Universiteit Leiden

Master Computer Science

Investigating End-to-End Arrow-based operations in Apache Spark

Name:	Mariska IJpelaar
Student ID:	s1961659
Date:	August 5, 2022
Specialisation:	Advanced Computing and Systems
1st supervisor:	Dr. A. Uta MSc
2nd supervisor:	Prof. Dr. Ir. N. Mentens

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In the last few years humanity generated unimaginable amounts of data, and will only generate more. Coming up with efficient methods to process all this data is vital to keep up with production. While good progress was made in speeding up network and storage devices, CPU performance stayed behind. Thus, a lot of research focuses on improving data analytics efficiency.

Apache Spark is a popular open-source analytics engine for large-scale data processing. It uses a row-wise format to represent its data. However, many analytical operations benefit more from a columnar format, such as Apache Arrow's in-memory format. While previous work already tried to integrate Arrow into Spark, costly conversions from row-wise to columnar data could not be prevented.

In this thesis, we investigate how one can best integrate Arrow's format in Spark, without overhead from format-conversions. In this study, we analysed how previous work integrated Arrow's format into Spark and what insights we can gather from it. From these insights, we created a Proof-of-Concept called Complete-Arrow-Spark (CAS). Within CAS we implemented a single end-to-end Arrow-based operator, to prove its feasibility.

We solved many challenges to make Spark understand Arrow's format. While we cannot process real-world applications yet, we produced promising results and found many areas of improvement.

Contents

Acronyms

Complete Arrow Spark (CAS)

Proof-of-Concept designed in this work, to integrate Apache Arrow's format into Apache Spark, while avoiding format conversions . 3, 26, 28, 29, 34, 36, 38, 39, 45, 46, 48–51

Directed Acyclic Graph (DAG)

A directed graph without directed cycles [54] 7–9, 38

Domain-Specific Language (DSL)

Computer language specifically designed for a certain domain [55] 11

Field-Programmable Gate Array (FPGA)

User-configurable integrated circuit, consisting of an array of programmable logic blocks and configurable interconnects [56] 11

Inter-Process Communication (IPC)

Mechanisms to allow communication between processes [58] 30, 47

Java Virtual Machine (JVM)

Virtual machine to run java bytecode in order to create portability [59] 21

Online Analytical Processing (OLAP)

System to analyze aggregated data from OLTP with complex queries [39] . . . 4, 13, 14

Online Transaction Processing (OLTP)

System to capture, store and process data from transactions [39] 4, 13, 14

Resilient Distributed Dataset (RDD)

An abstraction used by Apache Spark for cluster computing [64] . . 3, 7–10, 15, 21, 30, 31, 35, 36, 38

Single Instruction Multiple Data (SIMD)

Model to apply a single instruction on a multitude of data [61] 5, 11, 14, 15

User-Defined Function (UDF)

Function defined by the user in an environment where functions are normally built-in [52] 11

1 Introduction

According to Statista [7], we generated 64.2 zettabytes (10^{21}) of data in 2020, which is equal to 64.2 trillion gigabytes [62]. The Statista Research Department forecasted this amount to grow to more than 180 zettabytes in 2025. While we stored only two percent of the data generated in 2020, we still reached a total storage capacity of 6.7 zettabytes in that year [7]. These numbers show that processing large amounts of data efficiently is vital for keeping up with the total amount of data being generated.

In the previous decade we made good progress in gaining speedups for network and storage devices, however, CPU performance stayed behind [6]. Ousterhout *et al* [31] discovered that CPU power has become the bottleneck in modern data analytics frameworks. Thus, it has become important to invest in methods to make performance of analytics engines better. It is not surprising that in 2020, 47 percent of organisations reported to use 'Big Data Analytics' as a research method [9].

One such analytics engine is Apache Spark, a popular open-source unified analytics engine for large-scale (clustered) data processing. Its strength lies in its data parallelism and fault tolerance with the use of its *in-memory cluster computing* ([27], [53]). In 2019, Apache Spark was even seen as the "most important big data infrastructure technology" [8].

Spark's main representation for data is a row-wise format. While this format is useful for many types of operations, analytical workloads often benefit more from a columnar one [1]. The Apache Arrow project created such a columnar format. Arrow is an open-source project from the Apache Software Foundation, which defines a language-independent in-memory columnar format [40]. From Apache Arrow, a distributed scheduler called Ballista is currently at an early stage of development[1].

In previous work, others tried to integrate the Arrow format into the Spark engine. Rodriguez *et al* [33] used the Arrow Dataset API to add a connector between Apache Spark and Arrow-based data-sources, and analyzed how to tune several parameters for this framework. They concluded their framework was as fast as, or sometimes even faster than, vanilla Spark. Nonnenmacher *et al* [29], used the same Arrow Dataset API to create a Proof-of-Concept implementation to offload work from Spark to hardware accelerators based on Arrow. They also investigated the maturity of Spark's columnar processing with Apache Arrow. They found that Spark uses Arrow internally to exchange data with Pandas in PyArrow, but did not make the API public ([16], [14]). Additionally, the authors concluded that converting the Arrow format to Spark's row-based format is very costly and should be avoided.

During this work, we aim to answer the following research question: **How can one optimally integrate Apache Arrow's format in Spark without the need for format-conversions?**

In order to answer this question, we also investigate the following sub-questions:

RQ1. What can we learn from previous attempts at integrating Arrow's format into Spark?

RQ2. Is it feasible to create a Proof-of-Concept using the insights we gathered from previous experience?

[1] `https://arrow.apache.org/blog/2021/04/12/ballista-donation/`

To answer these questions, we make the following contributions:

i We analyze how a different setup impacts the performance of Native SQL Engine, a project by Intel to optimize Apache Spark through Apache Arrow integration.

ii We investigate the possibility of extending SpArrow [15], A Spark-Arrow engine focused on only the core-functionalities of Spark.

iii From the experience of previous work, we create a Proof-of-Concept called Complete Arrow Spark (CAS), analyse how it performs and how it can be improved.

In Section 2 we start by describing various terminologies and concepts required to understand the rest of this thesis. Additionally, we discuss related work in this section. In the following two sections we describe and evaluate the previous works that integrated Arrow's format in Spark. In Section 5, we explain our design for CAS in detail, and discuss some experiments that we performed to evaluate its performance. Finally, in our last Section, we conclude our findings and discuss future work to improve CAS.

2 Background and Related Work

2.1 Parallelism

Parallelism is a type of computation in which multiple tasks are performed at the same time [60]. When working with large amounts of data, parallelism is crucial to ensure running times remain manageable. We have two main types [19]:

1. **Horizontal Parallelism**: Here, data is partitioned and tasks are executed simultaneously on each partition. This type of parallelism is also referred to as **Data Level Parallelism** (in the context of a computer program) or **Intra-Operator Parallelism** (in the context of query processing).

2. **Vertical Parallelism**: Here, multiple tasks are executed simultaneously, often connected with each other such that tasks use each others output as their input. This type of parallelism is also called **Instruction Level Parallelism** (in the context of a computer program) or **Inter-Operator Parallelism** (in the context of query processing).

A great example which combines these two types of parallelism is `MapReduce`. Hadoop MapReduce is a software framework to process large amounts of data in-parallel. This framework consists of three major stages: **map**, **shuffle** and **reduce** [46].

Figure 1 shows an example of `WordCount`, a MapReduce application which counts the frequency of words in a text. In the map stage, the input is partitioned and saved as a `key-value` pair. In the example, the key is the word that is counted and the value is always the number '*one*' representing one occurrence of the word. In the shuffle stage, the pairs are distributed according to their keys. In the example, each key has its own list of 'one-occurrences'. Finally, in the reduce stage, the values are 'reduced' to their expected output. In our example, the lists are converted to an integer representing the number of occurrences of a word.

This framework uses Horizontal Parallelism because it partitions its data and applies its operators simultaneously on these partitions. Vertical parallelism becomes evident once you

realise the different stages can already work on intermediate results, and thus do not have to wait until previous stages have finished in their entirety. In our example, the reduce stage can already start 'counting' the number of occurrences before the shuffle stage has completed [11].

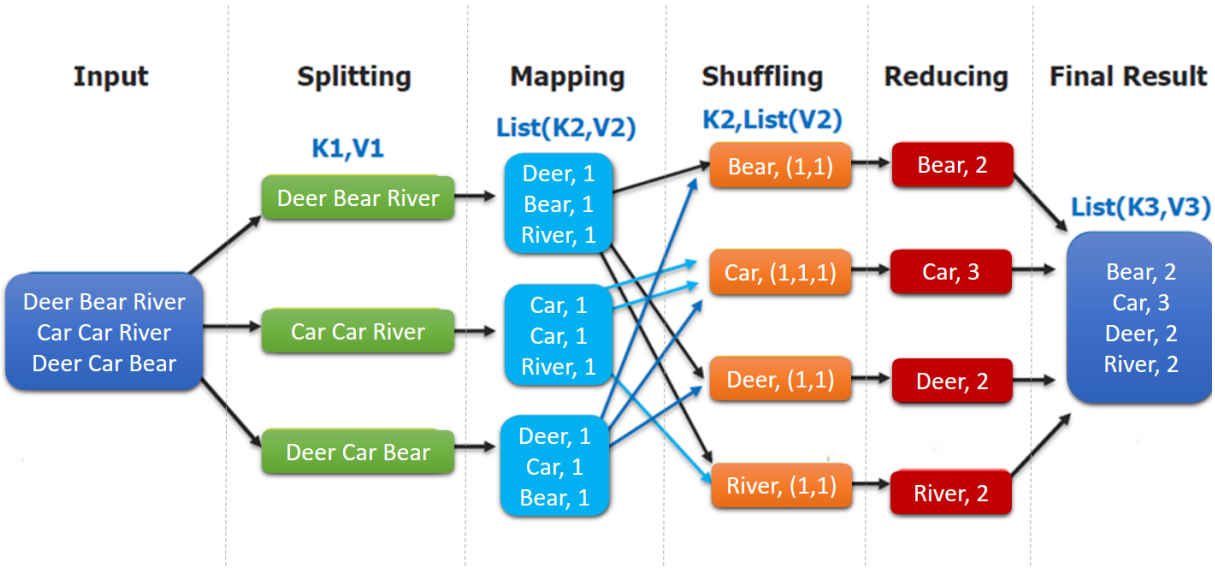


Figure 1: MapReduce Example – WordCount.

Source: Hadoop series 3: distributed computing framework MapReduce (FatalErrors [26])

2.2 Columnar and Row-Wise Databases

Traditionally, data was stored in relational tables using a row-wise format. This format was suitable for small amounts of data or small and frequent operations. However, as the amount of data and its type of operations changed, data in row-wise format became a bottleneck for certain applications and workloads. From the early 1970s onward, people started using Columnar Databases to alleviate this bottleneck [18]. In these databases data is stored in columns instead of rows. Figure 2 shows an example of a table saved both in row-wise format (left) and columnar-format (right).

Which format is suitable for data often depends on the type of operations that you perform on that data [1]. For example, many small `insert` operations will benefit from a row-wise format as the data can simply be appended. However, for columnar-data inserting a single row would result in multiple inserts (one for each column). Conversely, computing the average of a column will benefit from columnar data as all the required values are grouped together. For row-wise data such an operation would require access to widely-spread values. Another important consideration for choosing between these formats is compression. Since columnar data has many values of the same type grouped together, we can more easily compress this data, resulting in disk I/O improvements and less space needed for storage [1].

In essence, row-wise data is suitable for workloads that access a few rows across many columns, such as Online Transaction Processing (OLTP). These workloads generate many Inserts, Updates and Deletes [39]. A Columnar-format on the other hand is suitable for workloads that access a few columns across many rows such as Online Analytical Processing (OLAP), which are mainly bulk Selects and Joins [39].

	ID	Name	Age
Row1	065	Anna	20
Row2	152	Ben	46
Row3	526	Carlotte	12

065
Anna
20
152
Ben
46
526
Carlotte
12

065
152
526
Anna
Ben
Carlotte
20
46
12

Figure 2: Small Data in a Table saved in both row-wise (left) and columnar (right) format.

One widely used Columnar Database format is `Parquet`, an open-source file format for Columnar Storage [18], which we also use in this work.

2.3 Apache Arrow

Arrow is an open source project from the Apache Software Foundation [40]. It defines a language-independent in-memory columnar format, with bindings for many popular programming languages, including C, C++, Java, Python and more ([40], [41], [42]). Additionally, it provides zero-copy reads for data access without serialization overhead [41]. Its design ensures efficient data transfer between different systems and programming languages [42]. Additionally, Arrow is optimized for cache locality, pipelining and Single Instruction Multiple Data (SIMD) instructions [12]. For example, it supplies SIMD algorithms for various operators and generates tight-loop code to help the compiler optimize [12].

2.3.1 Arrow Architecture – Java

Apache Arrow supports APIs for many languages. While all language bindings share similar concepts, each environment has its own structures and building blocks. In this Section, we focus on the concepts as described in the documentation for the Java API [43].

Internally, Apache Arrow uses the `ArrowBuf` to represent contiguous data. This data resides in 'direct memory' rather than on-heap memory, because this allows other language bindings to directly access the data without copying, among other reasons. This memory is 'off-heap' and thus not managed by the Java Garbage Collector. Instead, Arrow uses reference-counting to know when it should release its buffers. We will elaborate more on memory management later in this section.

Managing these Arrowbuffers manually is often tedious and error-prone. Thus, Apache Arrow provides an abstraction layer called `ValueVector`, which manages the ArrowBuffers. This vector contain two buffers; one which expresses validity of the values (`ValidityBuffer`) and one which contains the actual data (`DataBuffer`). Users of ValueVectors are advised to maintain a documented 'Vector Life Cycle', although this is not enforced. In particular, the life cycle consists of the following stages (in-order):

1. creation: here we define the type and name of the vector.
2. allocation: here we allocate memory for the underlying buffers.
3. mutation: here we populate the vectors. Possibly, this also requires more memory to be allocated.
4. setting value count: we set the value count when we are done writing to it, and want to access it. The values now become available to read-calls.
5. access: here we read the values in the vector.
6. clear: when we do not need to use the vector anymore, we clear/ close it to release its memory.

Note that this order may contain loops (e.g. after access, you may mutate as long as you set the value count before accessing again).

To make a zero-copy slice of the ValueVector, Arrow provides the `TransferPair`, for which you can either transfer ownership, or share/split the ownership.

In general, we use ValueVectors as representation of columns in tabular data. To describe the structure of these columns, we use `Fields`, which contain the name, data-type and other metadata of the column. Additionally, they also describe if the column can have null values. If we want to describe the structure of a table, we combine these Fields in a `Schema`. A Schema contains a collection of Fields and metadata about the table.

As the value-count in ValueVectors is limited [24], it is good practice to work on the data in batches. To accommodate for this, Apache Arrow has a `VectorSchemaRoot`, a container for batches of data. This container keeps the shared information of the batches available, such as the Schema and allocation-information for the buffers. The VectorSchemaRoot is therefore very suitable to combine with streams. To (un)load the data, Apache Arrow provides the `VectorLoader` and `VectorUnloader`.

Since Apache Arrow implements its own reference-counting to manage memory, users should be very careful when managing the references to their ArrowBuffers within their ValueVectors. To deal with memory management, Apache Arrow offers the `BufferAllocator`.

This is an interface for allocating and freeing memory, as well as keeping track of currently allocated memory and allocation limits. Both ArrowBuffers and ValueVectors are associated with a BufferAllocator. BufferAllocators should be closed at the end of their usage, to check if there were any memory leaks. If there are, the BufferAllocator will report errors. Additionally, Arrow uses the `ReferenceManager()` to track the reference count. Each ArrowBuf is associated with a ReferenceManager. Whenever a ValueVector is closed, it will release it reference from the ReferenceManager, which in turn decrements the reference-count by one. If the reference-count reaches zero, the ReferenceManager notifies the associated BufferAllocator, such that memory may be released. Note that this BufferAllocator may be different from the allocator of the ValueVector who issued the last close. Arrow also offers a tree-based model for its allocators. They recommend to use one `RootAllocator` throughout the whole program, and create its children by calling `.newChildAllocator()`. For each child, we can configure another maximum allocation size, so we can adapt the allocators to different parts of the program. Aditionally, whenever we close a `BufferAllocator`, its children are also reported upon. Thus, we can have a better idea of where certain memory leaks occurred, if there are any.

2.4 Apache Spark

Apache Spark has become the most popular open-source unified analytics engine for large-scale data processing ([25], [53]). Many believe this popularity is because Spark is easy to use, fast and provides APIs for several popular programming languages such as Java, Scala, Python and R ([25], [20]). On top of this, you can also use it for almost all data storages, and it can be deployed in both cloud and on-premise platforms [20].

Spark gains many advantages from its *in-memory cluster computing*, and allows programmers to use *data parallelism and fault tolerance* paradigms without any extra effort [27]. To achieve this, it uses two main abstractions: **Resilient Distributed Datasets (RDDs)** and **Directed Acyclic Graphs (DAGs)** [27]. While we assume the DAG will be clear to most readers, we explain the RDD and various other components of Spark in the following subsections.

2.4.1 Spark Architecture

Before we can elaborate on the Resilient Distributed Datasets (RDDs), we first need to cover the general steps in a Spark application.

As shown in Figure 3, Spark works with one central coordinator and several distributed **Workers**. Within the cluster we have multiple components each with their own responsibilities and tasks:

- **Cluster Manager**: is responsible for allocating and managing the resources of the cluster [27]. Several types are supported, in particular these are: Standalone, Apache Mesos, Hadoop YARN, Kubernetes [45].
- **Executor**: is a process, launched by the Cluster Manager, that runs on a Worker node to execute tasks [36]. The Executor is also responsible for managing the data in memory or on-disk [45]. A Worker node can have multiple executors, but each executor only runs on one Worker node [36].
- **SparkContext**: is responsible for creating and distributing tasks over Executors [45].

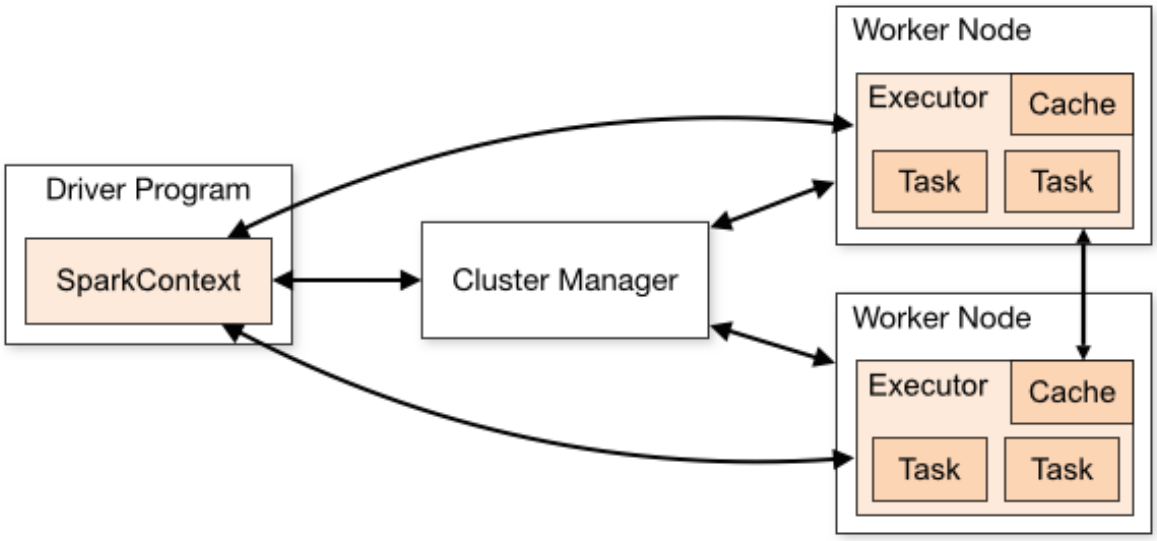


Figure 3: Spark Cluster Overview.

Source: Cluster Mode Overview (Apache Spark Docs [44])

- **Driver Program**: is responsible for setting up a SparkContext and monitoring the Executors ([27], [36]). The Driver Program always runs on the central coordinator (Master node) [37].
- **User**: is responsible for registering the Driver program to the Cluster Manager. This can either be done by invoking the `spark-shell` or by writing code through the Spark API for the supported programming languages [27].

The Workflow of running a Spark application on a Spark cluster is as follows [27]:

We assume a Spark cluster with a Cluster Manager is already running and that the User wrote a program with the intention of running it as a Driver Program. The User first registers the Driver Program to the Cluster Manager and the Cluster Manager starts its at the Master node. The Driver Program creates a SparkContext which converts the user code to a Directed Acyclic Graph (DAG), containing all operators that need to be performed. This DAG is referred to as a **Logical Plan**. Additionally, the SparkContext performs optimization techniques on the Logical Plan. Then it creates a **Physical Plan** which contains more details of how execution should occur. For example, the Physical Plan states the specific algorithm to use. From this plan, the Spark Context creates tasks to be executed by the Executors [37]. Once the SparkContext knows which tasks to run, it start communicating with the Cluster Manager to allocate the required resources. In particular, the Cluster Manager launches the Executors, which stay alive during the complete lifetime of the application. The Executors then register themselves to the Driver, such that the Driver can manage them. Finally, the Executors execute their tasks and return the result to the Driver.

