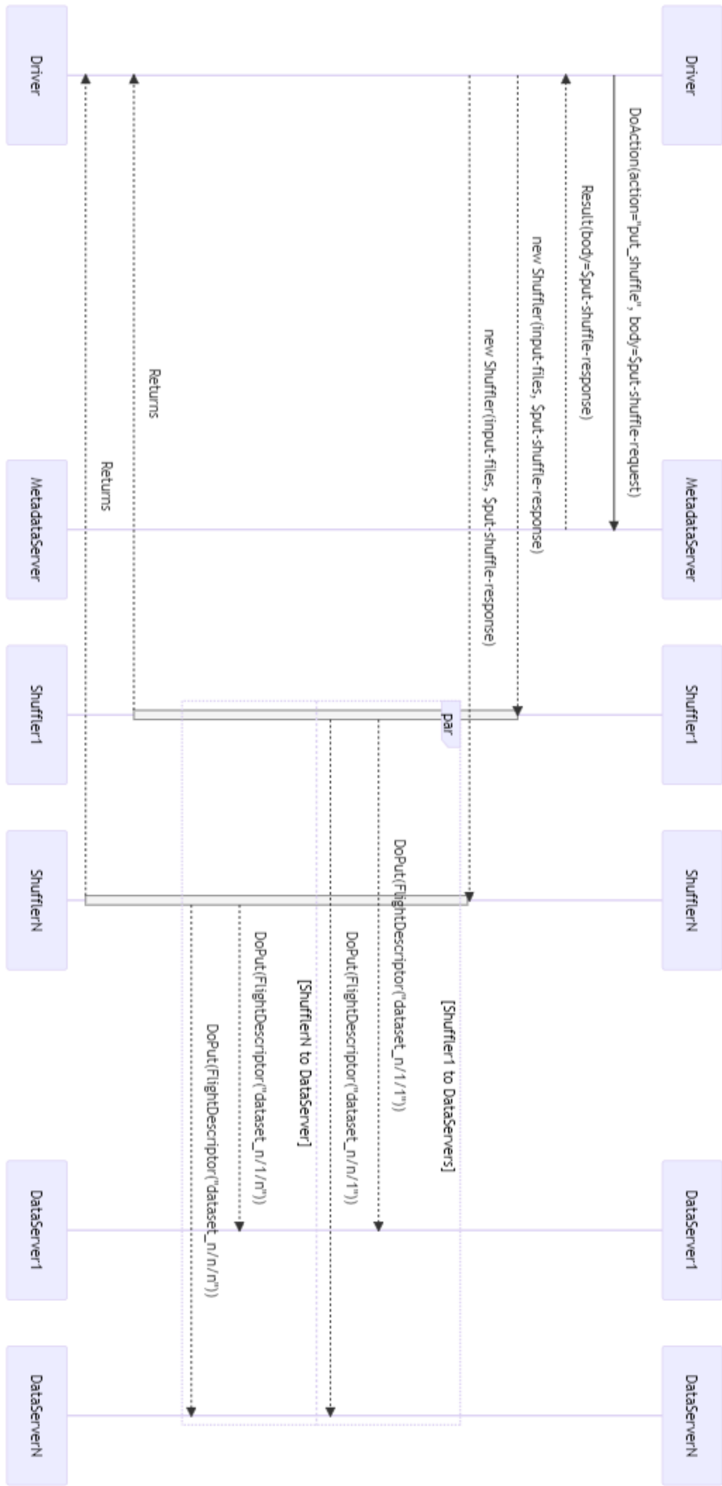

Figure 3.4: Sequence diagram of the driver-shufflers communication with the servers

Listing 3.1: View of the packets sent and received from the driver, shufflers and servers

```
put-shuffle-request
{id: "dataset_n", Reducers: n}

put-shuffle-response
{
    "partitions": [
        {
            "location": "grpc+tcp://dataserver1:5678",
            "descriptor": "dataset_n/1"
        },
        {
            "location": "grpc+tcp://dataserverN:5678",
            "descriptor": "dataset_n/n"
        }
    ]
}
```

3.3.2 Driver-Reducer communication with the servers

When all of the shuffler instances finish their job, the driver asks the metadata server for the information about the endpoints that are now containing the data resulting from the shuffle operation, and how to retrieve that data. The metadata server answers with the `flight-info` (Listing 3.2 lines 1-17) containing the list of endpoints, where each endpoint is relative to a single partition and contains the ticket that is required by the data server to identify the data relative to that partition and a list of locations where the data can be found (since there could be multiple data servers having the same data to avoid problems in case of one server failing to answer). In this case there is only one location for each partition.

With the list of endpoints received from the metadata server, the driver can now create the reducer instances, where each one is relative to a single partition to be reduced and is therefore created with the information on the endpoint relative to that partition.

