SpArrow: a Spark-Arrow Engine

Leveraging the Arrow in-memory columnar format to increase Spark efficiency in RDD computations

by

Federico Fiorini

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday November 24, 2021 at 11:00.

Student number:	4743105	
Project duration:	December 1, 2020 – November 30, 2021	
Thesis committee:	Dr. ir. J. A. Pouwelse,	TU Delft, chair
	Dr. J. S. Rellermeyer,	TU Delft, supervisor
	Dr. ir. Z. Al-Ars,	TU Delft

TU Delft

Abstract

The ever-increasing amount of data being generated worldwide, combined with the business advantages for companies in quickly and efficiently processing such data, have accelerated the research and development into big data analytics. Existing solutions for storing and computing data can no longer give the required processing performance, and technical advancements in I/O operations and networking have further increased the distance between the requirements and the provided solutions. A number of changes, such as the introduction of new memory representations, have been proposed to reduce the overhead of existing data analytics frameworks, but they still have not been fully integrated with such frameworks for a lack of market traction.
Apache Spark is among the most widely used frameworks for big data analytics, as it provides, among other functionalities, an extensive and easy-to-use API that allows to perform computations on a multitude of processing workloads. At the core of Spark sits the Resilient Distributed Dataset, an abstraction upon which other abstraction or workload-specific libraries have been built over the years. While the general development efforts focused on creating new abstractions on top of the core library, not enough attention has been put into solving intrinsic issues of the RDD's.
Thereby, this work proposes an integration between Apache Spark and the Apache Arrow in-memory data format, that can be leveraged to solve memory- and computational bottlenecks. Such integration is beneficial for a number of reasons, such as the reduction of serialization and storage overheads, thanks to an efficient columnar-oriented data format shared among several machines in a cluster. This work proves the feasibility of such approach, introducing a number of changes in Spark and Arrow that result in a proof of concept implementation. Furthermore, it proves that such integration can be performed without the need for expensive and disruptive changes to the existing API's, thus allowing existing workloads to be fully compatible with new, Arrow-based techniques. The performance advantages have been evaluated, resulting in an execution time speedup of approximately 14%, with the biggest improvement being 50% reduction in execution time for wide transformations. In addition, it allows for functionalities typically executed within Spark to be offloaded to Arrow, introducing further performance improvements, with a 20% reduction in execution time for offloaded functionalities, compared to pure Spark.

Preface

This thesis marks the end of my Master's in Embedded Systems and my journey at the Delft University of Technology.
Coming to the Netherlands has probably been the best decision I have ever taken in my life so far, as I had the opportunity to explore a different country, to get in touch with and embrace different cultures in an international environment, and to meet amazing people that I will forever remember, many of which have become over the years really close friends. Almost like a family abroad, and they really made this journey so far unbelievably great. I also had the opportunity to have different work experiences, such as the Formula Student Team Delft and an internship at Microsoft Netherlands, that contributed to my technical and, more importantly, personal growth. I will forever look at the achievements during these experiences with enormous pride, such as the first place overall at Formula Student Netherlands in Assen in 2018.
I am, in a way, grateful for the lows I have got to experience throughout this journey too, as they really helped me in getting to know myself and maturing.
I would particularly like to thank my supervisor, Dr. Jan Rellermeyer, for his guidance and support during this last chapter at the university. This thesis started long ago and was disturbed by a global pandemic and personal struggles, but Jan's supervision has always been a lighthouse for this work, with tips and tricks that helped shaping the goal of this thesis and the final implementation. I would also like to thank Dr. Zaid Al-Ars for his guidance as (unofficial?) co-supervisor, and for constantly motivating me during the good and, most importantly, the bad moments I have experienced.
Furthermore, I would like to personally thank each one of my closest friends here: you are the most important people I have here, and I will forever be grateful for all the good memories we shared, as well as your support when things did not really go well for me. Michele, Irene, Isabella, Samuele, Tullio, Benedetta and Martina: thank you from the bottom of my heart.
I would also personally thank each one of my friends back in my hometown. They have always been here for me before coming to the Netherlands, and I know for certain they will always be alongside me wherever I will be, and that is something to be grateful for. We haven't seen each other as frequently as we used to be, but every time I come back it feels as if I never left. I cannot name all of you unfortunately, otherwise I will need an extra chapter on this report. Do not worry, I will personally thank each of you the next time I come back, and I will forever thank you for being friends.
Last but most definitely not least, I would like to thank my parents for their unconditional support and love throughout these years. They were always with me while I was facing seemingly unsolvable challenges, and they were always supportive for each of the decisions I made during my time in the Netherlands. I am who I am thanks to you to some extent, as you allowed me to have such experiences.
I would like to dedicate this work to you.

Federico Fiorini
Rotterdam, November 2021

Contents

List of Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
FPGA	Field Programmable Gate Array
GC	Garbage Collector
GPU	Graphics Processing Unit
IPC	InterProcess Communication
I/O	Input/Output
JNI	Java Native Interface
JVM	Java Virtual Machine
KV	Key-Value
PoC	Proof of Concept
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
SIMD	Single Instruction Multi Data
SSD	Solid State Drive
SQL	Structured Query Language

List of Figures

1 Introduction

1.1. Context

In recent years, there has been a rapid growth in the volume of data being created daily through various Internet-connected sources. The boom of social media and the advent of Internet of Things (IoT) devices have dramatically contributed to such growth. Many companies are now using this amount of data, coming in various forms, to make intelligent, data-driven business decisions [20]. Such data can come in different formats and it can be divided into two main categories, *structured* (like traditional database values) and *unstructured* (such as social media feeds, pictures and audio). The sheer volume of the data required for analysis, combined with the amount of unstructured data available, makes traditional database systems unable to cope with the processing and analytics requests of today. Together, the volume of data and the processing speed required for its analysis, served as the main reason behind the introduction of *Big Data Analytics* frameworks.

Apache Spark is a unified analytics engine for large-scale data processing dating back to 2009, when it was started as a UC Berkeley research project. After its open-source release a year later, Spark's development community grew over the years, and moved to the Apache Software Foundation in 2013. Over the years, Spark has become one of the most active projects within the Hadoop ecosystem, maintained by a community of more than hundreds of developers, as well as one of the most popular frameworks for big data analytics [12, 35]. Nowadays, a number of companies, including Amazon, Autodesk, eBay and the NASA Jet Propulsion Laboratory use Spark for their data processing workloads or research [34].

Its core component manages the memory, scheduling and the cluster component where Spark processing jobs are executed, and it introduces the main abstraction used to handle collections of generic data, the Resilient Distributed Dataset (RDD in short). It is at the core of Spark, as all other abstractions and libraries are directly using RDDs for processing, or have their own collections built on top of them. Besides the core component, Apache Spark contains several other modules to support different analytics use-cases, such as Machine Learning processing, data streaming and real-time processing, and structured data processing (with the SQL library) [8]. Spark's popularity is boosted by its extended support for multiple programming languages, easy-to-use API's and speed in executing big data processing workloads. Its compatibility with other tools within the Hadoop ecosystem, together with fault-tolerance and lazy evaluation further cement Spark's market position among available analytics frameworks [36, 38].

Given the increasing amount of data and the subsequent need for storage and in-memory processing of such data, new memory representations have risen, to tackle problems related to the aforementioned needs. Typically, structured data is stored in SQL-based fashion, where data is represented in a row-oriented way. Each data item is thus represented as a row (sometimes called "record"), where each column represents a particular attribute of such item. Such memory representation provides ease of use, and it has been the common choice for databases over the years. Data can be stored in multiple formats too, with JSON and CSV being commonly used to store data in a row-oriented way. Within the Hadoop ecosystem, Apache AVRO is also used to store data efficiently in a row-oriented fashion [9].

Row-based data representations, however, come with some disadvantages: while it is easy to write a row to a database, reading and seek operations could introduce unwanted overhead. This becomes evident when performing data processing in a row-based database, where the induced latency for aggregating records has a

big impact on the performance. Furthermore, storing row-oriented data could pose a risk in terms of storage space, since compression cannot be efficiently be used. Serialization and de-serialization of records can be used to reduce disk space, but that introduces a significant overhead for I/O operations, as data needs to be first de-serialized before being processed.
To overcome such limitations, a new columnar-oriented data representation has been introduced for databases and, in general, in big data processing. An example of such data format is Apache Parquet [6], included within the Hadoop ecosystem and fully compatible with all the frameworks included in it. The main advantages of columnar-based formats are space-efficiency and query-efficiency. The first advantage leverages the uniformity of the data values of a single column to improve compression, thus reducing the overall disk space required to store data. The query efficiency comes from the inherent optimization in accessing columnar-based data in both sequential and random access, thus improving the overall performance for in-memory processing. While Apache Parquet only defines a columnar storage format, other tools from the Hadoop ecosystem define an in-memory data representation format that is columnar-oriented. Apache Arrow defines a language-independent columnar memory data format, defined in such a way to provide efficient analytic operations on modern hardware (such as CPU's and GPU's). It also supports zero-copy reads, and reduces the serialization overhead by implementing a shared communication format that can be used by multiple machines within a cluster [2, 25]. The shared communication and data layer introduced by Arrow proves to be beneficial in the field of big data processing, as it allows different systems and various frameworks within the Apache ecosystem to inter-operate efficiently, without overheads caused by data copies, conversion and serialization. This further strengthens the advantages introduced with its columnar-based format, and it makes Arrow a good candidate to be used to optimize Spark and solve some inherent issues related to the RDD library, which will be explored in this work.

1.2. Contribution

The main target for this work is to prove the feasibility of integrating the Arrow columnar data format into the Spark execution engine, in particular in the core section (as well called as Resilient Distributed Datasets API). The architectural and implementation challenges will be analyzed and solved, whenever possible, to create a proof of concept. Such proof will prove the feasibility of such integration, and it will be used as building block upon which other optimizations and more functionalities can be implemented in future works. This feasibility serves to show that Arrow can be fully integrated within the Spark ecosystem, as demonstrated by other works, which unfortunately never considered the core Spark package, but rather focused on accelerating SQL queries or improve other functionalities.
This work also embeds Parquet files as the main input data format, leveraging the similarities between the Parquet and Arrow columnar formats and their integrated features.
Furthermore, it proves that such integration can be performed using only the Java and Scala APIs, which can theoretically be interoperable since Scala code runs on the JVM as well. This allows for a more efficient proof of concept, that doesn't rely on native code generation or native code calls.
The proof of concept presented in this work doesn't aim to be a comprehensive and ready-to-market mature solution, but rather an initial building block upon which to build the extended set of features and optimization capabilities that both Arrow and Spark can offer. This work is implemented focusing on highlighting the advantages for local computation, that is when driver and executor are launched within the same JVM, but it can be quickly extended on a properly distributed environment without the need for additional configurations. That will be the final goal for an Arrow-Spark integration, as it will be clearer later, since Spark execution divides data in chunks of equal size (*partitioning*), and each chunk is computed in parallel. Allowing speedup in computation time on a single node will, therefore, show that the benefits will become more evident while moving to a distributed system.

1.3. Research Question

The main objective of this work can be summarized with the following research question:

Can Arrow in-memory data format be leveraged to improve the performance of Spark's core package and RDD computations, achieving zero-copy and no data serialization?

To answer this question, it's essential to have an in-depth analysis of the used frameworks and tools, implement a solution PoC and evaluate it against a benchmarked application, to see any performance improvements and challenges related to its development. In order to give a more robust and well-founded answer, the following sub-questions will have to be answered during this work.

1. Is Apache Arrow a feasible, near-native memory representation format that can be used for Spark's core functionalities, and do the two frameworks integrate without the need of native code?
2. Does the introduction of Apache Arrow affect the scalability of Spark applications?
3. Can Spark-based functionalities be pushed into Arrow, and is there a competitive advantage in doing so?
4. Is the performance of such integration competitive, and what are the associated bottlenecks and limitations of it?

1.4. Outline

The following chapter, Chapter 2, introduces all the tools and frameworks that will be used as baseline for the implementation of this work, which will be introduced in detail in Chapter 3.
Chapter 2 thus introduces Apache Spark, focusing more on its core (RDD API) package and the already-included columnar processing functionalities, which won't be used directly in this work but serve as a good introduction on what's already being done to integrate such tools with the Spark ecosystem. Furthermore, it will present Apache Arrow format and its Java API, which will be the pillar of the integration with Spark. It also introduces Apache Parquet, as it will be used as input data format for the evaluation section and the Arrow-based implementation.
As said before, Chapter 3 focuses on an in-deep explanation of the implementation, the general architecture of the solution and the main challenges faced in the development process. First, it presents the changes and additions required in the Spark core package to allow for Arrow data processing, then it presents the changes required in Arrow to allow for such integration to be successful. Besides, it shows some challenges risen during development and gives a hint on how this and future work aims to solve them.
After the general architecture is presented, Chapter 4 presents a detailed evaluation of the relevant components in the implementation, including (but not limited to) explanations on the reasons behind the results and some ways to achieve better results in the future.
Lastly, Chapter 5 presents a summary of all the results and reflects on whether the research question presented earlier has been answered and to which degree it has been, showing unsolved and new challenges and potential ways to solve them in future development of this work.

2 Background

2.1. Apache Spark

Apache Spark is a general-purpose cluster computing engine with API's and tools available in several programming languages (such as Java or Scala), in order to support a wide range of workloads.
To execute any workload using Spark, users need to write a *Driver* program, containing the necessary control flow to operate on the data. Such operations are then executed in parallel by several *Worker* nodes, each of which containing the actual logical unit that performs the computations (called *Executor*), who then return the results to the Driver program. [7]

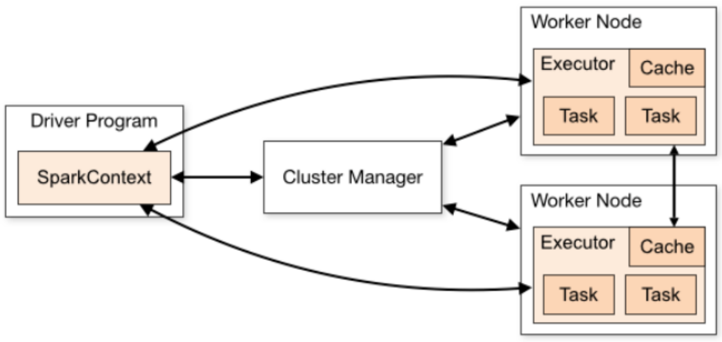

Figure 2.1: Overview of an Apache Spark operational cluster [7]

The main unit of execution in Spark is called *Task*, which is responsible to return the result to the Driver program or partition the data across multiple nodes.

2.1.1. Resilient Distributed Datasets

A Resilient Distributed Dataset (RDD) is the main abstraction in Apache Spark, and it sits at the core of the whole Apache Spark ecosystem. It's a read-only collection of data objects distributed across different nodes [41]. RDD's are designed to achieve fault tolerance and data parallelism, and they are created from different data sources (files on distributed file systems, high-level data collections on the Driver program, and so on) or by manipulating existing RDD's.
Such manipulations on existing RDD's are called *transformations*, and they can be divided into two different types: narrow transformations and wide transformations.