2.4.2 Resilient Distributed Dataset (RDD)

The two most important features of the RDD are in its name: *Resilient* and *Distributed*.

With *Resilient* we mean the ability to reconstruct data on failure. Spark enforces this by making its RDDs *immutable* and *lazy*. Immutable RDDs require copies for each change, while

lazy evaluations ensure that the operators are not immediately executed, but stored in the Directed Acyclic Graph (DAG). Additionally, Spark saves all intermediate results. If data gets lost, Spark can efficiently reconstruct the data by redoing the stored operation on the parent RDD.

RDDs are *Distributed* as their data is divided over multiple nodes. Spark realises this by creating multiple partitions from each RDD, which are spread over the nodes. Through this, Spark creates *implicit parallelism*, which does not require effort from the programmer.

Users can create RDDs in three ways:

1. With a dataset from external storage. For example, a `csv` file.
2. With an existing collection of data in the driver program. For example, a `List` in Python.
3. By performing an operation, or more specifically a transformation, on an existing RDD. Whenever, a new RDD is created in this way it contains a pointer to the RDD from which it was created, the parent RDD. This chain of dependencies is called **Lineage** [36].

There are two operators users can apply to their RDD.

Transformations are operators that create a new RDD [27]. Examples of transformations are `filter`, `union` and `map` [10]. Transformations can be applied to the whole dataset or to individual elements. However, because of the lazy evaluation feature of RDDs, these transformations are not applied immediately, but saved in the DAG [10].

Actions trigger the transformations to be performed, and return a result of another data type back to the driver [27]. The actual computations are performed in RAM to reduce the overhead of disk reads and writes [10]. Examples of Actions are: `count`, `first` and `reduce` [10]. Figure 4 illustrates the process of transformations and actions.

2.4.3 Spark SQL

Spark SQL is a model of Spark introduced in 2015 by Armbrust *et al* [2]. It integrates relational processing into Spark through a declarative API and contains an optimizer called Catalyst. In particular, it adds the DataFrame API for lazy relation operations. A DataFrame can be considered the same as a Resilient Distributed Dataset (RDD) of rows [29], except that it contains more structural information used for extra optimizations [47].

In version 1.6, Spark added the DataSet API as an extra abstraction layer [29]. In Scala, the DataFrame is defined as a Dataset of Rows [47]. A `Row`, in turn, is the stable representation of a row in a table, while internally Spark uses the unstable `InternalRow` [65].

Additionally, Spark added the SparkSession as a new entrypoint to interact with Spark functionalities, wrapping the SparkContext [21].

2.4.4 Columnar Processing in Apache Spark

Nonnenmacher *et al* [29] described how Spark performs Columnar processing. In particular, they discuss two main abstractions. The `ColumnVector` abstracts one column of in-memory data, and the `ColumnarBatch` abstracts a chunk of data, within one partition, by combining multiple ColumnVectors. They also explain that one specific implementation of the ColumnVector is the `ArrowColumnVector`, based on Apache Arrow.

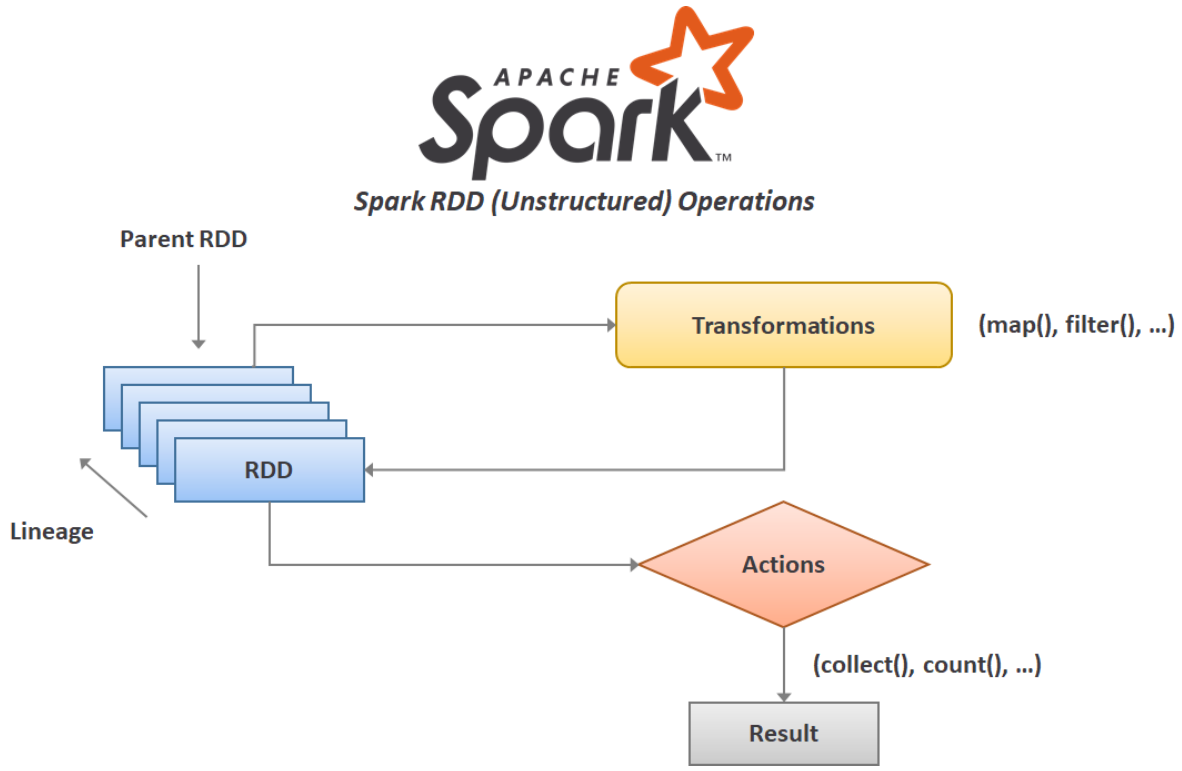


Figure 4: Spark RDD Operation Schema.

Source: Spark RDD (Low Level API) Basics using Pyspark (Medium [22])

In fact, one of the sub-questions answered by Nonnenmacher *et al*, was aimed at the feasibility of integrating Apache Arrow's format in Spark. The conclusions from this question gave us many insights for our own work. In particular, the authors of the thesis conclude: "Spark's columnar processing functionality does not yet fully integrate with Apache Arrow", mainly because the used structures were kept internal and were unavailable without the use of reflection techniques. Additionally, the paper concludes that converting the Arrow format to Spark's row-based format is very costly and should be avoided.

Insight 1

While Spark implements the ArrowColumnVector as an implementation of an Arrow-based vector, these structures were kept internal. Therefore, Apache Arrow is not yet fully integrated into Apache Spark. Additionally, conversions from this vector to a row-wise format are very costly.

Furthermore, they explain that Spark also extended their SparkPlans and rules, in order to accommodate for columnar processing. In particular, the SparkPlan was extended with the methods `supportsColumnar`, to tell if a plan can perform columnar execution, and `doExecuteColumnar`, to describe how this execution should be performed. Additionally, they added the conversion rules `RowToColumnarExec` and `ColumnarToRowExec`.

2.4.5 Previous Work on Improving Spark

Flare [13] is an accelerator back-end, which (i) generates native code, (ii) compiles queries as a whole instead of in stages, and (iii) optimizes User-Defined Functions (UDFs). The authors compare Apache Spark with 'best-of-breed query engines', and note that there is still a large performance gap. Additionally, they find Spark ineffective for UDFs. They aim to make Spark more performant in these areas, while keeping its expressiveness. With Flare, the authors gain 'orders of magnitude speedups', for both relational workloads and iterative functional processing. One of Flare's main purposes is to compile the entire query plan from the Spark optimizer to native code. To gain better performance with UDFs, Flare makes use of Domain-Specific Languages (DSLs) to turn these black-boxes into optimizable intermediate languages. Additionally, it provides options to bypass some of Sparks mechanisms, such as 'fault-tolerance for shared-memory-only execution'.

Nonnenmacher *et al* [29] aimed at accelerating the query execution of Apache Spark through offloading to other hardware accelerators, such as Single Instruction Multiple Data (SIMD) and Field-Programmable Gate Arrays (FPGAs). The thesis shows a gap between Spark's user-friendly API and the specialized knowledge required to offload work to special computing hardware. The authors investigated a method to reduce this gap, using Spark's internal information. In particular, they created a Proof-of-Concept implementation, which integrates Spark with Apache Arrow's ecosystem.

2.5 Vectorized and Compiled Queries

The classic query evaluation strategy, called the 'Volcano Model' [17], has been around since the 1990s [5]. In this model, a query is implemented as a chain of iterators which all propagate `next()` calls to their subsequent iterator. Additionally, each `next()` call may perform operations on the data. The amount of data processed by the iterators may get reduced through the different stages with various filter operators. This is similar to the width of a Volcano, which is smaller at the top than at the bottom, hence the name 'Volcano Model'.

In this system, we work with 'General Query Execution Code' [48], in which we have many abstractions and branches to process different kinds of queries and data types [23]. This allows us to combine arbitrary operators to process arbitrary data types [23]. This worked well in the past, but once the CPU started to become a bottleneck [31] this model became inefficient by the overhead from virtual function calls, instruction cache misses and branching [23]. Therefore, most modern database systems use either Vectorization [4] or Compiled Queries [28], both state-of-the-art paradigms, but very different systems [23].

In this Section we discuss both these paradigms. Additionally, we examine a case study in which Apache Spark gained significant speedups by changing its query evaluation strategy from the Volcano Model to both Compiled Queries and Vectorization.

2.5.1 Vectorized Execution

The main idea of Vectorized Execution is to apply a single instruction to multiple records of data, similar to the Single Instruction Multiple Data (SIMD) method [49]. Figure 5 visualizes this idea. On the left, it illustrates a 'Normal Instruction' in which a single instruction is applied to a single record of data, and returns a single result. On the right, we find a 'Vector Instruction', in which a single instruction is applied to multiple records of data, to return a multitude of results.

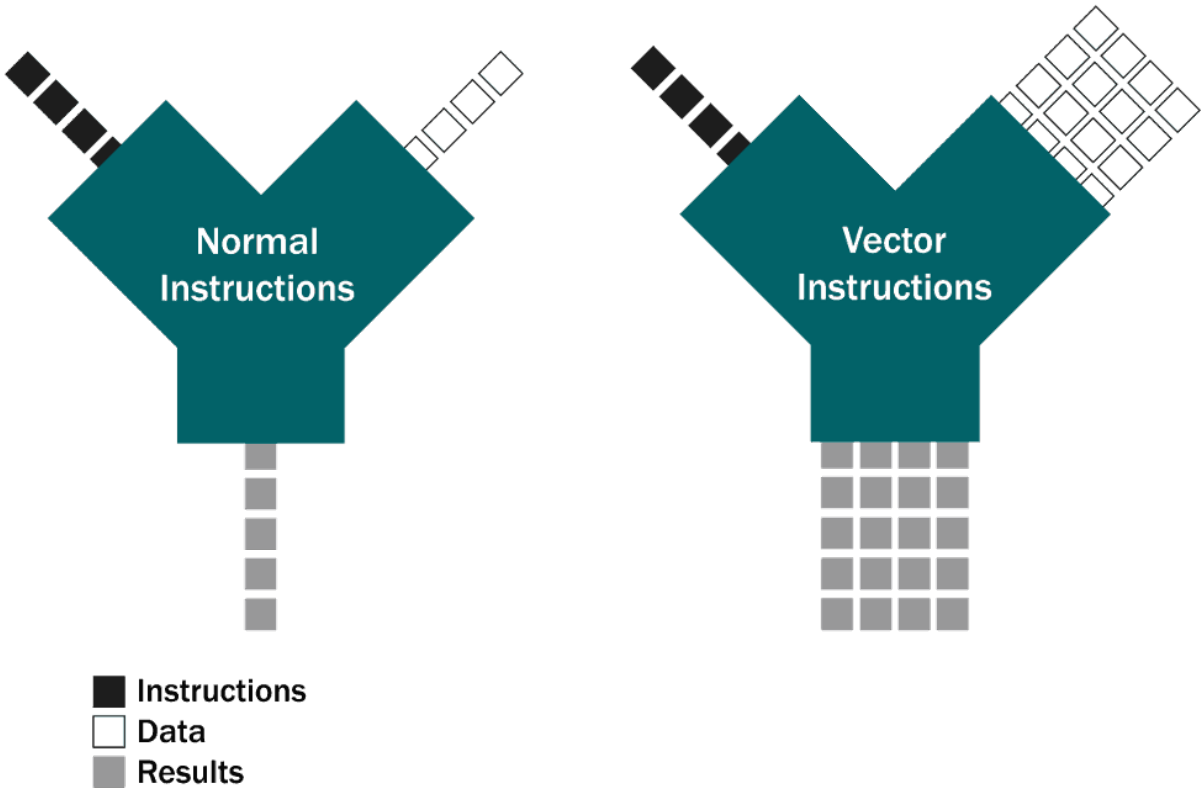


Figure 5: Comparison between 'Normal' (left) and 'Vector' (right) instructions.

Source: Vectorized and Compiled Queries – Part 2 ([50])

Vectorized execution is similar to the Volcano model as each instruction is still performed through iterators [23]. However, the main difference is that multiple records are processed in each `next()`-call, which reduces iterator-overhead and instruction cache misses ([23], [49]). In addition, the compiler can more easily perform optimizations such as loop unrolling [49].

This system is very suitable to be combined with columnar data [49]. Since we only need to load the columns we actually require, we save on I/O when reading from disk, and also improve data availability in caches [49]. Caches also profit from a Columnar format, as these benefit from data that reside side-by-side (spatial locality) [49]. Naturally, Apache Arrow's columnar format is also well suited for Vectorized Execution.

2.5.2 Compiled Queries

Kersten *et al* [23] explained the concept of Compiled Queries. The main idea of Compiled Queries is to implement a program that generates code, tailored to the specific data types and operators. In particular, the generated code should be free of excessive branching, abstractions and function calls, such that non-blocking operators are combined into a single loop [51]. Kersten *et al* further explain that, for this, each operator implements a push-based interface in the form of 'produce' and 'consume' functions. Specifically, a program consists of a 'Query Plan Tree' where the nodes represent the operators. During code generation, the tree is visited in a depth-first traversal. For each first visit the 'produce' function is called and for each last visit the 'consume' function is called. They further point out that with this strategy, the compiler is able to better optimize the code. Additionally, we can keep the data in registers such that fewer (load/ store) instructions are required. Finally, we also eliminate the need for virtual function calls. However, Kersten *et al* also discuss that the generated code is more difficult to comprehend and debug. Furthermore, generating and compiling the queries has some overhead, so the execution of the queries should take long enough to be beneficial. For this reason, they explain, Online Analytical Processing (OLAP) queries are well suited for Compiled Queries, while Online Transaction Processing (OLTP) queries would probably be too short.

2.5.3 Comparing Compiled and Vectorized Queries

Kersten *et al* [23] are the first to build a system in which they compare Compiled and Vectorized queries. They note that this was not easily done before since this requires "many implementation-specific choices". From their experimental results they gathered that both methods are efficient, with similar performance gains. However, they note some differences in areas of strength and weaknesses. We summarize their main results in Table 1.

Sompolski *et al* [38] also compare these two approaches. However, in this paper, the authors focus on finding the best approaches to combine Vectorization with Compiled Queries. To accomplish this, they first analyze the behaviours of these two models in three case studies: Project, Select and Hash Join. They further analyze if merging should be done by using Vectorization in Compiled Queries or the other way around. They conclude that we should always prefer to combine them. In particular, Compiled Queries should be extended with Vectorization, which outperforms both regular Compiled Queries and plain Vectorization. This is especially the case for Compiled Queries, which by itself can only provide marginal improvements, in addition to being more difficult to maintain. Specifically, the proposed approach is better at:

Metric / Query type	Best Approach
hiding cache misses (latencies), e.g. for hash joins	Vectorization
fewer CPU instructions, e.g. for cache-resident workloads	Compiled Queries
calculation-heavy queries	Compiled Queries
Online Analytical Processing (OLAP) workloads	Both
parallel cache misses, e.g. for memory-bound queries	Vectorization
Single Instruction Multiple Data (SIMD)	Both
(morsel-driven) parallelization	Both
Online Transaction Processing (OLTP) workloads	Compiled Queries
language support	Compiled Queries
compile time	Vectorized
easier profiling	Vectorized
adaptivity	Vectorized

Table 1: Vectorization versus Compiled Queries, results from Kersten *et al* [23].

(i) avoiding branch mispredictions, (ii) avoiding CPU cache misses, (iii) SIMD alignment, and (iv) parallel memory accesses.

2.5.4 Case Study: Vectorization and Compiled Queries in Spark

In the past, Apache Spark also made use of the Volcano model as its main query evaluation strategy. For the launch of their second major version, developers of Spark wanted to create a 10x speedup. For this, they looked at extending their query evaluation strategy to also use Vectorized Execution and Compiled Queries [34].

As a reasoning for their approach they considered two versions of an implementation of a SQL Query as shown in Listing 1.

```
01 |      select count(*) from store_sales where ss_item_sk = 1000
```

Listing 1: Example SQL Query.

For this simple query, the Volcano model would require classes for all stages (Aggregate, Project, Filter and Scan), and each stage would need to implement a `next()` method. They further reasoned that if they would ask a college freshman to implement the query from Listing 1, then they would implement something similar to the code in Listing 2. This code is fully tailored to the query, but not composable.

```
01 |      var count = 0
02 |      store_sales foreach { ss_item_sk =>
03 |          if (ss_item_sk == 1000)
04 |              count = count + 1
05 |      }
```

Listing 2: Hand-written Query Implementation.

Figure 6 shows a simple benchmark the developers ran, comparing the handwritten-code with the implementation based on the Volcano model. The developers claimed that the speedup as shown in the Figure is because the handwritten code contains no virtual function dispatches, enables loop unrolling and allows the intermediate results to be kept in registers.

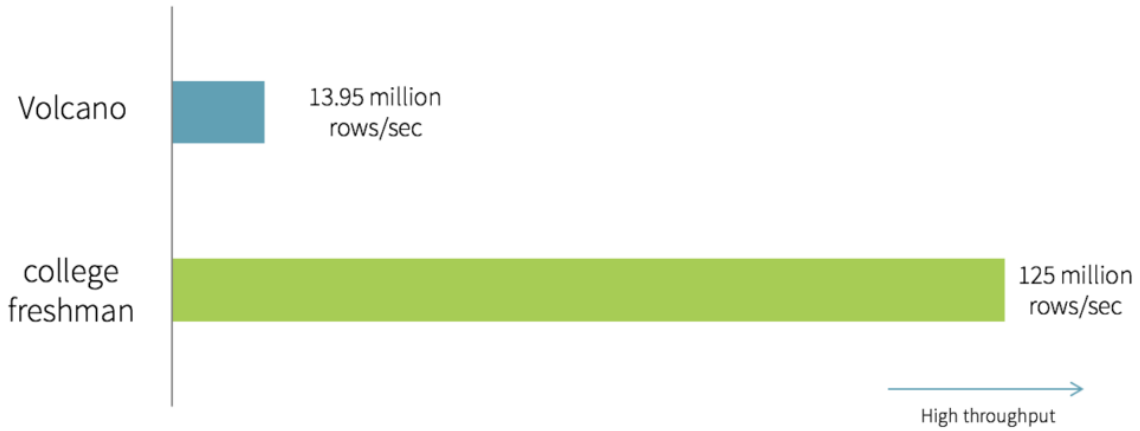


Figure 6: Volcano Model versus Hand-Written code.