Each reducer represents a client connecting to the endpoint's location where the data server containing the partition of interest can be found. Once connected to the data server, the reducer calls the `do_get` method with the ticket relative to that endpoint (Listing 3.2 lines 19-23), where the ticket is a value opaque to the client but is understandable by the server that will receive it. The data server gather all the files relative to the partition asked for and creates a stream of data composed by the concatenation of those files, which is sent to the reducer. The reducer is then reading one chunk at a time and is performing the same operations like before, grouping the records by keys for each chunk, summing up the values

relative to each key partially and then grouping all the records for the final sum. Then, the reducer stores the result in a csv file, which will contain the records with the key relative to that partition and the summed value of the clicks.

The sequence diagram of the communication between the driver, the reducers and the service is in the following Figure 3.5.

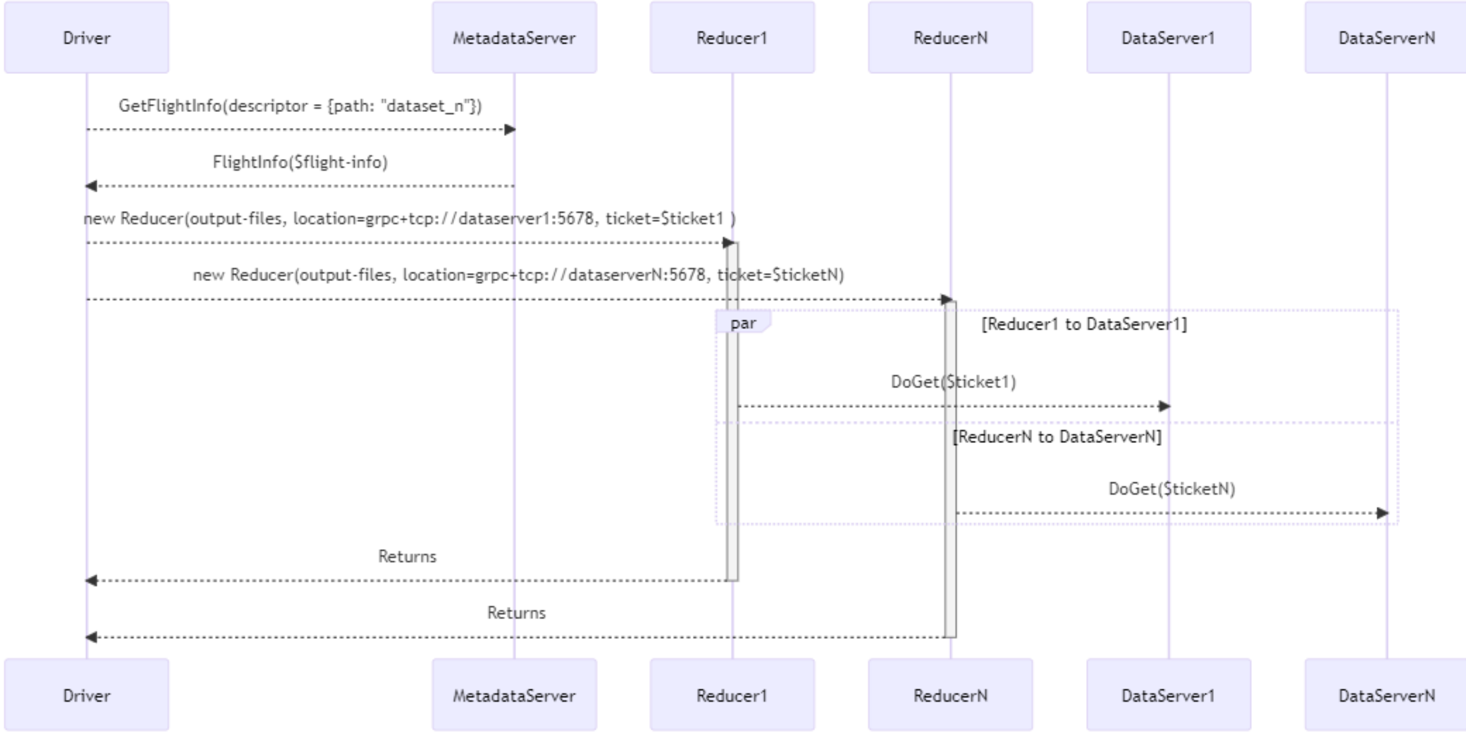

Figure 3.5: Sequence diagram of the driver-reducers communication with the servers

Listing 3.2: View of the packets sent and received from the driver, reducers and servers

```
flight-info
{
    "endpoints": [
        {
            "ticket": "dataset_n/1",
            "locations": [
                "grpc+tcp://dataserver1:5678"
            ]
        },
        {
            "ticket": "dataset_n/n",
            "locations": [
                "grpc+tcp://dataserverN:5678"
            ]
        }
    ]}
}

ticket1
"dataset_n/1"

ticketN
"dataset_n/n"
```

Chapter 4

Experimental evaluation

In this section, we aim to assess the effectiveness and performance of the proposed methodology through a comprehensive analysis of two distinct tests conducted for each version of the prototype. The two tests represents two opposites situations that may happen, to have the most extensive comparison in the best and worst scenario: having a very limited number of keys versus having one distinct key per row, which means that there can be millions. Each tests is executed with a different number of output partitions: 5, 10, 20, 40. By evaluating this, we can gain valuable insights into the strengths and limitations of each version.

