Transparently Accelerating Spark SQL Code on Computing Hardware

by

Fabian Nonnenmacher

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday August 19, 2020 at 2:00 PM.

Student number:	5154006	
Thesis number	Q&CE-CE-MS-2020-09	
Project duration:	February 1, 2020 – August 19, 2020 (30 ECTS)	
Thesis committee:	Dr. Zaid Al-Ars,	TU Delft, supervisor
	Prof. Peter Hofstee,	TU Delft, IBM Austin
	Dr. Claudia Hauff,	TU Delft
	Dr. Joost Hoozemans,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Through new digital business models, the importance of big data analytics continuously grows. Initially, data analytics clusters were mainly bounded by the throughput of network links and the performance of I/O operations. With current hardware development, this has changed, and often the performance of CPUs and memory access became the new limiting factor. Heterogeneous computing systems, consisting of CPUs and other computing hardware, such as GPUs and FPGAs, try to overcome this by offloading the computational work to the best suitable hardware.

Accelerating the computation by offloading work to special computing hardware often requires specialized knowledge and extensive effort. In contrast, Apache Spark became one of the most used data analytics tools, among other reasons, because of its user-friendly API. Notably, the component Spark SQL allows defining declarative queries without having to write any code. The present work investigates to reduce this gap and elaborates on how Spark SQL's internal information can be used to offload computations without the user having to configure Spark further.

Thereby, the present work uses the Apache Arrow in-memory format to exchange data efficiently between different accelerators. It evaluates Spark SQL's extensibility for providing custom acceleration and its new columnar processing function, including the compatibility with the Apache Arrow format. Furthermore, the present work demonstrates the technical feasibility of such an acceleration by providing a Proof-of-Concept implementation, which integrates Spark with tools from the Arrow ecosystem, such as Gandiva and Fletcher. Gandiva uses modern CPUs' SIMD capabilities to accelerate computations, and Fletcher allows the execution of FPGA-accelerated computations. Finally, the present work demonstrates that already for simple computations integrating these accelerators led to significant performance improvements. With Gandiva the computation became 1.27 times faster and with Fletcher even up-to 13 times.

Preface

Throughout the EIT Master School, the last two years become an enriching experience, which helped me to grow personally. I am thrilled that I had the chance to get to know two universities and live in two foreign countries. Living abroad, away from friends and family, sometimes can be scary. Therefore, I am thankful for meeting so many incredible people who made these two years an exceptional experience and become close friends.

I am grateful for being a part of the ABS group at the TU Delft Computer Engineering department for the last six months. I met there many inspirational people who are passionate about their vision of making FPGAs more accessible to Big Data applications. I am also proud that I was able to contribute to their vision and give something back to the support I have received during the work on my thesis. This excellent experience was only disturbed by the Corona situation's restrictions, which were out of the department's control. On the contrary, the team was putting a lot of effort into compensating these restrictions by setting up weekly meetings and tools for being available for us students.

I would particularly like to thank my supervisor Dr. Zaid Al-Ars, whose support and guidance always was exceptional and far beyond his mandatory duty. The work during my thesis was particularly motivating because of Zaid's excitement about all the small steps I achieved and his inspirational way of sharing his vision. I remember discussing with him after the first month when I was overwhelmed by the many possibilities and directions to go. Then, the discussions with him and his guidance helped me shape the goal of my work and, thereby, not to get distracted by too specific details.

While Zaid ensures all individual projects contribute to the big picture, his team are reliable supporters, who understand the used technologies into the smallest details. I want to express special thanks to Joost Hoozemans and Johan Peltenburg. They were always available for (technical) questions and discussion. Thereby, they helped me understand different parts of the big picture and decide which ideas are worth following.

Furthermore, I want to thank my fellow EIT companions, Héctor Bállega, and Shashank Aggarwal. They got stuck with me in Delft during the Corona lockdown and made it an unforgettable experience, despite the limited possibilities. Thanks for the excellent time and especially for the extraordinary BBQs we had!

Last, but definitely not least, I want to appreciate the unlimited support of my family. Knowing that I always have a safe harbor gave me the confidence to start this international adventure. Thank you for all the support and for giving me the freedom to gather new experiences.

Fabian Nonnenmacher
Eppingen, 3rd August 2020

Contents

List of Figures

List of Acronyms

ABS Accelerated Big Data Systems

API application programming interface

AWS Amazon Web Services

CPU central processing unit

DAG directed acyclic graph

DSL domain specific language

FPGA field-programmable gate array

GPU graphics processing unit

HAF hardware-accelerated function

HDL hardware description language

IPC interprocess communication

JDBC Java Database Connectivity

JIT Just-in-time

JMH Java Microbenchmark Harness

JNI Java Native Interface

JVM Java Virtual Machine

LLVM Low Level Virtual Machine

LSB least-significant bit

ODBC Open Database Connectivity

OS operating system

PoC Proof of Concept

RDD Resilient Distributed Dataset

RMI remote method invocation

SIMD single instruction multiple data

SQL Structured Query Language

SSD solid-state drive

UTF-8 8-Bit Universal Coded Character Set Transformation Format

1 Introduction

1.1. Context

With the creation of new digital business models, the amount of data created every day is increasing dramatically. In 2018, 33 zettabytes of new data have been created, which is 375 million times more than the size of the internet in 1997. Many companies join together and analyze the data, gathered from their users, and production processes to improve existing business models and to find new business opportunities. It is essential for them to be able to analyze a vast amount of data quickly and cost-efficiently [61].

Apache Spark is a unified analytics engine for distributed large-scale data processing. It was started in 2009 by a research project at UC Berkeley. Since then, Spark's popularity has increased rapidly and it has become one of the most used big data analytics platforms. Apache Spark is now used in many industries, including large internet companies such as eBay and Netflix. Furthermore, with more than 1000 contributors, it has developed to being the largest open source community in big data [15].

Besides the core component, which includes managing the memory, distributing the data, and coordinating the execution in a cluster, Apache Spark contains multiple modules to support further data-processing use cases such as streaming data and machine learning [15]. One of these modules is *Spark SQL*, which enables Spark to process structured data. This module integrates relational processing into Spark and allows the users to define their data processing queries in declarative style (e.g. SQL). Furthermore, it includes the highly extensible optimizer *Catalyst*, which optimizes the defined query for better performance before executing it [1].

There are many reasons for Spark's popularity, in particular, its fast in-memory processing, its ability to distribute workloads in clusters and the rich functionality provided by the whole ecosystem. Furthermore, many different blogs [6, 16, 63] discuss the easy-to-use APIs as an important reason for its rapid growth in popularity. Notably, the module, Spark SQL, allows defining complex data processing workloads with a short and precise code. This user-friendly API leads to a productivity boost for writing new queries and likewise improves the maintainability by providing helpful debugging information.

With the increasing popularity of new hardware such as SSDs and 10 Gps network links, the costs of IO operations have decreased and the CPU and memory have become the new performance bottlenecks. To address this problem, Project Tungsten was started, which includes several changes to the execution engine to increase its efficiency [56].

Nevertheless, these optimizations can also not overcome the fact that a CPU has limitations in executing computations in parallel. For many data-intensive workloads, it is, therefore, beneficial to make use of the SIMD capabilities of modern CPUs [32] or to offload computation-intensive work to graphics processing units (GPUs) [43].

Other computing accelerators are field-programmable gate arrays (FPGAs). These hardware chips contain a two-dimensional array of logic gates that can be reprogrammed. FPGAs allow developers to program an application-specific integrated circuit into the chip after fabrication. General-purpose CPUs are restricted in their ability to execute operations in parallel. In comparison, FPGAs do not have this limitation and can process data often faster and with lower energy consumption than CPUs [8].