Source: Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop (Databricks [35])

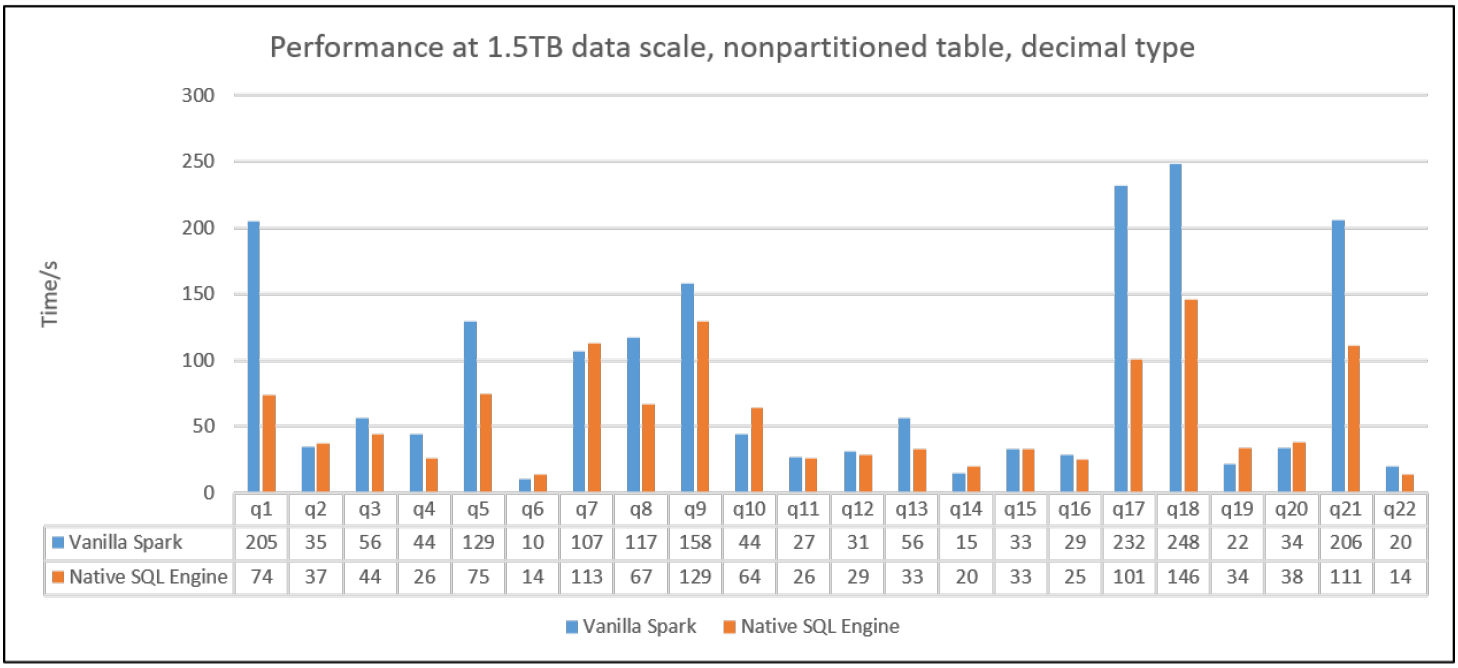
To generate their own handwritten code, they introduced 'whole-stage code generation', as inspired by Thomas Neumann's paper "Efficiently Compiling Efficient Query Plans for Modern Hardware" [28]. Their goal was to achieve the performance of handwritten code, while providing functionality of a general purpose engine. With the 'whole-stage code generation' they transform complete queries into a single function, compiled to bytecode at runtime. This technique works well for many operators and large datasets, but for more complex operators generating this code could be unattainable. For these types of operators they implemented vectorization, by processing batches of rows in a columnar format, instead of single rows. The query is then split into multiple single instructions, which are applied to the batches. However, since this puts intermediate results in memory again, they still prefer Compiled Queries over Vectorization.

3 Native SQL Engine – Evaluation

In this Section, we analyze the first previous work, specifically Native SQL Engine, that integrated Arrow's format into Spark. In particular, we examine the impact on performance when using a different setup than the one used by Native SQL Engine itself.

Native SQL Engine is currently integrated as the `Gazelle Plugin` in the `OAP`-project, an open-source project created by Intel and its community to optimize Apache Spark-SQL [30]. The Native SQL Engine, integrates Apache Arrow into Spark-SQL on three levels:

- Native reader through the Arrow Dataset API, to load columnar data to a Resilient Distributed Dataset (RDD) of ColumnBatches which have their data represented using the Arrow format.
- Offloading of execution work to `C++`-Arrow, to make use of Single Instruction Multiple Data (SIMD) optimizations.
- Columnar-shuffle to allow efficient compression.



	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22
Vanilla Spark	205	35	56	44	129	10	107	117	158	44	27	31	56	15	33	29	232	248	22	34	206	20
Native SQL Engine	74	37	44	26	75	14	113	67	129	64	26	29	33	20	33	25	101	146	34	38	111	14

Figure 7: Performance graph of Native SQL Engine, according to its User Guide[7].

Note that while the developers implemented many operators, not every operator is supported for columnar-execution. Unsupported operators will still require a conversion.

The Performance Section of the user guide[2], contains the result of two experiments, from which we consider only the first. In this experiment, they generated a dataset in parquet files compressing 1.5 TiB of data and a query-set using `spark-sql-perf`[3] and tpch-dbgen[4], based on the TPC-H Benchmark[5]. They ran each generated query on the dataset. For this, they used one master and three workers[6]. Figure 7 shows the results.

For our initial evaluation, we tried to reproduce this experiment, but locally on a single machine. To achieve this, we used the Configuration page[8] and the Performance page[9] provided by the Gazelle Plugin documentations. Since query 21[10] shows a good speedup in Figure 7, we decided to run this query for our evaluation, and compare it with the runtime of vanilla Spark. Figure 8, shows the complete query.

Additionally, we used the following configurations:

- We ran the experiments locally on an `eight-core Intel Xeon D-1548`, with `64 GiB DDR4` RAM. Specifically, we used the `m510` in the `Utah`-cluster from `CloudLab`, a scien-

[2] `https://oap-project.github.io/gazelle_plugin/1.3.0/User-Guide`, accessed at: 2022-07-27
[3] `https://github.com/databricks/spark-sql-perf`, accessed at: 2022-02
[4] `https://github.com/databricks/tpch-dbgen`, accessed at: 2022-02
[5] `https://www.tpc.org/tpch/default5.asp`, accessed at 2022-07-27
[6] Intel(r) Xeon(r) Gold 6252 CPU — 384 GiB memory — NVMe SSD x3 per single node
[7] source: `https://oap-project.github.io/gazelle_plugin/1.3.0/User-Guide`, accessed at: 2022-07-27
[8] `https://github.com/oap-project/gazelle_plugin/blob/master/docs/Configuration.md`, accessed at: 2022-02
[9] `https://github.com/oap-project/gazelle_plugin/blob/master/docs/performance.md`, accessed at: 2022-02
[10] generated by tpch-dbgen (`https://github.com/databricks/tpch-dbgen`, accessed at: 2022-02)
[11] source: (`https://github.com/databricks/tpch-dbgen`, accessed at: 2022-02)

```
select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey = l1.l_orderkey
                        and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists (
                select
                        *
                from
                        lineitem l3
                where
                        l3.l_orderkey = l1.l_orderkey
                        and l3.l_suppkey <> l1.l_suppkey
                        and l3.l_receiptdate > l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = ':1'
group by
        s_name
order by
        numwait desc,
        s_name;
```

Figure 8: SQL-Query 21 from tpch-dbgen[11].

Configuration	Value
spark.master	local[*]
spark.sq.extension	com.intel.sparkColumnarPlugin.ColumnarPlugin
spark.shuffle.manager	org.apache.spark.shuffle.sort.ColumnarShuffleManager
spark.executor.memory	12G
spark.memory.offHeap.size	48G
spark.memory.offHeap.enabled	true
(spark.jars configurations, adding Native SQL Engine jars)	

Table 2: Spark Configurations for Native SQL Engine.

tific infrastructure for cloud computing[12].

- We implemented our driver program in Scala.
- We let Spark use all threads available on the machine.
- We ran the evaluation 30 times.
- For the `tpch`-dataset generation, we cloned a particular branch from `tpch-dbgen`, required by `spark-sql-perf`, using:
  `git checkout 0469309147b42abac8857fa61b4cf69a6d3128a8 -- bm_utils.c`.
- We used a scalefactor of 100 GiB, which resulted in parquet files of about 34 GiB in size.
- We used the Spark configurations as described in Table 2.

Figure 9 shows the results of this experiment. The x-axis shows the running times in seconds for running query 21 for both vanilla Spark (left) and Native SQL Engine (right). In this Figure, we see that vanilla Spark runs the query in about 170 seconds, while Native SQL Engine runs this query in about 177 seconds. These results do not match with Figure 7, where Native SQL Engine is about twice as fast as vanilla Spark. Additionally, we observed that the running times in both experiments are relatively close together, while our dataset size is 15 times smaller than the dataset of the User Guide.

We expect these differences to be caused by the variations in setup. Some queries may be specifically suited for certain hardware. Using older, or just different, hardware may have a negative effect on performance. The effect of the data-size may be two-fold. On the one hand, Native SQL Engine offloads a lot of execution work to native-Arrow. While this may make the execution faster, it also adds more overhead. The gains for smaller datasets could simply be smaller because of this. On the other hand, the amount of work each executor has to perform may be more similar because the larger dataset was distributed, while the smaller one was run on a single machine.

While we are unsure what the exact causes are for the differences, it is clear that the performance results as shown in Figure 7 are not guaranteed for different setups.

[12] `https://www.cloudlab.us/`, accessed at: 2022-07-03

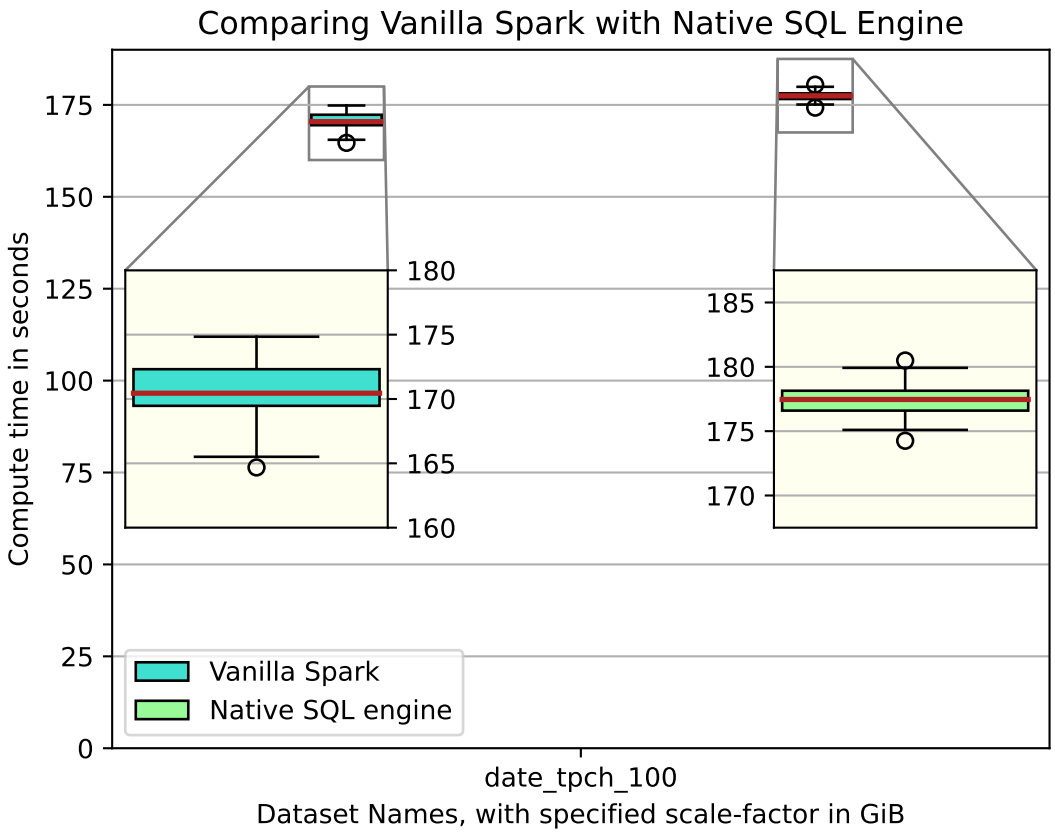


Figure 9: Results of comparing Vanilla Spark (left) with Native SQL Engine (right) on a dataset with size 34 GiB, plotted as boxplots.

Insight 2

The performance gain of Native SQL Engine is not robust against different setups.

4 SpArrow

In this Section, we analyze another previous work that integrated Arrow's format into Spark. "SpArrow: A Spark-Arrow Engine" [15], is a Proof-of-Concept for an integration between Apache Arrow's data format and the compute engine Apache Spark, focusing its design around Spark's Resilient Distributed Datasets (RDDs). Its main contribution has been implementing the `ArrowRDD`, with Arrow's `ValueVectors` as internal data.

In the rest of this Section, we first describe the design of SpArrow. Then, we discuss any limitations of this project which may prevent us from extending it properly. Finally, we evaluate its performance, and draw conclusions.

4.1 Design

In order to manage large amounts of data, Spark uses partitions to process smaller parts of data simultaneously. Similarly, SpArrow uses `ArrowPartitions`, built from an `ArrowRDD` using zero-copy slices from the RDD's ValueVector.

Any transformations on the RDDs fall under two categories: *narrow transformation*s, for which data remains on their current workers, and *wide transformations* for which data needs to be **shuffled** across multiple partitions. However, within SpArrow, there is also an important difference in whether the data preserves its type or not. If the type of the data needs to be changed (e.g. Int to String), then the underlying vectors also need to change. To deal with any of these transformations, SpArrow uses the `MapPartitionsArrowRDD`.

4.2 Limitations

As SpArrow is open source, we tried to use it as a baseline for our work. However, it has several limitations. The author describes these as: a 'limited set of data types', 'limited number of input vectors' and 'serialization could not be prevented completely'. In this Section, we extend the list of limitations with several issues that we identified when experimenting and expanding the SpArrow project.

Compilation Errors –
The first problem we encountered, was that the SpArrow project in the current "Arrow-Spark-Engine" repository does not compile[13]. Most of the compile-issues were difficult to solve and related to dependency management. To focus our work, we chose to only make the project compile for the `IntVector`-type of `ValueVectors`[14].

Scalability –
The SpArrow project author concludes that "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". Indeed we find several configurations ranging from 100k (100,000) to 10m (10,000,000) rows. However, the corresponding largest data file (on the github repository) was smaller than 3 GiB. We believe none of the used

[13] was verified not to compile at 2022-07-03
[14] see our fork: `https://github.com/MariskaIJpelaar/Arrow-Spark-Engine`

datasets were large enough to measure scalability for Spark workloads. Additionally, the usage of 'number of rows' in this way could lead readers to believe that datasets are sufficiently large.

Size Limits –
Once we started working with larger dataset sizes, we found more issues. We realised the most limiting issue was that the ValueVector is constrained in its size. The vector itself can have at most `Integer.MAX_VALUE` values, because it is indexed with `Int`s. Additionally, each ArrowBuf has a 2 GiB limit [24]. Consequently, the ArrowRDD, on which the SpArrow thesis is based, will not be appropriate for real-world workloads.

Furthermore, the SpArrow reader uses a greedy approach, meaning that the whole dataset is always read, even though only a small part may be needed. This is a problem if your file is larger than the amount of memory you have.

As another inconvenience for reading, SpArrow is only able to read in a single file, while parquet data is often distributed over multiple files.

Insight 3

The ArrowRDD of the SpArrow project is limited in the size of the data it can hold. Therefore, the current implementation of SpArrow is not appropriate for real-world applications.

Memory Management –
Within the Java Virtual Machine (JVM), memory management is normally done by the Garbage Collector. However, Apache Arrow uses off-heap memory for their ArrowBuffers. As a memory management strategy they keep a reference on-heap and perform reference counting. The user of the ArrowBuffers is responsible for releasing all references they make ([32], [43]). Unfortunately, SpArrow does not adhere to this convention, which causes the ArrowBuffers to never be released during the lifetime of the application.

Inefficiencies –
Increasing the datasize often reveals which functions are inefficient, as these have the worst impact on performance when scaling. We found `vecMin()` (minimumValue) to be such a function, as it first sorts the whole array and then returns the first value. In terms of complexity, this function runs in $\Omega(n \log(n))$ at best and $\mathcal{O}(n^2)$ at worst, while the naive solution would be $\Theta(n)$.

Other Limitations –
One of the limitations that SpArrow's author mentions is 'limited number of input vectors'. We noticed this limitation is caused by the use of `Tuple2`, a fixed-size collection, for managing the ValueVectors. We think this limitation can be easily solved, by using a less limiting collection such as `Array`.

4.3 Evaluation

To evaluate SpArrow, we started with reproducing the 'minimum value' experiment from their Section "4.5. Offloading Functionalities to Arrow". In this experiment, the author shows the performance impact of offloading operations to Arrow, compared to using a standard Spark operation, by comparing three methodologies:

1. Finding the minimum value of an Integer-typed Resilient Distributed Dataset (RDD) through vanilla Spark (`min()`), using `parallelize` to generate the data.

SpArrow Thesis	This Evaluation
Driver ran at headnode	Driver ran at compute node
2 compute nodes for workers	2 compute nodes for workers
dual 16-core AMD EPYC2 (Rome) 7282 CPUs	dual 8-core Intel E5-2630v3 CPUs
128 GiB RAM	64 GiB RAM
4TB SSD	2*4TB HDD
100 Gbit/s (= 12.5 GiB/s) InfiniBand interconnect	53 GiB/s FDR InfiniBand [57]
Apache Spark 3.3.0 SNAPSHOT from Nov 2021	Apache Spark 3.3.0 SNAPSHOT from May 2022
Apache Arrow (6.0.0), Parquet (1.12.2)	Apache Arrow (6.0.0), Parquet (1.12.2)
Repeats at least 20 times	Repeats at least 30 times

Table 3: Comparison Experimental Setup SpArrow Thesis and this Thesis.

2. Finding the minimum value of an Integer-typed ArrowRDD using SpArrow/ Arrow-Spark through the default operator (`min()`), by reading in a single parquet file.

3. Finding the minimum value of an Integer-typed ArrowRDD using SpArrow/ Arrow-Spark by reading in a single parquet file and offloading to Arrow.

All datasets contained a range of ascending integers ($[0, 1, \ldots, N)$, with $N \in \{$1 million, 5 million, 10 million$\}$).

During our evaluation, we did not have access to the DAS-6 cluster system yet, which is the system SpArrow was evaluated on. Instead, we evaluated SpArrow on the DAS-5 cluster system [3]. Table 3 shows the most important differences (and similarities) between our setup and the SpArrow's setup as described in the thesis.

Figure 10 shows the results from both our evaluation and the evaluation from SpArrow. Figure 10a shows the result from the SpArrow thesis. This graph displays the running times in seconds (y-axis) for the three methodologies against the input size (x-axis). For each input size (1m, 5m, 10m), we find the median value together with an error-bar. The plot also shows a trend-line to show the expected trend for any input-values in-between.

From this plot, we observe that the default `min()` operator of SpArrow always seems slower than the other two. Vanilla Spark's `min()` and the `vecMin()` operator are closer together. Overall, the SpArrow variant seems faster than the vanilla variant, although their variations have more overlap once the data size increases. We are unsure if the execution times include reading/ generating the data.

For our evaluation, we compared the same methodologies and also included the reading time for the data. Note that these times represent something different for the two variations. For vanilla Spark, this is only the setup time, as reading is done lazily. For SpArrow, this metric represents reading in all the data greedily.
We wanted to remain as close as possible to the original experiment, but the datasets we found were only for sizes $N \in \{$100 kilo, 1 million, 10 million$\}$. Additionally, we decided to use a separate plot for each parquet file, to better compare the various methodologies. In each of the remaining subfigures of Figure 10 we show the running times in seconds (y-axis) for each of measured components (x-axis). From left to right these are:

- *Vanilla Generate*: Vanilla Spark setup for lazy reading.

- *Vanilla Compute*: `min()` on an Integer-RDD.

- *SpArrow Generate*: SpArrow greedy reading.
- *SpArrow Compute Default*: `min()` on an Integer-ArrowRDD.
- *SpArrow Compute Offloading*: `vecmin()` on an Integer-ArrowRDD.

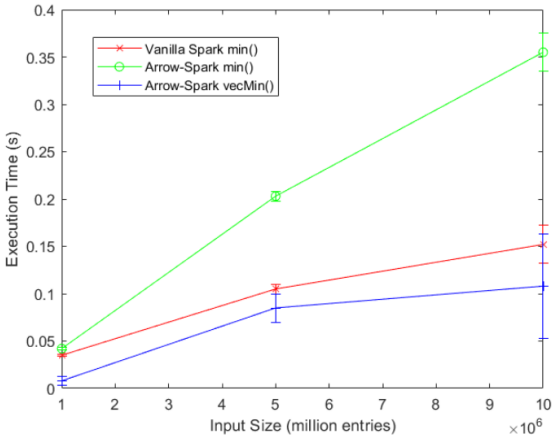
For these components we show boxplots to have a better understanding of the data distribution. We did remove the outliers to keep the y-axis range limited.