The machine used for those tests is composed by: CPU with 6 cores and 4.10 GHz base speed, 16 GB RAM at 2400 MHz. As a baseline for the executions, one can check its running time on the first row of the first test and see how the time changes on a different machine.

In the first case, the MapReduce program implemented in Python serves as the baseline for comparison. The tests have been designed to evaluate its performance under varying complexities. These tests will provide insights into the program's capacity on handling data and its performances.

Building upon the initial implementation, in the second version that extends the program using the Arrow framework, the evaluation will focus on measuring the impact of incorporating Arrow in terms of data processing speeds, reduced memory usage and overall performance gains.

The third and final version that extends the program further by incorporating Arrow Flight, analyzes its impact on the overall performance, by showing how the time changes when distributing some steps.

By conducting thorough evaluations on each case, this research aims to provide insights into the effectiveness of using Arrow and Arrow Flight to extend the MapReduce program implemented in Python. The results obtained from these evaluations will contribute to understanding the performance characteristics, scalability, and efficiency of the proposed extensions, thereby aiding in the identification

of potential areas for further optimization and improvement.

4.1 MapReduce implementation in Python

In this section, the performance of the very first version is analyzed, which consists of looking into the speed of the application on the different scenarios and how the data is distributed among the partitions. As mentioned in section 3.1, to simulate the behaviour of a distributed application, in every step each instance is being run on a different Python process. The number of processes is dependent on the step: it is equal to the number of input files in the map and shuffle steps, while in the reduce step it is equal to the number of output partitions. That is because, each mapper and shuffler are working on a single file while the reducer is working on each file relative to the partition it is interested in. In the following tests, since the program is creating multiple processes, the CPU is working at 100%, which may create some noise for the execution time. This is not relevant in this phase, but will be considered in the next phases.

The first test is executed with 10 input files, 10 unique keys (one per file in this case), 3 million rows per file, for a total of 30 million, and a number of partitions starting from 5 and doubling until 40. The parameters are:

- number of input partitions;
- number of elements per output partition;
- execution time (mean and standard deviation);
- reducer time.

The execution time is already composed by the reducer time, but the latter is also reported so that it is possible to see how the number of output partitions impacts on the parallelization of the reducer. For each number of output partition (or each case of the test), the program has been executed 5 times. The results are reported in table 4.1 and prove how increasing the number of partitions can be beneficial but can also increase the execution time (and memory consumption). First of all, with a low number of partitions, the keys are not evenly distributed, while increasing it they are, confirming the fact that more partitions tend to even the distributions out. Second, here the best result is obtained with 10 output partitions. Therefore, parallelizing more the execution is not optimal: this is due to the fact that having a single CPU more processes running together mean less computational power available. Moreover, all the processes have to be created and managed, requiring time and memory consumption. In a distributed situation, more partitions could behave better, but the problem of having to manage more nodes remains.

N partitions	N elements per partition	Execution time Mean (s)	Execution time Std dev (s)	Reducer time (s)
5	[3,1,2,2,2]	24.571	0.327	5.631
10	[1,0,0,0,2,2,1,2,2,0]	20.720	0.460	3.554
20	max 1 per partition	22.398	0.680	4.280
40	max 1 per partition	24.490	0.728	5.742

Table 4.1: MapReduce performance with 10 unique keys

The second test is executed with 5 input files containing 3 million rows, each one with a different key, for a total of 15 million unique keys. This is to see that by increasing the number of keys, the keys among the partitions will be evenly distributed even with few partitions, which is why there is a different parameter that just checks if the output partitions are uniformly distributed or not. The results are in table 4.2

In this case, the lower amount of partitions appears to have the best execution time, but the best reducer time is again obtained with 10 partitions. Increasing after 10 partitions the time is worsening like before. The reason for the execution time being better with 5 partitions can be related to the behaviour of the shuffler: it uses the same number of processes (that is the number of input files) every time, but the number of files that it has to process is increasing with the number of output partitions. That is because, for each input file, the shuffler has to write `n partitions` number of temporary files for the reducer. So, with 5 partitions the shuffler writes 50 files, while with 10 it has to write 100 files, which slows down the execution time. Moreover, the reducer time is much higher with respect to the previous test, even if this test has half the rows. This is due to the fact that all the keys are unique, which requires more computation time and memory occupation to store all the values for each key, while before it was much faster to just increase the counter for the same key.

N partitions	Is uniform	Execution time Mean (s)	Execution time Std dev (s)	Reducer time (s)
5	y	20.481	0.500	6.373
10	y	20.838	0.281	6.073
20	y	22.323	0.240	6.606
40	y	25.608	1.109	8.455

Table 4.2: MapReduce performance with 15 million unique keys

The tests here are to give a general view of the performance of the program.

In a real scenario, the steps may be running on different machines and the best parameters may change, but the main idea remains the same: increasing the number of partitions is improving the performances with some limitations and until a certain point, therefore it is very important to look into the optimal number of partitions for a MapReduce program.