With the exploding volume of data and the increasing complexity of processing pipelines, it is nearly impossible for traditional computer architectures to keep up with the arising performance requirements. Hence, many experts in high-performance computing are working on *heterogeneous computing* platforms that accelerate data analytics applications by offloading parts of the workload to suitable hardware accelerators such as GPUs and FPGAs [24].

Spark has also recognized the trend of heterogeneous computing. By adding new features, such as columnar data processing (Spark-27396) and accelerator-aware task-scheduling (Spark-24615), to the recently released major version 3.0.0, they have laid a foundation for integrating Spark with different hardware accelerators.

The Accelerated Big Data Systems (ABS) group at the TU Delft Computer Engineering department is also working on making the vision of heterogeneous computing platforms reality. With their Fletcher framework, they simplify the integration of vendor-specific FPGAs into other data processing tools [48]. This integration is based on Apache Arrow, a language-agnostic in-memory columnar format that enables data exchange between different processes without serialization overhead [3]. Spark is also compatible with the Arrow format and uses it mainly to transfer data to and from python processes [20]. Similarly, this work contributes to the greater vision of the ABS group by studying Spark's heterogeneous computing features. It analyzes how Spark queries can be executed on different hardware accelerators based on the Apache Arrow format without losing the benefits of the user-friendly API.

1.2. Contribution

As described in the previous section, the module Spark SQL provides an API where a data scientist describes a data analytics query in a declarative style, which is understood structurally. The main aim of the present work is to accelerate the query execution by using this structural information to offload parts of the query to other hardware accelerators.

The use of hardware accelerations often requires specific knowledge to decide if acceleration is beneficial. The vision is that a data scientist uses Spark's user-friendly API and defines a declarative query, and an algorithm uses execution statistics to decide on using the available hardware accelerators as efficiently as possible. Following this visionary idea, this work hides the acceleration and uses the query's internal structure to accelerate the execution *transparently*.

Motivated by the other work of the ABS group, this work concentrates on integrating Spark with two hardware accelerators from the Apache Arrow ecosystem. Firstly, this work evaluates the suitability of Gandiva [51], which uses modern CPU SIMD capabilities to process Arrow data. Secondly, it integrates Spark SQL with Fletcher [48] to accelerate the execution with FPGAs. Additionally to the accelerators, this work also integrates Spark with the Arrow Parquet file reader, to demonstrate a whole workflow based on the Arrow format.

The main contribution of this work is the elaboration of the technical feasibility of the described integrations. The functionality and maturity of the associated technical components are analyzed and a Proof of Concept (PoC) is implemented to demonstrate the workability. Nevertheless, the implementation does not fully integrate a specific feature set. Instead, it solves several significant challenges and demonstrates which additional challenges exist and should be addressed in further work.

Even though Spark's key functionality is to distribute computational work in a cluster, this work focuses on accelerating the execution on a single-node. Spark divides the data into chunks (*partitions*) and distributes them to different executor threads, which work in parallel [66]. However, this work does not consider any challenges related to the distributed setup (e.g. availability of hardware accelerators). Instead, it concentrates on a single executor and improves the performance of the individual execution. Nevertheless, the implementation is fully compatible with Spark's executor model and can also be executed in a distributed setup as long as the required hardware accelerators are available.

Finally, the work evaluates the performance of the PoC implementation on some exemplary use cases. It demonstrates first performance improvements and provides insights on necessary preconditions for powerful acceleration.

1.3. Research question

The main goal of this work is described by the following main research question:

Can Spark SQL's internal structural information of the query be used to accelerate the query execution by offloading work to hardware accelerators based on Apache Arrow?

Answering this question requires the analysis of the technologies, a PoC implementation to gain further insights, and an evaluation of the performance improvements. To give a well-founded answer to the main question, the following subquestions will also be answered.

1. Does Spark SQL provide sufficient extension points that allow the provision of different hardware accelerators?
2. How mature is Spark's columnar processing function and is it compatible with the Apache Arrow memory format?
3. Which performance improvements can be identified and what are potential bottlenecks?

1.4. Outline

Chapter 2 introduces the tools integrated during the present work and provides their internal technical details necessary for the integration described later. This chapter introduces Apache Spark, Apache Arrow, and Fletcher. Thereby it focuses on the components and features relevant for this work.

Based on this background information, Chapter 3 presents the general architecture of the present work and introduces general concepts necessary for integrating accelerators with Spark. Firstly, it presents how Spark can be extended with custom columnar processing implementations. Secondly, it discusses necessary modifications, so that Spark's columnar processing functionality becomes compatible with the Apache Arrow format. Thirdly, it presents the present work's approach to exchange Arrow data between Java/Scala code and native C++ libraries, which enables Spark to use accelerators not implemented in Java.

Afterward, these concepts are applied to concrete accelerator implementations presented in Chapter 4. This includes the Dataset Parquet Reader and Gandiva from the Arrow project and the Fletcher runtime, which allows executing FPGA-accelerated functions on Arrow Data.

Chapter 5 evaluates the performance of these accelerated implementations by comparing them to the unmodified Vanilla Spark. This chapter discusses the effect of the different accelerators and evaluates the impact of the batch size.

Finally, Chapter 6 summarizes the results and reflects on the formulated research questions.Furthermore, it shows unsolved challenges and further work necessary to make the vision of a Spark-based heterogeneous computing platform, including Fletcher, reality.

2 Background

2.1. Spark SQL

Spark SQL is a module that enables Spark to process structured data. Additionally to the underlying Spark RDD API, Spark SQL allows the users to define declarative queries and uses the structure of the queries internally together with the structure of the data to optimize the execution [20].

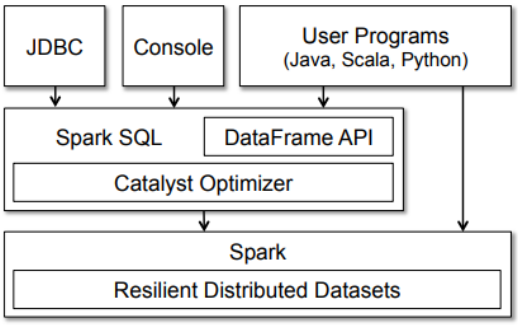


Figure 2.1: Interfaces to Spark SQL, and interaction with Spark [1]

Spark SQL is built on top of the functional programming API and extends Spark's functionality (Figure 2.1). It exposes an SQL interface, which can be accessed through JDBC/ODBC or a command-line tool. Therefore, end-users and applications can interact with Spark SQL without the users having to write any code. Furthermore, Spark SQL includes the DataFrame API, which can be called from all programming languages supported by Spark [1, 20].

This work evaluates several very recent features of Spark. Hence, the implementation and testing done during this work are based on Apache Spark version 3.0.0 [60], which is the most recent version at the time of writing. This new major version was released in June 2020 and was not available at the beginning of this work. Consequently, the first evaluations and implementations began with the publicly available preview version (`3.0.0-preview2`) but were later migrated to the officially released version.

2.1.1. Spark RDD API

Apache Spark is a general-purpose cluster computing engine that offers a functional programming API, allowing users to manipulate Resilient Distributed Datasets (RDDs) that are data collections distributed among different nodes [1]. These RDD abstractions are specially designed for efficient fault-tolerance and data reuse. In contrast, to other frameworks, Spark

also provides an abstraction for distributed memory and allows the users to explicitly persist intermediate results in memory and control the partitioning of the data [66].