Figure 10b shows the result for the 100k dataset. While we cannot compare with the exact data in Figure 10a, we can compare the expected trend. As expected, 'Vanilla Generate' is almost negligible. Following the line in Figure 10a of 'Vanilla Spark min()', we find it conforms to our median value of 'Vanilla Compute'. It does show that we have a higher variance in our data, which could be explained by the difference in experimental setup (such as older hardware). Note that 'Vanilla Compute' also includes reading in the data. The next component 'SpArrow Generate', which is the SpArrow greedy reader, often takes more than half of the whole 'Vanilla Compute'. When we compare 'SpArrow Compute Default' with 'Arrow-Spark min()' we see that indeed the default computation takes longer than all other components. However, we do want to note that our median is higher than SpArrow's median for a larger (1 million) input size. The cause for this could either be a difference in version or hardware. We expect the former, as slowdown because of hardware would also cause 'Vanilla Compute' to run longer. Finally, we find that 'SpArrow Compute Offloading' differs a lot from 'Arrow-Spark vecmin()', as it should have been a lot faster than 'Vanilla Compute', but is actually often slower. Again, we expect this to be caused by a different version. For the one million dataset and ten million dataset many observations remain the same, except that the SpArrow computations in our evaluation are much slower than the 'Arrow-Spark' computations in Figure 10a.

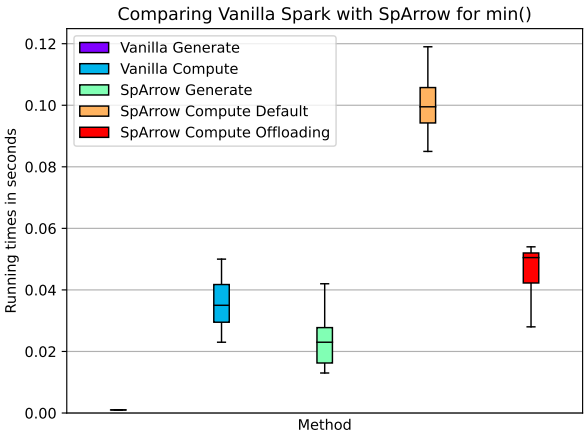
In Section 4.2 we went through some limitations that we found in the SpArrow implementation. Since we were working on extending SpArrow, we aimed to resolve some of these limitations. In particular:

- *Scalability*: We tried to increase the dataset size to perform more appropriate scalability evaluations. However, the ArrowRDD as created in SpArrow requires all its data to be in memory, in the form of (at most two) ValueVectors, where each ValueVector can have at most `Integer.MAX_VALUE` values, and each ArrowBuf can have at most 2 GiB allocated. Within this constraint, we tried to extend the scalability evaluation to somewhat larger datasets. While we realise this still does not represent real-world workloads, we hope this reveals a better expected trend in terms of scalability.
- *Memory Management*: We provided a temporary solution to the memory-management issue by simply reducing the reference-count to zero at the end of each experiment. While not suitable for larger datasets, this allowed us to work with somewhat larger data for experiments where we repeated within the same SparkSession.
- *Inefficient* `vecMin()`: We implemented our own algorithm for finding the minimum value, so we did not have to sort the whole ValueVector first.
- *Single File Reader*: We extended the greedy reader such that multiple parquet files in a single directory can be read.

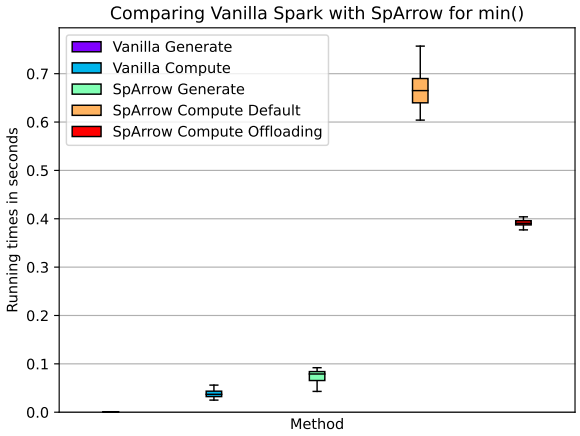
Finally, we also let Spark read from the parquet files, instead of generating its own data, such that the partitioning remains the same for both methodologies, resulting in a fairer comparison.



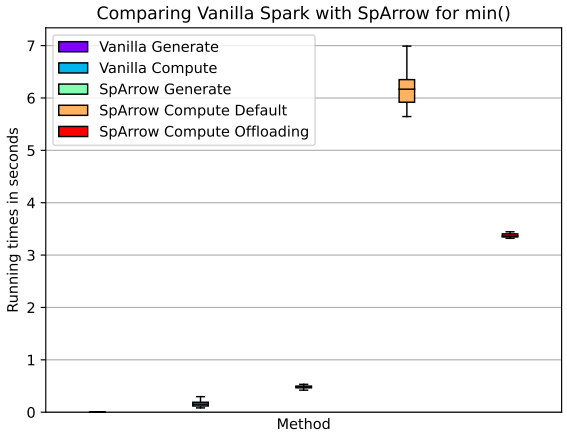
(a) From SpArrow thesis: "Comparing different implementations and design for finding the minimum value in an RDD".



(b) Our SpArrow Evaluation for a dataset of size 100k.



(c) Our SpArrow Evaluation for a dataset of size 1m.

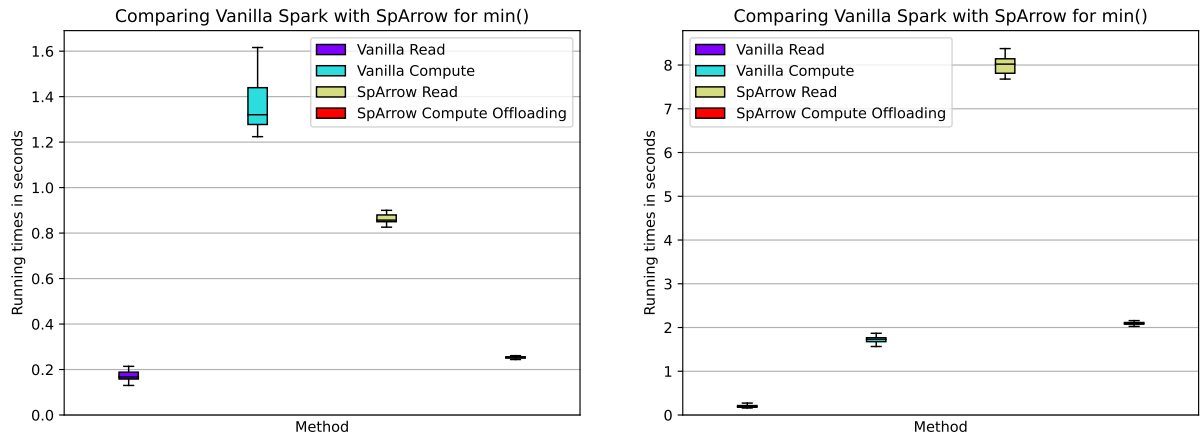


(d) Our SpArrow Evaluation for a dataset of size 10m.

Figure 10: Results both from the SpArrow Thesis and our Evaluation comparing `min()` on both vanilla Spark and SpArrow and `vecMin()` on SpArrow .

We evaluated above improvements in a similar setup as before. For the first evaluation, we re-used the parquet file of size 40 MiB, containing ten million rows. For the second we created our own dataset of size 685 MiB, containing a hundred million rows. For both evaluations, we left out the 'SpArrow Compute Default', because it takes a very long time and does not provide a meaningful contribution anymore.

Figure 11 shows the result of these evaluations. When comparing the plot in Figure 11a with the plot in Figure 10d, we immediately notice that 'SpArrow Compute Offloading' has become more than 10x faster. The main contributor to this speedup is our new `vecMin()` implementation. Additionally, we find that both 'Vanilla Read' and 'Vanilla Compute' have become slower, because we let Spark read in the data as well. We even observe that SpArrow has become faster than vanilla Spark. Finally, we find that 'SpArrow' read also became



(a) Improved SpArrow Evaluation with a dataset of 10 million rows (40 Mib compressed parquet-files).

(b) Improved SpArrow Evaluation with 685 Mib compressed parquet-files as dataset.

Figure 11: Improved SpArrow Evaluations

somewhat slower, although not by much.

When we compare Figure 11a with Figure 11b, however, we see that SpArrow lost its advantages in speedup. While 'Vanilla Compute' and 'SpArrow Compute Offloading' show equal runtimes, 'SpArrow Read' has become significantly slower. We may thus conclude that while computation seems to scale well, reading becomes a large bottleneck once dataset sizes start to increase.

Insight 4

For larger datasets, the SpArrow-reader becomes a bottleneck.

4.4 Conclusion

Working with SpArrow has definitely been an interesting experience. While often frustrating, we also learned a lot about Apache Spark and Apache Arrow, since our knowledge about those was rather limited at the start of this project. In addition, we were also better prepared to create our contribution, and could take a lot of inspiration from the process in working with SpArrow.

However, we must conclude that the current SpArrow implementation is too limited to be useful for real-world applications, even if we would extend it. It is simply not possible to run larger workloads on the ArrowRDD, on which SpArrow is based.

Insight 5

It is not feasible to extend SpArrow because of its current size limitations. It could, however, be used as a source of inspiration.

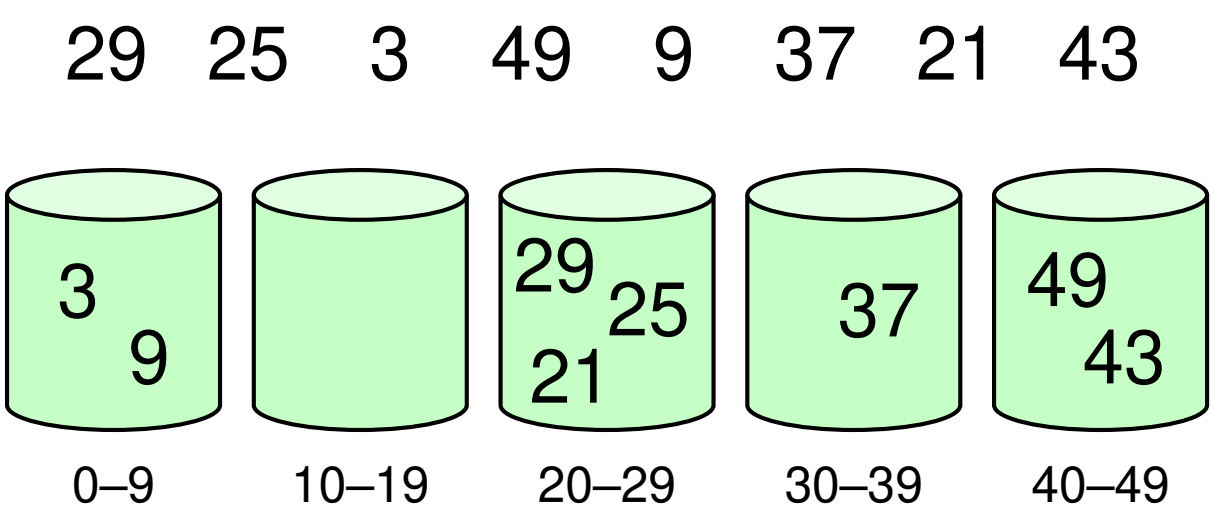


Figure 12: Example of a bucket-distribution during Bucketsort[16].

5 End-to-End Arrow Spark

In this Section, we discuss and evaluate our Proof-of-Concept 'Complete Arrow Spark (CAS)'. We named it 'Complete' as all operators we add to Apache Spark will be end-to-end Arrow-based. CAS is a Proof-of-Concept as we will not implement all possible operators. Instead we focus on implementing a single end-to-end Arrow-based operator.

For the choice of this operator, we wanted to make sure that it makes sense to perform it in Apache Spark. For example, some filter operators can be offloaded to parquet itself to reduce the amount of data that is loaded into memory, and does not require Spark. Thus, we chose to do a sort operator as this requires data movement through a shuffle-stage, which we cannot do without Spark.

Additionally, a columnar-format has to make sense for our operator. With this constraint, we chose to implement a multi-column sort, since this keeps the tabular format intact during shuffling, which ensures the data-format matters.

In the following subsections, we describe our design and implementation choices for this multi-column sort. We end this Section by performing experiments on our Proof-of-Concept, in which we cover several configurations and compare with vanilla Spark.

5.1 Design: Sorting in Spark

In this Section, we start by describing the various stages Spark uses for sorting. We first give an overview of all stages, followed by a detailed description of the stages, containing both vanilla Spark's components and the components from Complete Arrow Spark (CAS).

Spark uses a type of bucket-sorting[15] where the data is first partially-sorted by range. This ensures the whole dataset is sorted after the buckets are sorted locally, which is useful for distributed sorting. Figure 12 shows an example of these buckets. In this Figure, we also see that determining the right ranges for each bucket is important to get uniform distributions. Spark aims to create an uniform distribution, by first sampling its input-data. Each partition contains a single bucket.

[15] https://en.wikipedia.org/wiki/Bucket_sort

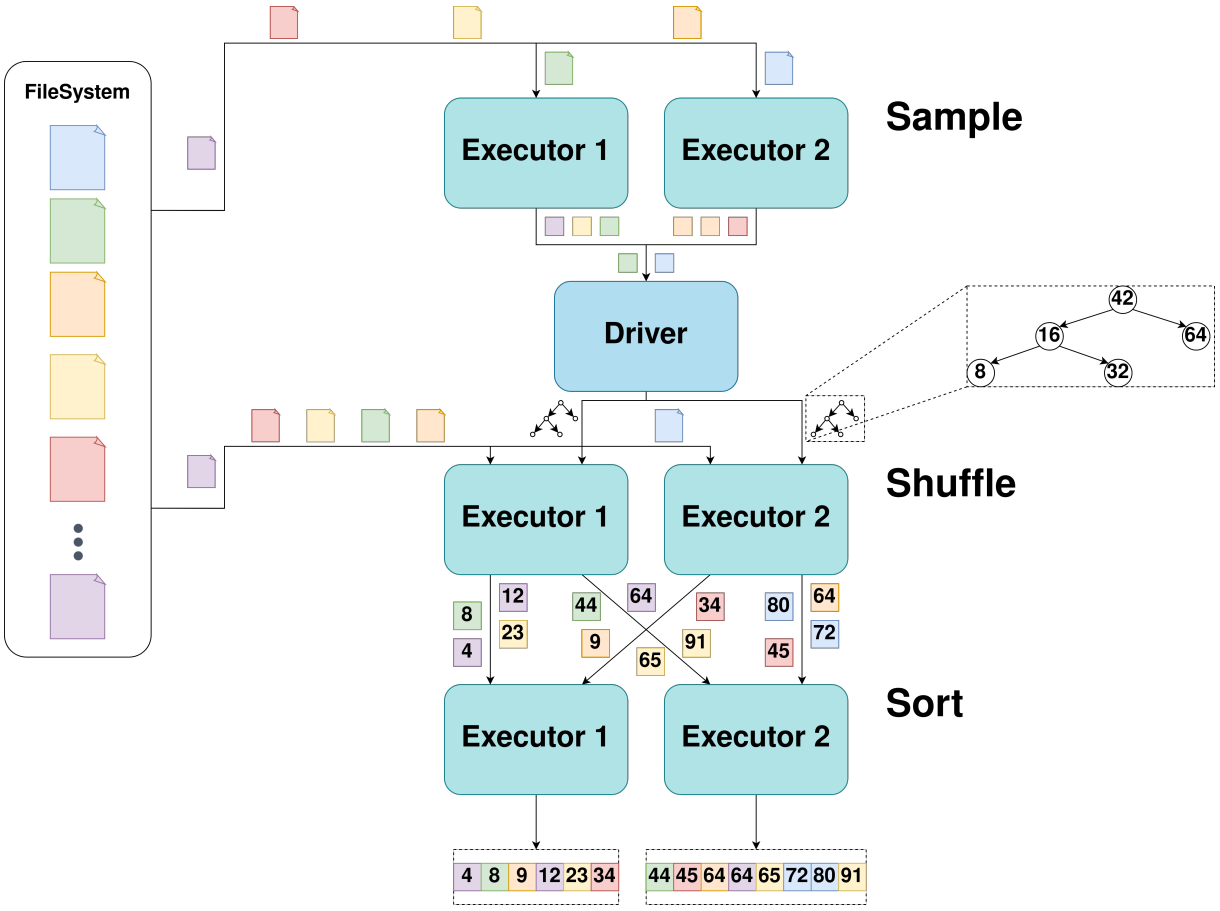


Figure 13: Spark Stages for Sorting.

Figure 13 gives an overview of the three main stages for sorting in Spark. In its first stage, 'Sample', Spark determines the range-bounds for the buckets/ partitions. Each executor samples their own partitions of the data. In the Figure, the data resides on a (distributed) file-system, but other sources are possible as well. The driver collects the samples from each executor and determines the range-bounds.

The next stage is 'Shuffle'. In this stage, the executors read in their partitions of the data and distribute them according to the range-bounds.

Finally, we have the 'Sort' stage in which each executor sorts its buckets of data. Spark uses Timsort[17] for this stage.

In the remainder of this design-section we will describe the 'sampling' and 'shuffle' stage in more detail.

5.1.1 Sampling

In this section, we describe the sampling stage.

We illustrate the main steps in Figure 14. To determine the range-bounds, Spark uses the `RangePartitioner`, which contains the `rangeBounds` object, an array of Rows with

[16] source: By Zieben007 - Own work, CC BY-SA 4.0 `https://commons.wikimedia.org/w/index.php?curid=59456452`

[17] `https://en.wikipedia.org/wiki/Timsort`, according to DataBricks [63])

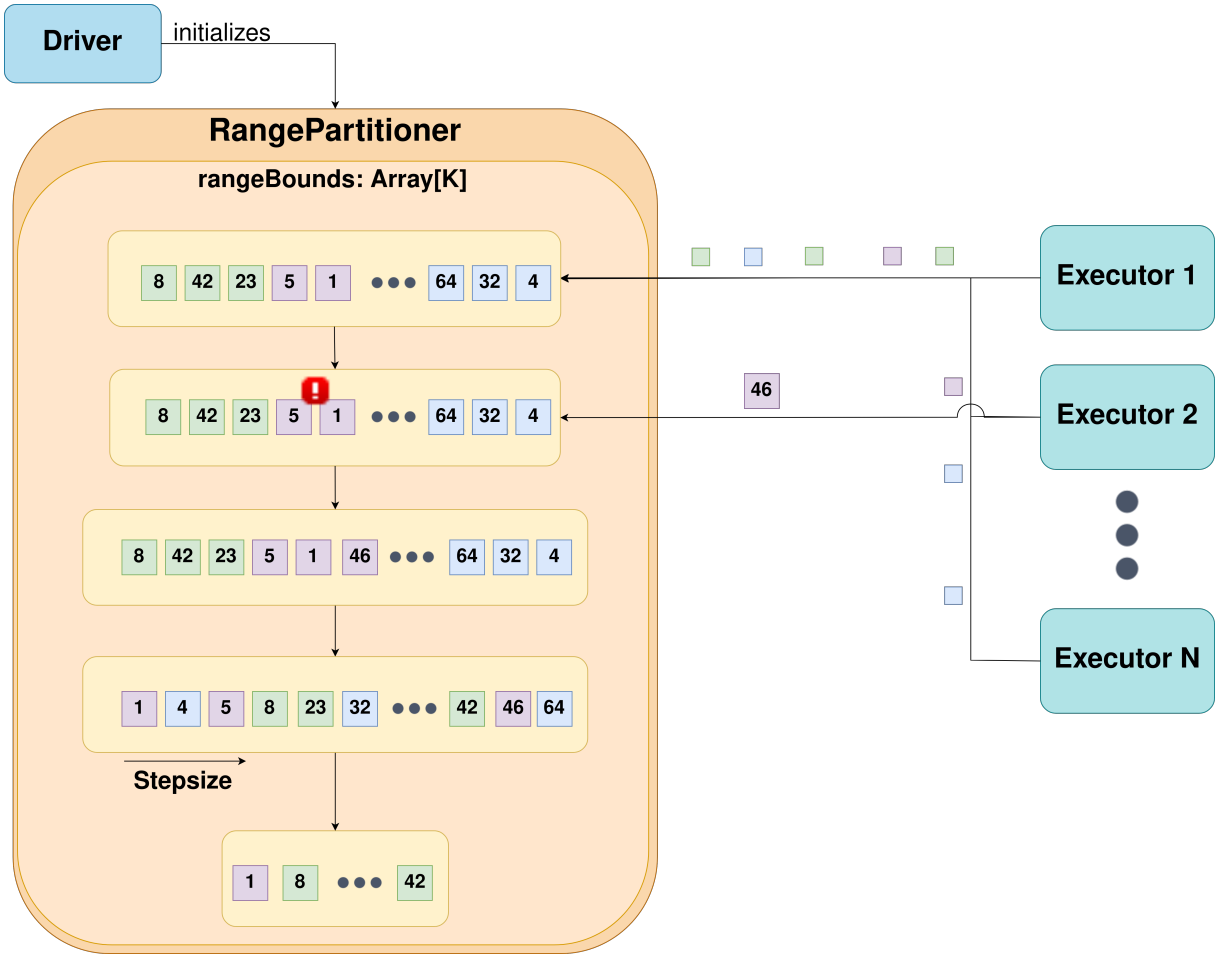


Figure 14: Sampling in Apache Spark.