4.2 Arrow local

Since the program is simulating a distributed environment, multiple Python processes are running simultaneously for every step in the MapReduce application. This is limiting the performance evaluation because, having a single local machine without much computing power, those processes are consuming all the CPU when running, which leads to some noise when evaluating the execution times. In order to have more accurate assessments, a single process will be used for every step and the partial execution times will be saved. The program is now being run 100 times and then the execution times are averaged, so that to obtain the execution time of each step (and the total), the time taken by each step will be divided by the number of processes that it should be divided into. Therefore, for example, if the mapper would use a number of processes equal to the number of input partitions, which are 10, the time taken by the mapper with a single process is divided by 10.

In all the following tables, the mapper time is reported even if it should not change in the different cases, since it is relative to the input files only, parameter that is not changing in the same test. This is for the sake of completeness and clarity when showing also the total time, and also to verify that the assumption is correct.

The first test is executed with 10 input files, with 10 total unique keys. The number of total records is 15 million, with 1.5 million per file. The results are in Table 4.3 and show how the best execution time is with 10 output partitions like before, but this time it is very close to the time with 20 output partitions. This is because with 10 partitions the shufflers have less files to create and write into, but the reducer has more work to do having to process more data per partition, while by increasing the number of partitions the shuffler requires more time to create more partial files (since it creates `n partitions` files per input file), but the reducer is significantly faster due to the fact that it has to process less data. Also, in this case, having less keys than partitions, some instances of the reducers would have no data to process.

N partitions	Map time (s)	Shuffle time (s)	Reduce time (s)	Total time (s)
5	1.133	3.026	2.444	6.604
10	1.162	3.329	1.249	5.741
20	1.171	3.962	0.652	5.785
40	1.153	4.962	0.324	6.438

Table 4.3: Performance with 10 unique keys using Arrow

In the second test, with the same number of records but one unique key per record (which makes it 15 million unique keys in total), things change. The results in Table 4.4 prove that the number of partitions in a scenario more plausible (with lots of unique keys) is different that what was found before. In this case, the best

result is the one with 20 output partitions, while 40 partitions is still doing better than 10.

Here, every reducer instance is working at the same capacity since the keys are many and the partitions end up being uniformly distributed, which means that increasing the number of partitions leads to the reducers having to process less data. This is optimal until a point where the time taken by the shuffle is increasing more than how the time taken by the reducer is decreasing.

As expected, the reducer time is halving when the number of partitions is doubling, which confirms the fact that the keys are uniformly spread meaning that the number of records to be processed by the reducer is decreasing proportionally to the increase in the number of partitions.

The shuffler time however is increasing much more, which could be related to how the number of partitions impacts on its performance: 10 partitions means 10 files written per shuffler, 20 partitions means 20 files written and so on. Having the same number of input files, the shuffler still has to process the same number of records, but it has to create and write into more temporary files as output. For what can be deduced by the results in Table 4.4, the shuffler time increases more when more files have to be written, and at an higher rate of how the reducer time decreases.

N partitions	Map time (s)	Shuffle time (s)	Reduce time (s)	Total time (s)
5	1.549	3.321	6.615	11.485
10	1.581	3.806	3.347	8.734
20	1.548	4.450	1.687	7.685
40	1.530	5.791	0.896	8.216

Table 4.4: Performance with 15 million unique keys using Arrow

In Figure 4.1 is shown all the times at the different steps, per iteration, before being averaged (with 20 output partitions, that is the best case). This is to prove that the time is approximately similar in all iterations with some small variance, so that the considerations made before are valid.

When taking into accounts both tests and their performances, it can be seen in Figure 4.2 how the best scenario is clearly performing better, since as said having less keys means less effort mostly for the reducer but also for the shuffler and mapper. In the figure, the times for each step are compared, at the variation of the number of partitions. This comparison highlights the biggest difference in time between the two tests, which lies in the reducer time, that is the one more influenced by the number of output partitions and keys. The shuffler and mapper times are very similar, since the number of input files and records is the same. The shuffler takes more time when there are more unique keys since it has to process more separate records and write them in different files.