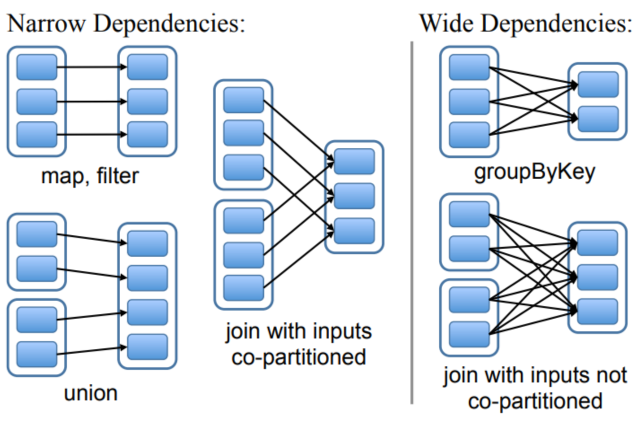
Narrow Dependencies:
Wide Dependencies:
map, filter
join with inputs co-partitioned
groupByKey
union
join with inputs not co-partitioned

Figure 2.2: Examples of narrow and wide dependencies. RDD's and their partitions are shown as rectangles [40]

In a narrow transformation, data from each partition is computed directly on the Worker node where that partition resides, whereas wide transformations require data from multiple partitions to be passed around ("*shuffled*") across multiple partitions. It is evident that such transformations require more computational steps to achieve a result, thus are the ones that impact performance more.
RDD transformations are not immediately executed: they are *lazily evaluated*, which means that they are computed only once an operation requires data to be sent back to the Driver program, or written to a file on a shared file system. Such operations are referred to as *actions*, and Spark supports a wide range of such operations to perform any kind of operation on the data (examples include, but are not limited to, counting the number of values of the RDD or returning the first element of it).
Another key difference between transformations and actions is that the former always return another RDD, albeit with changes to the underlying data, whereas the latter return a value (or a set of them). Actions are what triggers all transformations to be applied to the initial RDD, and can be used to materialize data during intermediate processing steps as well.

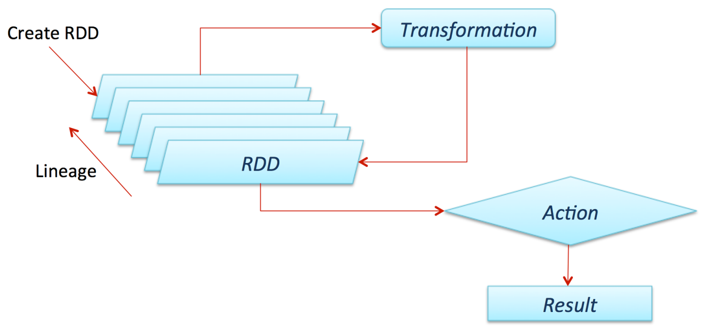

Figure 2.3: Example of how transformations and actions operate on an RDD [23]

Since transformations are "lazy", Spark needs to keep track of all the changes to the original RDD before they can be materialized (i.e. being actually computed): this "lineage", that is, the list of dependencies the resulting RDD has with the original one, is kept under the so-called *Direct Acyclic Graph* (*DAG*). Such lineage allows Spark to achieve efficient fault-tolerance (hence the term "Resilient") in such a way that, if a partition got lost, the RDD would hold enough information to be able to build it again, just by looking through the various dependencies with the original data source. [41]
The Spark Context from Figure 2.1 is the "container" where the DAG is created for each RDD. Upon triggering an action on an RDD from the Driver program, the Spark Context creates the associated DAG and divides it into small computational chunks (each of which representing an individual transformation on the aforementioned RDD), that are called *Stages* in Spark terminology. Each individual stage is then divided into several tasks on the Worker nodes to be run in parallel. Each DAG and its resulting stages and tasks are created and computed by the so-called *DAG Scheduler*, a core functionality within the Spark Context that serves as a

scheduling layer for all DAG-related executions in Spark.
Over the years, Spark has seen a number of changes and improvements to always ensure optimal performance for big data analytics workloads. Such changes arose from highlighting performance bottlenecks and potential "stragglers" (that is, slow executing tasks in a particular stage of a Spark application), and the development community itself has started various projects to solve them. Several studies have shown that the recent advances in SSD technology and faster network inter-connections have significantly improved I/O capabilities and performance, while CPU has remained roughly the same. In addition, the increased amount of data being shared across a computing cluster has increased the need for costly, CPU-bound data serialization and de-serialization [28, 32]. But it's not just CPU performance that negatively impacts the performance of big data processing in Spark, memory usage plays a big role too. In Java, objects are allocated in a memory region called *Heap*. In order to optimize memory usage and avoid keeping unnecessary data in memory, the JVM automatically and periodically checks the references for each Java object, removing all un-referenced objects from memory. The component responsible for this procedure is the *Garbage Collector* (GC), and it continuously monitors the object allocation and referencing with assumptions and heuristics to estimate the object life-span [27]. When such heuristic is not correct, however, the overhead increases and objects are not properly managed: this normally doesn't happen in normal Java programs, as objects' life-span is sufficiently small and deterministic.
In contrast, Spark can determine the exact life-cycle of its allocated objects throughout the whole computation using their lineage graph, thus allowing it to manage such objects more efficiently than how the JVM would do. Another problem related to the JVM object allocation is the inherent overhead of their memory layout: a simple 4-byte string, for example, can result in more than 24 bytes for its internal representation, as the JVM adds an object header and hashing [32]. Both of these memory shortcomings introduce undesirable overheads in Spark computation, and create an unwanted bottleneck.

2.1.2. Project Tungsten

Prior to release 1.5, Spark's development efforts have been focusing on improving I/O operations and enabling the use of more efficient input data and storage formats, such as the Parquet format: such introductions have further increased the impact that memory overhead and CPU bottlenecks have created, thus introducing the need for an umbrella solution to solve both issues at the same time.
The developers community has hence focused their attention into optimizing CPU and memory utilization, in order to reduce the gap between computation and I/O: Project Tungsten is the code-name for such development, aimed at optimizing Spark's execution engine and bringing its performance closer to the one of modern hardware devices [17]. This can be achieved by focusing on improved memory management, code-generation and SIMD operations. Project Tungsten has been introduced in Spark 1.4 with its foundation blocks, including a re-designed memory management, and it has been fully integrated from Spark 2.0 when features such as Whole Stage code generation and vectorization have been introduced. As part of the foundation blocks, Project Tungsten has introduced an explicit memory management implementation based on `sun.misc.Unsafe`, which includes C-style memory access features. This allows Spark to manipulate raw memory portions with explicit memory allocation and de-allocation, hence the term unsafe. The result of this implementation is called `UnsafeRow`. As the name suggests, this new component allows to work with row-based data, thus it is mostly used to optimize the SQL engine, and the DataSet and DataFrame API. Another disadvantage of this components is that it still uses a row-oriented data representation in memory, so it does not fully exploit the advantages of the columnar data format. In addition to this, when working with columnar data, conversions from row to column and vice-versa are required, thus introducing further unnecessary overhead. Project Tungsten also introduced off-heap memory management as part of its foundation. *Off-Heap* memory refers to all the memory sections that are outside of the JVM-managed memory area, thus it is not checked by the GC. As there is no garbage collection being performed by the JVM, this memory portion offers the advantage of reducing the related overhead of such operation, but the main problem with using it for Java-like objects is that it requires serialization and de-serialization to write data off-heap or read from it. As a consequence of this, removing the GC overhead just introduces another one.
Project Tungsten, therefore, is not the definitive answer for the CPU and memory bottlenecks of Spark. The main downside of it is that it has been developed with DataFrames in mind, thus little to no components can be used effectively in RDD-based workloads (only off-heap memory management can be used from the core library of Spark). In addition to this, previous works have already shown that the columnar support introduced by Tungsten is still limited, and it requires row-to-column conversion to be suitable for Spark's internal data representation [26]. Furthermore, it still does not eliminate the serialization overhead that is

created when moving data across different partitions in a cluster, since it does not focus its attention to I/O operations.
To further optimize Apache Spark, and in particular its RDD-based workloads, another step has to be taken: integrating it with existing frameworks that allow for columnar-oriented memory management and reduce serialization overheads while data is partitioned across multiple machines.

2.2. Apache Arrow

Apache Arrow is a framework for in-memory analytics that specifies a standardized language-independent columnar format for data representation, designed to efficiently use modern hardware. Originally developed from a seed of Apache Drill [4], it was announced by the Apache Software Foundation in 2016 and it became the de-facto standard for columnar in-memory data processing.
Having the API and format specifications available in multiple programming languages (including C++, Python and Java), Arrow aims to increase interoperability between different Big Data processing solutions. This is further achieved with its format: being language-independent means that multiple processes or systems can share data between them without the need for serializing or de-serializing data, which accounts for most of the performance overhead in big data solutions, as well as memory copies. Furthermore, being columnar allows for further optimization, such as ease of indexing, denser in-memory storage and possibility to be offloaded to hardware accelerators like GPU's or FPGA's. [1, 10]

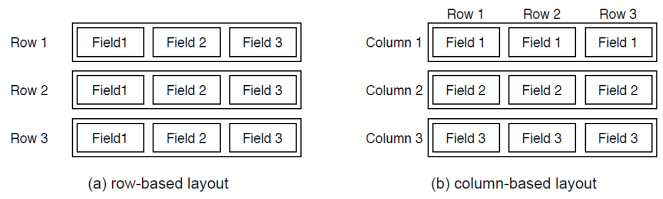

Figure 2.4: Row-oriented vs. Column-oriented memory layout [26]

2.2.1. Arrow Format

The Arrow Columnar Format specifies a language-agnostic physical memory layout for storing data structures, as well as a protocol to share data among different systems using the columnar data representation. Alongside the aforementioned benefits, the columnar format also allows for constant-time random access, zero-copy of shared memory and data adjacency (which simplifies sequential access) [11].
Since the different language implementations share the specification of the columnar format, the Arrow project provides some useful terminology to refer to different concepts and data structures. The fundamental data structure for Arrow is the *Array*, or *Vector* (since the Java API uses the latter term, it will be kept throughout the entire document): an *Arrow Vector* is a sequence of values of the same type, each of which having known length. Each element of the vector can be either *primitive* or *nested*, depending on whether the data type does contain child types. This work focuses primarily on primitive data types, as they represent the most common ones used in many Big Data applications, but it provides the basic logic to allow for nested data types too.
Each vector is defined by a logical data type, a sequence of buffers and two 64-bit signed integers. The first of the two integers represents the *length* of the vector, whereas the second represents the number of null values. They have the same representation in metadata, as the whole vector may only contain null values.
The buffers in an Arrow vector can be of three different types, depending on the logical type of the values that a particular vector holds. The *data buffer* contains the actual value of each element of the array. Since a vector can have null elements, the *validity buffer* specifies whether a particular element (determined by the bit of the vector) is null. It's worth noting that this vector is optional in case the vector doesn't have null elements. A third (also optional) buffer, called *offset buffer*, contains the start index of each element, and it's present in case the logical type is of variable width (such as strings).
In order to simplify the PoC, this work expects all vectors being ingested into Spark to have the same length, when coming from the same input source. That is, if a file contains two vectors, they can have different data types but have to share the same length (null count may vary). According to the Arrow documentation, the buffers should be aligned when allocated, that is having addresses at multiple of 8- or 64-bites. That means that, sometimes, some padding (over-allocation) is necessary [11].

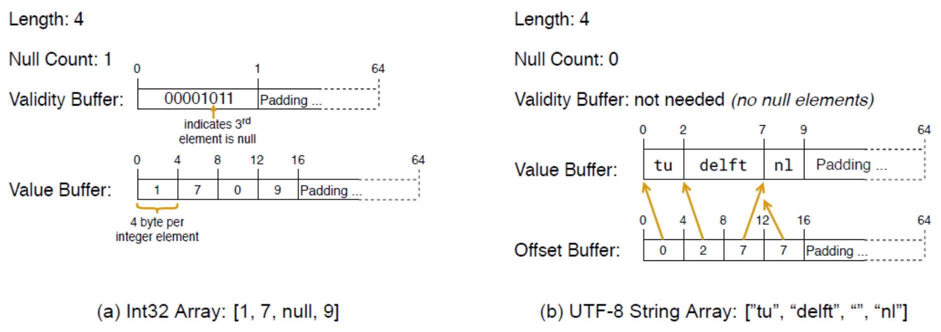
Length: 4
Null Count: 1
Validity Buffer:
00001011
Padding ...
indicates 3rd element is null
Value Buffer:
1
7
0
9
Padding ...
4 byte per integer element
(a) Int32 Array: [1, 7, null, 9]
Length: 4
Null Count: 0
Validity Buffer: not needed (no null elements)
Value Buffer:
tu
delft
nl
Padding ...
Offset Buffer:
0
2
7
7
Padding ...
(b) UTF-8 String Array: ["tu", "delft", "", "nl"]

Figure 2.5: Schematic memory representation of Arrow vectors and their buffers [26]

2.2.2. Java API

As mentioned earlier, this implementation relies on the Arrow Java library. The main reason behind this choice is the interoperability between the Scala and Java programming languages, where the first one is the choice used for the Spark core API. This further reduces the implementation complexity and potential failure points, which could arise with native code calls using the Java Native Interface (JNI). Related works, such as previous work performed by the ABS group [26] allow for native code implementations, but this was out of scope for this PoC.
The Java API provides a basic interface to Arrow data structures, called `ValueVector`: it is an abstraction used to store sequences of values having the same data type in an individual column. Depending on the logical type of the vector's elements, several classes implement this interface to provide type-based logic, while the `ValueVector` interface provides the common methods.
According to the documentation, it is important to follow the correct vector life-cycle. Upon vector creation, users first need to allocate enough memory to hold all the column values for the vectors, then it is possible to proceed populating the vectors. Note that the memory allocation does not require users to explicitly allocate the individual buffers depending on the logical type, which is performed by the implementing class internals, but they are required to set the vector values according to the specific data type. Once the vector has been populated, the total value count has to be set. At this stage, the vector holds a certain amount of values (either set or null), and it can then be accessed: value getters or vector readers can be used, depending on the user's implementation. Once the vector has been accessed, and it is no longer used, it can be cleared (that is, memory for the underlying buffers and metadata is released). It is important that user code follows these steps in order, to avoid unexpected behaviour or undefined vector states [3]. It is also possible to create zero-copy slices of the vectors, in order to have logical sub-sequences of an initial `ValueVector`.
Vectors are usually contained in an higher-level abstraction, called `VectorSchemaRoot`: this class can hold the actual vectors themselves or data batches (called `ArrowRecordBatch` in the Java library), as well as meta-data information regarding the data type of each vector (called `Field`) and the overall schema definition. In this implementation, the `VectorSchemaRoot` is used during the conversion between Parquet and Arrow, but it is not used for data transferring or RDD creation: the vectors themselves are used to create the RDD's inside the Spark ecosystem, and are being shared among the various nodes of the cluster.
The main advantage introduced by the Arrow Java API is that the buffers for each vector are based on Java's Unsafe API: each `ArrowBuf`, in fact, serves as a facade over the underlying memory that provides several access API to read and write data into a chunk of direct memory. This makes Arrow suitable for off-heap operations and, since management of this area is provided by Spark from version 1.4 (as mentioned earlier), it is logical to integrate these two frameworks together to solve memory-related bottlenecks.

2.3. Apache Parquet

Apache Parquet is an open-source, columnar data storage format available to any project within the Hadoop ecosystem. It leverages various features such as record shredding and assembly algorithms, and was built to store complex nested data, which makes it more efficient than other storage methods (including row-based or simple flattening of data structures) [6].