(generic-)type `K`. Similarly, Complete Arrow Spark (CAS) uses the `ArrowRangePartitioner`. If we have N shuffle-partitions, then both partitioners gather $N - 1$ range-bounds.

Both partitioners sample from the executors in their first step. Spark uses a heuristic to determine the total sample-size, and uses this to determine the sample-size-per-partition k. Each executor then fills a reservoir of at most k samples from its partitions, and sends it to the used partitioner (ArrowRangePartitioner or RangeParititioner) at the driver-side. If a partition sampled too few rows, the used partitioner samples again on that partition. For this sampling, Spark uses a different method than for the initial sampling.

For the initial samples, each executor first fills a reservoir with the first k rows. Then, the remaining rows are distributed semi-randomly over the reservoir, possibly replacing previously placed rows. This ensures the reservoir always has (at most) k samples. For the rebalance-sampling, Spark implemented a type of Bernoulli sampling[18] and aims at a target sample-count of around k. Both sampling methods are semi-random, as Spark uses fixed seeds.

Additionally, each sampled row receives a weight, depending on the sample size and the size of the data it was sampled from.

After they collected the samples, both partitioners sort them and select the range-bounds according to the weight and a certain stepsize. Duplicate values are skipped. Note that in the ArrowPartitioner, we first remove the duplicates before selecting them, so we do not have to

[18] `https://en.wikipedia.org/wiki/Bernoulli_sampling`

check for duplicates during selection.

Finally, the driver sends the used partitioner (ArrowRangePartitioner or RangeParititioner) to each executor. As the partitioner contains both the range-bounds, and the methods to use them, the executors are now able to distribute their rows.

5.1.2 Shuffling

For each row in each partition, the executors determine to which partition they should distribute it. For this, they use the range-bounds which they received from their Partitioner. In particular, they perform a binary-search[19] on the range-bounds to determine in which range a certain row falls. Each executor serializes the rows, which match the range-bounds of other executors, into temporary files and sends these to the other executors.

5.2 Implementation

In this Section, we describe the implementation details of Complete Arrow Spark (CAS) in a bottom-up approach. We start by explaining core components of our work, and end by giving an overview of the added Physical Plans.

5.2.1 ArrowColumnarBatchRow

As a basis for our work, we had to find a representation for the Arrow-backed data. Spark already implemented the `ArrowColumnVector`, a ColumnVector backed by Arrow-data. Thus, we are able to reuse this structure as internal representation. However, Spark expects `InternalRows`, not `InternalColumns`, so, we must tell Spark how to interpret these columns. For this purpose, we created the `ArrowColumnarBatchRow` as an immutable wrapper around the ArrowColumnVectors, inspired by Spark's `ColumnarBatch` and `ColumnarBatchRow`, with a few differences.

The ColumnarBatch contains an Array of `ColumnVectors` and cannot be used as an InternalRow. The ColumnarBatchRow represents a single row in the batch as InternalRow, but cannot be accessed as a batch. Since we still wanted to have batched-access, we decided to make the ArrowColumnarBatchRow a batched InternalRow.

As the InternalRow implementes `SpecializedGetters`, we must define methods to retrieve the i-th value. This brings up the question: what do we consider the i-th value in a tabular batch? We came up with two possible solutions:

1. The ArrowColumnarBatchRow represents an one-dimensional array of multiple appended rows. The i-th value must then be mapped to a j-th column, and k-th row. This disallows the getters from having batched-access.

2. The ArrowColumnarBatchRow represents a row of Arrays (`ArrayData`). The i-th value is then a batch of the i-th column, and the getters return a `ColumnarArray`.

Since the first solution prevents us from using batched-execution, we decided not to apply it. Instead, we chose the second solution, as batched-access reduces overhead and possibly reduces instruction cache misses ([23], [49]). Figure 15 shows an example of an ArrowColumnarBatchRow.

[19] `https://en.wikipedia.org/wiki/Binary_search_algorithm`

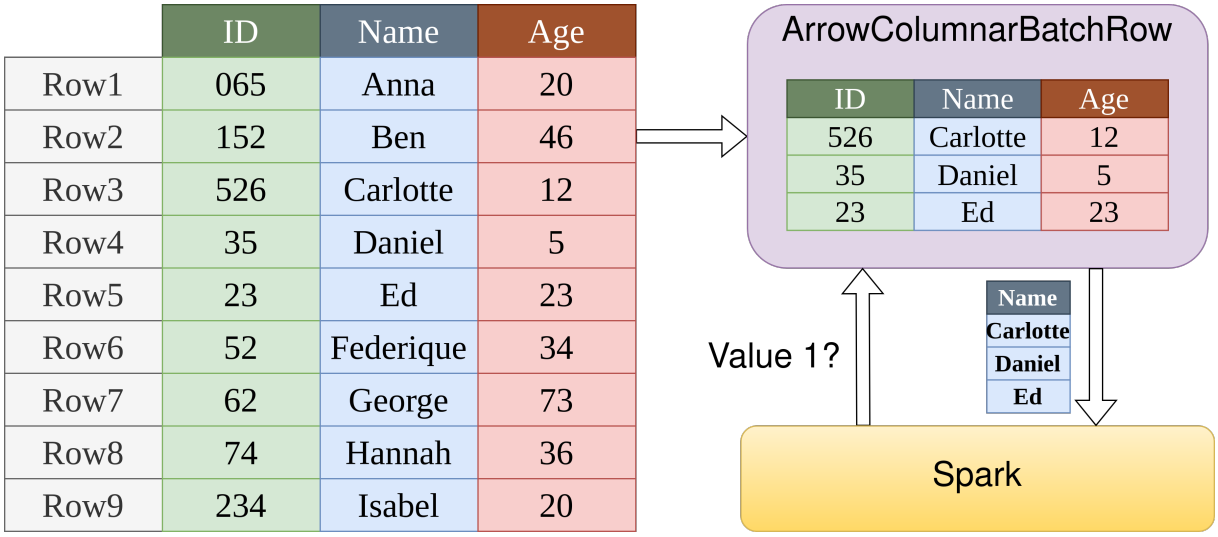


Figure 15: ArrowColumnarBatchRow, created from a batch of a table.

A problem addressed by Nonnenmacher *et al* [29] is that the data within the ColumnarArray is private, so we have no easy access. Similar to the solution of Nonnenmacher *et al*, we created the `ArrowColumnarArray` as a wrapper around the `ColumnarArray`, which uses reflection-hacks to retrieve the ArrowColumnVector.

5.2.2 ArrowRDD

While we implemented ArrowColumnarBatchRows as InternalRows, sometimes we want Spark to perform other actions than it would normally do for InternalRows. In these cases, we must specifiy these actions in Arrow-backed Resilient Distributed Datasets (RDDs). For example, we must tell Spark how the driver should retrieve data from its workers. To generalize this, we created the `ArrowRDD`, an abstract structure which Arrow-backed RDDs may inherit from. In particular, we implement two operators for this RDD: `collect` and `take`. For `collect`, we gather all ArrowColumnarBatchRows from all partitions and return them in an Array. For `take`, we gather only the ArrowColumnarBatchRows which represent the first n rows.

For both operators, we looked at how Spark implemented them and found three main stages:

1. Encode the data at the worker-side into byte-arrays.
2. Gather the byte-arrays at the driver side.
3. Decode the byte-arrays back into the data we want.

For the take operator, the second and last step are combined in a loop, such that the result can be gathered batch-by-batch until all rows are collected.

For encoding and decoding, we use Arrow's Inter-Process Communication (IPC) format[20]. In particular, we load and unload data into a VectorSchemaRoot, which is used by the ArrowStreamWriter to write batches to streams. To accomplish encoding, we first convert the ArrowColumnarBatchRow to an ArrowRecordBatch, as this is the format that the VectorSchemaRoot requires for loading.

[20] Arrow IPC format: `https://arrow.apache.org/docs/java/ipc.html`

For the collect operator, we also provide methods to encode/decode extra data. This is required if an RDD contains other data next to the ArrowColumnarBatchRow, such as weights. To accomodate for this, we allow the user to pass extra encoding and decoding functions.

5.2.3 Memory Management

An important lesson we learned from using SpArrow [15] is that memory management is important when using Arrow. While memory management is already difficult in the non-distributed case, setting up memory management for distributed data processing is very tricky. In particular, it becomes difficult to manage ownership. While we only need a single RootAllocator for a single-machine, we now have distributed memory spaces, where each should have its own RootAllocator. This means we need to close our allocator once our worker is done with its task.

Within Spark, however, tasks are defined for partitions, not for workers. It is easy to determine when a task is finished, but not when all tasks for a worker are finished. Thus, we decided to create an `ArrowPartition`, with a single purpose to hold a RootAllocator. Now, each Arrow-based Resilient Distributed Dataset (RDD) should only work with partitions of our ArrowPartition-type. We enforce this in our abstract ArrowRDD in which we also close the allocator. This guarantees that each ArrowBuf, which is allocated through the partition's RootAllocator, will either be closed or reported on.

Often, we need to have access to the RootAllocator saved in the ArrowPartition. This is straightforward when we actually have access to the partition of the RDD, but we lose this information if we perform a mapping operator on an ArrowRDD. For this purpose, we created the `ArrowMapPartitionsRDD`, similar to Spark's `MapPartitionRDD`, with the main difference that the mapping-function also receives the RootAllocator of the Partition in its arguments. The ArrowRDD defines methods to supply these mapping-functions, similar to Spark's `mapPartitionsInternal`, `map` and `mapPartitionsWithIndex` methods.

Another challenge is that using the Arrow API creates references, and we need to carefully manage them. Initially, we kept it simple by only using the RootAllocator. This works, as long as we do not forget to release our references, but if a memory leak gets reported, there is no indication where the leak occurred. Our second approach, therefore, included the use of ChildAllocators. Each reference we made, got its own allocator. This made debugging a lot easier, because each leak was accompanied with its own 'stack-trace' of allocators. However, an issue with this approach was that we got a very deep tree of allocators, which we could not close until the very end. For larger datasets, we soon filled up our memory with many small objects. For our third idea, we wanted to create a wide tree instead, such that we were able to clean up in the meantime. By providing useful names for our ChildAllocators, this would still provide enough debug-information. We implemented this by making sure each ChildAllocator is a direct descendant of the RootAllocator. This approach, however, still contained one problem. While multiple allocators can have a reference to an ArrowBuf, each original owner can make extra allocations, which are not shared. These extra allocations only get transferred, if there is an actual transfer of ownership. The allocator holding these extra allocations, should be the last allocator to close. In the current setup, this is not guaranteed when two ChildAllocators share an ArrowBuf. Thus, we had to ensure the RootAllocator always is and remains the original owner of an ArrowBuf. To accomplish this, transfer of ownership is only done between RootAllocators. ChildAllocators should always receive a reference and never an ownership.

Figure 16 illustrates the described reference-management approaches. The upper left image

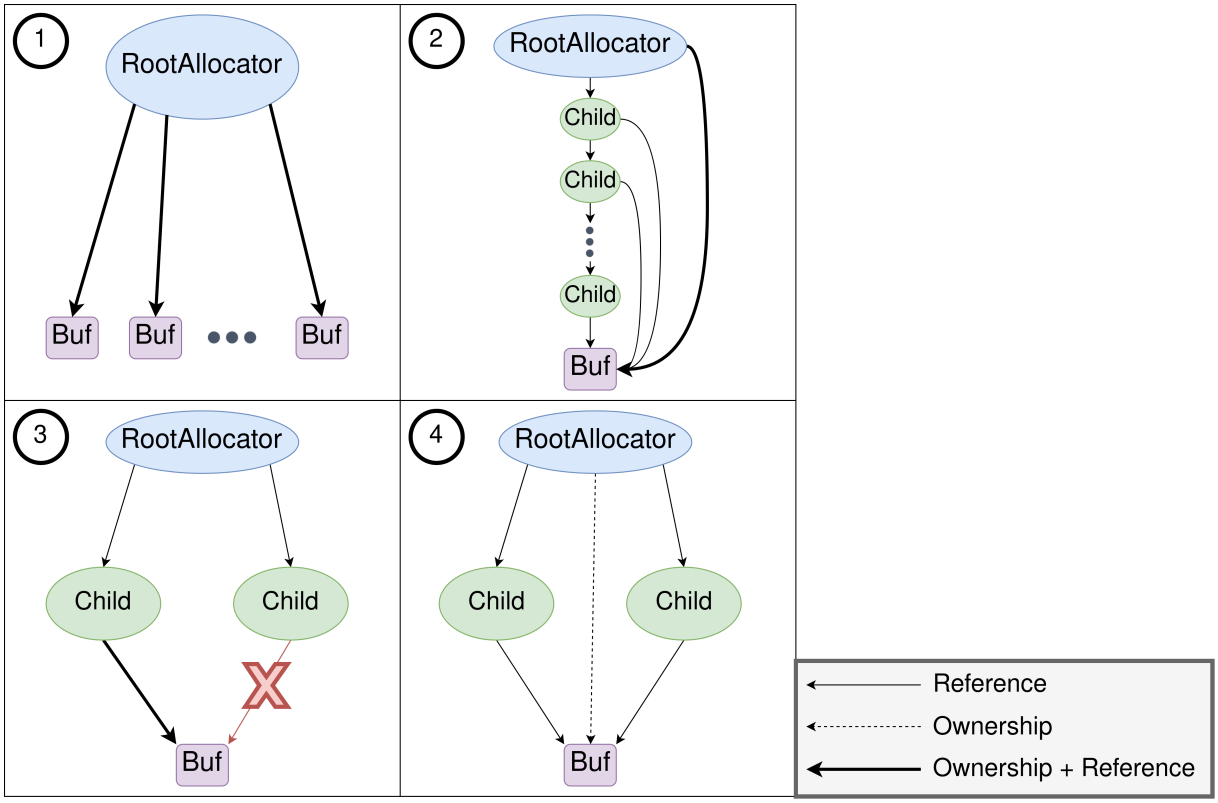


Figure 16: Four Strategies for Reference Management.

shows one RootAllocator as owner of all ArrowBuffers. The upper right image shows a deep tree of ChildAllocators, all having a reference to a single ArrowBuf. The lower left image shows a wide tree, where the left ChildAllocator is the owner of the ArrowBuf, and the right ChildAllocator may not get a reference. Finally, the lower right image shows a wide tree, where ChildAllocators get references, and the RootAllocator gets ownerhship.

Insight 6

When combining Apache Spark with Apache Arrow, reference management becomes tricky. One the one hand, we do not want to work with only a single RootAllocator. On the other hand, we should be careful with the number of ChildAlloctors we use. We should also be careful about the difference between owning a reference, and owning an actual ArrowBuf.

5.2.4 ArrowColumnarBatchRowBuilder

With the use of the Arrow API, it is quite easy to split an ArrowColumnarBatchRow into multiple batches, with the use of slices. However, there is no trivial method to combine multiple ArrowColumnarBatchRows into a single ArrowColumnarBatchRow. Nevertheless, it is a useful operation to have when we want to merge batches which are smaller than a target batchsize, such as when working with samples. There is also no cheap way to do this, as we need to copy data in order to merge multiple batches. With our ArrowColumnarBatchRowBuilder, we do support merging multiple batches into one, optionally up to n rows. We put effort in trying to make it more efficient, but it remains an expensive operation.

First of all, we had to make a distinction between the first batch and following batches. For the first batch, we determine the amount of columns and their types. We assume subsequent batches follow this schema, just like one would use a VectorSchemaRoot. Additionally, we assume all values are valid.

From this first batch, we create an array of ArrowColumnVectors and not an actual ArrowColumnarBatchRow yet. Then, the user of the ArrowColumnarBatchRowBuilder may append an arbitrary amount of ArrowColumnarBatchRows to these columns, as long as the number of values does not exceed `Integer.MAX_VALUE`, as ValueVectors are bound by their integers for indexing. Finally, the user may finalize the columns, or create an ArrowColumnarBatchRow, by using the `build` methods.

When copying the data from the original batch to the merged-batch, we want to copy all values at once. However, through the Arrow API, we could only copy one value at a time. So, we implemented the functionality to copy multiple values at once ourselves. For the data itself this is relatively straightforward, as we can copy multiple bytes at once. Of course, we do need to make sure we allocate enough memory for this. However, we also need to update the ValidityBuffer, where the validity of each value is represented per-bit and not per-byte. So, we wrote our own `ValidityRangeSetter` method in order to set the validity of a range of values. First, we use Arrow's `BitVectorHelper` to retrieve the start-and-end bits- and bytes-indices. We then read the required bytes into a byte-array, use bit-wise operators to set the right bits to valid, and write the required bytes back again.

Another point to consider is that we need to keep track of the number of bytes we have written. This is relatively simple, as we can retrieve the `readableBytes` from a ValueVector. However, it is important to keep track of this number per-column, as this may differ for columns of different types.

Finally, we had to make sure to conform to our reference-management strategy, as explained in Section 5.2.3. Specifically, we need to ensure the RootAllocator of the original batches will also be the original owner of the merged-batch. This requires a few steps. When processing the first batch, we assign its RootAllocator to the newly created batch. Then, when building the new ArrowColumnarBatchRow, we transfer its references to new ChildAllocators and close the references from the RootAllocator. Through this, the RootAllocator owns the ArrowBuffers, but no references to it, and the ChildAllocators own references to the ArrowBuffers but do not own the buffer itself.

5.2.5 SparkComparator

Apache Arrow provides several java-algorithms, such as sort- and search- algorithms. However, these algorithms only work for a single ValueVector, not for an array of them. Thus, to make use of these algorithms, we need to create a single ValueVector from our ArrowColumnarBatchRow. Luckily, Arrow provides the `UnionVector`, which we can use as representation of our batch.

Addtionally, we need to supply a `VectorValueComparator` to the algorithms, which defines how we can compare values in one or two ValueVectors. While Arrow supplies some default comparators, these are too generic for our purposes. Thus, we created two types of 'SparkComparators', in order to supply specific comparators to the algorithms.

The `SparkComparator` is meant to compare a ValueVector based on Spark's SortOrder and Arrow's VectorValueComparator. The SortOrder defines two type of orderings; SortDirection and NullOrdering. The SortDirection defines if the ordering is Ascending or Descending, and the NullOrdering defines if null-values come first or last. By overriding the `compare` and

compareNotNull methods of the VectorValueComparator, we are able to combine the provided VectorValueComparator and SortOrder.

The SparkUnionComparator is meant for an UnionVector where ValueVectors have different priorities. This comparator is actually a collection of multiple (ordered) SparkComparators, and returns the first non-zero compare result of all its SparkComparators.

5.2.6 ColumnDataFrame

We aimed to make Complete Arrow Spark (CAS) fully end-to-end Arrow-based. This also includes an Arrow-based DataFrame. In the end, we refrained from actually implementing such a DataFrame, because the data collected from the DataFrame to the driver is meant to be small. We believe an Arrow-based DataFrame will not provide a significant advantage, but will incur overhead.

Instead, we created a ColumnDataFrame as a base for creating an actual Arrow-based DataFrame in the future, should this become needed. Our ColumnDataFrame is defined as a Dataset of ColumnBatches. A ColumnBatch, much like the ColumnarBatch from Spark, contains multiple columns (TColumns) which represent a batch of the Dataset.

To be able to use this ColumnDataFrame, we also had to create our own ColumnDataFrameReader, which creates a ColumnDataFrame from a readable source. Mostly, we only had to implement this because we wanted to return a different type. So, we extended the default DataFrameReader and overrode the required functions. However, as with a lot of classes created by Spark, some member variables we needed were private. To prevent too much code copying, we used reflection hacks to re-use the private member variables of the DataFrameReader.

Finally, we also had to implement a ColumnEncoder, which is needed to encode ArrowColumnarBatchRows to ColumnBatches. This was done, similar to Spark's RowEncoder through the Whole-Stage CodeGen.

5.2.7 Physical Plans

In order to extend Spark we make use of Spark Extensions[21]. In particular, we insert our own strategies to transform Optimized Logical Plans to our Arrow-based Physical Plans. These extensions can be enabled by calling .withExtensions(ArrowSparkExtensionWrapper.injectAll) on the SparkSession.Builder. In total we created four different physical plans.

ArrowScanExec –
The ArrowScanExec is a Physical Plan which creates a FileScanArrowRDD given a provided ArrowFileFormat. Its approach is similar to Spark's DataSourceScanExec.

The ArrowFileFormat is an abstract FileFormat, with two main purposes:

1. Define the buildArrowReaderWithPartitionValues(...) method. This method must return a read-function which returns an Iterator of ArrowColumnarBatchRows, given a PartitionedFile and RootAllocator to allocate with. A PartitionedFile represents a part of a single file, and describes, among others, its filePath, start-offset and length (in bytes).

2. Implement the inferSchema(...) method, which transforms the Schema of the Parquet file, such that Spark thinks we are working with rows containing Array-typed values (conforming our ArrowColumnarBatchRow).