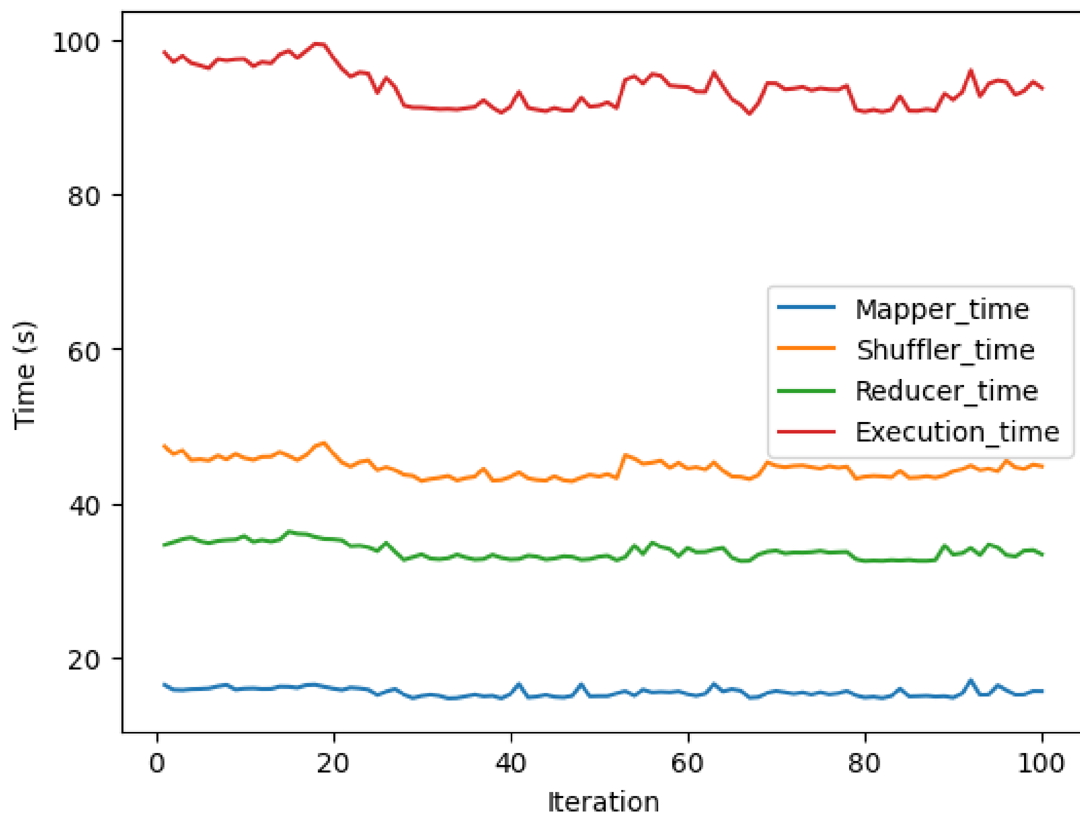

Figure 4.1: Time taken for each step in 100 iterations, with 20 partitions

The reducer time is incredibly slow with few output partitions when lots of unique keys are present, while it starts off already fast with few keys. This is relative to how the reducer needs to store all the key-value pairs and compute the desired reduce function for each of them, which leads to poor performances when having few reducers to compute huge amounts of data, doing even worse when increasing the number of keys to process, since more distinct keys mean more key-value pairs to store in memory and more computations to perform. Even if in this version the reducer has improved the memory consumption by using Arrow's data structures, there still is a Python list summing up the values, which is bigger with more unique keys, which also means that there are more separate computations to be done thus increasing the time required.

The results in Figure 4.2 also shows how the reducer time is decreasing a lot with the increase of the number of partitions, even faster in the case of 15 million keys, which is coherent to the statement made before about the time halving when doubling the number of partitions. Therefore, even with a huge amount of distinct keys, by increasing the number of partitions, therefore increasing the number of

reducers, the reducer reaches a performance very close to the best scenario.

The shuffler time however is increasing more in the worst scenario, which can be due to the fact that more partitions gives more effort to the shuffler, having it more files to write, and that effort is increasing even more if there are much more keys to be written. Having more keys to be written would mean that, for every record batch that is being read, the shuffler has to write in every output file, since it is highly improbable that one partition is not being considered. However, with a small number of distinct keys, when reading a record batch, it can happen that the keys have to be written in just some partitions, which means that the shuffler would have to write to a small number of output file at that iteration, making it faster. For these reasons, it is very important to consider the trade-off between these two performances in a real case.

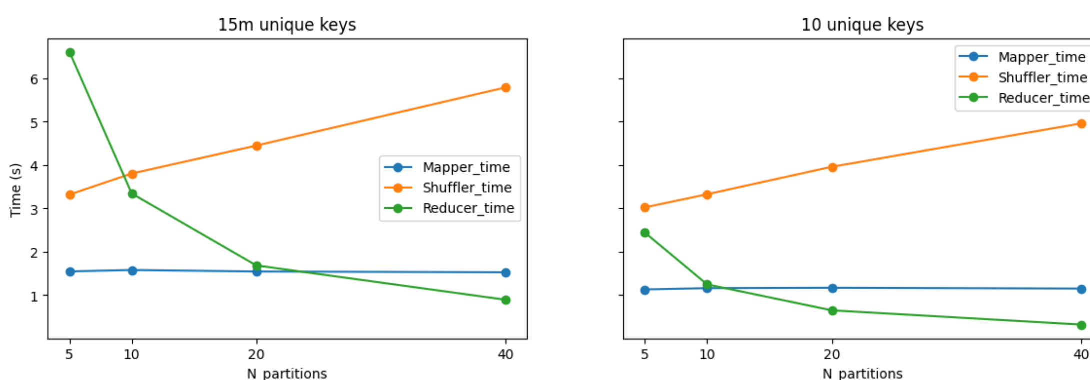

Figure 4.2: Time taken for each step, per partition, in the two tests

Finally, the total time taken for each partition in the two tests is shown in Figure 4.3. Here the difference between those two scenarios is much more evident, and it also shows how the number of partitions affects differently the performances on a different number of unique keys.