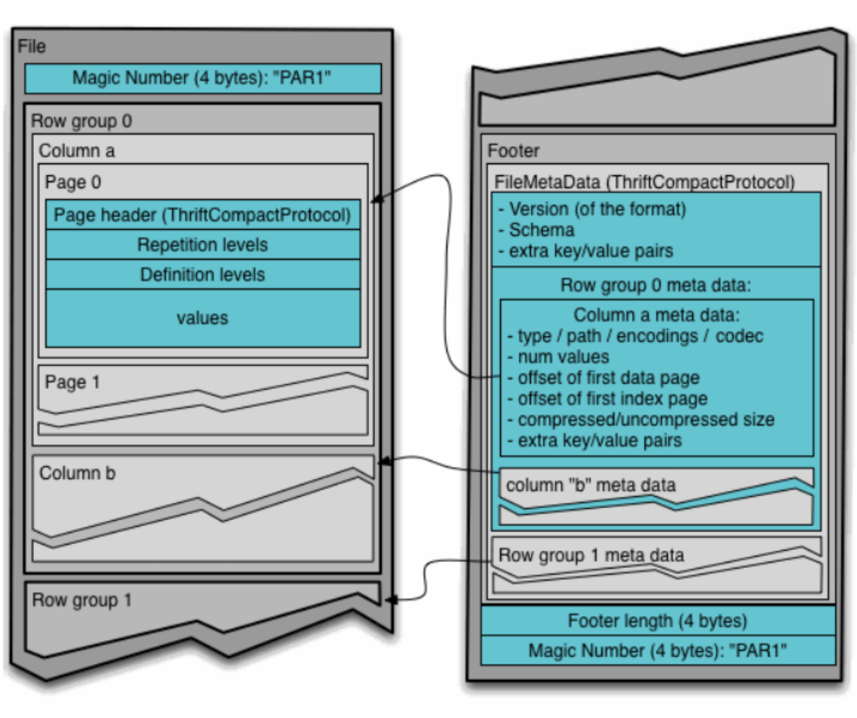
File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

Figure 2.6: Example of a Parquet file representation [6]

A Parquet file is split into *row groups*, which contain a piece of each column in the dataset. Such piece, called *column chunk*, is a portion of the data for a specific column. It is guaranteed that chunks are allocated contiguously within the file. Column chunks are then divided into multiple *pages*, which are indivisible units of data (in terms of compression and encoding). In addition, all files include metadata for that file and for each column, in order to allow readers to extract the columns correctly and with the necessary types.
Each column has a specific data type, and nesting is allowed within columns. This work, however, focuses only on primitive data types and plain columns, that is, with no nesting.
Parquet is available in multiple programming languages, and an implementation of a Parquet reader for Arrow is already included in the C++ library. In order to keep the complexity at a minimum, however, this work provides a new Java implementation for a Parquet reader, included in the Arrow API, that leverages some functionalities already present in the Java Parquet API. Such functionalities, implemented in the `parquet-mr` package, allow to convert a Parquet schema from an input file into an Arrow schema, which will then be used to create the `VectorSchemaRoot` and its vectors. The reader for Parquet files used in this work is the default implementation of the Java Parquet API, based on Hadoop files.

2.4. Related Work

Spark popularity has, of course, created a lot of opportunities to research how to solve some of the issues being faced by Spark and how to optimize the performance of workloads running on it. The Apache development team itself has, over the years, proposed a number of changes to increase the performance of big data processing, and to tackle some of the issues faced by the users. As already mentioned, Project Tungsten aims to optimize Spark's shortcomings in terms of CPU and memory utilization, but it fails to substantially improve RDD operations, focusing instead on SQL queries. This can also be seen with the introduction of columnar-based data processing (in contrast with Spark's default row-based processing), starting from Spark 2.0 and considered to be one of the building blocks upon which Project Tungsten was developed [22]: analyzing Spark's source code [1] reveals that the columnar implementations have been included only for DataFrame and DataSet operations. Furthermore, as already discovered by other research work on the same subject, Spark does yet provide full functional support for columnar operations [26].
Different Spark implementations, such as Spark-MLlib [5], are implemented on top of the Spark core API and

[1]Available at: https://github.com/apache/spark

directly use RDD's for their execution, providing workload-based performance improvements and optimizations. However, they do not solve inherent issues related to Spark's RDD's.

Another approach used to optimize Spark's performance is to offload some of the computation on hardware accelerators, such as GPU's (which are already included in most data-centres and can serve as an off-shelves alternative to CPU's) or FPGA's. The latter is recently seeing an increase in usage, as an alternative to CPU computation, mainly because of the increased effort in solving intrinsic problems and issues with such technology. Thus, *heterogeneous computing* is becoming a more and more feasible alternative for high-performance computing and big data analytics [19].

One such example of this approach is Spark-GPU [39], which enables Spark workloads to be run on GPU's, introducing block processing and data buffering (in either row- or columnar-based format) in native memory. This allows to execute Spark workloads exploiting GPU's intrinsic high parallelism and high memory bandwidth. This approach proves to improve Spark's execution time, but it incurs into extra complexity and overhead when copying data from native memory into GPU memory, as well as complexity introduced by creating GPU-friendly code execution components that can be integrated with the default Spark's execution engine. Another example of computational offloading to GPU's is HeteroSpark [24], where GPU's are connected to Worker nodes in a cluster. Internally, it uses Java RMI for data forwarding to the GPU, but it incurs in high overhead given by data serialization and de-serialization.

As already said, FPGA's can be used to accelerate (parts of) Spark computation, as an alternative to GPU's. One example of this integration is Falcon Computing's (now part of Xilinx) own Blaze implementation [16]. This runtime system allows to accelerate big data queries with FPGA's, and it introduces an accelerator-friendly version of Spark's RDD library. The main disadvantage of such solution is its limited amount of algorithms, thus reducing the overall flexibility. Other approaches include Apache Arrow as in-memory columnar data format for FPGA integration, leveraging its great integration with FPGA's internal processing [14, 26]. These works have similar implementations, as they leverage and improve Spark's built-in columnar processing, aiding existing functionalities with tools such as a native Parquet reader and Gandiva [18]. F. Nonnenmacher's work [26], in particular, has been developed in the ABS Group at the TU Delft, and provides an integration with Fletcher [29], an FPGA-based framework that directly interfaces with Apache Arrow. These implementations, however, do not integrate the RDD API nor solve its limitations, as they focus their attention on the SQL component of Spark. During this work, an Arrow-based Spark integration has been presented [31], that uses a similar approach to the aforementioned implementations, albeit for the acceleration part. In particular, this work creates an integration between the Arrow C++ Dataset API, including a native Parquet reader, and Spark's Dataset API (that is, the `ColumnarBatch` implementation used to represent columnar data). Given its newness, not many comparisons could be drawn for a comprehensive overview of this solution.

Apart from accelerating Spark through the use of heterogeneous computing hardware, which, as it can be seen from the ongoing research, focuses its attention more towards higher-level Spark libraries (in particular the SQL package), it is possible to improve RDD operations by solving some inherent issues related to it. Two of the more evident problems are in the form of memory management and the cost of shuffle operations in wide transformations. As already explained, while Project Tungsten introduced some memory management techniques that can be leveraged during RDD operations, these are not fully exploiting the advantages that a fully-RDD-compatible columnar-based data representation could bring. Leveraging off-heap memory, for instance, would still cause data serialization overhead while writing to or reading from such memory region. On the other hand, some research efforts have been made to improve shuffle performance, and reduce its induced overhead in the total execution time. Existing work on this topic aim at reducing the number of read/write operations being performed during shuffling, or by re-designing the shuffling mechanism used by Spark for operations such as join or reduce. The first approach [37] targets the bottlenecks associated with CPU scheduling and I/O operations in shuffle "spill" operations, that is, when the output of a map stage results in data being written in shuffle files in disk. It leverages the Unsafe-based shuffle manager in combination with Java's NIOBuffers and large buffer sizes to reduce the overall amount of read and write operations. The second approach also reduces the number of I/O operations for shuffle tasks, but it works by implementing a re-designed shuffling management system to replace Spark's existing one. OPS [15] is an open-source distributed shuffle management system that proposes early-merging of intermediate data from the map stage in memory (rather than disk), early-scheduling based on Spark's partitions and early-shuffling of such intermediate data being already available. This ultimately results in the map and shuffle stage to be executed almost in parallel, based on the scheduling of the map tasks and their completion. SCache [30] proposes a similar approach, where the shuffle I/O operations are executed in parallel with normal map computations, leveraging pre-scheduling based on heuristics and pre-fetching based on shuffle size predictions.

3 Implementation

3.1. Overview

This chapter explores the various challenges that integrating the Arrow format with Apache Spark poses, and the implementation used to solve them.
In general, this work follows the rule of performing non-disruptive changes to the Spark API, so that Arrow-based and normal Spark workloads could be executed within the same *Spark Context*: the main benefit of such approach is that enabling the Arrow-Spark engine proposed in this work on an existing cluster would not require complex setup or external tools, and that all programs can properly execute within user-space. This way, different users can have heterogeneous workloads executing on the same, existing hardware.
The details of the implementation will be discussed in the following sections, but in general the approach is to enable the creation of RDD using Parquet-generated Arrow vectors. Swap Scala data types with Arrow is, however, not straightforward: using RDD's as plain containers of Arrow-backed vectors of data would require a complete change of Spark's execution engine, as it has been developed to expect primitive data types. Therefore, the obvious choice is to retrieve the values contained in the vectors for the actual computation part, while vectors are used to carry the data around the cluster in a standardized and language-independent manner, with the RDD being the logical container of the vectors. Since they can be sliced in a zero-copy fashion, data partitioning is still possible and, therefore, normal Spark execution can be used for computation. Such implementation, however, heavily relies on Scala reflection to enable the framework to obtain the relevant run-time information on the values' type.

3.2. General Architecture

The implementation directly integrates the changes in the API packages of each framework, exploiting their modularity and extensibility, to show the overall feasibility of this solution. Therefore, the implementation of this work [1] directly follows the structure of each framework, as it can be seen in Figure 3.1. A more detailed explanation of the changes per framework will be discussed in the following sections. In order to simplify the implementation, without losing any features, the Spark Scala API and the Arrow and Parquet Java API have been used in this work: this allows for a seamless integration between the two programming languages, as Scala code is executed on the JVM and is perfectly interoperable with the Java libraries. This solution, however, requires the design of a Parquet file reader to be included within the Arrow Java library, as well as tools to perform a conversion between Arrow and Parquet schemas, which is essential for the feasibility of this work.
Given that Scala is a strongly-typed language, it was not possible to provide a fully encompassing solution for complex data types, such as Tuples, as it would have required implementing all the different cases. This implementation is therefore used as a building block for further extensions, with the creation of the main logic and the support for base-case types (such as integers, strings and 2-valued tuples).

[1]The code repository can be found at https://github.com/fedefiorini/Arrow-Spark-Engine

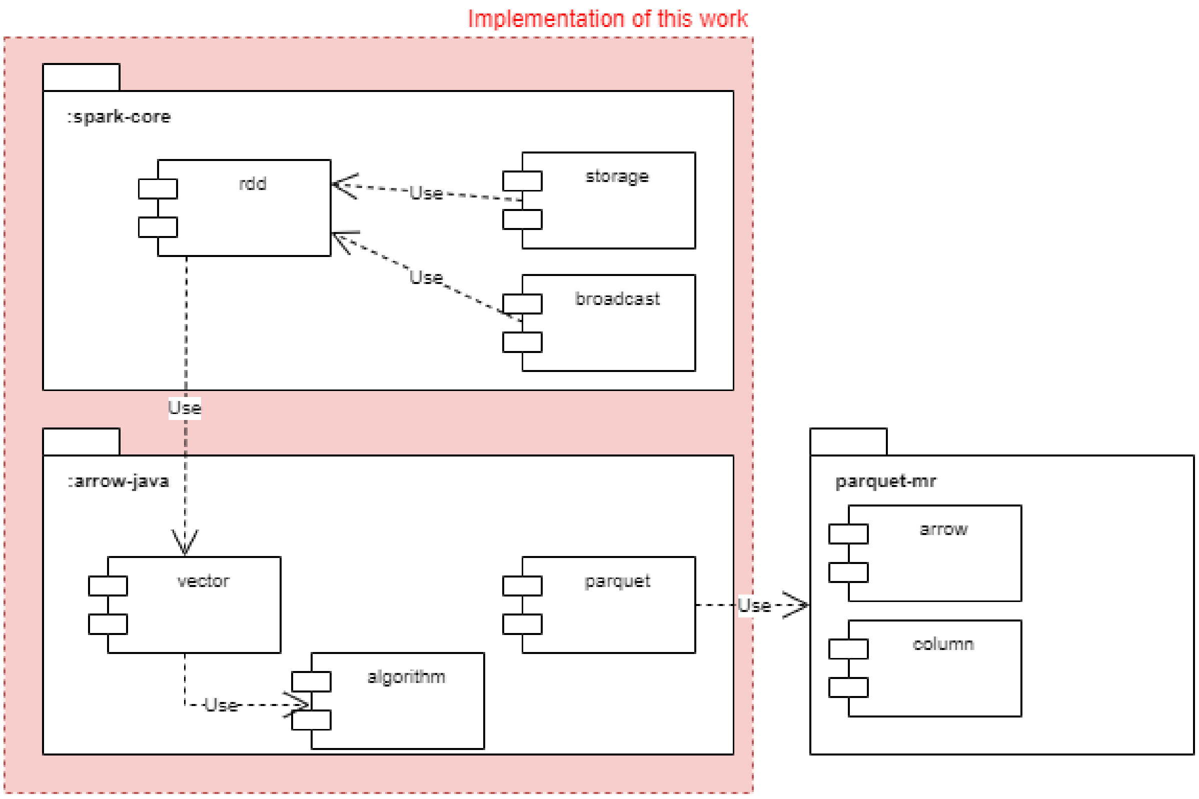

Figure 3.1: Architecture overview: structure of the changed packages and their dependencies

3.3. Introducing Arrow vectors in Spark

This section goes into detail in all the changes required to allow Spark to with with the Arrow in-memory format, and they can be divided further into two categories: **Arrow-based** and **Arrow-enhanced**.
The former describes the implementation of components that required Arrow as data format, and can't work otherwise. The latter includes all changes to the Spark core package that have been modified to support Arrow in their normal operations. A clearer overview of the changes can be seen in Figure 3.2, where the distinction between the two categories is highlighted for better comprehension.
As mentioned earlier, the whole implementation follows the naming conventions and modularity of the Spark-core package. The changes to MemoryStore and BlockManager allow to use the off-heap memory for Arrow-based RDD computation, which has inherent benefits that will be discussed in more details in the following chapter.

3.3.1. Arrow-based RDD

The RDD API provides several methods to create an RDD and partition its data according to the user requirements, the most common of which are starting from a file stored in the Hadoop File Systems (HDFS), which uses the old MapReduce API, or starting from a sequence of data being defined in the Spark driver program.
In order to create an Arrow-based RDD, however, users need to start from an Arrow ValueVector (or, better, a collection of them), which can be retrieved from an Arrow file or a more-commonly used Parquet file. Such integration requires a specific implementation of an RDD capable of holding such vectors, and perform all the required computations on them without ever changing the format of the underlying data. Thanks to Spark's extensibility, it was possible to create the required classes just by extending (or implementing, in the case of interfaces) the native Spark components. The implementation takes inspiration from the `ParallelCollectionRDD` class.
An overview of the procedure described above can be seen in the following Figure 3.3, which depicts the different ways in which pure Spark and Arrow-Spark create the RDD.