RDDs are an immutable collection of records that can be created by referencing data in external storage systems (e.g. shared filesystem) or transforming other RDDs. A transformation describes a deterministic operation such as *map*, *filter*, or *join*, that can be applied to one or multiple RDDs and creates a new RDD. The new RDD then stores the information how it was derived from other RDDs. As a result, the RDD does not need to be materialized the whole time. Instead, the information can be used to recreate the RDD at any time (e.g. in a case of failure) [66].

The second type of operation is an *action*. This is similar to a *transformation*, as it is applied to an RDD. However, in contrast, it returns a materialized result value. Therefore, executing an *action* triggers a computation. In Contrast, *transformations* are evaluated lazily. They are only computed when required by an action. Internally, Spark creates a lineage directed acyclic graph (DAG), which stores the dependencies between the RDDs and only executes the transformations when required by an *action* to compute a result [19, 66].

In general, there are two different types of transformations. Firstly, there are the *narrow* transformations, where each partition of the parent RDD can only be used once to compute a partition of the resulting RDD. Secondly, there are wide transformations where partitions of the parent RDD may be used many times. Examples of the different types of transformations are shown in Figure 2.2 [66].

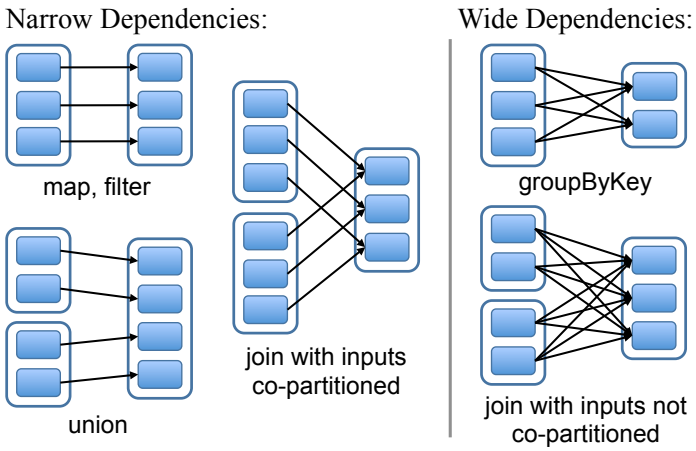


Figure 2.2: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles [66].

To use Spark, developers write a *driver program* by using the Spark RDD API. As shown in Figure 2.3, this driver program controls the lineage DAG and forwards partitions (subsets) of RDDs to a cluster of worker nodes. The driver program then invokes the computation of the transformations and actions and gathers the results from the worker nodes [66].

In the distributed setup, it becomes apparent that *narrow transformations* can be executed much faster than wide ones. For narrow operations, a worker node can compute the partition of a new RDD from the partitions of the parent RDD without having to exchange any data with the other worker nodes. Additionally, further optimization, such as pipelining, can be applied, whereby multiple transformations may be grouped and executed in one pass. In contrast, wide transformations are slow. They require a data exchange (*shuffle*), which has a negative impact on the performance because of the network latency.

As described in the previous chapter, this work focuses on the evaluation of different approaches for the improvement of a single worker node's performance. Translated into Spark's

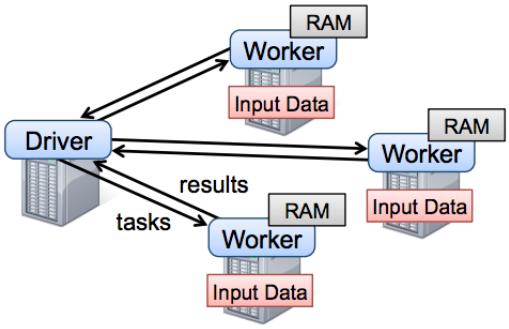


Figure 2.3: Spark runtime. The user's driver program coordinates the computation on a worker node cluster [66].

terminology, this means this work focuses on improving the execution of *narrow transformations*.

2.1.2. DataFrame API

As described above, the DataFrame API allows users to control Spark SQL from another programming language. At the time of writing, the API is available for Scala, Java, Python, and R [20]. The central abstraction is a *DataFrame* which is equivalent to a table in a relational database. Simultaneously, a DataFrame can also be viewed as an RDD of rows. The combination of these two principles allows the user to manipulate a DataFrame with either relational operators, such as `where` and `groupBy`, or with procedural operators similar to the RDD API, such as `map` or `filter` [1].

Apart from the relational API, a user can register a DataFrame as a temporary table and query it using SQL [1]. With version 1.6, Spark introduced the additional DataSet abstraction which extends the API with a strongly-typed, object-based API for procedural operators [41]. The following code example shows on a simple use case that all three approaches can be mixed easily to create functionally equivalent computations.

```
1  // import a json file as DataFrame
2  val df: DataFrame = spark.read.json("employees.json")
3
4  // Using the DataFrame API
5  val r1 = df.where(col("age") < 30).count()
6
7  // Using a temporary table with pure SQL
8  df.createTempView("employees")
9  val r2 = spark.sql("SELECT count(*) FROM employees WHERE age < 30 ")
10
11 // Using the strongly-typed DataSet abstraction
12 case class Employee(name: String, age: Long)
13 import spark.implicits._
14 val dataset = df.as[Employee] // map each record to the Employee class
15 val r3 = dataset.filter( employee => employee.age < 30).count()
```

Listing 2.1: Three different ways of using the DataFrame API

The DataFrame operations are similar to the RDD transformations in that they are lazy and are only executed when an "output operation" is called. The sequence of operations defined by the user represents a *logical plan* which is then optimized before execution [1].

The three approaches from the previous example are functionally equivalent, but their logical plan representations are not. In particular the lambda expression passed to the `filter`

method of the DataSet API is executed within the JVM as in any other java function. As a result, Spark SQL does not understand the lambda expression structurally and is, therefore, limited in the abilities to optimize the lambda expression or to execute it in another environment. Analyzing the JVM byte code would be one solution to overcome this limitation, which is proposed and discussed in feature Spark-14083 of the Spark backlog [59]. However, this work highly depends on understanding the structure of the logical plan. Therefore, only logical plans created by the other approaches are considered in the implementation of this work.

The DataFrame API supports atomic SQL types such as boolean, integer, double, decimal, and string and complex types such as structs and arrays [1]. This work analyzes the technical feasibility of integrating Spark with other tools and focuses only on the simple case, the atomic data types.

2.1.3. Catalyst - the query optimizer

As already discussed, one of the main advantages of using Spark SQL is that it uses the structural information of the data and the query to optimize the latter before executing it. Within Spark SQL, the responsible component is called Catalyst. This optimizer follows an extensible design so that new optimization techniques can be added easily by internal and external developers. For adding new rules to Catalyst, no complex domain specific language (DSL) is required. Instead, Catalyst is based on functional features of the Scala programming language, such as pattern-matching [1].

Within Catalyst, every query is represented by a *tree* of operations (e.g. filter) and expressions (e.g. “x > 5”) that can be manipulated by applying *rules* to them. The most common way to define a rule is to use Catalyst’s `transform` method, which iterates through a tree and replaces every subtree that matches a particular pattern [1].