[21] Spark Extensions was introduced here: https://databricks.com/session/how-to-extend-apache-spark-with-customized-optimizations

We implemented one instance of the ArrowFileFormat, called `SimpleParquetArrowFileFormat`, which returns a read-function to read in a Parquet file and generate ArrowColumnarBatchRows.

The FileScanArrowRDD is an ArrowRDD with as main purpose to turn an `ArrowFilePartition` into an Iterator of ArrowColumnarBatchRows. Its approach is similar to the `FileScanRDD` in that it wraps the Iterators for each PartitionedFile into a single Iterator. The ArrowFilePartition is simply a combination of a FilePartition and an ArrowPartition.

ArrowShuffleExchangeExec –
The ArrowShuffleExchangeExec is a Physical Plan which creates a `ShuffledArrowColumnarBatchRowRDD` given a Partitioning, and the produced Resilient Distributed Dataset (RDD) from its child. Note that in our case, this is a FileScanArrowRDD. For this RDD, it needs a `ShuffleDependency`, and for this it creates an `ArrowColumnarBatchRowSerializer`, `ArrowBypassMergeSortShuffleWriter` and `ArrowRangePartitioner`.

The ArrowColumnarBatchRowSerializer's main purpose is to serialize and deserialize a series of ArrowColumnarBatchRows. While we use a similar method as for encoding and decoding, there are still some notable differences, which made getting it right tricky:

- Multiple serialized instances may be send to the same deserializer. This is a problem, as we would normally read until our ArrowStreamReader cannot read in more batches. For this, it expects certain headers and footers, and throws an error if the format is not as expected. Thus, we cannot keep reading until the end of the file if we want to avoid exceptions. On the other hand, we also cannot simply check on the end of file, as we should try to read a byte for that, meaning we might break the header. Our solution was to write an additional byte (we chose the character 'B') before the stream-header. However, the additional byte is required to be written after the compression-header, as the bytes in the deserializer should go through the decompressor directly.

- For deserialization, we receive a batched InputStream. This means that while there may be x bytes left to read, you can only read a maximum of y bytes each time, with $y < x$. This is a problem, as our readers expect certain parts to be of a certain size. If they cannot read in this size, then they throw an exception. As a solution we simply read in all bytes from the InputStream in an ArrayBuffer. This is not the most elegant solution, yet it is simple and effective.

The ArrowBypassMergeSortShuffleWriter is a ShuffleWriter responsible for writing the batches to the right partitions, similar to Spark's `BypassMergeSortShuffleWriter`. For this, it receives an Iterator of key-value pairs. In our implementation, a value is an ArrowColumnarBatchRow and its corresponding key is an array of integers, mapping each batch-index to a partition-id (generated by the ArrowRangePartitioner). The ArrowBypassMergeSortShuffleWriter then distributes the rows in the batch according to the mapping.

The ShuffledArrowColumnarBatchRowRDD is an ArrowRDD with as main purpose to turn an `ArrowShuffledRowRDDPartition` into an Iterator of ArrowColumnarBatchRows. For this, it uses the information in the ShuffleDependency. This step mostly involves deserializing the partition using the ArrowColumnarBatchRowSerializer-instance. Additionally, it expects it partitions to be `ArrowShuffledRowRDDPartitions`, which is a combination of an ArrowPartition and a ShuffledRowRDDPartition. The ShuffledRowRDDPartition simply contains its index and start-and-end indices.

ArrowSortExec –
The ArrowSortExec is a Physical Plan which is part of the Whole-Stage CodeGen and generates

code in order to sort partitions in an RDD. We implemented it similarly to Spark's SortExec. In this plan, there are two main functions to implement: `doProduce` and `doConsume`.

In doConsume, SortExec receives the rows to sort, and adds it to its Sorter. In ArrowSortExec, we use an ArrowColumnarBatchRowBuilder to combine the received batches into a single batch to sort.

In doProduce, SortExec first calls `produce` on its child, such that it will send the rows to sort. Then, it sorts the rows and sends each row individually to its parent. For ArrowSortExec, we also first call `produce` on the child, to receive the ArrowColumnarBatchRows to sort. Similarly, we sort the gathered batch and send it in its entirety to its parent.

ArrowCollectExec –
The ArrowCollectExec is a Physical Plan which acts as a wrapper around other Arrow-based plans. In particular, it ensures that operators such as `collect` and `take` are handled properly through the right ArrowRDD methods. Whenever a strategy adds an Arrow-based Physical Plan, it should insert the ArrowCollectExec plan to ensure proper propagation. For example, our `ArrowFileSourceStrategy` builds the plan as follows: `ArrowCollectExec(ArrowScanExec(scanExec))`, where `scanExec` is a FileSourceScanExec.

Example Driver Program –
In order to activate these plans, we made an example driver program. This program is also used in the experiments in Section 5.4 and Section 5.5. Figure 17 shows fragments of this code. The first fragment shows how we perform a multi-column sort using vanilla Spark. The second fragment shows how we perform a multi-column sort using Complete Arrow Spark (CAS). Both fragments consist of two stages. The first stage prepares the input RDD for sorting. This includes both reading in the data, and shuffling it. The second stage performs the actual sorting.

Note that sorting is an expensive operator. Therefore, Spark tries very hard to skip it if it thinks it is not required. To force Spark to actually sort all the data, we therefore also count all our sorted rows. While it may seem sorting is not required when you only count your rows, we verified Spark still performs the sorting. Additionally, we had to write our custom count function, so we could close the batches at the same time.

With this driver program, Spark generates the physical plans as shown in Figure 18. As we can see, the plans for vanilla Spark and CAS are quite similar. Stage 1 contains the scanner-plans, and shuffle-plans, and Stage 2 contains the sort-plans. A difference is the added ArrowCollect-plans, which we added manually to ensure `collects` and `takes` are properly propagated. The AdaptiveSparkPlan is automatically added by Spark, to perform final optimizations.

5.3 Limitations

Unfortunately, our current implementation has some limitations which prevents it from being useful in a real-world application. Specifically:

- As we focused only on a multi-column operator, we do not support all possible operators
- While we have theoretical support for an arbitrary number of columns and arbitrary data types, we only tested one or two columns, and integer-typed data.
- We have flaws in our memory management, which prevents us from processing (compressed) datasets of 85 GiB in size. Therefore, our evaluations in the coming sections

```
/** vanilla Sorting */
...
val df = spark.table("vanilla")
val cols = df.columns
val sorted_df = df.sort(cols(0), cols(1))
/** Stage 1 */
val rdd = sorted_df.queryExecution.executedPlan.execute()
/** Stage 2 */
spark.sparkContext.runJob(rdd, iter => iter.length).sum

...

/** Complete-Arrow-Spark (CAS) Sorting */
val cdf: ColumnDataFrame =
    new ColumnDataFrameReader(spark)
        .format("org.apache.spark.sql.execution.datasources.
    SimpleParquetArrowFileFormat")
        .loadDF(dir.path)
val cCols = cdf.columns
val sorted_cdf = cdf.sort(cCols(0), cCols(1))
/** Stage 1 */
val arrowRDD = sorted_cdf.queryExecution.executedPlan.execute()
/** Stage 2 */
val arrowFunc: Iterator[InternalRow] => Int = { case iter:
    Iterator[ArrowColumnarBatchRow] =>
     iter.map { batch =>
       try {
         batch.numRows
       } finally {
         batch.close()
       }
     }.sum
}
spark.sparkContext.runJob(arrowRDD, arrowFunc).sum
```

Figure 17: Fragments Driver code.

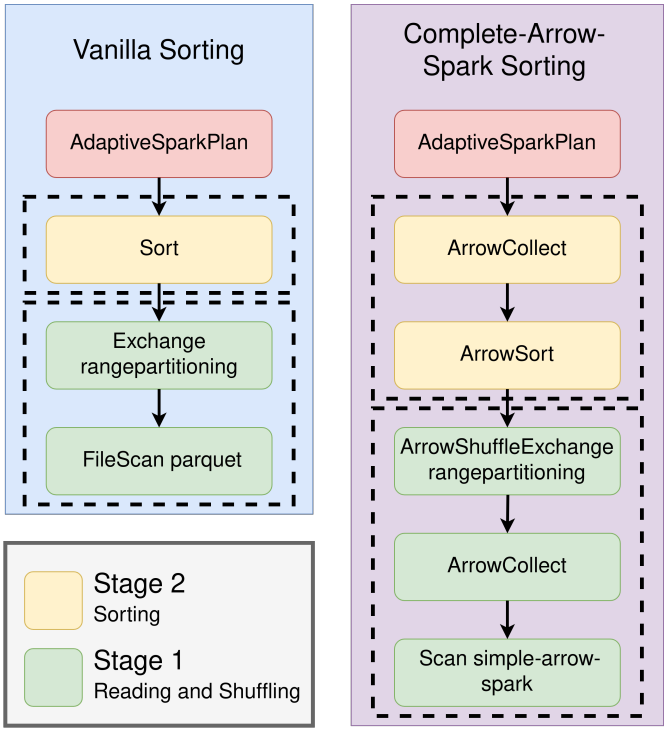


Figure 18: Physical Plans for vanilla sorting (left) and sorting with CAS extension (right), depicted as Directed Acyclic Graph (DAG). Each physical plan describes an operation to transform the RDD produced by its child to a new RDD

will only be on: 6.5 MiB, 172 MiB and 3.5 GiB compressed Parquet files. While we realise these sizes are far from real-world applications, we hope to find some indication of performance impact when scaling workloads. Additionally, we start a new SparkSession for each run in our experiments, to prevent any out-of-memory errors to occur after a while for the 3.5 GiB-experiments.

- We are still significantly slower than Spark. We will elaborate on this in Section 5.5.4.

5.4 Strategy Evaluations

During the implementation of Complete Arrow Spark (CAS), we experimented with multiple strategies for different parts of our project. In this Section, we describe these strategies, the experiments we performed and discuss the results.

5.4.1 Setup

For all experiments in this Section, we performed the multi-column sort on multiple datasets. We ran each experiment on the DAS-6 distributed cluster system [3]. Table 4 shows some specific configurations. Additionally, we used the driver program as introduced in Figure 17 in Section 5.2.7. Finally, unless specified otherwise, we use a reading-batchsize of 4096 rows, as we will elaborate on in Section 5.5.2.

Configuration	Value
Spark Cluster mode	Standalone
Number of workers	4
Size of Parquet files	6.5 MiB, 172 MiB and 3.5 GiB
CPU	AMD EPYC 7402P 24-Core 48-threads
L1i, L1d, L2, L3 cache	32k, 32k, 512k, 16384k
RAM	128 GiB
Storage	0.8 TiB NVMe
Interconnect	100 Gbit/s (= 12.5 GiB/s) InfiniBand
Apache Spark version	3.3.0 SNAPSHOT from July 2022
Apache Arrow version	6.0.1
Apache Parquet version	1.12.3
Amount of runs	10

Table 4: Configurations used for the Strategy Evaluation experiments.

5.4.2 Parquet Reader

In the SpArrow thesis [15], the parquet-file reader was based on the Parquet-To-Arrow converter from Trivedi[22]. Figure 11b in Section 4.3 showed that for larger datasets, this reading strategy became a bottleneck. Therefore, we also implemented a native-reader through Apache Arrow's Dataset API, similar to previous work ([33], [29]). In this section, we compare both Trivedi's reader and the native-reader in Complete Arrow Spark (CAS). To have a fair comparison, we allowed a maximum of `Integer.MAX_VALUE` rows per batch (batchsize) for both readers.

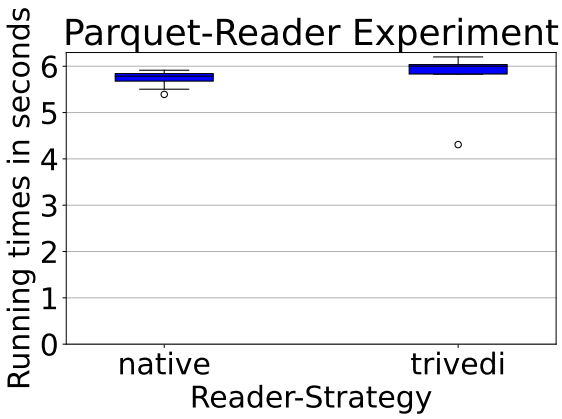
Figure 19 shows the result for this experiment. For each dataset, we have a different subplot. For each subplot, we show both the native-strategy (left) and trivedi-strategy (right). For each strategy, we show the running times in seconds, displayed in a boxplot, for stage 1 (reading and shuffling). In general, we cannot find a significant difference between the two approaches. Only for the 172 MiB dataset in Figure 19b, there seems to be a notable difference, in favour of the native reader.

While, we could not find a significant difference between the two readers, they still have different advantages. Trivedi's reader is easier to setup, as we do not require any native-bindings. However, we expect the native reader will be faster for significantly larger datasets ($\geq$ 100 GiB), although this would need to be validated through experiments. Additionally, the current implementation of Trivedi's reader does not have a setting to configure batchsizes. However, we show in Section 5.5.2, that this configuration can have a significant impact. For this last reason, we use the native-reader in all following experiments.

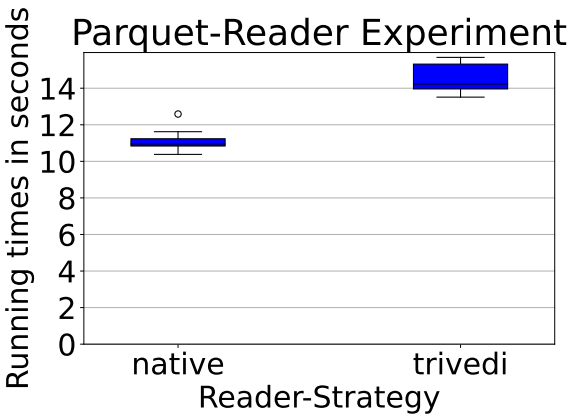
Insight 7

For compressed parquet-files up to 3.5 GiB, there does not seem to be a difference in performance between Trivedi's reader and a native reader.

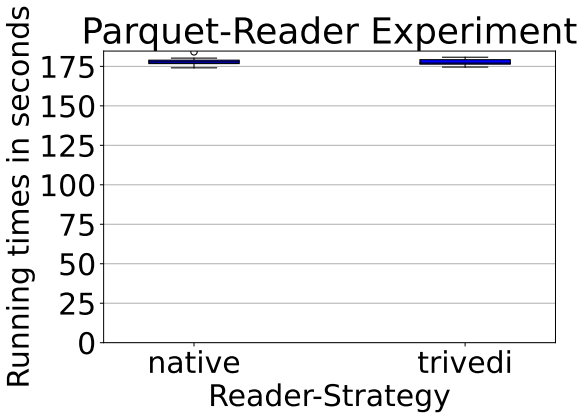
[22] `https://gist.github.com/animeshtrivedi/76de64f9dab1453958e1d4f8eca1605f`



(a) Results for 6.5 MiB dataset.



(b) Results for 172 MiB dataset.



(c) Results for 3.5 GiB dataset.

Figure 19: Results from Parquet-Reader Experiments.

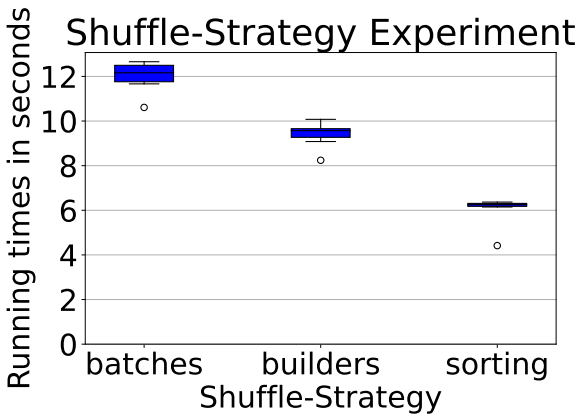
5.4.3 Shuffle Strategy

In the shuffle-stage we have to assign each row in an ArrowColumnarBatchRow to a partition. Since we have to do this for the whole dataset, it is crucial that we do this in an efficient way.

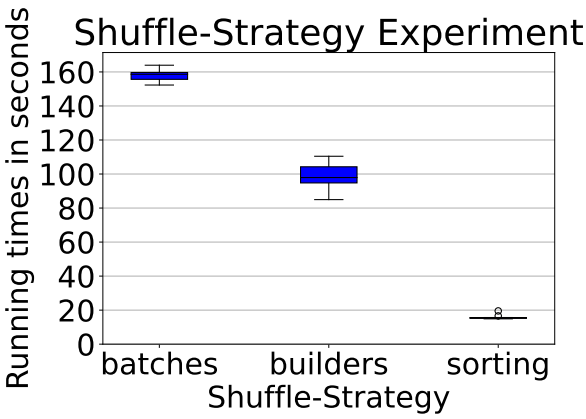
Initially, we created an ArrowColumnarBatchRow for each row separately, and combined them into a single ArrowColumnarBatchRow in the end, with the help of the ArrowColumnarBatchRowBuilder (as introduced in Section 5.2.4). As we kept all these ArrowColumnarBatchRows, we needed to keep many small objects, which gave memory issues for larger datasets.

As a solution, we immediately passed the separate rows to an ArrowColumnarBatchRowBuilder. In this approach, we created as many builders as we had partitions, and build up new batches row-by-row. While this was a better solution than creating many batches, builders are better suited for a few large batches, not many small ones.

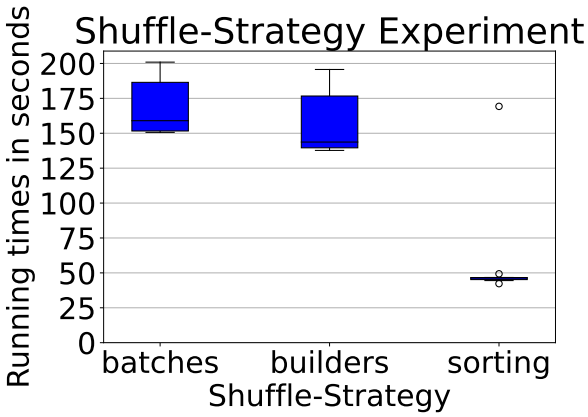
Finally, we had the idea to first sort the batch by its assigned partition-ids, and then split it according to the partition-ids boundaries. From the ArrowRangePartitioner, we already have the mapping from index to partition-ids in an array of integers. Thus, if we find the permutation required to sort this array, then we can apply it on the batch to sort it as well. However, as we do not have many partitions, we needed to find a sorting algorithm which works well for



(a) Results for 6.5 MiB dataset.



(b) Results for 172 MiB dataset.



(c) Results for 3.5 GiB dataset.

Figure 20: Results from Shuffle-Strategy Experiments.

many duplicates. We could not take a sorting algorithm like counting sort[23], as we need to keep information about the indices. Instead, we decided to use quicksort with three-way-partitioning as done in Dijkstra's approach[24]. Meanwhile, we kept track of the boundaries of the different partition-ids, such that it was easy to split our sorted batch later on.

We evaluated the three described approaches and show the results in Figure 20. For each dataset, we have a different subplot. For each subplot, we show the different strategies, from left to right these are: row-by-row approach (batches), builder approach and sorting approach. For each strategy, we show the running times in seconds, displayed in a boxplot, for stage 1 (reading and shuffling).

In all three plots we find significant differences between the three approaches, where the builder-approach is often faster than the batches approach, and the sorting approach is often faster than the other approaches. The result for the 3.5 GiB dataset, however, is a bit different than for the others. For this dataset we find that the running times of the builder-approach are closer to the running times of the batches-approach.

[23]Counting sort: `https://en.wikipedia.org/wiki/Counting_sort`

[24]Dutch National Flag Problem: `https://www.baeldung.com/java-sorting-arrays-with-repeated-entries#1-dutch-national-flag-problem`

Insight 8

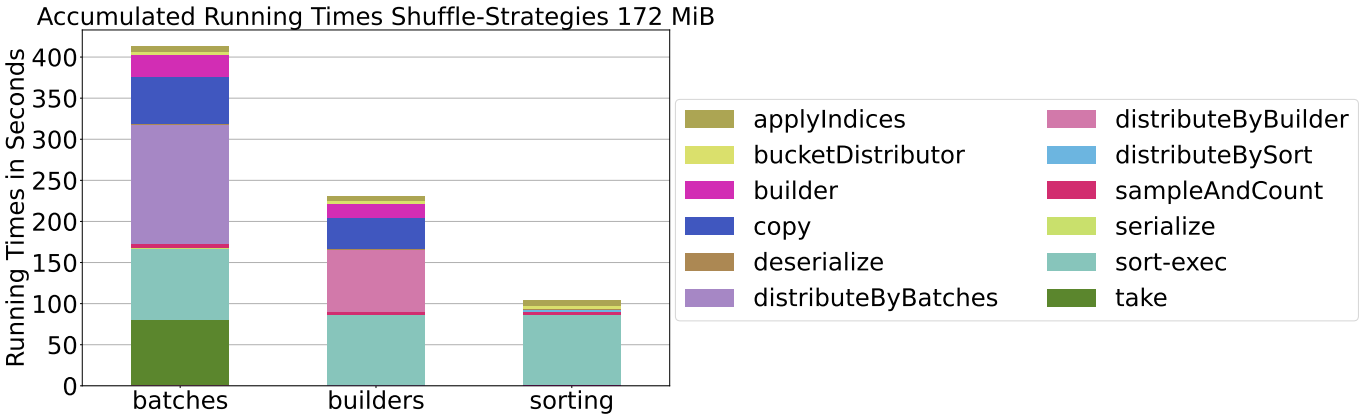
During shuffling, it is better to first sort the ArrowColumnarBatchRow and then split it, instead of first splitting it and then distributing it, at least for (compressed) datasets up to 3.5 GiB.