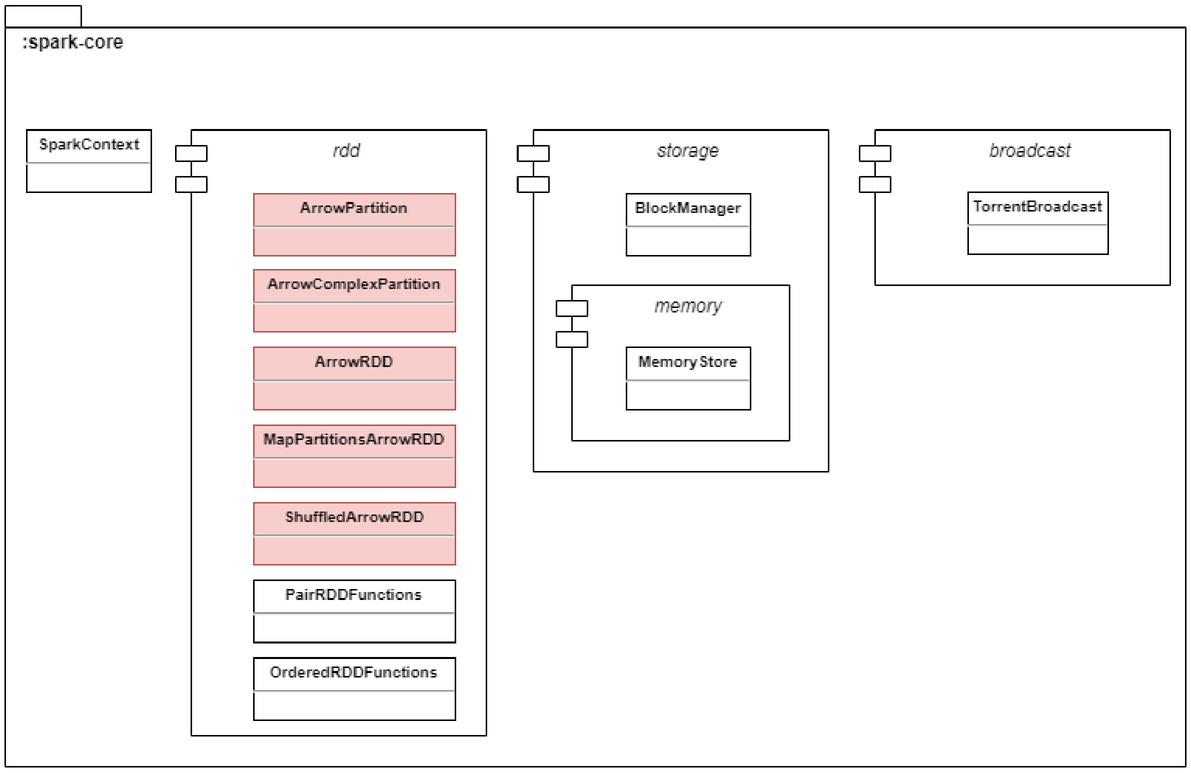

Figure 3.2: Spark-core changes overview

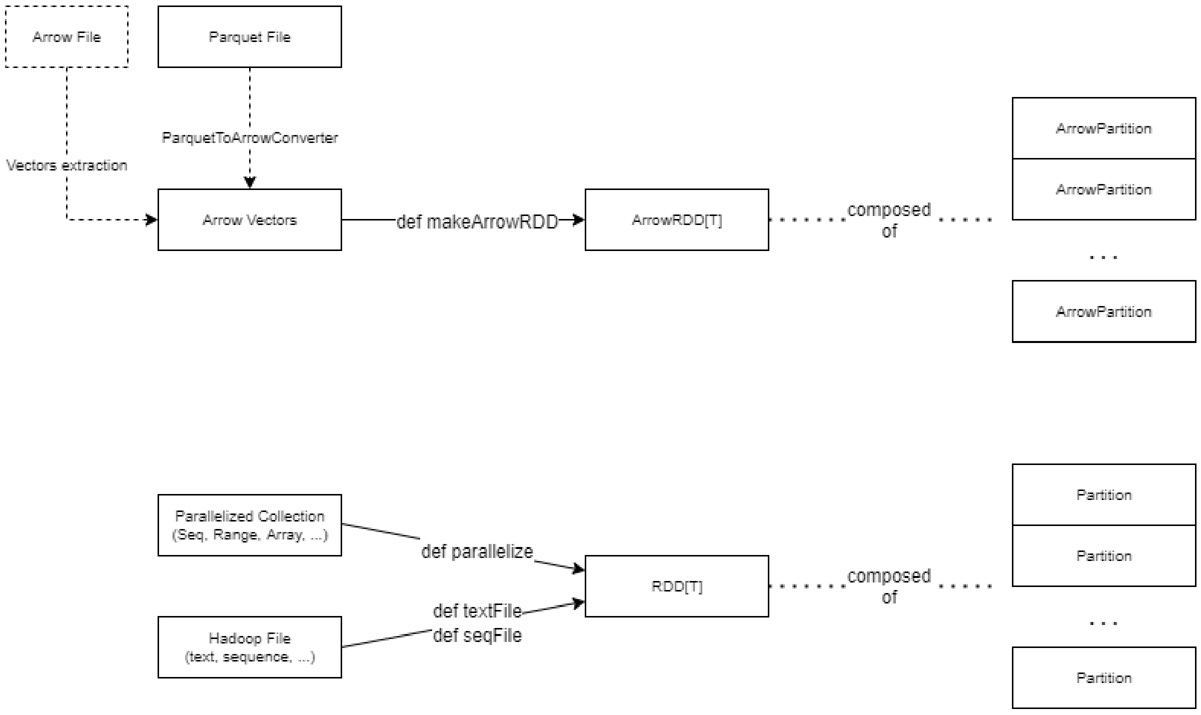

Figure 3.3: RDD creation techniques (Spark vs. Arrow-Spark)

Once the collection of ValueVector(s) has been retrieved, the method `makeArrowRDD` included in `SparkContext` proceeds to create the Arrow-backed RDD, whose type is the one of the values of the ValueVector (i.e. Integers for IntVector, Long for BigIntVector and so on). The starting vector can be split in different sub-vectors, each of which holding the same number of values, depending on the partitioning level defined during the RDD

creation. Such partitioning is performed using the built-in Arrow feature of slicing, which performs zero-copy division of vectors according to user-defined characteristics.
In order to extend the abstract RDD class, users need to define the following methods: `getPartitions`, used to return the set of partitions of this RDD, and `compute`, which computes a given partition of the original RDD. In the Arrow-Spark implementation, the first method simply creates a set of `ArrowPartitions`, each of which having a slice of the original ValueVector(s) used. It's important to note that an `ArrowRDD` can only be created with one or two ValueVector(s) as input, but it can be extended in the future to hold several more vectors (there's no virtual limit for this). In case two vectors are being used, the creation of the partition also creates an array of the vector slices, in order to preserve the original input format.

3.3.2. Arrow Partition

The `ArrowPartition` class provides the basic logic to work with Arrow-backed vectors in Spark. It's further sub-divided into normal and complex, the latter holding complex vectors such as `StructVector`. There's some differences between the two, but the main logic remains pretty much the same.
An `ArrowPartition` is created from a slice of the original vector used to create the RDD, and it further includes methods derived from the `Externalizable` interface to allow it being transferred between the Driver program and the Executors when performing computations in Spark. This topic will be discussed in more details in the next sub-section, but it's worth mentioning that, with this approach, the impact of data serialization can be reduced, thus improving execution time.
This class provides two separate iterators, depending on the number of input vectors used to create the RDD and called from the `compute` method in the RDD, which are used to get the individual vector values once they're required by the Spark execution.
Furthermore, this partition offers two different methods to efficiently convert the data as a result of a transformation, in such a way that the Arrow format is maintained throughout the code execution (thus, the vectors can always be retrieved and passed around in their original format): the first method, called `transformValues`, simply applies a transformation to the vector's values if that doesn't require a vector change (i.e. a function f : T => T), whereas the second method, `convert`, changes the underlying vector while applying a transformation on its values, in order to maintain the Arrow logic. That's the case, for example, of a transformation f : T => U between Integers and String values (i.e. a common toString): if the transformation would be applied without having the underlying vector converted to the proper type, the whole computation would fail since they're of incompatible types, as discussed in the previous chapter.
This functional separation has been proven beneficial, considering that vector conversion is a costly but required operation, when the workload includes transformations with the same data type. A detailed evaluation of it is included in the next chapter.

3.3.3. Transformations

As explained in the previous chapter, Spark allows for two different types of transformations.
Under normal Spark execution, transformations such as `map` or `filter` would create a so-called `MapPartitionsRDD`, which is an RDD that applies the given function to each partition of the parent RDD. The function is then applied to each value contained in the partition, which is also converted to the correct data type by the JVM itself. With Arrow-based data structures, that conversion and transformation is not trivial.
A solution could be leveraging the built-in **Iterator** in the `ArrowPartition`, but that would only apply the transformation without converting the underlying vector holding the values, resulting in runtime exceptions.
To overcome this issue, as mentioned earlier, `ArrowPartition` contains two different functions that apply the given function to the vectors, converting them to the correct type if needed.

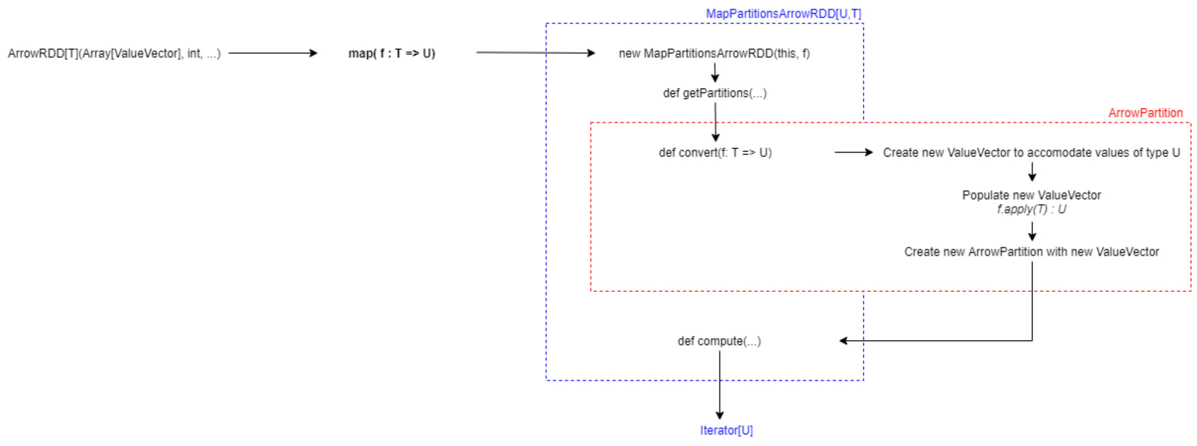
ArrowRDD[T](Array[ValueVector], int, ...)
map(f : T => U)
MapPartitionsArrowRDD[U,T]
new MapPartitionsArrowRDD(this, f)
def getPartitions(...)
ArrowPartition
def convert(f: T => U)
Create new ValueVector to accomodate values of type U
Populate new ValueVector
f.apply(T) : U
Create new ArrowPartition with new ValueVector
def compute(...)
Iterator[U]

Figure 3.4: Overview of any (narrow) transformation execution in Arrow-Spark. It only shows the case for vector conversion

The main reason for pushing the conversion functionalities to the Arrow Partition is that it's the actual "container" of the vector, after it's being sliced. Applying the same logic at another abstraction layer (such as the RDD itself) would incur in logical issues, as the RDD itself doesn't hold the data but it's rather an abstract container for it.
Once the values are successfully transformed and, if required, the vector converted to the right type, the usual Spark execution takes over: first, the `compute` function is called and, then, the actual iterator is retrieved from it. This way, a proper transformation can be called on Arrow-backed RDD's without disrupting the standard Spark environment.
In case of wide transformations the logic for the shuffle stage and the function application has been kept similar to the standard Spark execution, in order to maintain interoperability between the two different solutions. Only small changes in the `getPartitions` method for `ShuffledRDD` have been implemented, that uses Arrow-backed partitions from the previous stage. `PairRDDFunctions`, which is used to perform transformations on key-value-based RDD's, has been changed too, in such a way that both Arrow-backed and standard shuffled RDD's can be seamlessly called from within. This change is required, since all wide transformations (such as *reduceByKey* or *aggregateByKey*) are performed on key-value pairs. `OrderedRDDFunctions` has also been changed, so that sorting functionalities are enabled to work with Arrow-based RDD's too. The main difference between standard Spark and Arrow-Spark, when it comes to wide transformations, can be seen in the serialization operations that are required when shuffling the data in the various partitions.

3.3.4. Serialization in Arrow-Spark

In any distributed application, serialization plays an important role on the performance.
The main idea of integrating Arrow as data format for Spark's execution is to reduce the impact of data de-/serialization, as well as data copying. Spark's execution normally involves serialization several times, as RDD's and partitions are moved around the network (between the driver and executor programmes) under the concept of `Task`: in Spark, there's two separate kinds of tasks, namely the ResultTask and ShuffleMapTask. The former represents a task that sends data to the driver application as a result of a transformation, whereas the latter represents the output of a shuffle, in which data in different partitions is moved around depending on the function being called.
To avoid unwanted data serialization, and to maintain interoperability with the existing Spark eco-system, the Arrow partitions implement the methods `writeExternal` and `readExternal`. They allow for custom- and user-based writing and reading to object streams, in such a way that the whole Arrow-RDD is passed around efficiently, along with the underlying vectors.
This reduces the impact of serialization, as it will be seen in the Evaluation section, but it's surely a sub-optimal solution: it's possible to dive further into this topic, avoiding serialization entirely, but that would likely require disruptive changes to the Spark scheduler and the Task and Stage implementations, increasing the complexity of the solution.
With the Externalizable implementation, vectors can be passed around the system efficiently, and re-created on the executor nodes for the actual computations.

3.3.5. Data Typing and Reflection

This subsection described the challenges arising while using Arrow-based vectors as the underlying data type for RDD's.
The Scala Spark API usually uses reflection, in particular `ClassTag`, to ensure that all components can operate regardless of the data type being used. Some additional functionalities are added for specific data types, such as KV-pairs or double-precision floating point numeric types (i.e. Double). According to the Scala API [33] , the class tag contains information of a given class T that's available at runtime, but not at compile time. This is particularly useful when instantiating sequences or arrays whose type is unknown at compile time, or when creating a generic implementation for any given runtime type. It's a powerful but limited construct, since not all information are available at runtime as the JVM erases whatever is not required for the execution. This makes using `ClassTag` particularly tricky to use when converting the vectors in RDD transformations, as some times the resulting type (i.e. U in a f : T => U function) can't be properly inferred.
This could be sufficient under normal circumstances, but it actually creates issues when a transformation requires the knowledge of all the static type information: this lack of information given by `ClassTag` becomes evident for such transformations that require knowing the exact types of a Tuple, defined as
`Tuple2[+T1, +T2](_1:T1, _2:T2)`, such as `map(x => (x, x.toString))`. In this case, the Class Tag doesn't provide any information on T1 and T2, which makes such transformation impossible.
This issue can be circumvented introducing `TypeTag`, as part of the scala.reflect.runtime.universe package:a type tag encapsulates the runtime type representation of a compile-time type. This can be seen in the example below:

```
println(classTag[(1,"1")]  //prints "scala.Tuple2"
println(typeTag[(1,"1")]   //prints "(Int, String)"
```

Applying this logic requires some additional changes in several places in the code, in particular in the `ArrowPartition` and `ArrowRDD` implementations, plus the classes extending them, but it's necessary to correctly perform vector conversion successfully during any transformation.

3.4. Arrow Changes

This section details all the changes to the Arrow Java API that allow the integration with the Spark core package. It also includes the Parquet reader and Parquet to Arrow converter, which were not previously available in the Java API: this change allows for a more complete and easier usage of such API, without the need to include native code calls (through the Java Native Interface) to the C/C++ API.