By using Scala’s pattern matching syntax, these rules can be implemented quite easily. For example, the following code snippet defines a rule for *constant folding* of additions. When applying this rule to the expression `x+(1+2)`, this results in the new expression `x+3` (see Figure 2.4) [1].

```
1 tree.transform{
2   case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
3 }
```

Listing 2.2: Constant folding rule defined with pattern matching

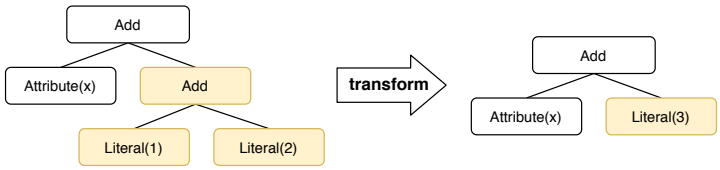


Figure 2.4: Applying constant-folding rule to the Catalyst tree for the expression `x+(1+2)`

As shown in Figure 2.5, the optimization of Catalyst is conducted in 4 phases: Firstly, the logical plan is analyzed and all references are resolved. Secondly, the plan is logically optimized and then mapped to different physical executors. Finally, the best physical mapping is chosen on a cost-based model and used to generate Java byte code. The following describes these 4 phases in detail [1].

As described in the previous section, there are multiple ways to define a Spark SQL query. However, all approaches result in a tree of operations and expression, which is called an *unresolved logical plan*. During the analysis phase, all references (e.g. column names) are

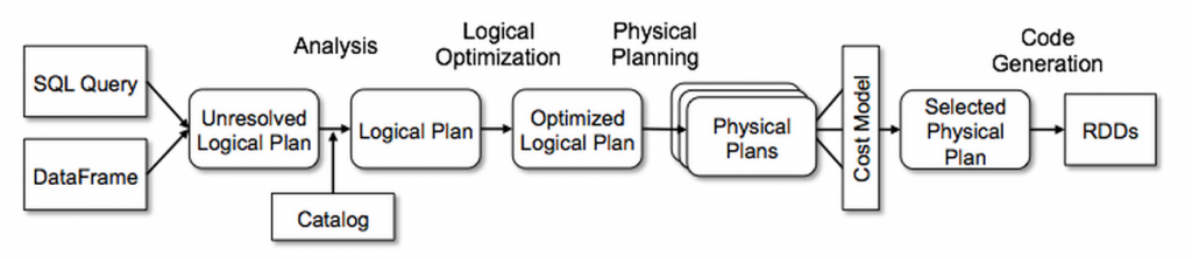


Figure 2.5: Phases of query planning in Spark SQL [1]

resolved and their data types are determined. This information is then used to validate the query, for example, by ensuring that the data types are compatible between operations [1].

The resolved logical plan is then optimized for a faster execution by applying a set of rules. These rules include constant folding, predicate pushdown, projection pruning and boolean expression simplification [1].

The next phase is to find the best way to execute a logical plan. Every logical operation is mapped to at least one potential physical operator, which is a concrete implementation of logical operations containing detailed instructions to compute the desired result [41]. Conceptually, this mapping could lead to multiple variants from which a cost-based model chooses the best fitting one. However, the current implementation of Spark maps most logical operations to precisely one physical operator. A different physical operator is only used for Join operations on small data sets. Therefore, there is no need for a complex cost-based model and it has not been implemented at the time of writing [1].

Additionally, in this phase, further physical optimizations such as *predicate pushdown* are applied. *Predicate pushdown* means that some filter operations are not executed by Spark, but instead directly by the data source. As a result, fewer data must be loaded into the memory or transferred over the network. For example, Parquet files are organized in chunks and store various statistics about each chunk's data, such as the maximum value of a column. When applying predicate pushdown, these statistics are used to decide early if a chunk contains data matching the filter expression and the loading of not matching chunks into memory can be avoided [12, 41].

Especially for operations parameterized with expression trees, such as filter or projection, the evaluation of the expression trees is a costly operation. For avoiding this, Catalyst generates Java byte code from the expression tree and applies this generated code to every row, which is processed. With the *Wholestage Code Generation*, introduced in Subsection 2.1.4, Spark enhanced this functionality and started to generate Java byte code from the whole physical plan and not only from the expressions of one physical operator.

Catalyst's extensible design makes it easy for third-party developers to add new rules to all phases of the optimization. The present work, uses this API extensively, to provide different implementations for certain logical operations.

2.1.4. Project Tungsten

The use of SSDs and faster Ethernet connections led to a massive increase in I/O performance. CPU and memory access has now become the new bottleneck in big data processing [41]. Project Tungsten was started to overcome this, combining different changes to the execution engine to improve the efficiency of memory and CPU [14]. In the following, some of these changes, such as memory management and code generation are presented in-depth. Then, columnar processing for better SIMD support is discussed in the next section.

Memory management

Generally the *JVM garbage collector* is responsible for managing the whole life cycle of Java objects and their memory usage. The garbage collector is a very complex and powerful component optimized for a wide variety of applications. However, Spark wants to achieve the highest performance possible and wants to avoid the overhead through the Java memory layout and the garbage collector [41, 64].

The memory layout of Java objects is designed for typical workloads and is not optimized for small memory consumption. Every object stored in memory contains additional headers and hash codes. Which results in a simple 4 byte UTF-8 String object consuming 48 bytes memory [64].

The garbage collector continuously monitors the references of an object. After it detects no reference exist anymore, it destroys the object and frees the memory. To manage the objects as efficiently as possible, it estimates the life span based on many heuristics. However, whenever this estimate is wrong, the garbage collector is not handling the objects ideally. In contrast to the garbage collector, Spark manages the data flow through the computation and knows the life-span of the data objects exactly. It can use these insights to manage the memory more efficiently [41, 64].

For managing the memory format of the data, Spark has introduced in version 1.4 the `UnsafeRow`, a binary representation of a data row. The implementation is based on Java's internal `sun.misc.Unsafe` package, which provides advanced functionality that allows C-style memory access, such as explicit allocation, deallocation and pointer arithmetics [64].

The memory layout of an `UnsafeRow` consists of three different regions (Figure 2.6). The first region, the *null bit set region*, indicates, with a bit, whether a value of the field is null or not. This region is beneficial for non-null filtering because, for this, loading the actual value is not necessary. The *fixed-length value region* has reserved an 8 byte spot for every field of the row. Values fitting into 8 bytes such as `int`, `long` or `double` are stored directly in their reserved spot. When the values are larger (or undefined), they are stored instead in the *variable-length value region*. The spot in the *fixed-length value region* is used to refer to the location by storing the starting offset and length. Additionally, all regions are 8-byte aligned, so that they fit exactly into 64 bit CPU registers [41].

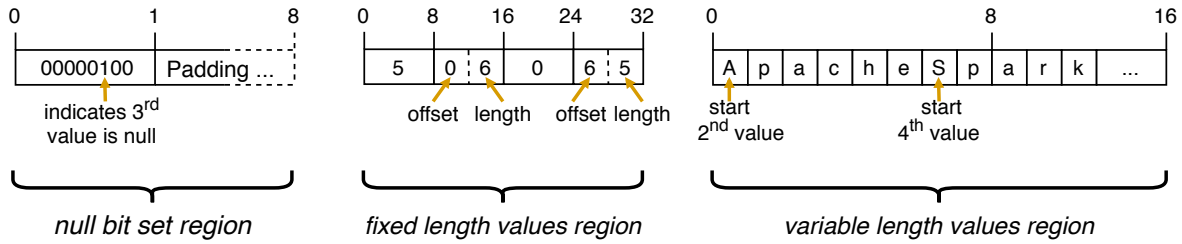


Figure 2.6: Memory layout of Spark's UnsafeRow with the values: [5, "`Apache`", *null*, "`Spark`"]

Whole Stage Code Generation

As described in the previous section, the initial version of Spark SQL Catalyst already contained a code generation step. It has converted trees of expressions (e.g. a filter predicate) into Java byte code and has executed it when processing a row. With Apache Spark 2.0, an improved version of code generation was introduced. The new function generates one coherent piece of code of all physical operators executed on the same node (*"a stage"*) [41, 57].