To analyze what causes this difference, we also monitored the running times of several functions separately. Specifically, we accumulated the total time spent in a single function for a complete experiment, per worker. Note that a high accumulated running-time does not always correspond to a high running-time in the experiment itself, as it may be spread over many partitions. Thus, we cannot use these running times to compare which approach performs better, but only what critical operators are for each approach.

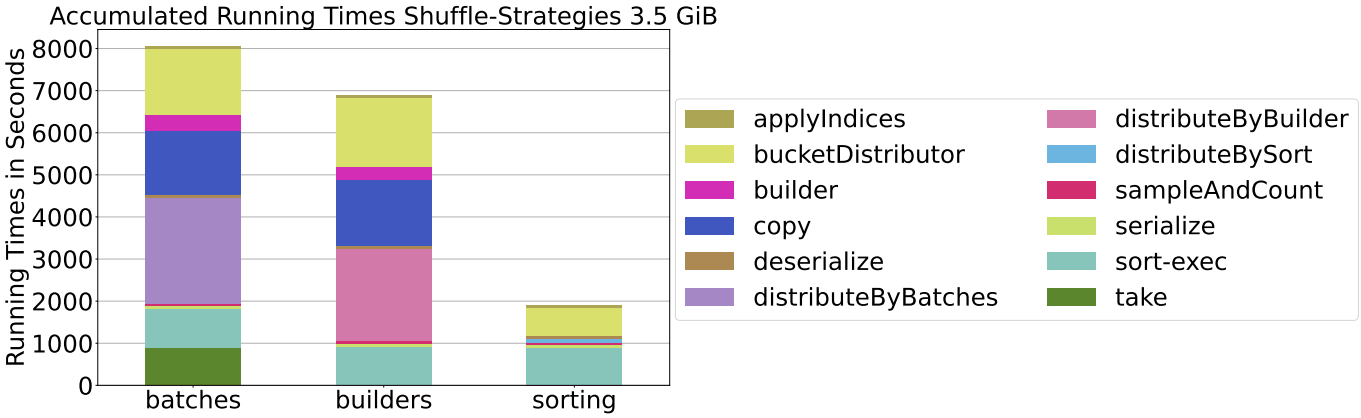
Figure 21 depicts the maximum accumulated running times, per dataset and per strategy. In Figure 21a we show the accumulated running times per function for the 172 MiB dataset. To keep the number of functions manageable, we left out functions with a lower accumulated running time than one second. In Figure 21b, we show the accumulated running times per function for the 3.5 GiB dataset. Here we left out functions with a lower accumulated running time than 50 seconds. For both plots, we show on the x-axis from left to right: row-by-row approach (batches), builder approach and sorting approach. The functions we find in both plots are:

- applyIndices: permute a batch according to a provided set of integers.
- bucketDistributor: produce a mapping from batch-indices to corresponding partition-id.
- builder: usage of ArrowColumnarBatchRowBuilder.
- copy: create a reference of a(n) (slice of an) ArrowColumnarBatchRow.
- deserialize: deserialize one or more serialized batches.
- distributeByBatches: distribute row-by-row (batches).
- distributeByBuilder: distribute by ArrowColumnarBatchRowBuilder.
- distributeBySort: distribute by sorting partition-ids.
- sampleAndCount: sampling method as described in Section 5.1.1.
- serialize: serialize ArrowColumnarBatchRows to shuffle.
- sort-exec: actual sorting.
- take: merge multiple ArrowColumnarBatchRows into one, using the ArrowColumnarBatchRowBuilder (called by distributeByBatches).

Both subplots show the same influence of the strategies on performance. Apart from the actual sorting, the running times are defined by the various distribute-functions. Between the batches-strategy and builders-strategy we find the most obvious difference to be the disappearance of 'take'. This makes sense, as we are using the ArrowColumnarBatchRowBuilders directly in the builders-strategy instead of using 'take' at the end. We also find that when using the sorting-strategy, the actual sorting plays the only dominant role in performance.



(a) Results for 172 MiB dataset.



(b) Results for 3.5 GiB dataset.

Figure 21: Aggregated Running Times from Shuffle-Strategy Experiments.

However, if we compare the 172 MiB dataset with the 3.5 GiB dataset, we find that for the latter, 'bucketDistributor' also plays an important role. The complexity of the bucketDistributor is bound by both the dataset size and the number of partitions. In fact, if we have n rows in our dataset, and we want to distribute them over k partitions, then the complexity of bucketDistributor is bound by $\mathcal{O}(n \cdot \log(k-1))$. Since larger datasets lead to more partitions, in general, we thus see that the bucketDistributor becomes non-negligible.

Above observation could be a cause for the differences in Figure 20, although further experiments would need to validate this.

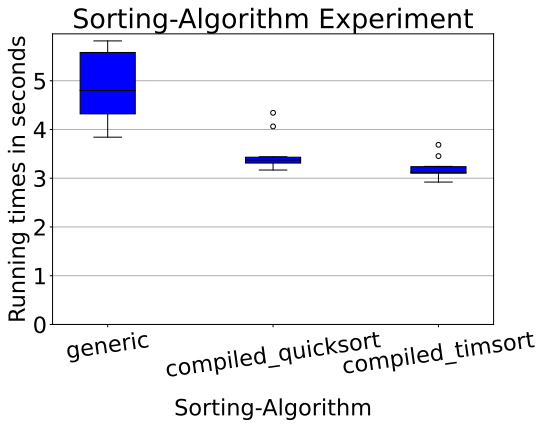
Insight 9

The running time of distributing rows over partitions becomes non-negligible if the number of partitions is large enough.

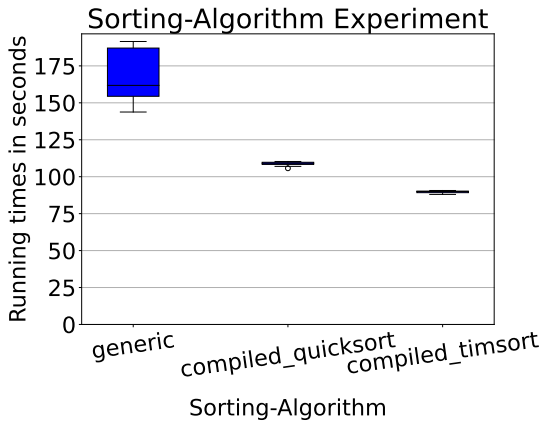
5.4.4 Sorting Strategy

Since sorting is an expensive operator, we put effort into making it faster as well.

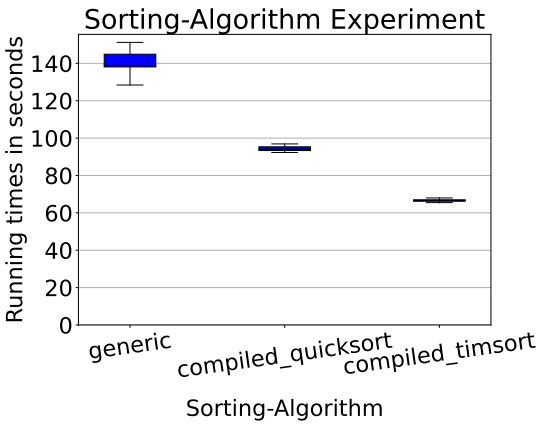
Initially, we used Arrow's java-algorithms since these are generic such that we have the-



(a) Results for 6.5 MiB dataset.



(b) Results for 172 MiB dataset.



(c) Results for 3.5 GiB dataset.

Figure 22: Results from Sorting-Algorithm Experiments.

oretical support for all vector-types. However, as we learned from Compiled Queries [28], abstractions are not always best for performance. Thus, we implemented our own compiled-sorting algorithms. One similar to the Arrow-algorithm (Quicksort[25]), and one similar to what Spark is using.

While theoretically we could still support all datatypes in the compiled algorithms, we chose to only implement them for integer-types, as a Proof-of-Concept. However, it is easy to add new types, as one only has to define how two non-null values can be compared.

We evaluated the three described algorithms and show the results in Figure 22. We have a different subplot for each dataset. Within each subplot we show the different algorithms, from left to right: generic Quicksort (generic), compiled Quicksort, compiled Timsort. For each algorithm, we show the running times in seconds, displayed in a boxplot, for stage 2 (sorting).

For all subplots we find similar results. The compiled algorithms are always faster and more stable than the generic Quicksort algorithm. Additionally, we find that compiled Timsort is often better than compiled Quicksort, and gets better for larger datasets.

[25]https://en.wikipedia.org/wiki/Quicksort

Configuration	Value
Spark Cluster mode	Standalone
Number of workers	4
Size of Parquet files	6.5 MiB, 172 MiB and 3.5 GiB
CPU	AMD EPYC 7402P 24-Core 48-threads
L1i, L1d, L2, L3 cache	32k, 32k, 512k, 16384k
RAM	128 GiB
Storage	0.8 TiB NVMe
Interconnect	100 Gbit/s (= 12.5 GiB/s) InfiniBand
Apache Spark version	3.3.0 SNAPSHOT from July 2022
Apache Arrow version	6.0.1
Apache Parquet version	1.12.3
Amount of runs	30

Table 5: Configurations used for the Strategy Evaluation experiments.

Insight 10

For sorting, we benefit from compiled algorithms over generic algorithms. Additionally, Timsort seems to perform better for somewhat larger (compressed) datasets, at least up to 3.5 GiB.

5.5 Experiments

In this Section, we perform experiments on several configurations from Complete Arrow Spark (CAS). We describe these configurations, and discuss the results of the experiments. Finally, we also compare CAS with a multi-column sort through vanilla Spark.

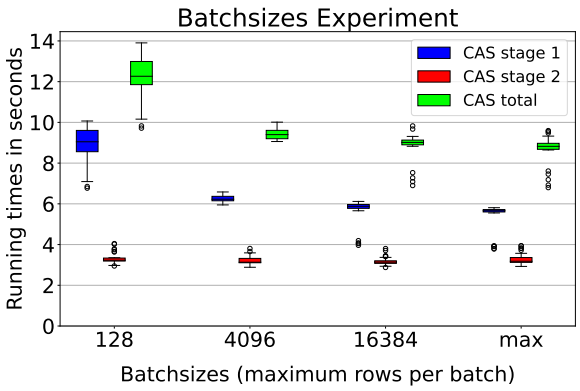
5.5.1 Setup

For all experiments in this Section, we performed the multi-column sort on multiple datasets. We ran each experiment on the DAS-6 distributed cluster system [3]. Table 5 shows some specific configurations. Additionally, we used the driver program as introduced in Figure 17 in Section 5.2.7.

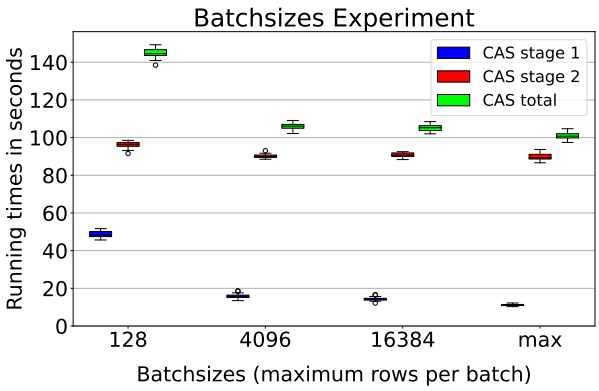
5.5.2 Batchsizes

As described in Section 5.4.2, we can configure the maximum number of rows in a batch (batchsize), for the native reader. According to Rodriguez *et al* [33], these batchsizes have an impact on their performance. In particular, too low batchsizes cause too much conversion and memory-copy overhead, while too high batchsizes are not cache-friendly. They also found that it is better to overestimate batchsizes. In this Section, we investigate if these observations hold for Complete Arrow Spark (CAS) as well.

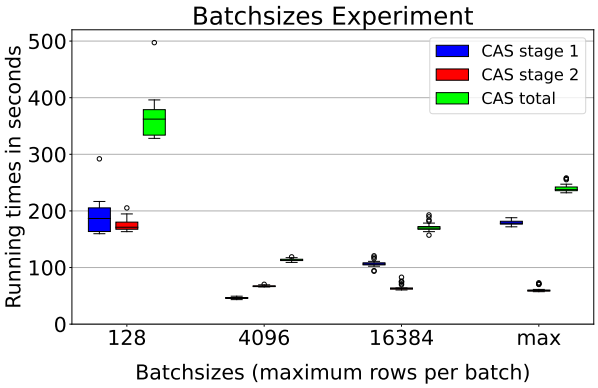
Figure 23 shows the results of this experiment. We plot each dataset in a different subplot. Within each subplot we show the different batchsizes. From left to right, these are: 128, 4096, 16384 and max, where max is `Integer.MAX_VALUE`. For each batchsize, we show the running times in seconds, displayed in a boxplot, for stage 1 (reading and shuffling), stage 2 (sorting), as well as the total running times.



(a) Results for 6.5 MiB dataset.



(b) Results for 172 MiB dataset.



(c) Results for 3.5 GiB dataset.

Figure 23: Results from Batchsizes Experiment.

One similarity we find among all subplots is that, indeed, too low batchsizes (128) show the most negative impact on performance. However, we find the most notable differences in performance for the 3.5 GiB dataset. We expect this is because the other datasets are too small to even come close to the largest batchsizes. Similar to the work of Rodriguez *et al* [33], the results for the 3.5 GiB dataset show that larger batchsizes are not always better. For CAS, the 3.5 GiB dataset benefits most from a batchsize of 4096 rows. Another difference with the other plots is that a too low batchsize may even affect stage 2 (sorting).

Running more extensive experiments may help with fine-tuning the optimal batchsize.

Similar as to Section 5.4.3, we also monitored the running times of several functions separately in this experiment. Specifically, we accumulated the total time spent in a single function per worker. Note that a high accumulated running-time does not always correspond to a high running-time in the experiment self, as it may be spread over many partitions. Thus, we cannot use these running times to compare which batchsize performs better, but only what the critical operators per batchsize are.

Figure 24 shows the maximum accumulated running times per batchsize, for the 3.5 GiB dataset. To keep the number of functions manageable, we left out functions with a lower accumulated running time than 100 seconds. The functions in the plots are:

- applyIndices: permute a batch according to a provided set of integers.

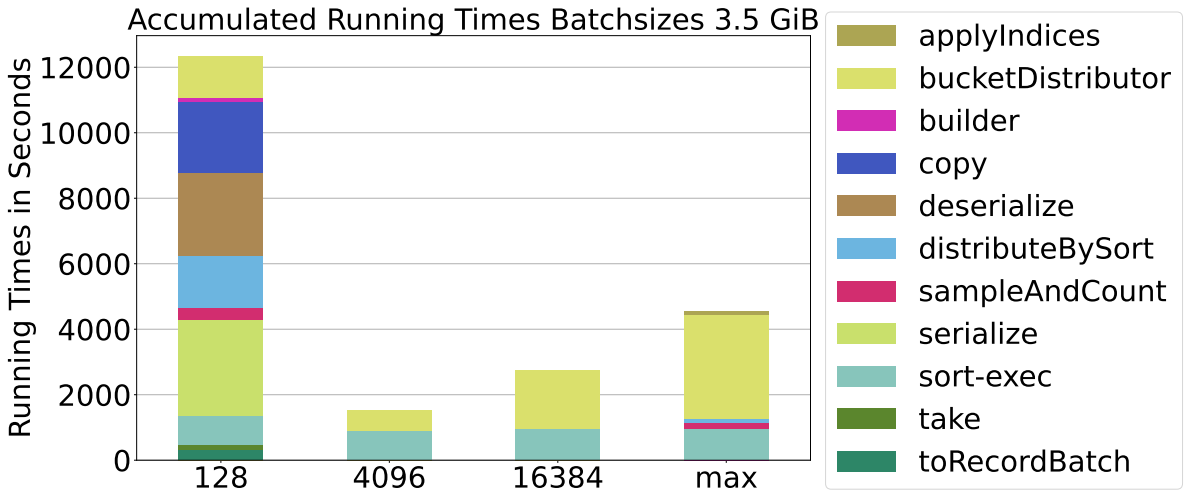


Figure 24: Aggregated Running Times from Batchsize Experiments.

- bucketDistributor: produce a mapping from batch-indices to corresponding partition-id.
- builder: usage of ArrowColumnarBatchRowBuilder.
- copy: create a reference of a(n) (slice of an) ArrowColumnarBatchRow.
- deserialize: deserialize one or more serialized batches.
- distributeBySort: distribute by sorting partition-ids.
- sampleAndCount: sampling method as described in Section 5.1.1.
- serialize: serialize ArrowColumnarBatchRows to shuffle.
- sort-exec: actual sorting.
- take: merge multiple ArrowColumnarBatchRows into one, using the ArrowColumnarBatchRowBuilder.
- toRecordBatch: convert an ArrowColumnarBatchRow to Arrow's Inter-Process Communication (IPC) format.

From the Figure we see that a batchsize of 128 rows indeed causes overhead in serialization and copying. A larger batchsize than 4096 rows causes the 'bucketDistributor' to run longer as there are more rows to distribute per batch. For the other extreme, we have maximum batchsizes of `Integer.MAX_VALUE` rows, for which we find that "applyIndices" and "sampleAndCount" take longer.

Insight 11

Within CAS, the maximum number of rows in a batch (batchsize) has influence on performance. Too small batches, such as those with 128 rows, give too much overhead in serialization and copying. Too large batches, such as those with `Integer.MAX_VALUE` rows, also affects performance negatively as each batch gets too much work. A good batchsize is 4096 rows, at least for (compressed) datasets up to 3.5 GiB.

5.5.3 Number of Shuffle Partitions

As a final configuration parameter, we consider the number of shuffle partitions, as we believe it may influence both stages of our driver program. For stage 1 (reading and shuffling) it affects the number of partitions to distribute to. We expect a lower number of shuffle partitions to affect this positively, as we can distribute batches faster over fewer partitions. For stage 2 (sorting) it affects the batchsizes to sort on. We expect a lower number of shuffle partitions to affect this negatively, as the work-per-batch will increase. However, a too high number of partitions may also affect this stage negatively, as the overhead could become too big.

Figure 25 shows the result of this experiment. We plot each dataset in a different subplot. Within each subplot we show the different number of shuffle-partitions. For each number of shuffle-partitions, we show the running times in seconds, displayed in a boxplot, for stage 1 (reading and shuffling), stage 2 (sorting), as well as the total running times.

Similar to previous experiments, we find the number of shuffle-partitions to affect the 6.5 MiB and 172 MiB datasets very little. Only for the smallest number of partitions in the 6.5 MiB dataset we find that it affects variance a lot. Although we also notice outliers in the same direction for the higher number of shuffle-partitions.

In the 3.5 GiB dataset we find the results are as expected. For the lower number of shuffle-partitions we see stage 1 and stage 2 are farthest apart, in favour of stage 1. As the number of shuffle-partitions increases, we find stage 1 runs somewhat slower, while stage 2 runs significantly faster. Although, 500 shuffle-partitions seem to give a negative effect to the stability of the runtimes. For the total running times, 200 shuffle-partitions (Spark's default), seem to be optimal. More extensive experiments could fine-tune this number.

Insight 12

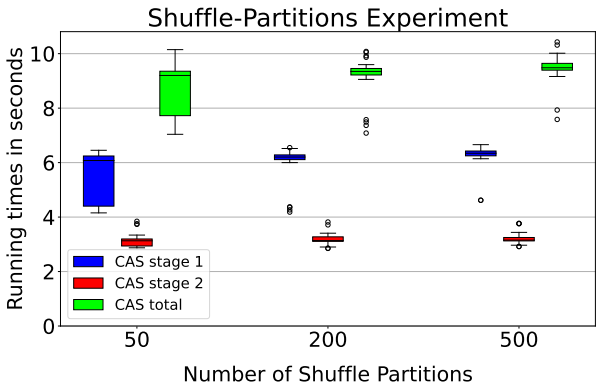
The amount of shuffle-partitions has influence over the running times of Complete Arrow Spark (CAS). More shuffle-partitions have a small negative impact on the read-and-shuffle stage, while they have a significant impact on the sort-stage. For the 3.5 GiB (compressed) dataset, an optimal number of shuffle-partitions seems to be 200.