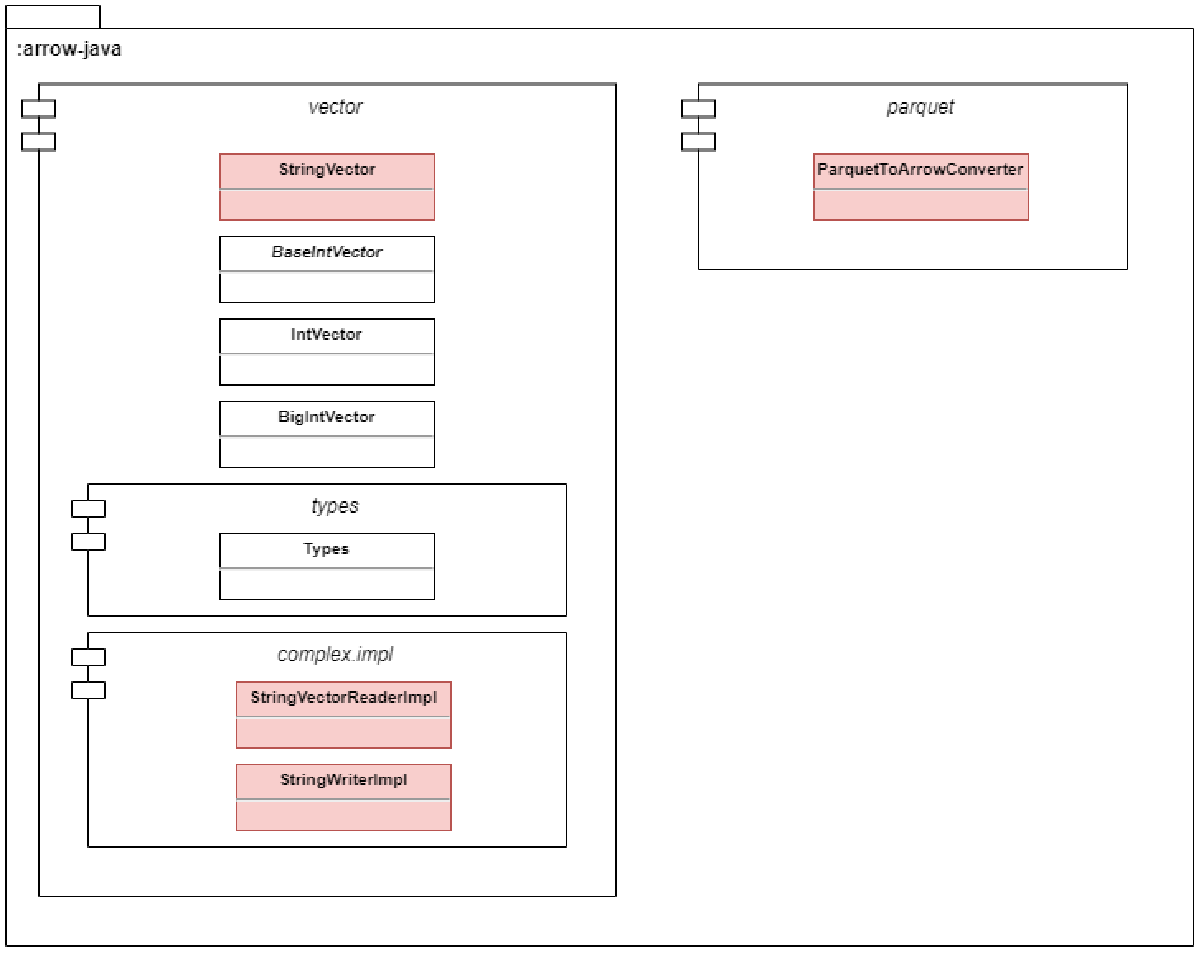

Figure 3.5: Arrow changes overview

3.4.1. Parquet to Arrow

This package contains the necessary logic to read a Parquet file from within the Arrow Java API, and to convert it into a usable Arrow data structure (in particular, a `VectorSchemaRoot`): this can be seen as a holder for a set of vectors (called `FieldVector`) that can be loader or unloaded. It also holds information on the Arrow data such as the schema, which can be useful to determine whether a vector of a particular data type is present or not.
In order to obtain usable information and data structures, helper methods are added that allow users to retrieve vectors of a particular type or the generated Arrow schema. An overview of this class operations can be seen in Figure 3.6:

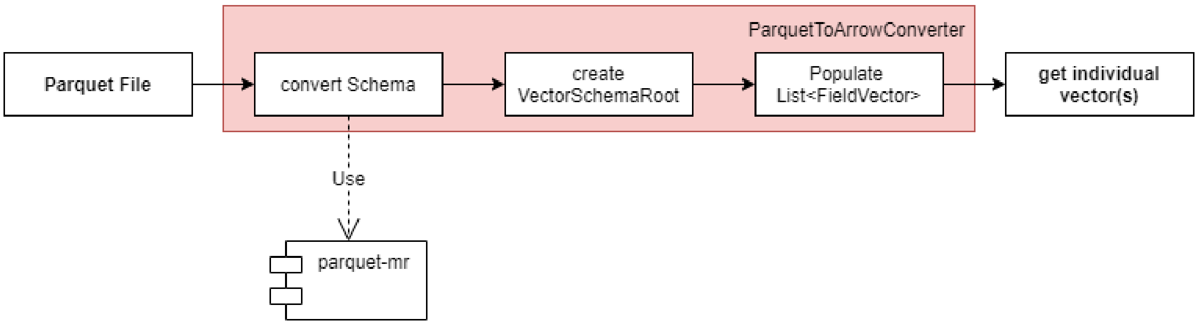

Figure 3.6: Parquet-to-Arrow simplified overview

The implementation is loosely based on Animesh Trivedi's work[2], with a few key differences.
First, this implementation uses functions from the latest stable *Arrow* and *Parquet* releases, whereas the previous implementation used mostly deprecated methods (which won't be available anymore in future releases).
Secondly, most of the schema conversion, from Parquet to Arrow schema, is performed "hardcoding" the necessary methods and data structures that create a usable Arrow schema, which doesn't allow for extensibility or completeness. This implementation, on the other hand, uses functionalities available in the `parquet-mr` package, such as `SchemaMapping` and `SchemaConverter`, which take a Parquet schema and provide a one-to-one mapping to an Arrow schema, thus allowing a more complete solution.
The function that populates the Arrow vectors according to the data in the Parquet file is, unfortunately, based on Trivedi's solution: this is because no other available methods were found in the aforementioned packages that could allow for a simple and efficient vector creation. This can be better implemented in further development of this work, but for the moment the changes introduced can already show some timing improvement compared to the old implementation, as it will be discussed in the next chapter.

3.4.2. StringVector

In the Arrow data format specification, variable-width data (such as String) can be represented using two different vectors: `VarBinaryVector` and `VarCharVector`. As the names suggest, the first one is used mostly for binary data (i.e. in the form of byte arrays), whereas the second for VARCHAR values, such as String data.
Unfortunately, the default implementation of the two different vectors doesn't really allows for String values to be efficiently set as values, since

- `VarBinaryVector` only allows to set its values as byte array
- `VarCharVector` wraps String values in a Hadoop-like abstraction called **Text**, which is not directly compatible with java- or scala-based String values, therefore making it unusable for common data transformations

Therefore, in order to allow proper String data manipulation within the Arrow-Spark ecosystem, a new vector type has been implemented. The new `StringVector` (and all its required helper classes, such as Readers and Writers for its data) has thus been included in the Arrow Java API, allowing to manipulate String-like data and perform vector re-construction during the conversion phase for each transformation.

3.4.3. Pushing functionalities to Arrow

In order to prove the feasibility of this solution, and to add some more scientific relevance to this work, it's important to see whether some functionalities can be pushed to Arrow when using this Arrow-Spark engine, rather than using the default Spark implementation.
An example of such functionalities can be finding the minimum value of a numeric vector (such as IntVector), which can already be done solely in Spark by calling the `min()` function on the RDD: this action simply calls an action, called `reduce`, which reduces the elements of the RDD to a single value, according to the specified commutative and associative binary operator. This of course triggers a job execution, which is then called on each RDD's partition in order to return the desired result.
As a proof of concept, the above mathematical function can also be called directly from the vector themselves: using algorithms available in the Arrow Java API, each vector can individually find its minimum value and communicate such value to the Arrow-based partition holding the vector, which then passes the result to the RDD. This way, the minimum value can be effectively be computed in Arrow itself, rather than from within the Spark framework.
An evaluation of this method and a comparison with the default Spark operations has been included in the following chapter, and it serves to prove that it's possible to push some functionalities to Arrow, especially for numerical-value vectors.

[2]code implementation available at: https://gist.github.com/animeshtrivedi/76de64f9dab1453958e1d4f8eca1605f

4 Evaluation

4.1. Setup

The measurements are executed on the DAS-6 cluster system [13], consisting of one headnode (on which the Spark driver program executes) and, otherwise specified differently, two compute nodes (where the worker nodes reside). The system consists of a dual 16-core AMD EPYC2 (Rome) 7282 CPU (2.8GHz), with 128GB RAM and two 4TB SSD for storage. The cluster nodes used have a 100Gbit/s InfiniBand interconnect, and they run Centos8 as operating system.
This implementation uses Apache Spark version 3.3.0 (the latest available source code), and the latest stable releases for Apache Arrow (6.0.0) and Parquet (1.12.2). In addition, for building and execution purposes, the experiments use Maven 3.8.3, Scala 2.12.15, SBT 1.5.5 and Temurin Java 8 (formally AdoptOpenJDK).
In order to test the ArrowRDD capabilities, different input Parquet files have been used, ranging from 100,000 to 10,000,000 rows. To make the evaluation consistent, and to benchmark Spark performance, text files have been used that follow the same number of entries and data type. A list of all the files used for this evaluation can be found in the code repository, under the *data* folder [1]. The Parquet files follow a similar structure as the ones included in the Chicago Taxi trip data, available for public use. The files have been created starting from available JSON files of the desired number of entries, then converted into Parquet files for evaluation.
For the execution time evaluation, the Intel HiBench microbenchmark suite [21] has been used, in particular the Word Count and Scala Sort benchmarks. All other experiments have been performed using custom-made transformations, which can be found in detail in each section.
All the experiments have been performed enabling Off Heap memory, in order to clearly see the advantages given by introducing the Arrow format, for Spark execution. This can be configured using the following two options:

- `spark.memory.offHeap.enabled = true`
- `spark.memory.offHeap.size = 1g`

Both options can be configured under `conf/spark-defaults.conf` and used at runtime. The latter option specifies the amount of bytes available in the off heap memory region, in this case 1GiB. Of course this value could be set to a larger value, but it has been kept somewhat limited to mimic low-memory scenarios too.
The performance evaluation for each scenario has been obtained enabling Spark logging, which is then visualized on the web-based *History Server* for further insights. In order to obtain meaningful results, and to reduce the impact of noise in the results, experiments have been performed 20 times (or more, when needed) and averaging out the median value for each run. In addition, to avoid "cold-start" problems in Spark, each experiment has been performed after running some other, unrelated, computation on the cluster.

4.2. Parquet

The performance of the proposed Parquet to Arrow converter has been measured by reading and converting a Parquet file consisting of 10,000,000 rows, and it is then compared to A. Trivedi's design. The two implementations are compared on total execution time, but also on each code section. The three code sections

[1] https://github.com/fedefiorini/Arrow-Spark-Engine/tree/main/data

that have been measured, and compared, are **Parquet Reading**, **Schema Conversion** and **Vector Population**. Given that the last component has almost been ported to the proposed implementation, time differences there are less relevant for this evaluation. Figure 4.1 shows the comparison between the two implementation,

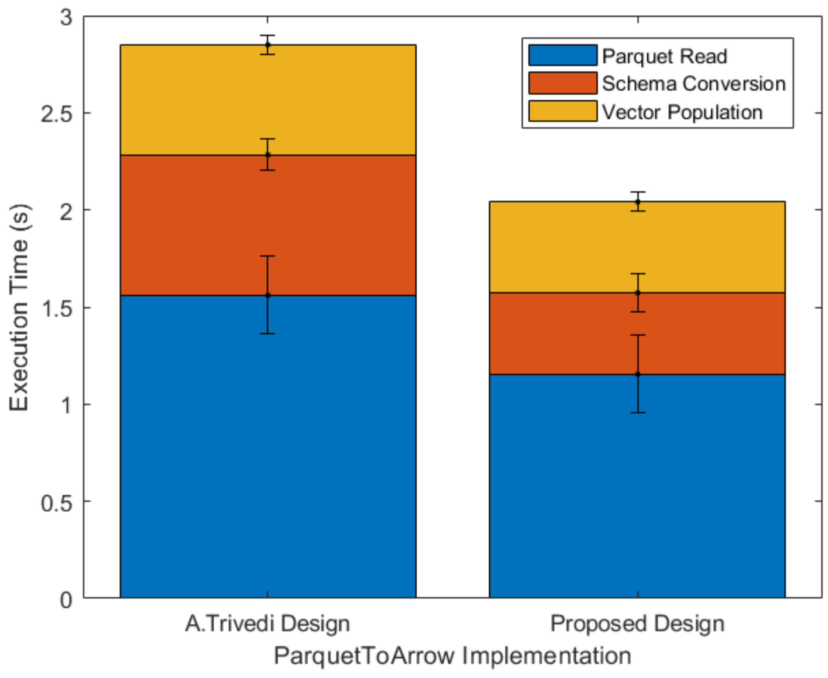

Figure 4.1: Comparing different implementations of the ParquetToArrow converter, with indication of the different code sections in focus

and it can be clearly seen that, on average, reading the Parquet file takes most of the total execution time (roughly around 55%). This proves that reading Parquet files and retrieving the important information, such as the schema and metadata on each column, is a costly operation. Unfortunately, this is necessary to allow the Arrow-Spark engine to work, so it has to be taken into account for future work. However, using built-in Parquet-Arrow functionalities it is possible to speed up the reading process, allowing the converter to keep only the relevant information required to convert the schema.
The schema conversion leverages components in the *parquet-mr* package too, thus increasing robustness and performance. It allows for a faster conversion between a Parquet schema and an Arrow schema, ensuring that all possible data types can be converted too (rather than "hard-coding" the ones required by each experiment). For vector population, unfortunately, no methods are currently available in the Parquet source code. Future work may aspire to further decrease this processing time, only given the implementation of such methods in future releases.
On average, the proposed design can achieve a reduction in execution time by approximately 28%, which proves to be a huge advantage in reducing the overall execution time of a Spark-Arrow workload. Unfortunately, the impact of file I/O on built-in Spark functions (using Hadoop files as input) is negligible on the overall performance, so reading Parquet files still has a disadvantage towards this solution. Given the importance of new columnar formats in big data processing, however, still proves the importance of this implementation.

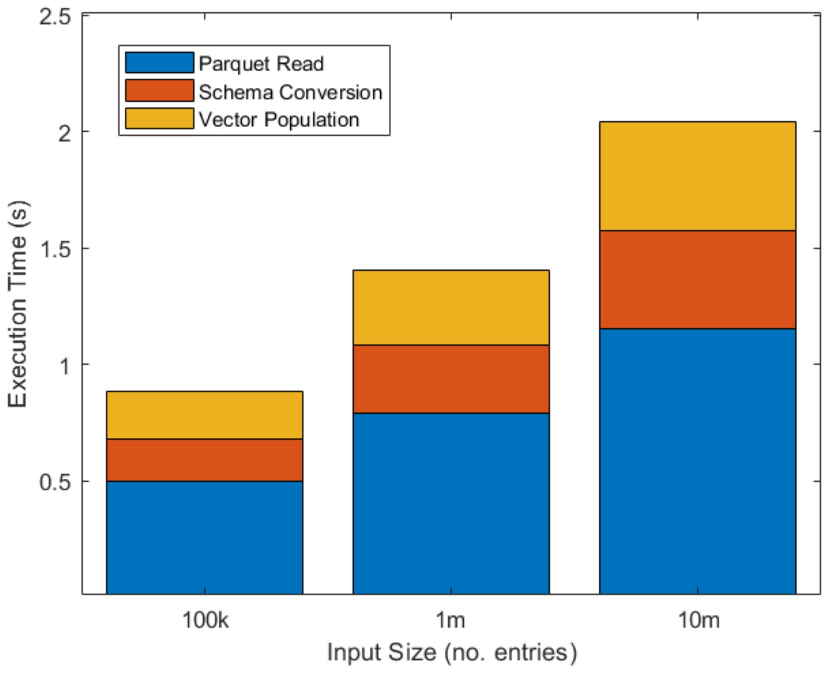

Figure 4.2: Parquet to Arrow conversion comparison for different input data sizes

As it can be seen from the picture above, increasing the input data size clearly increases the overall execution time of the converter. A way to speed up the overall execution could be performing a Parquet read only once during the whole execution: this will greatly affect the performance, as this costly operation can only be performed at the beginning of long and complex workloads, thus reducing the overall impact of it.

4.3. Narrow Transformations Considerations

As it was mentioned in the previous chapter, this implementation heavily relies on vector conversion as the building block for all types of narrow transformations being performed on an ArrowRDD.
This vector conversion is necessary when the underlying vector needs to accommodate values of a different data type, and it becomes more essential in case of two types being incompatible with each other: while a conversion between Long and Integer may still work, a conversion between Integer and String requires a totally different type of vector (as already pointed out in the introduction). The former example of a numeric-based conversion clearly represents an edge case, so it would always be recommendable to change the underlying vector according to the resulting data type of the transformation.
Given that converting a vector while performing transformations on the same data type could result in unnecessary overhead, a solution could be of just updating the values in the vector according to the transformation. This can only be performed on same-type transformations (e.g. Int => Int), and it has an immediate benefit, which can be seen in Figure 4.3.