Similar to other database systems, previous versions of Spark used a query execution strategy based on the volcano iterator model [57]. In this model, every processing step imple-

ments a `next()` method that returns the next processed data record. Internally, this method calls the `next()` method of the predecessor and applies its transformation to it. Thereby, a chain of `next()` methods is created, which allows processing one data record completely independently of the other records [29].

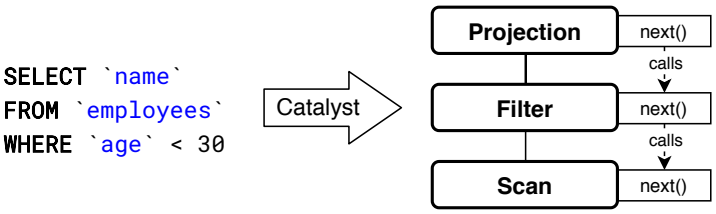


Figure 2.7: Example of volcano model based execution

An example of this is shown in Figure 2.7 on a query filtering for employees less than 30 years old. Catalyst maps the shown SQL query to a physical plan consisting of three operators. Calling the next method on the *projection* operator results in a chain of `next()` calls and finally returns the name of the first employee younger than 30 years. The whole stage code generator combines multiple operators into one function, similar to the following pseudo-code:

```
Iterator scanIterator = ... //database reader

while (scanIterator.hasNext()){
  // get next record from database
  Row row = scanIterator.next();

  // filter
  if (! (row.age < 30) )
    continue;

  // projection
  Row newRow = new Row(row.name);

  append(newRow);
}
```

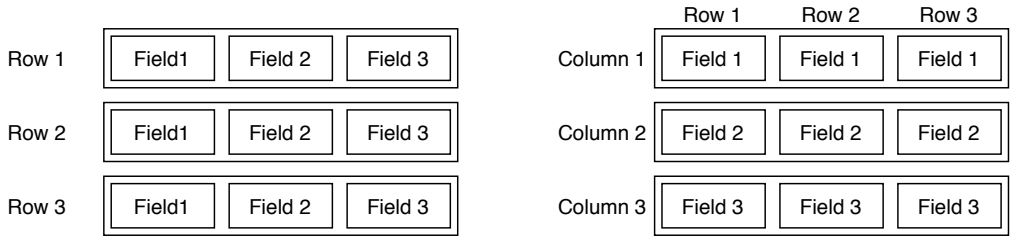
Listing 2.3: Pseudo Java Code demonstrating the whole stage code generation output

In comparison to the volcano model, the generated code has various advantages. Firstly, it contains fewer *virtual* function calls. Secondly, the volcano model requires that the intermediate records are stored in memory (function call stack). For the generated version, it is sufficient to hold most intermediate records in the CPU registers. Moreover, modern compilers, like the Java Just-in-time (JIT) compiler, optimize the code, for example, with pipelining, prefetching, and instruction reordering. The compiler can use these optimization techniques much better for coherent code, than for complex function call graphs [57].

2.1.5. Columnar processing

As part of Project Tungsten, Apache Spark 2.0 started to use a columnar memory format for individual physical operators such as the parquet file reading [41]. In the just-released version 3.0, the next step has been taken with feature Spark-27396 that extends Catalyst's public APIs with a generic design to specify columnar-based implementations for all physical operators [59]. At the time of writing, Spark itself does not contain many columnar-based implementations. Nevertheless, the new design, introduced in Spark-27396, is a fundamental basis for the implementations of this work and is therefore, described as follows:

Spark's initial memory follows a row-based layout (Figure 2.8a). With this approach, every row (or data record) is serialized as one chunk into memory. As the layout of the `UnsafeRow` (described in the previous section) shows, multiple different data types are combined and written in a contiguous block of memory. In contrast, in the column-based layout (Figure 2.8b), the same fields of multiple rows are grouped and stored as one block in memory [41].



(a) row-based layout (b) column-based layout

Figure 2.8: Comparison of row-based and column-based memory layout

Generally, in a row-based layout, a single record can be read or modified easily, however for many use cases (e.g. aggregating a single field) much unnecessary data is read. For column-based layouts, it is the opposite. Only the relevant fields can be loaded into memory, but reading a whole row leads to multiple memory accesses [31].

In 1966, Michael Flynn [27] classified different computer architectures based on their handling of data-level parallelism and task-level parallelism. The category single instruction multiple data (SIMD) describes a computer architecture, where multiple processors apply the same operations to different data streams simultaneously. However, the computer still has only a single instruction memory and control processor. This architecture implements *data-level parallelism* and is suitable for many data processing use cases such as matrix computations or image processing. Today most GPUs fall into this category and also modern CPUs come with a separate SIMD unit for such use cases [32].

Modern implementations typically load a whole vector of data into the memory and process all elements of this vector simultaneously. Naturally, this computer architecture works well together with the column-based memory layout. The processor can load a whole column without reorganizing it and can process multiple data rows at once. Accordingly, the columnar memory-layout is an important precondition for processing data in heterogeneous clusters using different computing hardware such as GPUs and FPGAs.

Analyzing Spark's source code [58] reveals the details of the columnar-processing implementation. The most important abstraction is the `ColumnarBatch`, representing a chunk of data and combining multiple `ColumnVector`. Important to notice is that a `ColumnarBatch` is a chunk of data within one partition. As shown in Figure 2.9, Spark splits the full dataset into partitions, which are processed in parallel by multiple executor threads. Typically, a single partition is then processed row-by-row within a single thread. However, the `ColumnarBatch` implementation splits the data of one partition once more and enables the processing in batches.

The `ColumnVector` is an interface abstracting one column of in-memory data. Spark includes multiple implementations of this interface. An implementation based on Apache Arrow (`ArrowColumnVector`) is also part of the project. Even though this is a long term goal, the Spark contributors have decided not to use the Arrow-based implementation as the current default. In the discussion of feature Spark-27396 in Spark's issue tracker [59] the contributors decided not to expose any Arrow-related APIs, before the release of an Arrow major version. Instead, the Arrow-based implementation is internally used to exchange data with Pandas [18].

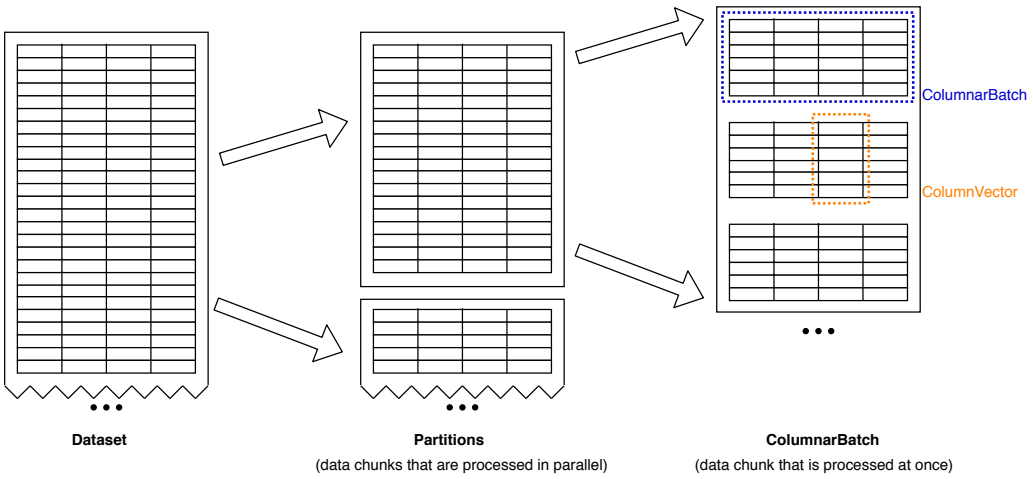


Figure 2.9: Different levels of data splitting in Spark.