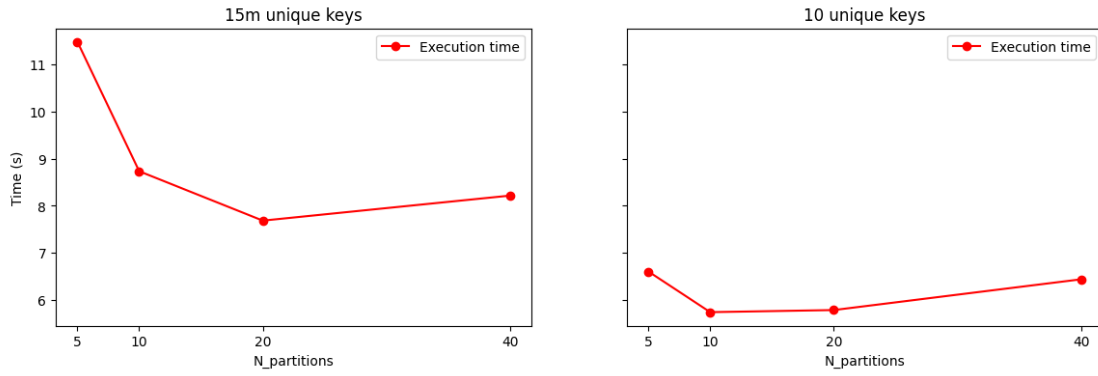

Figure 4.3: Total time taken per partition in the two tests

4.3 Arrow distributed

In this section we look at the results obtained by the final, and most important, version of the prototype. Here the difference from the previous version is in the distribution of the shuffle, so that the temporary files stored by the shuffle step are now being stored in a cluster of data servers separate from the local machine, where the reducer is reading from. However, in this tests the data servers are still being created on the same machine, in order to showcase the behaviour of the application and see how adding additional steps to communicate between different services, simulating a client-server behaviour, impacts on the performances of the application. For a complete testing it would be necessary to have access to a different machine or a cloud, where the data servers could then be created so that there is actual separation between the client and servers, which would probably slow down the execution times by having to take into account the speed of the net as well. Since it is not as important as showing the potentiality of this prototype, it is not done.

The evaluation performed here is still done on two tests, each of them with 4 cases varying the number of partitions, and being executed 100 times per case. The results are averaged on the number of processes that should be running in parallel like before. The focus of this section will be more on how the speed of the shuffler and the reducer changes when adding the additional steps to distribute the data in between.

The record batch size is specified in the mapper to be 8196, which means that each batch is composed by 8196 rows or records. The total size in bytes of each record batch is composed by: 8196 * (size of int32) + 8196 * (size of string) + (size of metadata). In the test with all distinct keys, it is 213096 bytes, which would be different in the other case depending on the length of the keys. Since in the shuffle phase the record batches sent out are not of the specified batch size, but are smaller due to the shuffler not wanting to store all the data from different record batches (it process one record batch at a time and writes it out in the temporary files splitting the keys in the different partitions), their size would be the initial size divided by the number of output partitions. That size is how much data is being transferred between the shufflers and the data servers, and read from the data servers to the reducers, at each time, because those instances are processing one record batch at a time.

In Table 4.5 the performance for the test with 10 unique keys is showed, and the results are not much worse than before. This can be related to the data servers being on the same address as the clients, but it gives a good idea of how adding a distributed step increases the performance times. Arrow Flight is doing an excellent work in not worsening much the performance by adding little overhead and complexity to implement a distributed step that is fundamental.

The best number of partitions is 20, which is different from the local version in this case, even if not by much. This proves one more time that 20 partitions is the best number in this scenario.

N partitions	Map time (s)	Shuffle time (s)	Reduce time (s)	Total time (s)
5	1.185	3.141	2.827	7.154
10	1.158	3.326	1.494	5.979
20	1.192	3.929	0.692	5.814
40	1.175	4.964	0.340	6.480

Table 4.5: Distributed performance with 10 unique keys using Arrow Flight RPC

Table 4.6 shows the results for the test with 15 million unique keys using Arrow Flight, and the results are promising like before. The best number of partitions is still 20, with 40 being better than 10 in this case, since the reducers are doing much less work compared to the shufflers.

N partitions	Map time (s)	Shuffle time (s)	Reduce time (s)	Total time (s)
5	1.513	3.371	6.931	11.816
10	1.536	3.686	3.540	8.762
20	1.519	4.418	1.786	7.723
40	1.534	6.002	0.954	8.490

Table 4.6: Distributed performance with 15 million unique keys using Arrow Flight RPC

In order to better visualize the difference in the two tests, for each step, Figure 4.4 shows how the speed changes in each step of the two tests, when the number of partitions is increasing. There is again a clear difference between the reduce time in the two cases when the number of partitions is low, but that difference is decreasing when using more output partitions. The mapper time is approximately similar as expected, and the shuffler time is also similar, but is increasing more in the test with 15 million unique keys when enlarging the number of partitions. This is related to the number of keys, since with 10 keys, when using more partitions than 10, some of the partitions will be empty and at most 10 partitions are being used, therefore limiting the increase in time. With much more keys, however, when increasing the number of partitions, all of the partitions are being used, therefore proportionally augmenting the time required to perform the shuffle operation.