5.5.4 Comparison with Vanilla Spark

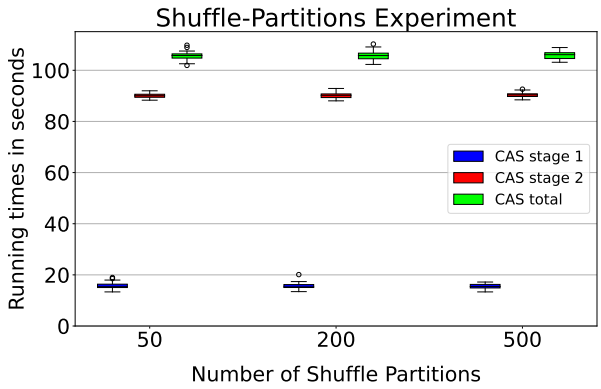
Finally, we combine all our optimal configurations to compare Complete Arrow Spark (CAS) with vanilla Spark.

Figure 26 shows the results of this comparison. We plot each dataset in a different subplot. For both CAS and vanilla Spark we show the running times in seconds in a boxplot, for stage 1 (reading and shufling), stage 2 (sorting) as well as the total running times.

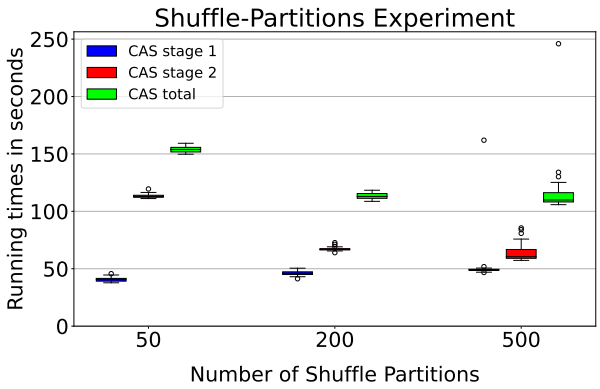
Unfortunately, all plots show that CAS is slower than vanilla Spark. However, there are still some differences between the different dataset sizes. For the 6.5 MiB dataset, we are around 1.5 times slower than Spark. Just like Spark, our stage 2 (sorting), is faster than our



(a) Results for 6.5 MiB dataset.



(b) Results for 17 MiB dataset.



(c) Results for 3.5 GiB dataset.

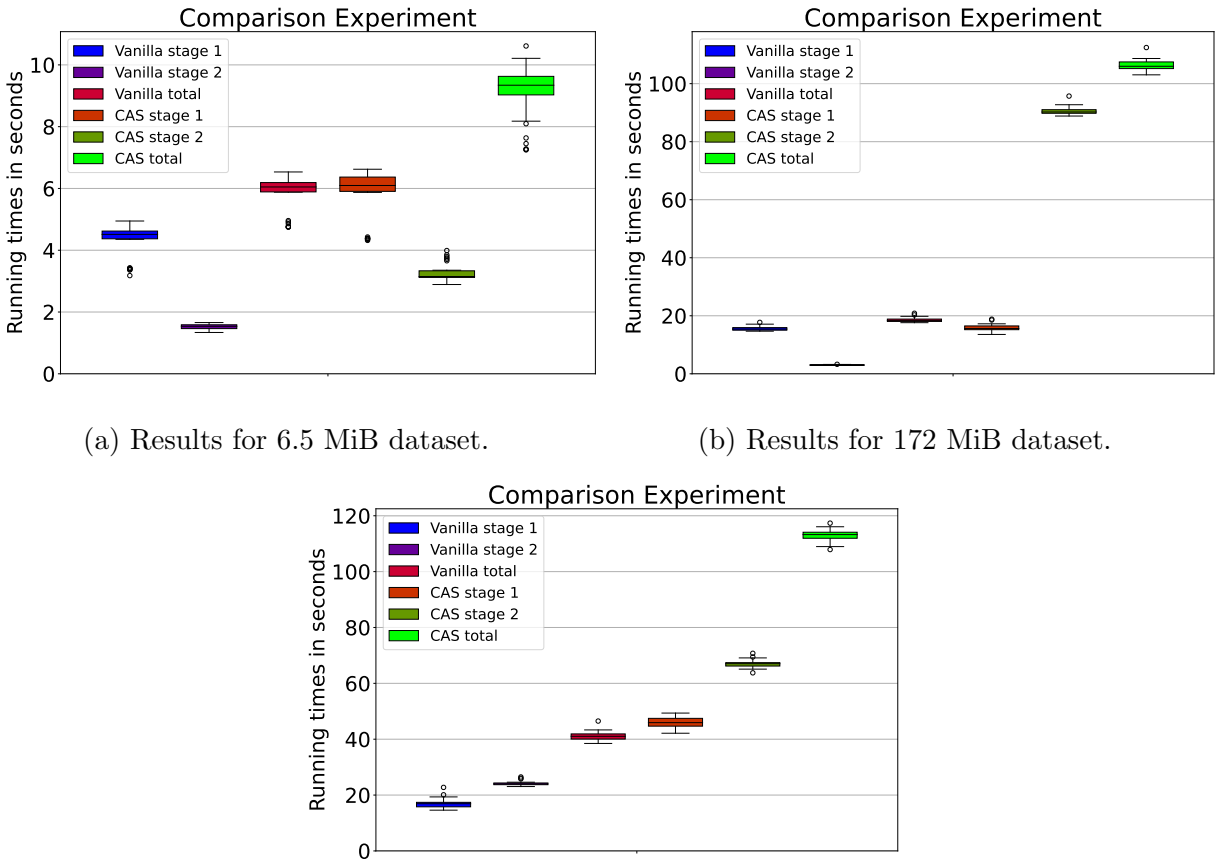
Figure 25: Results from Shuffle-Partitions Experiment.

stage 1 (reading and shuffling). For the 172 MiB dataset, our total running time is around 5.5 times slower than Spark. Interestingly this is only caused by stage 2 (sorting). For stage 1 (reading and shuffling), we are actually about as fast as Spark. For the 3.5 GiB dataset, our total running time is around 2.75 times slower than Spark. While our stage 2 (sorting) is still slower than stage 1 (reading and shuffling), we found that stage 2 is actually faster than stage 2 for the 172 MiB dataset. We expect this to be caused by the number of partitions. As the 172 MiB dataset is smaller than the 3.5 GiB dataset, it will produce fewer shuffle-partitions, which means the work-per-batch for sorting is larger.

With plenty of room for improvements, we are confident further development can bring the running times even closer to Spark's.

Insight 13

CAS is around 1.5 - 5.5 times slower than Spark for still relatively small datasets. However, we found the running times are sensitive to configurations which either cause too much overhead, or too much work-per-batch. Thus, we expect more extensive experimentation might bring our running times closer to Spark's. Additionally, more effort needs to be put into sorting, as this is slower in all configurations.



(a) Results for 6.5 MiB dataset.

(b) Results for 172 MiB dataset.

(c) Results for 3.5 GiB dataset.

Figure 26: Results from comparing CAS with vanilla Spark.

6 Conclusion

In this work we investigated how one can best integrate Apache Arrow in Apache Spark, without format-conversion overhead. For this, we started by analyzing previous work of Arrow integrations into Apache Spark. From the insights gathered during this analysis, we created Complete Arrow Spark (CAS), a Proof-of-Concept integration of Arrow into Spark, without format conversions.

Finally, we gather our main conclusions by answering the following sub-questions:

RQ1. **What can we learn from previous attempts at integrating Arrow's format into Spark?** From Native SQL Engine, we learned that a different setup and configuration can have a significant impact on the performance, even for similar queries and data-formats.

From SpArrow, we learned that we should pay close attention to size limits and memory management. Additionally, we found that the parquet-reader as used by SpArrow, becomes a bottleneck for larger (+0.5 GiB) datasets. Finally, SpArrow itself cannot be used for real-world applications in its current state, because of its size limits.

RQ2. **Is it feasible to create a Proof-of-Concept using the insights we gathered from previous experience?** We proved the feasibility of integrating Apache Arrow's format into Apache Spark, without having format-conversions. A significant challenge in this

implementation is the right use of reference management. Additionally, we investigated the impact of several strategies and configurations. We found that the parquet-reader strategy does not provide any differences in performance for the dataset sizes we evaluated on. What did matter, were the shuffle-strategies, number of shuffle-partitions, sorting-strategies, and the batchsizes. Finally, we compared our work with vanilla Spark, and found that we are around 1.5 - 5.5 times slower.

6.1 Future Work

As described in Section 5, there is still much work to be done before Complete Arrow Spark (CAS) is ready for real-world applications. In this Section, we list the most important future work that can be done.

Support Larger Datasets –
The current biggest limitation of CAS is the dataset sizes it can process. The most important future work will be to solve this issue. We expect this work to be focused on solving Arrow's reference management, and making current strategies robust against large amounts of data. In particular, we expect better bounds to the number of rows in ArrowColumnarBatchRows will help already in this aspect.

Reference Management –
As we learned from Section 5.2.3, reference management for Arrow within Spark is very tricky, and error-prone. To better manage this, one could redesign the ArrowColumnarBatchRow. In particular, it is important to realise that most of our problems are created through the Arrow API. Whenever we create a slice of an ArrowBuf, we receive a reference, which we need to manage. Probably, however, we only need one reference which we may release at task-completion. Thus, as a future work, one could replace all Arrow's slicing, by a custom slicer, which does not create references. We expect this will make reference management much more straightforward.

More Extensive Experiments –
In Section 5.4 and Section 5.5, we analyzed the performance impact of several configurations and strategies. We believe these configurations can be fine-tuned even more, such as batchsizes and the number of shuffle partitions. Additionally, once CAS can process larger amounts of data, the impact of the parquet-reader should be re-evaluated.

Extend supported operators –
Once CAS can efficiently multi-column sort large amounts of data, support for more operators can be added.

References

[1] Amazon. What is a columnar database? https://aws.amazon.com/nosql/columnar/, 2022. [Online; accessed 2022-07-26].

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[3] 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. In *IEEE Computer, Vol. 49, No. 5*, pages 54–63. IEEE, 2016.

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237. Citeseer, 2005.

[5] N BSc Laurens and P Arjen. Exploring query re-optimization. 2021.

[6] Databricks. Project tungsten: Bringing spark closer to bare metal. https://www.slideshare.net/databricks/spark-performance-whats-next, 2016. [Online; slide 5; accessed 2022-07-27].

[7] Statista Research Department. Amount of data created, consumed, and stored 2010-2025. https://www.statista.com/statistics/871513/worldwide-data-created/, 2022. [Online; accessed 2022-07-27].

[8] Statista Research Department. Importance of big data infrastructure technologies worldwide 2019. https://www.statista.com/statistics/919419/worldwide-critical-big-data-technology/, 2022. [Online; accessed 2022-07-27].

[9] Statista Research Department. Use of big data analytics in market research worldwide 2014-2020. https://www.statista.com/statistics/966892/market-research-industry-big-data-analytics, 2022. [Online; accessed 2022-07-27].

[10] Dhanya Thailappan. Understanding the basics of apache spark rdd. https://www.analyticsvidhya.com/blog/2021/08/understanding-the-basics-of-apache-spark-rdd/, 2021. [Online; accessed 2022-07-26].

[11] didierc. Vertical and horizontal parallelism. https://stackoverflow.com/a/15303852, 2013. [Online; accessed 2022-07-25].

[12] Dremio contributors. What is apache arrow? capabilities & benefits. https://www.dremio.com/resources/guides/apache-arrow/, 2022. [Online; accessed 2022-07-25].

[13] Grégory M Essertel, Ruby Y Tahboub, James M Decker, Kevin J Brown, Kunle Olukotun, and Tiark Rompf. Flare: Native compilation for heterogeneous workloads in apache spark. *arXiv preprint arXiv:1703.08219*, 2017.

[14] Robert Joseph Evans. Spip: Public apis for extended columnar processing support. https://issues.apache.org/jira/browse/SPARK-27396, 2022. [Online; accessed 2022-07-27].

[15] Federico Fiorini and Jan S. Rellermeyer. Sparrow: A spark-arrow engine: Leveraging the arrow in-memory columnar format to increase spark efficiency in rdd computations. Master's thesis, Delft University of Technology, 2021. [url: http://resolver.tudelft.nl/uuid:9aa12ee8-dcb2-4791-b751-7555332d7a18; accessed 2022-06-12].

[16] The Apache Software Foundation. Pyspark usage guide for pandas with apache arrow. https://spark.apache.org/docs/3.0.0/sql-pyspark-pandas-with-arrow.html, 2022. [Online; accessed 2022-07-27].

[17] Goetz Graefe. Volcano, an extensible and parallel query evaluation system; cu-cs-481-90. 1990.

[18] Hevo contributors. What is columnar database? – a comprehensive guide 101. https://hevodata.com/learn/columnar-databases/, 2021. [Online; accessed 2022-07-25].

[19] JavaTPoint contributors. Types of database parallelism. https://www.javatpoint.com/data-warehouse-types-of-database-parallelism, 2022. [Online; accessed 2022-07-25].

[20] Jean-Yves Stephan, Ocean for Apache Spark team. How to be successful with apache spark in 2021. https://www.datamechanics.co/blog-post/how-to-be-successful-with-apache-spark-in-2021, 2020. [Online; accessed 2022-07-26].

[21] Jules Damji, Databricks. How to use sparksession in apache spark 2.0. https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html, 2016. [Online; accessed 2022-07-31].

[22] Sercan Karagoz. Spark rdd (low level api) basics using pyspark. https://miro.medium.com/max/1400/1*2uwvLC1HsWpOsmRw4ZOp2w.png, 2020. [Online; https://medium.com/analytics-vidhya/spark-rdd-low-level-api-basics-using-pyspark-a9a322b58f61; accessed 2022-07-26].

[23] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, 2018.

[24] Micah Kornfield. Is there a size limitation/problem when writing the apache arrow format with the java api. https://stackoverflow.com/a/57986494, 2019. [Online; accessed 2022-07-25].

[25] Michael Ortega, Databricks. 5 reasons to become an apache spark expert. https://databricks.com/blog/2019/01/15/5-reasons-to-become-an-apache-spark-expert.html, 2019. [Online; accessed 2022-07-26].

[26] FatalErrors Morton. Hadoop series 3: distributed computing framework mapreduce. https://www.fatalerrors.org/images/blog/7dba45d44581f9f1be9fb869f3cb6db3.jpg, 2020. [Online; https://www.

fatalerrors.org/a/hadoop-series-3-distributed-computing-framework-mapreduce.html; accessed 2022-07-25].

[27] Neha Vaidya. Apache spark architecture – spark cluster architecture explained. https://www.edureka.co/blog/spark-architecture/, 2022. [Online; accessed 2022-07-26].

[28] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[29] F.M. Nonnenmacher and Z. Al-Ars. Transparently accelerating spark sql code on computing hardware. Master's thesis, Delft University of Technology, 2020. [url: hhttps://repository.tudelft.nl/islandora/object/uuid:f588ca1d-e4ae-4bf4-96ed-221d483b559d?collection=education; accessed 2022-07-25].

[30] OAP Contributors. Optimized analytics package (oap) – overview. https://oap-project.github.io/latest/, 2022. [Online; accessed 2022-07-27].

[31] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[32] Vladimir Pligin. A fatal error has been detected by the java runtime environment when ignite native persistence is on. https://stackoverflow.com/a/67787771, 2021. [Online; accessed 2022-07-25].

[33] Sebastiaan Alvarez Rodriguez, Jayjeet Chackrabroty, Aaron Chu, Ivo Jimenez, Jeff LeFevre, Carlos Maltzahn, and Alexandru Uta. Zero-cost, arrow-enabled data interface for apache spark. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2400–2405. IEEE, 2021.

[34] Sameer Agarwal, Davies Liu and Reynold Xin. Apache spark as a compiler: Joining a billion rows per second on a laptop. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html, 2016. [Online; accessed 2022-07-26].

[35] Sameer Agarwal, Davies Liu and Reynold Xin. Apache spark as a compiler: Joining a billion rows per second on a laptop. https://databricks.com/wp-content/uploads/2016/05/volcano-vs-hand-written-code-1024x397.png, 2016. [Online; https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html; accessed 2022-07-26].

[36] Sarfaraz Hussain. Understanding the working of spark driver and executor. https://blog.knoldus.com/understanding-the-working-of-spark-driver-and-executor/, 2019. [Online; accessed 2022-07-26].

[37] Shalini Goutam. Apache spark logical and physical plans. https://blog.clairvoyantsoft.com/spark-logical-and-physical-plans-469a0c061d9e, 2021. [Online; accessed 2022-07-26].

[38] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40, 2011.

[39] StichData. Oltp and olap: a practical comparison. https://www.stitchdata.com/resources/oltp-vs-olap, 2022. [Online; accessed 2022-07-26].

[40] Siddharth Teotia and Srini Penchikala. Columnar databases and vectorization. https://www.infoq.com/articles/columnar-databases-and-vectorization/, 2018. [Online; accessed 2022-07-25].

[41] The Apache Software Foundation. Apache arrow: A cross-language development platform for in-memory analytics – what is arrow? https://arrow.apache.org/, 2022. [Online; accessed 2022-07-25].

[42] The Apache Software Foundation. Apache arrow overview. https://arrow.apache.org/overview/, 2022. [Online; accessed 2022-07-25].

[43] The Apache Software Foundation. Apache arrow: Quick start guide. https://arrow.apache.org/docs/java/quickstartguide.html, 2022. [Online; accessed 2022-07-26].

[44] The Apache Software Foundation. Cluster mode overview. https://spark.apache.org/docs/latest/img/cluster-overview.png, 2022. [Online; https://spark.apache.org/docs/latest/cluster-overview.html; accessed 2022-07-26].

[45] The Apache Software Foundation. Cluster mode overview – cluster manager types. https://spark.apache.org/docs/latest/cluster-overview.html, 2022. [Online; accessed 2022-07-26].

[46] The Apache Software Foundation. Mapreduce tutorial. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html, 2022. [Online; accessed 2022-07-25].

[47] The Apache Software Foundation. Spark sql, dataframes and datasets guide. https://spark.apache.org/docs/3.0.0/sql-programming-guide.html, 2022. [Online; accessed 2022-07-31].

[48] Tilak Patidar. Vectorized and compiled queries — part 1. https://medium.com/@tilakpatidar/vectorized-and-compiled-queries-part-1-37794c3860cc, 2018. [Online; accessed 2022-07-26].

[49] Tilak Patidar. Vectorized and compiled queries — part 2. https://medium.com/@tilakpatidar/vectorized-and-compiled-queries-part-2-cd0d91fa189f, 2018. [Online; accessed 2022-07-26].

[50] Tilak Patidar. Vectorized and compiled queries — part 2. https://miro.medium.com/max/1400/1*krn3NPwCERy6hk0xKB5CZQ.png, 2018. [Online; https://medium.com/@tilakpatidar/vectorized-and-compiled-queries-part-2-cd0d91fa189f; accessed 2022-07-26].

[51] Tilak Patidar. Vectorized and compiled queries — part 3. https://medium.com/@tilakpatidar/vectorized-and-compiled-queries-part-3-807d71ec31a5, 2018. [Online; accessed 2022-07-26].

[52] Wikipedia contributors. User-defined function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=User-defined_function&oldid=1041475611, 2021. [Online; accessed 26-July-2022].

[53] Wikipedia contributors. Apache spark — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Spark&oldid=1096788006, 2022. [Online; accessed 26-July-2022].

[54] Wikipedia contributors. Directed acyclic graph — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Directed_acyclic_graph&oldid=1099254255, 2022. [Online; accessed 26-July-2022].

[55] Wikipedia contributors. Domain-specific language — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=1091164524, 2022. [Online; accessed 26-July-2022].

[56] Wikipedia contributors. Field-programmable gate array — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=1090640152, 2022. [Online; accessed 26-July-2022].

[57] Wikipedia contributors. Infiniband — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=InfiniBand&oldid=1092818112, 2022. [Online; accessed 2022-07-04].

[58] Wikipedia contributors. Inter-process communication — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Inter-process_communication&oldid=1099231051, 2022. [Online; accessed 28-July-2022].

[59] Wikipedia contributors. Java virtual machine — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=1098980164, 2022. [Online; accessed 26-July-2022].

[60] Wikipedia contributors. Parallel computing — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=1095305790, 2022. [Online; accessed 2022-07-25].

[61] Wikipedia contributors. Single instruction, multiple data — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Single_instruction,_multiple_data&oldid=1090482442, 2022. [Online; accessed 26-July-2022].

[62] Wikipedia contributors. Zettabyte era — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Zettabyte_Era&oldid=1095807873, 2022. [Online; accessed 27-July-2022].

[63] Reynold Xin. Apache spark the fastest open source engine for sorting a petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html, 2014. [Online; accessed 2022-07-30].

[64] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[65] zetaprime. Differences between spark's row and internalrow types. `https://stackoverflow.com/a/56763598`, 2019. [Online; accessed 2022-07-31].