The implementation of feature Spark-27396 provides an API that enables physical operators to process data in batches. The base class of the physical operators `SparkPlan` was extended with new methods. To create a operator with columnar processing support, the following two methods need to be implemented:

```
/**
* The base class for physical operators.
*/
abstract class SparkPlan extends QueryPlan[SparkPlan] {

  /**
  * Return true if this stage of the plan supports columnar execution.
  */
  def supportsColumnar: Boolean = false

  /**
  * Produces the result of the query as an `RDD[ColumnarBatch]`
  */
  protected def doExecuteColumnar(): RDD[ColumnarBatch] = {
    throw new IllegalStateException(s"column support mismatch")
  }
  //...
}
```

Listing 2.4: Excerpt from class `SparkPlan` [58]

A new transformation phase was added to the Catalyst optimizer to combine columnar and non-columnar operators (Figure 2.10). This phase introduces additional operators that either convert the memory into the columnar format (`RowToColumnarExec`) or the other way around (`ColumnarToRowExec`). Two new extension points have been introduced to add custom transformation rules. The first set of rules (*pre*) is executed before inserting the transformation and allows injecting operators with columnar-processing support. The second set of rules (*post*) allows further addition of optimizations or replacement of transformations with custom implementations.

2.2. Apache Arrow

In 2016, the Apache Foundation announced the Apache Arrow project, which defines a language-agnostic in-memory columnar format embedded in a software framework. This mem-

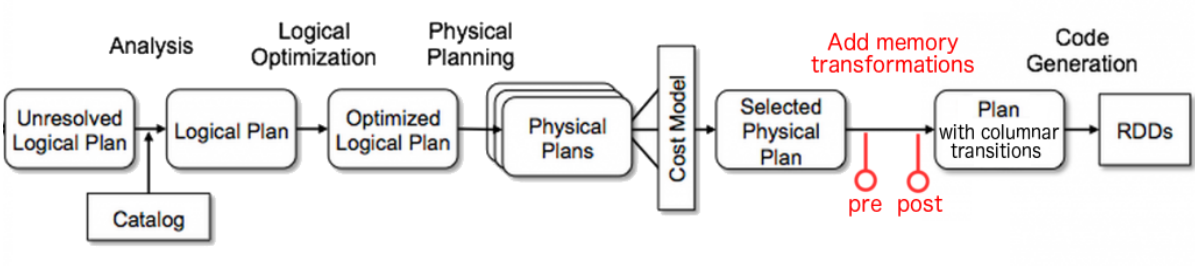


Figure 2.10: Additional memory transformation phase in Catalyst planning (based on [1])

ory format and the multi-language implementations enable different processes to share data between them. Because of the shared layout, this can be done without serialization, deserialization, or memory copies. Therefore, Apache Arrow usage is especially beneficial for multi-system workloads, in which the overhead of the cross-system communication can be reduced dramatically. As discussed in Subsection 2.1.5, the columnar layout is especially beneficial for modern hardware such as CPUs and GPUs and was, therefore, a logical design decision for Arrow [28].

Arrow provides implementations for a wide variety of programming languages to read and write the defined memory format. The Java and C++ implementations are especially relevant for the present work. Furthermore, it includes additional computing libraries and functionality supporting interprocess communication (IPC). In this work, the included Parquet file reader and Gandiva, a toolset for executing SIMD operations, are used [3].

During the work on this thesis, Arrow version 0.17.1 has been released and is used for all implementations.

2.2.1. Arrow columnar format

The *Arrow Columnar Format* defines a physical layout to store data-structures in-memory. This layout is optimized for sequential access (data adjacency), constant random access, and SIMD operations. Additionally, in shared-memory, it can be accessed from multiple processes without copying data.

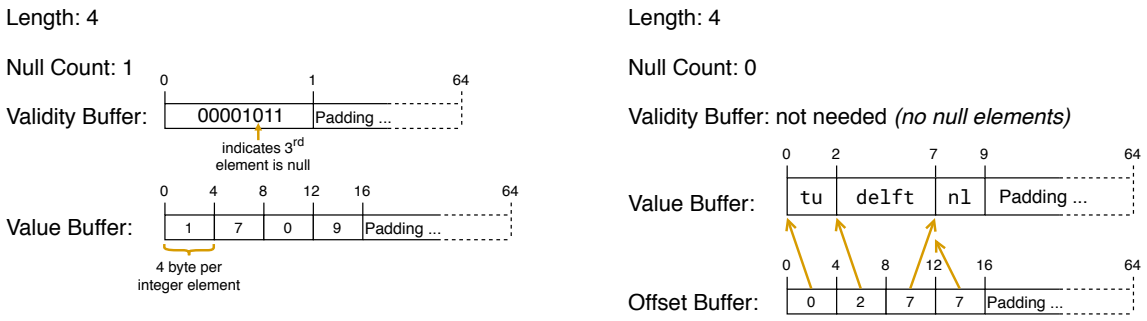
In Arrow, a sequence of values with a known length, all having the same data type, is called an *array*. The elements of an array can be nested and can contain further child types. However, for this work, the nested types are not relevant and therefore only the so-called *primitive types* which do not have any child types are considered. *Primitive types* can either have a fixed bit-width (e.g. an integer) or a variable size (e.g. a string).

An Arrow array is defined by two signed 64bit integers and different memory buffers. In Arrow, a buffer represents a contiguous memory block. Firstly, the two integers store the length of the array and the number of null elements. The first buffer is called *validity buffer* and uses one bit to indicate if an array element is null or not. Within this buffer, least-significant bit (LSB) numbering is used and a “0” indicates that an element is null. This buffer is optional when either all or zero elements of the array are null.

The *value buffer* stores the values of the elements of an array. As Figure 2.11a shows, for every element of a fixed bit-width type, a slot of the matching size is used. For data types with a variable length, every element has a different size. Therefore, a further *offset buffer* stores the start index of every element (see Figure 2.11b).

The Arrow documentation recommends aligning the buffers on 64 bytes. This means the memory address and the length of a buffer should be a multiple of 64. Naturally, buffers are than larger then required but instead, they are optimized for loading into the cache [2].

For IPC, Arrow introduces the *RecordBatch* container that holds multiple arrays. Besides



(a) Int32 Array: [1, 7, null, 9]

(b) UTF-8 String Array: [”tu”, “delft”, “”, “nl”]

Figure 2.11: Schematic representation of the columnar memory layout of Arrow arrays

the buffers' memory addresses, the processes also exchange metadata about the structure of the data. This metadata is called *Schema* and is transferred in Flatbuffer format. By using this wide-spread format, it can be serialized into many different programming languages [2].

2.2.2. Arrow Java library

For this work, the Java library is used extensively. The custom memory layout cannot be easily accessed with the regular JVM's memory management and therefore a custom implementation is provided.

The Arrow library manages the memory independently of the garbage collector (*off-heap*). This is similar to Spark's `UnsafeRow` implementation, discussed in Subsection 2.1.4. The source code [4] shows that the memory allocation is abstracted by the `BufferAllocator` interface, which provides methods to allocate new Arrow buffers. Internally, the default implementation `BaseAllocator` uses Netty's Buffer API [53] to access the off-heap memory.