Since in reality we do not expect to have a small number of unique keys, the expected behaviour of this application in a real scenario is more coherent with the second test, where it is more important to consider the increase in the shuffle time.

Figure 4.5 shows the difference in the two tests for the total execution time, so that it is easier to understand the performances. The second test is, as expected,

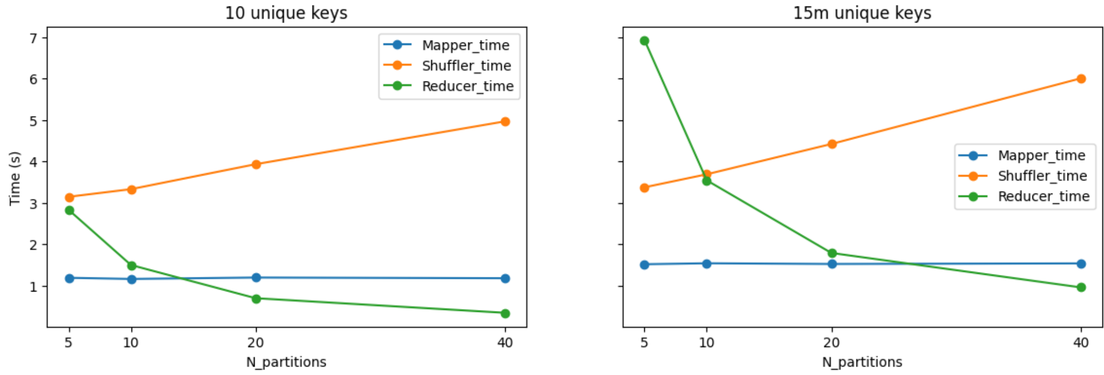

Figure 4.4: Time taken for each step in the two distributed tests

slower than the first, which is related to the number of input keys, but there is a bigger time decrease as the number of partitions increases, to a point where the direction of the slope changes, indicating the best possible parameter.

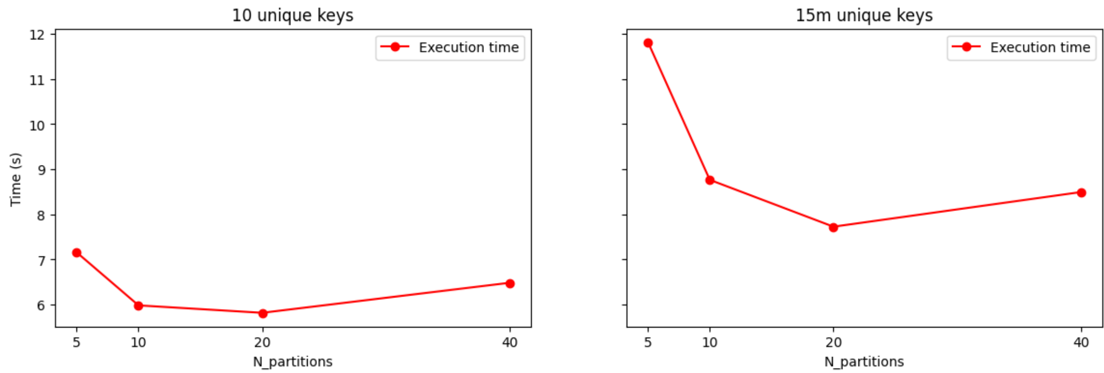

Figure 4.5: Total time taken for each step in the two distributed tests

Finally, we can compare the results in the two tests for the different versions using Arrow: the local and the distributed ones. Table 4.7 contains the total execution times in the two versions when increasing the number of partitions, with 10 unique keys. Here it is possible to notice how the difference, when using the optimal numbers, is not big, indicating how using Arrow Flight to distribute the steps is not as impactful as one may think: the complexity added by this framework is minimal.

Table 4.8 shows the difference in execution time for the two version, when testing with 15 million unique keys. Here the difference is again very small, not like before but again to a level that proves how Arrow Flight is an incredible framework. The best result here is clearly obtained with 20 partitions in both versions, and the

N partitions	Local total time (s)	Distributed total time (s)
5	6.604	7.154
10	5.741	5.979
20	5.785	5.814
40	6.438	6.480

Table 4.7: Performance comparison between the local and distributed versions with 10 unique keys

difference is very small with 10 or 20 partitions.

N partitions	Local total time (s)	Distributed total time (s)
5	11.485	11.816
10	8.734	8.762
20	7.685	7.723
40	8.216	8.490

Table 4.8: Performance comparison between the local and distributed versions with 15 million unique keys

Finally, a different parameter is tested: `N_input_files`, which is the number of input files of the application, also known as number of input partitions that the dataset has been split in. This parameter determines the performance of the mapper and shuffler, since for those steps one instance is created per input file. Therefore, having more input partitions would increase the number of mappers and shufflers running, which may improve the overall speed of the program. However, increasing the number of instances running also impacts on the memory and the computing power being consumed at the same time, which is an important factor to consider.