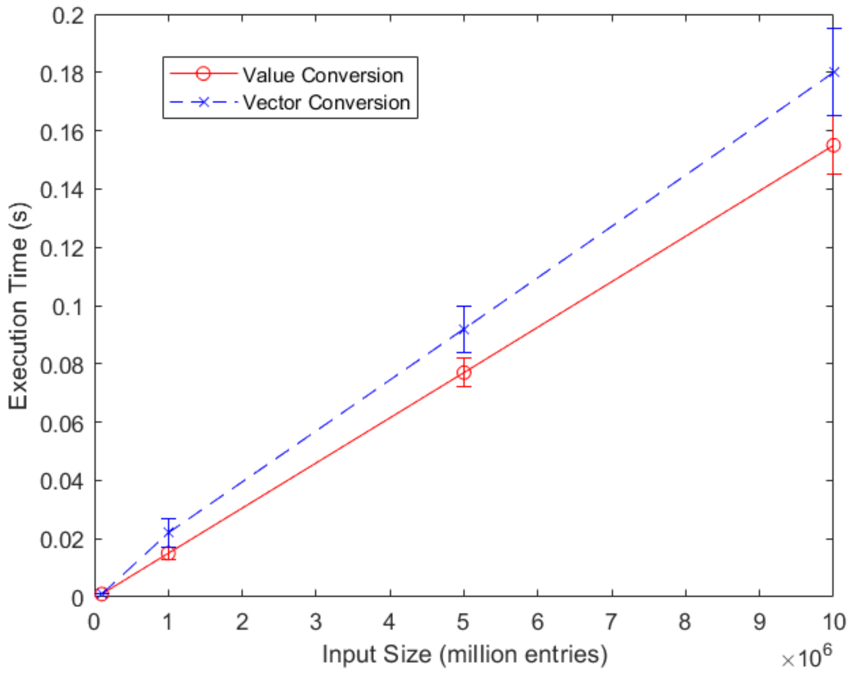

Figure 4.3: Comparing execution time for different approaches for transformations on an ArrowRDD

For the same transformation, re-allocating the vector clearly takes up more time, on average, than transforming the values that the original vector holds. This shows that a *value* conversion is preferable when the transformation allows it, and that logic has been delegated to the `GetPartitions` method in the `MapPartitionsRDD`, which then checks the original and return data types of a given transformation, and calls the methods accordingly. Overall, for functions with the same data type (any `f: T => T`) applied on an ArrowRDD, performing a value conversion has a speedup between 15% and 30%, depending on the input data size (the speedup increases with the size).

4.4. Benchmarks

4.4.1. WordCount

In order to have an effective comparison between Vanilla Spark and the proposed Arrow-Spark implementation, the WordCount micro-benchmark has been used. It simply counts the occurrences of any word in a text, and it consists of a series of transformations on an RDD of String values, defined as shown below:
`flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_ + _)`
In order to be used with Arrow-Spark, the first transformation has been adjusted to match the starting data type from a Parquet file: given that the data is retrieved as byte arrays, in order to be manipulated further in the code as string values, it has to be converted as such. The function used for it is
`map(b => new String(b, StandardCharsets.UTF_8))`. This function allows to use the following two transformations properly.

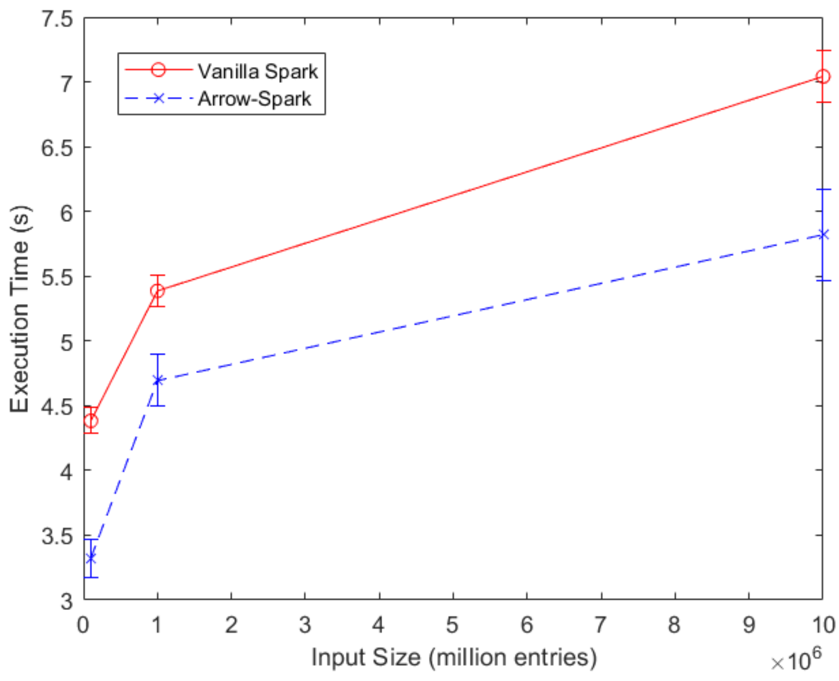

Figure 4.4: Comparing Vanilla Spark and Arrow-Spark execution time for different input sizes

Figure 4.4 shows the direct comparison of the total execution times between pure Spark (in red, solid line) and the implementation of this work, depicted with a blue dashed line, for the WordCount benchmark for different input sizes. It is possible to note that, on average, Arrow-Spark has more variability from the median execution time more than Spark, which results more stable (and, in addition, it increase with increasing the input size). Nevertheless, Arrow-Spark usually outperforms Vanilla Spark (between 9% and 18%) for the WordCount benchmark.

This performance analysis evaluates the total execution time for the executors, and it takes into account other metrics such as the task de-serialization. It does not consider the scheduling delay, which can be fine-tuned accordingly for different workloads, and it proved to be quite stable for each experiment.

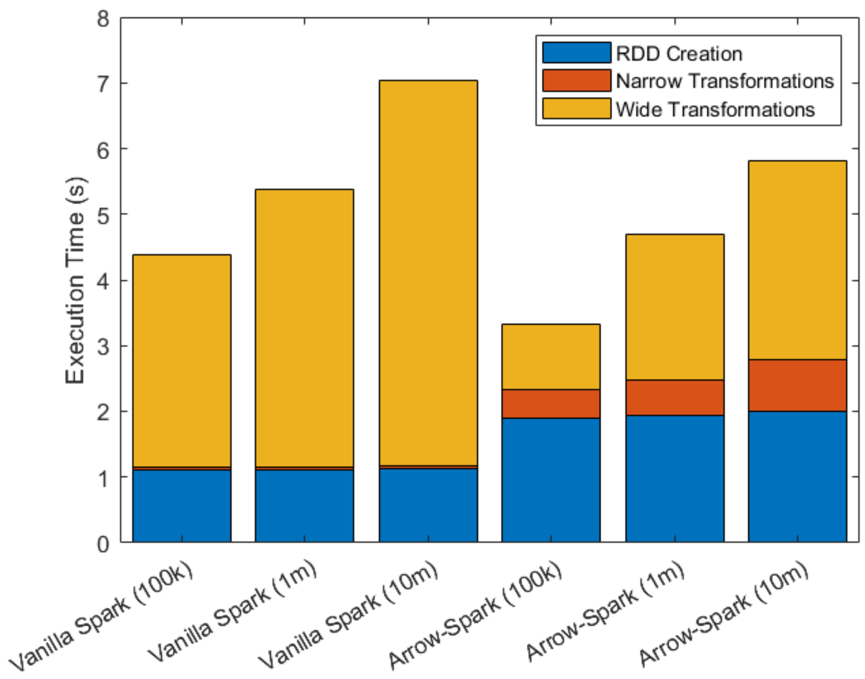

Figure 4.5: Comparing Vanilla Spark and Arrow-Spark execution time, focusing on different tasks being performed

The figure above shows the execution time breakdown for Vanilla Spark and Arrow-Spark, with varying input sizes. It is possible to see how the different tasks take up a different amount of time, depending on the implementation: in Vanilla Spark, wide transformations take up most of the execution time, and the narrow transformations are, usually, negligible in terms of impact. Arrow-Spark, on the other hand, introduces some overhead for RDD creation and narrow transformations, the latter having an increase in the overall execution time when increasing the input data size. This happens because of the increased number of vector accesses and conversions, which negatively affects performance. One possible way to limit the impact of narrow transformations could be devising an efficient partitioning strategy, knowing the input size in advance. As you can imagine, however, such partitioning strategy could have a negative effect on the overall performance, in such a way that the network between the Driver and Worker nodes could saturate when increasing the amount of communication needed to get a result (that is, increasing the partitions too much). Unfortunately, such side-effect of working with Arrow data structures is unavoidable, since it is embedded in the Arrow format itself. Given the overall impact that such performance slowdown has, compared to other big advantages that this implementation gives to Spark execution, it would be advisable to focus the optimization efforts in other parts of the overall execution.
The most visible advantage, as it can be seen in Figure 4.5, is the reduction in execution time for wide transformations (in this case, the reduceByKey): Arrow-Spark can outperform Vanilla Spark by approximately 50%. The columnar format, along with the serialization trick explained earlier, do allow for such performance improvement, especially in the shuffle map stage. This experiment shows the feasibility of Arrow-Spark workloads and the performance advantages that such implementation gives to the overall execution time. Together with the improvements, the performance breakdown in the picture above also highlights some of the limitations of the present work, which will be explained in the following chapter and, hopefully, targeted together in further development. In general, the main benefit of the columnar format have been evaluated, and this work shows that moving to such format is the perfect route to perform overall Spark optimizations.

4.4.2. ScalaSort

The microbenchmark ScalaSort has been used to further evaluate the proposed solution, allowing for a better comparison between Arrow-Spark and pure Spark for different types of workloads. The ScalaSort benchmark, as the name suggests, sorts the data in a given RDD according to a particular ordering, in this case ascending (that is the default case). It is similar to the more popular TeraSort, with the key difference that it does not use Hadoop's TeraInput class as a wrapper for input data, which could not be used for this evaluation (because

the *Text* wrapper on string-based data does not work well with Arrow-Spark data).
The benchmark consists of a series of transformations on a string-based RDD, defined as
`map((_, 1)).sortByKey(true, numPartitions).map(_._1)`
The first transformation creates a `Tuple2` value for each string (given that sorting using this particular function only works with tuple-like data), then it gets sorted and re-partitioned accordingly (in this case, the number of partitions is determined using the default `RangePartitioner`). The other parameter of the sorting function is a boolean, used to indicate whether the sorting has to be performed in an ascending (such as this case) or descending fashion. The final transformation, then, only takes the first value in the tuple, that is, the text value, from the sorted data (since the number in the tuple is not relevant).
In order to be used with Arrow-Spark, the first transformation also includes an extra step, that is
`map(b => new String(b, StandardCharsets.UTF_8))` to allow Arrow-Spark to work with string-based values (exactly like the WordCount experiment).

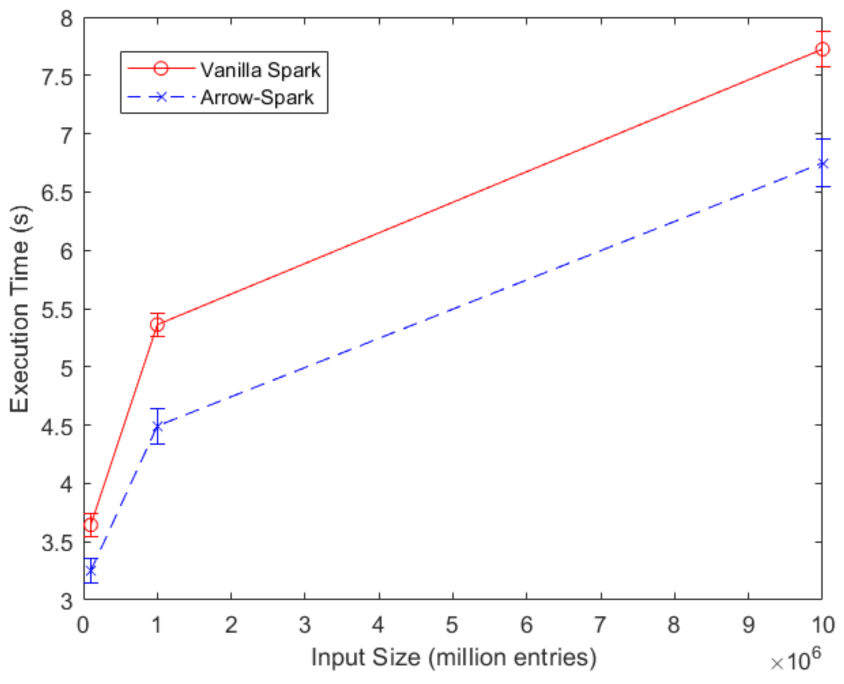

Figure 4.6: Comparing Vanilla Spark and Arrow-Spark execution time for different input sizes (Sort example)

Figure 4.6 shows the the total execution time of the ScalaSort benchmark for Vanilla Spark and Arrow-Spark, with varying input datasets. As for the WordCount example, Arrow-Spark (the blue, dashed line in the figure) is shown to improve the total execution time for this workload, albeit with a smaller improvement compared to the example before. In general, it is possible to see that the variability of the execution times, compared to the median value reported in the picture, is less prominent than the WordCount example, proving once again the robustness of this implementation and its associated performance benefits. Arrow-Spark can, in general, out-perform pure Spark workload between 6% and 15%, showing the advantages of using the columnar format and Arrow vectors in sorting operations. The results of this evaluation include, as before, the task de-serialization time: an in-depth analysis of the results will follow in the next sections.

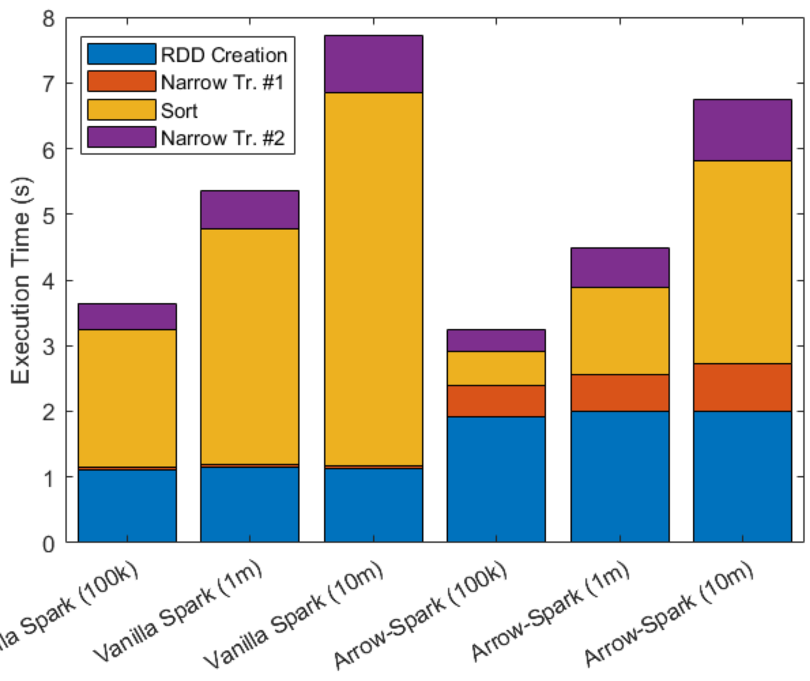

Figure 4.7: Comparing Vanilla Spark and Arrow-Spark execution time, focusing on different tasks being performed

In the picture above, like the WordCount example, it is possible to see the performance breakdown for each individual task performed by Spark and Arrow-Spark for this benchmark. Once again, the RDD creation and narrow transformation times take up longer in the Arrow-Spark example, as discussed earlier. What is interesting to note is, however, how the second `map` transformation has a much bigger impact on Vanilla Spark than the other case, and how, for sufficiently small input datasets (around 100,000 entries), such transformation is actually quicker in the Arrow-Spark case. The execution time of this transformation then increases with the increase in the input dataset, and for more than 1,000,000 entries it is still marginally slower than pure Spark workloads.
In this experiment, it is possible to see how the performance improvement for wide transformations (the sorting, in this case) is slightly less prominent than the WordCount example, but it still out-perform Vanilla Spark by approximately 60%, depending on the input size. Such improvement is in line with what it has been observed before. As already mentioned, this experiment once again proves the feasibility of Arrow-Spark and its performance advantages, along with some inherent disadvantages that this integration brings with it (that will further be analyzed in the following sections and chapter).