In the Java library, the `VectorSchemaRoot` is a central container, holding data batches. Unlike the `RecordBatch` in other language implementations, the `VectorSchemaRoot` can be seen more as a pipeline, through which data flows. The `ValueVector` interface is the java abstraction of an Arrow array and, therefore, stores a sequence of values having the same type. As previously described, the memory allocation is not handled by the garbage collector, so the user is responsible for allocating and freeing the memory used by a `ValueVector`. The interface provides methods for this, which are forwarded to the internal `BufferAllocator` [2, 4].

2.2.3. Parquet reader

Apache Parquet is a columnar data storage format from the Hadoop ecosystem. It is designed to store complex and nested data structures and supports efficient compression and encoding schemas. The format is not limited to a specific tool and is supported by most data processing frameworks within the Hadoop ecosystem [46].

The columnar storage format of Parquet works well together with Arrow, so in 2018, the Apache Foundation moved the parquet C++ library into the Arrow project [2].

Additionally to this library, Arrow includes the *Arrow C++ Datasets* component, which provides a higher abstracted API for processing Parquet files. Besides the pure reading and writing functions, this component addresses issues such as parallel processing, handling distributed files, partitioning and filtering [4].

These additional features better suit the functionality provided by the Spark Parquet reader

implementation. Consequently, this work uses the *Arrow C++ Datasets* component, even though it is still in an alpha/beta status at the time of writing. Due to this early development stage, no extensive documentation was available, and mainly, the tests in the Arrow repository [4] were used to understand the usage of the API.

2.2.4. Gandiva

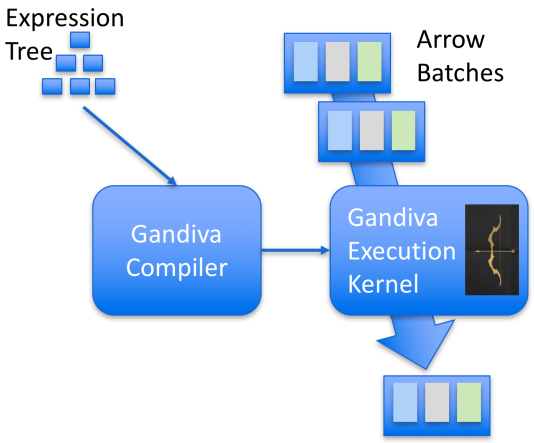


Figure 2.12: Gandiva architecture overview [51]

The company Dremio is developing a Data Lake Engine of the same name based on the Apache Arrow format. As part of their product, they have developed Gandiva, an LLVM-based execution kernel, for running analytical executions on Arrow data and donated it in 2018 to the Apache Arrow project [5].

Figure 2.12 shows the two main components of Gandiva. Firstly, the runtime expression compiler takes a tree of Gandiva expressions as input and converts them into assembly code. Using the JIT compilation capabilities of the LLVM compiler internally, the resulting assembly code is highly optimized for the underlying hardware. Secondly, Gandiva contains an execution kernel which consumes Arrow arrays and applies the generates assembly code to it [51].

As previously discussed, most modern CPUs include SIMD units and can simultaneously process multiple data points. Additionally, data based on the Arrow memory layout can be loaded cache-friendly into the CPU [2]. These properties support the LLVM compiler's auto-vectorization function [21], optimizing the assembly code for the execution on CPUs with SIMD support.

Gandiva's main function is implemented in C++. However, it provides an additional Java integration because it was developed as part of the Java-based Dremio. The Java API allows the definition of expression trees and the execution of them on Arrow data. Internally, the expression trees are mapped to a Protobuf structure and passed to the underlying native code that interacts with the LLVM compiler.

2.3. Fletcher

Following the vision to make data stored in Apache Arrow format accessible to tools in any computing environment, the Accelerated Big Data Systems group at the TU Delft Computer Engineering department has developed Fletcher, a fully-open sourced vendor-agnostic FPGA acceleration framework [30, 48].

FPGAs are programmable devices, whose internal circuit can be configured after manufacturing. By applying different parallelism techniques to the circuit, the FPGAs outrun CPUs for

certain computations [25]. Therefore, FPGAs are being used for numerous compute-intensive big data applications, such as data decompression [26], image processing [33], and genomics algorithms [34, 49].

The Fletcher framework has two main capabilities summarized in Figure 2.13. On the one hand it supports the development of hardware-accelerated function (HAF) by generating templates that include FPGA bitstreams to access the Arrow data. On the other hand, the Fletcher runtime component simplifies the integration and execution of HAF.

This work elaborates on calling the Fletcher runtime component from Spark and, therefore, the development of HAF and Fletcher's functionality is not considered here.

The Fletcher runtime includes an API that provides high-level functions to prepare and send Arrow data to the FPGA, to control the execution of the HAL on different platforms (currently supported: Amazon EC2 F1 and OpenPOWER CAPI) and to read the result data back after the execution [30, 48].

The Fletcher runtime has also been designed to be language-agnostic, by using the language-independent Arrow format. Generally, Fletcher can be used from all languages that support the Apache Arrow format. At the time of writing, implementations of the runtime library exist for Python and C++ [30].

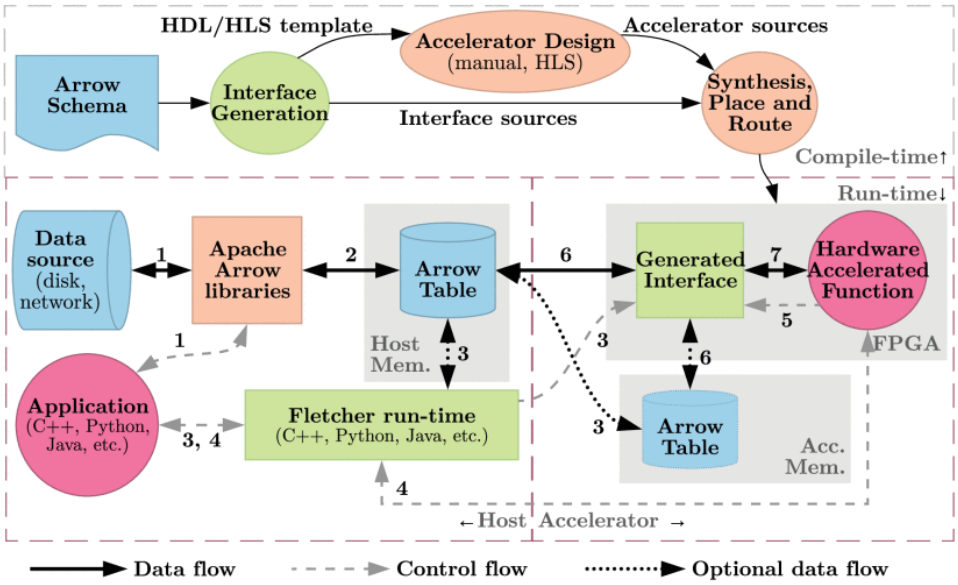


Figure 2.13: Fletcher overview [48]

2.4. Related work

Many computationally intensive workloads, e.g. machine learning algorithms containing dense matrix algorithms, are good candidates for GPU acceleration [24]. Because GPUs are wider spread than FPGAs, there also exists more work discussing the integration of GPUs with Spark. Generally, it seems that there are two main approaches to integrate GPU accelerators with Spark.