For this scenario, the same two different tests are being run (with 10 unique keys or 15 million) and the input data size is the same as before (15 million records). This means that if the data is partitioned in 5 partitions, there are 15.000.000 / 5 records per partition, so that by increasing the number of input partitions the data per partition is less. The values being tested for this parameter are: 5, 10, 20.

Figure 4.6 shows the results for executing the two tests in this case. Here it can be noticed how, by increasing the number of input partitions, the speed improves incredibly. The reducer time is still the same (since it depends on the number of output partitions), while the mapper and shuffler time are improving. Unfortunately, this improvement comes with some costs: in order to increase the parallelization of the program, the number of machines where the steps have to run is increasing as well, that could each run one or more instances of the steps,

depending on the size of the dataset. However, if the dataset is big, to parallelize properly we would need many distributed machines, where each machine has a considerable cost. Also, more instances mean more overhead and time to run and coordinate them. Therefore, it is important to balance the costs and benefits in this choice, finding the right amount of input partitions. Being the shuffler the most time consuming operation, the figure shows that increasing the input partitioning improves its time, but the improvement is fading the more partitioning is done. In this case, 10 input partitions could be considered a good option, since the improvement when using 20 is good but it would require 2 times more the number of processes and therefore more costs.

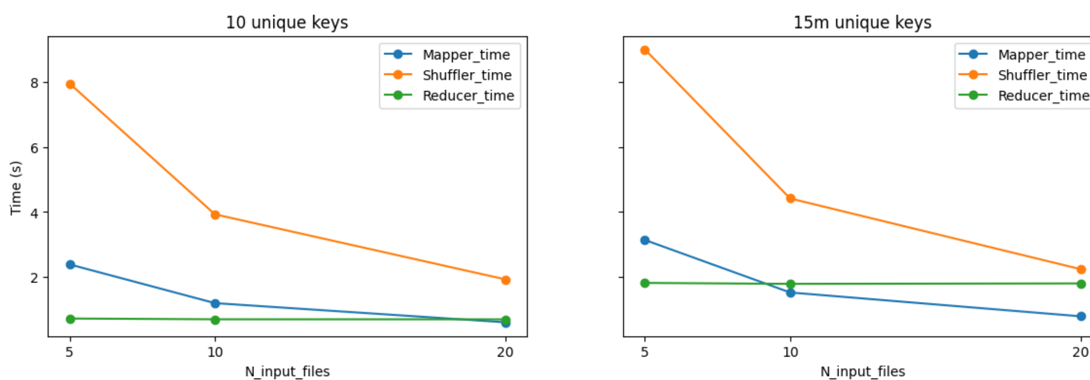

Figure 4.6: Time taken for each step varying the number of input files in the distributed version

Chapter 5

Conclusions

While the primary objective of achieving a fully distributed shuffle step in DataFusion using Arrow Flight RPC was not accomplished within the scope of this thesis work, the prototype implementation provides a solid foundation for further exploration and development. The findings from this work contribute to the understanding of leveraging Arrow Flight RPC for distributed shuffle operations and pave the way for future research and optimization in the field of distributed data processing.

As for future work, there are several promising directions to pursue. One potential avenue is integrating the distributed shuffle step prototype within DataFusion itself. By incorporating the prototype into the core functionality of DataFusion, users can leverage the benefits of distributed shuffling seamlessly within their data processing pipelines.

Moreover, conducting extensive scalability testing with larger datasets and distributed clusters would provide a clearer understanding of the system's behavior under various workloads and deployment scenarios. Evaluating the prototype's performance on real-world use cases and benchmarking against other distributed data processing frameworks can further validate its effectiveness and competitiveness.

In conclusion, this thesis work laid the groundwork for a distributed shuffle step in DataFusion using Arrow Flight RPC. The prototype implementation showcased the potential benefits of leveraging Arrow Flight RPC for distributed shuffling and offered insights into its performance characteristics. By integrating the prototype within DataFusion, as well as exploring advanced optimization techniques and conducting scalability testing, future research can continue to enhance the capabilities and performance of the distributed shuffle step, making significant contributions to the field of distributed data processing.

Bibliography

[1] URL: https://blog.devgenius.io/apache-arrow-2d72137d9e84 (cit. on p. 5).

[2] Dejan Simic. «Apache Arrow: Read DataFrame with zero memory». In: (). URL: https://towardsdatascience.com/apache-arrow-read-dataframe-with-zero-memory-69634092b1a (cit. on p. 6).

[3] URL: https://blog.djnavarro.net/posts/2022-05-25_arrays-and-tables-in-arrow/ (cit. on p. 7).

[4] URL: https://grpc.io (cit. on p. 8).

[5] URL: https://protobuf.dev (cit. on p. 8).

[6] URL: https://arrow.apache.org/docs/format/Flight.html (cit. on pp. 9–11).