4.4.3. Off Heap vs. On Heap

All the experiments have been performed enabling the Spark execution engine to use off heap memory, in order to prove the advantages that using Arrow-based data structures can bring to the execution time. In order to make the comparison more complete, it is good to look at the same comparison being performed while using the standard, on heap memory for the execution.

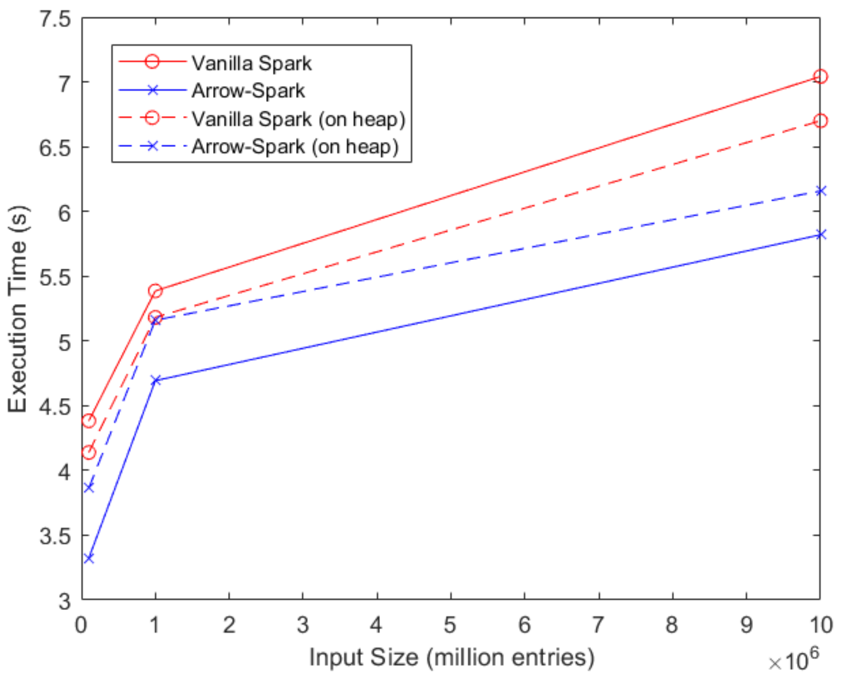

Figure 4.8: Comparing the total execution time for Vanilla Spark and Arrow-Spark, further comparing on heap and off heap memory execution (WordCount)

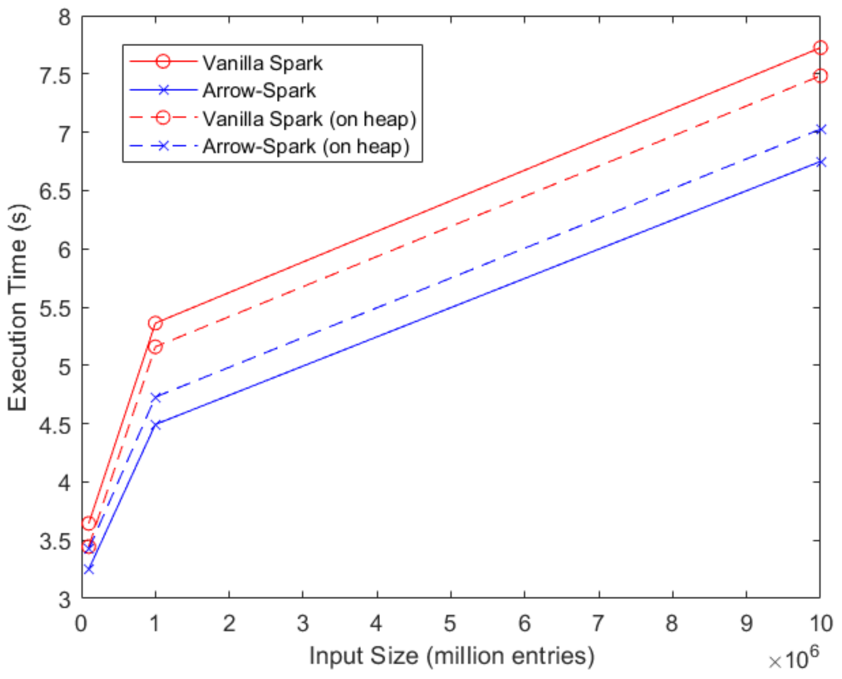

Figure 4.9: Comparing the total execution time for Vanilla Spark and Arrow-Spark, further comparing on heap and off heap memory execution (ScalaSort)

Figure 4.8 and Figure 4.9 show the total execution time for Vanilla Spark (in red) and Arrow-Spark (in blue): the solid lines represent the off-heap execution time, as already presented in the previous section, whereas the dashed line represents the execution time in case of (default) on heap execution. It is possible to note how,

in both benchmarks, Arrow-Spark's performance suffers negatively, while Vanilla Spark has a sudden performance benefit for using this particular memory region. The main reason for Spark's improvement can be see in the amount of serialization required to read and write data to and from the off heap region, and the fact that the row-oriented data format can benefit from being processed in the heap memory region. The garbage collection time has not been taken into account, but it was marginal with respect to the overall execution time. Arrow-Spark, on the other hand, really benefits from being executed in the off heap region: that is due to the columnar format being particularly suitable for such execution, and because the garbage collection in the heap region has a bigger impact than in the case of pure Spark (due to some internal optimizations in the execution engine).
This result shows how Arrow data structures, and the columnar format, can really improve Spark execution time when leveraging the right memory portion. Nevertheless, Arrow-Spark can still outperform Vanilla Spark in both workloads and for all the measured data sizes, albeit with a much smaller performance improvement.

4.4.4. Further Considerations

As already mentioned in the previous evaluations, all the results shown so far do take into account the task de-serialization time for the execution time measurements. That is because, with the executor's compute time, the de-serialization time is the second biggest component that influences the overall execution time. It could be seen from the experiments that such time was giving the most variability in the Arrow-Spark experiments, thus showing how there is still some work to be done to achieve zero serialization in Spark.

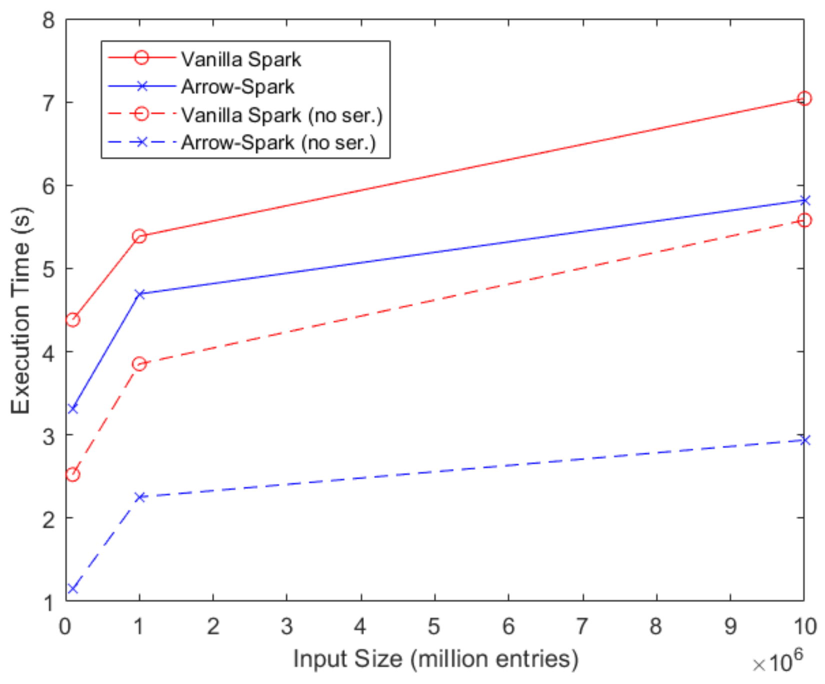

Figure 4.10: Comparing the total execution time for Vanilla Spark and Arrow-Spark, taking into account serialization impact

Figure 4.10 highlights the difference in execution time when taking into account the task de-serialization time, and when considering only the executor compute time: the red line shows the Vanilla Spark execution time for the WordCount example, while the blue line, again, shows Arrow-Spark. The solid line represents the execution time when task de-serialization is factored in, whereas the dashed line represents only the compute time of the executors.
It can clearly be seen that Arrow-Spark gives a big improvement in the executor compute time, but it still suffers from some serialization that reduces the overall performance improvements that could potentially be achieved. From the picture above, in addition, it is possible to point out that the impact of such task de-serialization is bigger for Arrow-Spark than it is for pure Spark, especially in its variability. It has been seen, in all the experiments, that such serialization time really changes from execution to execution, making Arrow-Spark a little bit more unpredictable in terms of total execution time (while still being more performing

than pure Spark for both workloads evaluated). This impact can be explained as follows: in this implementation, Arrow-Spark still heavily relies on the default Spark tasks, which include the RDD and the function to be executed on it, and that are carried over to the worker nodes to be executed. At this stage, the task is then de-serialized and made ready to be executed. As mentioned in the previous chapter, this work "tricks" the Spark serializer by using the *Externalizable* methods, which make Arrow data structures available to be passed around in the cluster without any major changes in the execution engine. Removing this component in future works, and creating a proper Arrow engine (with potentially disruptive changes to the standard execution engine), could greatly improve the overall performance of Arrow-Spark workloads. Focusing only on the compute time for executors, the performance improvement can reach approximately 50% in the total execution time.

4.5. Offloading Functionalities to Arrow

In the previous chapter, the possibility to offload functionalities to Arrow has been presented, and it shows an alternative to using Spark-based functionalities where an Arrow-based implementation can be used and performed directly on Arrow vectors. In order to prove the feasibility of this solution, an evaluation has been performed with three different methodologies being compared with each other: the experiment consists of finding the minimum value of an Integer-based RDD, where three different methods have been compared. The first method is the standard `min()` operation on a normal Spark RDD, created using the `parallelize` method with different integer sequences (starting from 1,000,000, up to 10,000,000 entries). The second implementation uses the same `min()` function, this time over an `ArrowRDD`, created using a Parquet file with a single integer vector with the same number of rows. The third method, that is what this work focuses on, is the `vectorMin()` implementation, in which the minimum value for each vector is calculated in the vectors themselves, then pushed to the driver program. The same Parquet files have been used in this case, too.

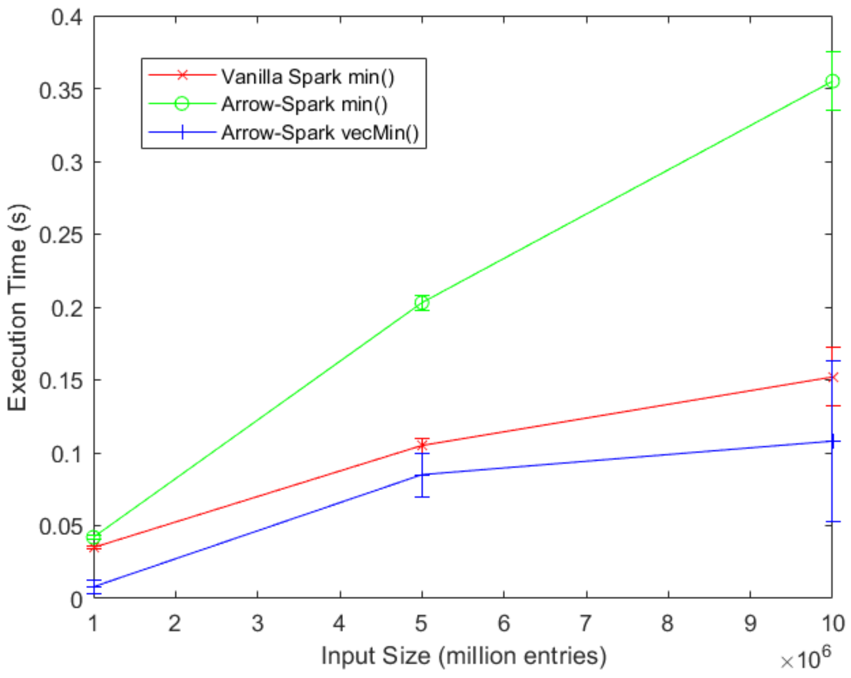

Figure 4.11: Comparing different implementations and design for finding the minimum value in an RDD

Figure 4.11 shows the result of this evaluation, where each implementation has been tested with different input sizes, and the total execution time noted down. The proposed implementation proves to be beneficial, especially when compared to the standard implementation on Arrow-based RDD: the reason for such slowdown is due to the function accessing the whole vector and compare it value-based, while the proposed design uses, on the other hand, built-in algorithms (in the *vector* package) to sort the vector with zero-copy, then returning the first value of each vector slice. This is then compared with the other values and returned.

It is interesting to note that the proposed design drifts severely when increasing the input vector size, more than the Vanilla Spark default implementation.
On average, the proposed design out-performs Vanilla Spark by around 20%, and such performance increase can also be related to using off-heap memory allocation for the experiments. This method has been proven beneficial for Arrow data structure, while Vanilla Spark suffers a slight performance degradation using this memory region.

5 Conclusion and Future Work

5.1. Conclusions

This work's primary goal is to answer the following research question **"Can Arrow in-memory data format be leveraged to improve the performance of Spark's core package and RDD computations, achieving zero-copy and no data serialization?"**, highlighted in Section 1.3, by proposing a proof of concept implementation and highlighting the challenges that arose from it. Given the size and generality of such question, the following sub-questions have been created and answered in depth:

1. *Is Apache Arrow a feasible, near-native memory representation format that can be used for Spark's core functionalities, and do the two frameworks integrate without the need of native code?*
   This question was answered by analyzing the Arrow format and its Java-based API, highlighting what was required to be included into Arrow before it could be integrated with Apache Spark. The present work provides a Java-based Parquet reader that does not require native code calls (through JNI), thus reducing the overall complexity of this solution. Furthermore, such reader can also be used to perform an efficient conversion between a Parquet schema to Arrow, allowing this input data format to be used and leveraged in any subsequent workload. In order to efficiently work with string-based values, a new type of Arrow vector has been implemented, that can fully integrate with the existing Arrow ecosystem and leveraged when performing Spark transformations.
   Section 3.3 illustrates the changes required to integrate Arrow vectors within the Spark RDD ecosystem, and the subsequent evaluation with different benchmarks and use-cases proves that such integration is feasible, albeit with certain limitations that will be presented in the following section. Overall, integrating Arrow with Spark is proven to be feasible, and such integration can be achieved without resorting to native code calls.

2. *Does the introduction of Apache Arrow affect the scalability of Spark applications?*
   The proposed implementation has been evaluated against a number of different workloads and scenarios, and compared with pure Spark workloads of the same type. The evaluation results show that Arrow-Spark can on one hand improve the efficiency for such workloads, but it still gives some disadvantages when increasing the input data size. Overall, the performance improvement is there, and this integration is not affected by scaling out Spark workloads, since the experiments proved to be successful for different configurations.

3. *Can Spark-based functionalities be pushed into Arrow, and is there a competitive advantage in doing so?*
   This work focused part of its attention into the feasibility of offloading some functionalities to Arrow, instead of using the default Spark implementation. Section 3.4.3 includes the changes being performed to allow for such sub-question to be answered, with the results of the subsequent evaluation being highlighted in Section 4.5. The proposed implementation aims at offloading finding the numerical minimum in integer-based RDD's, and the implementation used in this thesis proves to give approximately 20% speedup in performance, compared to the default implementation.

4. *Is the performance of such integration competitive, and what are the associated bottlenecks and limitations of it?*

The proposed implementation has been evaluated and compared to Vanilla Spark workloads, and the results show a performance improvement on average of 12% for the total execution time. The advantages of the current work can be clearly seen in the shuffle stage, where the Arrow columnar format can out-perform Spark by approximately 50%. This evaluation shows that the proposed design is a good starting point for Spark optimizations, its feasibility and competitiveness. Along with the performance improvements, the evaluation shows the associated shortcomings of this implementation, that are explained in detail in the following section. Among the most noticeable, the requirement of performing non-disruptive changes to the Spark execution engine is proven to reduce the potential performance improvement, since the task de-serialization for Arrow-Spark based workloads impacts the total executor time more than in Vanilla Spark workloads, and with more drift in the results associated with it.

5.2. Future Work and Recommendations

The present work has focused on showing the feasibility of integrating the Arrow in-memory data format with the Resilient Distributed Dataset API in Spark-core, thus reaching a complete integration between Arrow and Spark ecosystems (as the core package was still a missing link between these two frameworks). It proves that such integration is possible, showing the relative advantages of having an Arrow-based Spark execution, from data ingestion in columnar format (using Parquet) to result output.

During the implementation and evaluation of this work, however, more challenges arose and created some improvement points that have to be discussed, in order to put this work in a more mature and competitive stage: this implementation thus serves as a main building block for future solutions, that could aim at minimizing some performance degradation exposed by the new Arrow-based execution, and at further reducing serialization throughout the workload execution. While some of these new challenges and improvement points have already been touched in the Evaluation chapter, it's good to summarize them here, in order to have a clear indication of where to start for future work.

First, a more extended set of data types and number of input vectors can be implemented in the future: this work proves the feasibility of this integration, thus focuses its attention on providing the necessary logic to operate with examples of fixed-width and variable-width vectors, disregarding all potential native Arrow vector types. In addition, so far only two vectors at most could be used to create an ArrowRDD: this is due to the strong typing in Java, which doesn't allow for a generic implementation of Tuples, for example. The solution implemented focuses on Tuple2, but the same logic can be implemented for any tuple available.

As already mentioned, serialization plays a huge role in the overall performance for Spark (and, consequently, Arrow-Spark) workloads. This implementation offers an initial way to avoid as much serialization as possible, without eliminating it completely. Therefore, in future work, the possibility to dramatically change the Spark execution engine for Arrow-based RDDs has to be taken into account (such implementation would likely require disruptive changes to Spark's `DAGScheduler`, as well as the implementation of the tasks and resulting stages of execution). Hypothetically, such changes will further speedup execution in case of wide transformations, making Arrow-Spark a serious contender for ready-to-market Big Data solutions in the future. Such changes can also be assisted with a fully-fledged `ShuffledArrowRDD` implementation, which would require a complete change in the transformation logic to exploit Arrow-backed vectors to full potential.

Such changes can potentially limit the impact of the necessary slowdown incurred by narrow transformations, where vector conversion (and vector access) play an important role and negatively impact the overall execution time, thus achieving competitive performances for huge input datasets (with a row count greater than 10,000,000).

This implementation focuses on a first integration between Arrow and Spark in the core RDD API, but it doesn't get evaluated with hardware accelerators. Following the ongoing research at the ABS group from TU Delft, it will be beneficial to see whether some parts of the execution can actually be accelerated using FPGA-based frameworks, such as Fletcher[29], to fully leverage the Arrow columnar format during code execution. This can also show the greater potential of pushing more and more functionalities to Arrow, as the columnar format can be perfectly integrated in purpose-built hardware accelerators, in order to fully reduce overall execution time.

A
Measurement Results

This appendix contains various tables with the exact measurements from the experiments of chapter 4. The experimental setup is the same as explained in Chapter 4.1. The implementation of these experiments is available in the repository of this work, under the section "evaluation" [1].

A.1. Parquet to Arrow Conversion

This section refers to the evaluation of the Parquet to Arrow conversion implementations presented in Chapter 3.1.1 and evaluated in Chapter 4.2. Each implementation has been evaluated 10 times, then the results averaged out to provide a final and robust evaluation. The old implementation is based on A.Trivedi's work [2]. The current implementation (which will be used for each of the experiments performed in the Evaluation chapter of this work) is evaluated for different input dataset sizes, in terms of number of rows in the file. This time, only the aggregate execution time is measured, averaging out 20 runs for each dataset.

Metric	**A. Trivedi Implementation**	**Current Implementation**
Parquet Read	1.561	1.154
Schema Conversion	0.723	0.419
Vector Population	0.566	0.468
Total	**2.850**	**2.041**

Table A.1: Comparing the total reading and conversion time between the old implementation, and the one proposed in this work, which leverages built-in Parquet functionalities for Arrow conversion

Input Size	100k	1m	10m
Execution Time	0.886	1.402	2.041

Table A.2: Comparing the total execution time for the Parquet-To-Arrow conversion for different dataset sizes.

A.2. Transformations comparison

This section reports the evaluation measurements for the vector and value conversion strategies, as well as the results of the comparison between Vanilla Spark and Arrow-Spark narrow transformations.
The results only show the total executor compute time, leaving out the task de-serialization time from the comparison. This can show a more in-detail comparison between the different strategies involved and the comparison between the different workloads, reducing the unpredictability of the serialization, serving as a building block for future development.
Each experiment has been repeated 20 times, averaging out the median value obtained from the logs.

[1]https://github.com/fedefiorini/Arrow-Spark-Engine/tree/main/src/main/scala/nl/tudelft/ffiorini/evaluation
[2]Available at: https://gist.github.com/animeshtrivedi/76de64f9dab1453958e1d4f8eca1605f

Input Size	100k	1m	5m	10m
Vanilla Spark	0.013	0.013	0.014	0.014
Arrow-Spark (value)	0.003	0.015	0.077	0.155
Arrow-Spark (vector)	0.003	0.022	0.092	0.180

Table A.3: Execution time of a narrow transformation, comparing Vanilla Spark with value and vector conversion in ArrowRDD

A.3. Benchmarks

This section reports the evaluation results of the WordCount and ScalaSort benchmarks, where Vanilla Spark and Arrow-Spark have been compared to each other. Each experiment reports the total execution time as well as the execution breakdown, to further highlight the advantages (and disadvantages) of the proposed implementation. The total execution time is then compared when the task de-serialization is not factored in. The experiments have been repeated 20 times, averaging out the median value obtained for the logs. When considering only the executor compute time (removing task de-serialization), each experiment has further repeated 5 more times, keeping the average of the median value obtained from each experiment.
While considering the on-heap execution time, the experiments have been repeated 20 times, averaging out the median value obtained from the logs, just as in the off heap case.

A.3.1. WordCount

Metrics	Vanilla Spark			Arrow-Spark		
–	100k	1m	10m	100k	1m	10m
RDD Creation	1.100	1.107	1.121	1.889	1.928	1.996
Narrow Transf.	0.052	0.053	0.058	0.435	0.558	0.800
Wide Transf.	3.321	4.226	5.863	0.998	2.207	3.024
Total	**4.383**	**5.386**	**7.042**	**3.322**	**4.693**	**5.820**

Table A.4: Comparing the execution time breakdown for Vanilla Spark and Arrow-Spark for the WordCount benchmark, factoring in task de-serialization

Input Size	100k	1m	10m
Vanilla Spark (off heap)	4.383	5.386	7.042
Arrow-Spark (off heap)	3.322	4.693	5.820
Vanilla Spark (on heap)	4.139	5.182	6.700
Arrow-Spark (on heap)	3.863	5.160	6.158

Table A.5: Comparing the execution time breakdown for Vanilla Spark and Arrow-Spark for the WordCount benchmark, executed using on heap memory only

Input Size	100k	1m	10m
Vanilla Spark (ser.)	4.383	5.386	7.042
Arrow-Spark (ser.)	3.322	4.693	5.820
Vanilla Spark (no ser.)	2.525	3.853	5.580
Arrow-Spark (no ser.)	1.150	2.255	2.938

Table A.6: Comparing the total execution time for Vanilla Spark and Arrow-Spark for the WordCount benchmark, with and without the impact of task de-serialization

A.3.2. ScalaSort

Metrics	Vanilla Spark			Arrow-Spark		
–	100k	1m	10m	100k	1m	10m
RDD Creation	1.103	1.155	1.127	1.922	1.992	1.998
Map 1	0.045	0.045	0.050	0.475	0.575	0.722
Sort	2.086	3.580	5.666	0.522	1.326	3.084
Map 2	0.410	0.582	0.882	0.332	0.600	0.945
Total	**3.644**	**5.362**	**7.725**	**3.251**	**4.493**	**6.749**

Table A.7: Comparing the execution time breakdown for Vanilla Spark and Arrow-Spark for the ScalaSort benchmark

Input Size	100k	1m	10m
Vanilla Spark (off heap)	3.644	5.362	7.725
Arrow-Spark (off heap)	3.251	4.493	6.749
Vanilla Spark (on heap)	3.445	5.159	7.484
Arrow-Spark (on heap)	3.423	4.728	7.027

Table A.8: Comparing the execution time breakdown for Vanilla Spark and Arrow-Spark for the ScalaSort benchmark, executed using on heap memory only

A.4. Arrow-based Functionalities

This section reports the evaluation results of offloading functionalities to Arrow, specifically calculating the minimum value in an RDD. The evaluation has been performed comparing the proposed design, offloading such computation to the vector themselves, with the default Spark implementation. This default implementation has been performed on a normal RDD, created starting from a Scala sequence of integers, and on an ArrowRDD, where the data has been fed to the application through custom-made Parquet file with an integer vector. The experiments have been repeated 10 times for each input data size, then the median value from each has been averaged out and reported here.

Input Size	1m	5m	10m
Vanilla Spark min()	0.035	0.105	0.152
Arrow-Spark min()	0.042	0.203	0.355
Arrow-Spark vecMin()	0.008	0.085	0.108

Table A.9: Comparing the execution time of finding the minimum value of an RDD, with default Spark implementation against the proposed design

Bibliography

[1] Apache arrow documentation, . URL https://arrow.apache.org/docs/index.html#. (Visited on: 2021-10-24).

[2] Apache arrow (website), . URL https://arrow.apache.org/. (Visited on: 2021-10-24).

[3] Arrow java api (documentation-website), . URL https://arrow.apache.org/docs/java/vector.html. (Visited on: 2021-10-25).

[4] Apache drill website. URL http://drill.apache.org/. (Visited on: 2021-10-26).

[5] Spark mllib: Spark's scalable machine learning library. URL https://spark.apache.org/mllib/. (Visited on: 2021-10-28).

[6] Apache parquet website (documentation). URL https://parquet.apache.org/documentation/latest/. (Visited on: 2021-1).

[7] Cluster operations in apache spark, . URL https://spark.apache.org/docs/latest/cluster-overview.html. (Visited on: 2021-10-25).

[8] Apache spark website, . URL https://spark.apache.org/docs/latest/index.html. (Visited on: 2021-10-24).

[9] Apache. Apache avro documentation (latest), . URL https://avro.apache.org/docs/current/. (Visited on: 2021-10-30).

[10] Apache. The apache software foundation announces apache arrow as top-level project (blog post), . URL https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces87. (Visited on: 2021-10-25).

[11] Apache Arrow. Arrow columnar format specification (website). URL https://arrow.apache.org/docs/format/Columnar.html. (Visited on: 2021-10-25).

[12] Amazon Web Services (AWS). What is spark? (website). URL https://aws.amazon.com/big-data/what-is-spark/. (Visited on: 2021-10-28).

[13] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016. doi: 10.1109/MC.2016.127.

[14] W. Chen and C. Hung. Accelerating spark sql workloads to 50x performance with apache arrow-based fpga accelerators, 2020. URL https://databricks.com/session_na20/accelerating-spark-sql-workloads-to-50x-performance-with-apache-arrow-based-fpga-accelerators. (Visited on: 2021-10-28).

[15] Yuchen Cheng, Chunghsuan Wu, Yanqiang Liu, Rui Ren, Hong Xu, Bin Yang, and Zhengwei Qi. Ops: Optimized shuffle management system for apache spark. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388160. doi: 10.1145/3404397.3404430. URL https://doi.org/10.1145/3404397.3404430.

[16] Falcon Computing. Blaze source code (github). URL https://github.com/falcon-computing/blaze. (Visited on: 2021-10-28).

[17] Databricks. Project tungsten introduction. URL https://databricks.com/glossary/tungsten. (Visited on: 2021-10-24).

[18] Dremio. Announcing gandiva initiative for apache arrow. URL https://www.dremio.com/announcing-gandiva-initiative-for-apache-arrow. (Visited on: 2021-10-28).

[19] D. Eaton. Turning big data challenges into opportunities with fpga-accelerated computing. URL https://www.datacenterdynamics.com/en/opinions/turning-big-data-challenges-opportunities-fpga-accelerated-computing/. (Visited on: 2021-10-28).

[20] SAS Insights. What is big data? URL https://www.sas.com/en_us/insights/big-data/what-is-big-data.html. (Visited on: 2021-10-29).

[21] Intel. Hibench microbenchmark suite. URL https://github.com/Intel-bigdata/HiBench. (Visited on: 2021-09-20).

[22] Spark Issue Tracking (Jira). Spark-12785 (implemented in-memory columnar format). URL https://issues.apache.org/jira/browse/SPARK-12785. (Visited on: 2021-10-28).

[23] A. Kuntamukkala. Spark cheat sheet. URL https://mas-dse.github.io/DSE230/docs/Spark%20Cheat-Sheets%20(DZone).pdf. (Visited on: 2021-10-25).

[24] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348, 2015. doi: 10.1109/NAS.2015.7255222.

[25] Maximilian Michels. Apache arrow: The hidden champion of data analytics (blog), 2020. URL https://maximilianmichels.com/2020/apache-arrow-the-hidden-champion/. (Visited on: 2021-10-30).

[26] F. M. Nonnenmacher. Transparently accelerating spark sql code on computing hardware, 2020. URL http://resolver.tudelft.nl/uuid:f588ca1d-e4ae-4bf4-96ed-221d483b559d.

[27] Oracle. Java garbage collection basics (tech-guide). URL https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. (Visited on: 2021-10-27).

[28] Kay et al. Ousterhout. Making sense of performance in data analytics framework. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 293–307. USENIX: The Advanced Computing Systems Association, 2015. ISBN 9781931971218. URL https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-ousterhout.pdf.

[29] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277, Los Alamitos, CA, USA, sep 2019. IEEE Computer Society. doi: 10.1109/FPL.2019.00051. URL https://doi.ieeecomputersociety.org/10.1109/FPL.2019.00051.

[30] Rui Ren, Chunghsuan Wu, Zhouwang Fu, Tao Song, Yanqiang Liu, Zhengwei Qi, and Haibing Guan. Efficient shuffle management for dag computing frameworks based on the frq model. *Journal of Parallel and Distributed Computing*, 149:163–173, 2021. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2020.11.008. URL https://www.sciencedirect.com/science/article/pii/S0743731520304159.

[31] Sebastiaan Alvarez Rodriguez, Jayjeet Chakraborty, Aaron Chu, Ivo Jimenez, Jeff LeFevre, Carlos Maltzahn, and Alexandru Uta. Zero-cost, arrow-enabled data interface for apache spark, 2021.

[32] J. (Databricks) Rosen. Deep dive into project tungsten: Bringing spark closer to bare metal (slides), 2015. URL https://databricks.com/session/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal. (Visited on: 2021-10-24).

[33] Scala-lang. Classtag (api definition). URL https://www.scala-lang.org/api/current/scala/reflect/ClassTag.html. (Visited on: 2021-11-01).

[34] Apache Spark. Companies and organizations, . URL `https://spark.apache.org/powered-by.html`. (Visited on: 2021-10-29).

[35] Apache Spark. Apache spark history, . URL `https://spark.apache.org/history.html`. (Visited on: 2021-10-25).

[36] JanBask Training. What is apache spark? (blog). URL `https://www.janbasktraining.com/blog/what-is-spark/#2`. (Visited on 2021-10-29).

[37] Vandana, Velamuri Monica, and Narendra Kumar Parambalath. Shuffle phase optimization in spark. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1028–1034, 2017. doi: 10.1109/ICACCI.2017.8125977.

[38] VTeams. Spark: An ultimate tool for data engineering (blog). URL `https://vteams.com/blog/most-popular-tool-for-data-engineering/`. (Visited on: 2021-10-29).

[39] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283, 2016. doi: 10.1109/BigData.2016.7840613.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[41] Matei A. Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.