On the one hand, we have the approach of re-implementing high-level Spark operations, e.g. from Spark's machine learning library MLib, without changing the interfaces. This approach requires the implementation of full algorithms on top-of Spark's RDD API and is therefore not very flexible and requires a separate implementation for every interface function. HeteroSpark [42] is a framework providing such GPU accelerated machine-learning workloads. Internally, it uses Java RMI to forward data to the GPU, which requires costly data serial-

ization and deserialization operations. The present work avoids this serialization by using Apache Arrow. Also, IBM's approach in 2016 presented by Rajesh Bordawekar in [9, 10] follows this approach and provides mainly alternative implementation for Spark's MLib. Because their implementation is based on a prior version of Spark, they are not making use of the new columnar processing feature and copy the data of a whole partition into the device with the CUDA framework. This brings the downsides that data conversions from Spark's row-based format are required, and the size of a partition is limited to the device's memory.

On the other hand, the second approach is similar to the present work and uses Spark SQL's internal representation to generate GPU code. This approach is very flexible because it adapts to all kinds of workflows, but it is also more complicated because many aspects must be considered during the generation. This approach is, for example, followed by the framework Spark-GPU [65]. They implement their own "GPU-RDD" which buffers the data in native memory, to avoid costly transformations. Another project is the Spark RAPIDS Accelerator [45, 55], developed by NVIDIA, which integrates Spark with the RAPIDS suite [54], a bundle of open-source software libraries for running analytic pipelines on GPUs. Robert Evans, a member of NVIDIA's team behind this project, has contributed the columnar processing functionality, introduced in Subsection 2.1.5. Therefore, it is evident that this project is based on the new feature. Unlike the present work, it is not based on Apache Arrow but uses a custom ColumnVector implementation optimized to exchange data with RAPIDS.

Also, when accelerating Spark with FPGAs, the same two approaches can be seen. The Kestrel Runtime from Falcon Computing [13] is an accelerator management tool for heterogeneous computing clusters. Through its compatibility with Apache Spark, it allows accelerating big data queries with GPUs and FPGAs. This runtime is based on the open-source Blaze runtime system [35], which allows implementing FPGA-accelerated algorithms on top of Spark's RDD. Like before, these algorithms can be designed interchangeably with Spark's operations and enable a transparent acceleration. However, a disadvantage is the lack of flexibility, and users have to choose from a defined set of accelerated algorithms. Also, InAccel's FPGA orchestrator comes with a Spark integration. They provide a Machine Learning suite available on the AWS marketplace, which overloads Spark's MLib functionality and, therefore, their integration falls into the same category [39].

The only work found which falls into the second category is Bigstream [7]. This integration uses Spark's physical plan to evaluate if a bitfile templates (FPGA acceleration) is available to accelerate the Spark execution. Thereby, they follow a similar approach to the present work, but due to their closed-source implementation, no further analysis was possible.

During the final phase of the present work, another FPGA integration [11] was presented on the "Spark+AI Summit 2020" conference. The demonstrated integration was based on Intel's "Spark Native SQL Engine" [36] project. The goal of this open-source project is to enable Spark SQL for vectorized SIMD optimizations. In favor of this, the team has implemented an Apache Arrow-based version of Spark's columnar processing and has integrated it with a Native Parquet Reader, Gandiva, and Columnar Shuffle operations. Although the details of the FPGA integration are not publicly available, this shows very well that this project follows a similar idea as the present work. It has already integrated matching accelerators and has solved similar challenges. Due to the very recent publishing on GitHub, the present work could not use synergies or compare results with the "Spark Native SQL Engine" project. Due to the higher development effort, this project is in a more robust and mature state, than the PoC implementation of the present work. Therefore, Section 6.2 suggests further evaluation of this project to figure out if synergies can be used.

3

Architecture and general concepts

3.1. General structure

To integrate the previously discussed tools, the architecture of the implementation has to take into account the different programming languages of the third-party libraries used. Therefore, the implementation of this work[1] is organized into three different modules (Figure 3.1). Firstly, the *arrow-processor-native* module is written in C++ and is responsible for calling related C++ libraries. Furthermore, this module provides an interface, which is called from the *arrow-processor* module by using the Java Native Interface (JNI). The *arrow-processor* is a facade around the low-level native interface and provides a higher abstracted Scala interface based on the abstractions from the Java Arrow library. The third module is the *spark-extension* module, which extends Spark SQL and connects it with the custom accelerators implemented in the *arrow-processor* module. Both modules, *spark-extension*, and *arrow-processor* are implemented in Scala for easier integration with Spark. However, Scala code is executed on the JVM and is fully interoperable with Java libraries, such as the Arrow library.

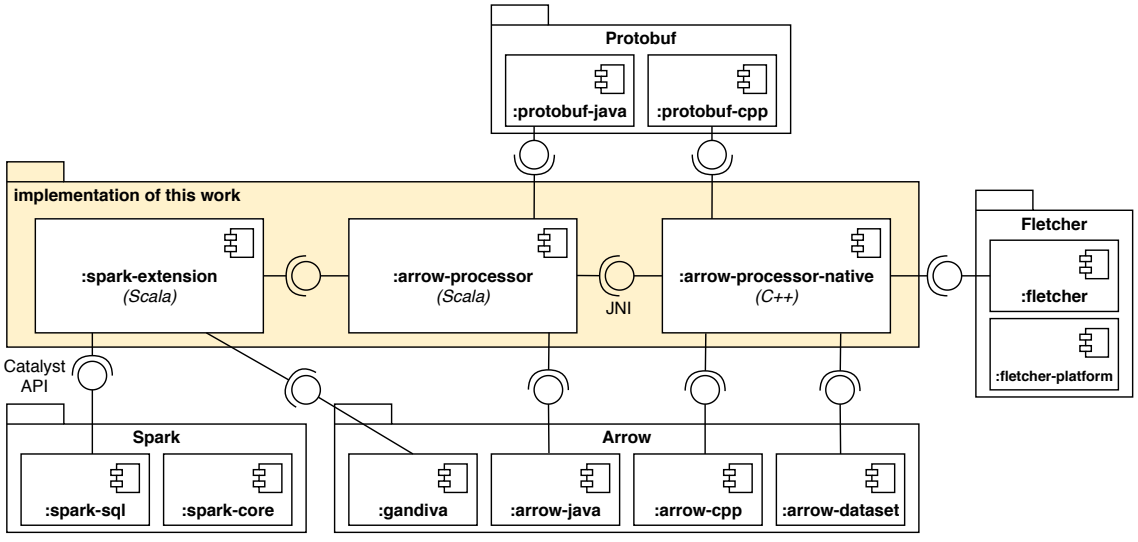


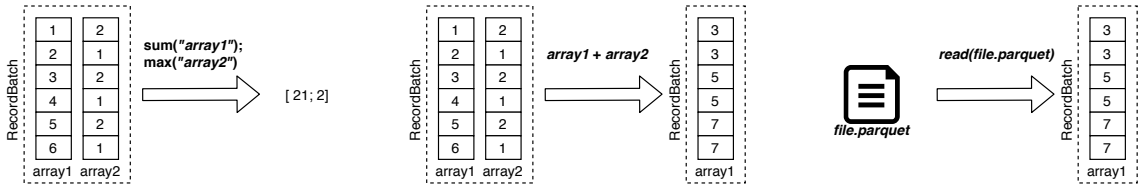
Figure 3.1: Architecture overview: Structure of the implementation and used third-party libraries (dependencies between the third-party libraries are not shown)

The general idea of the implementation is that Spark's main function stays unchanged. Spark remains responsible for coordinating the complete execution and for passing the data

[1]The implementation of this work can be found on https://github.com/fnonnenmacher/spark-arrow-accelerated

between the different execution steps. However, with the extensions of this work, individual execution steps are offloaded to Arrow-based hardware-accelerators of third-party libraries. These execution steps always relate to the processing of data batches. In Spark terminology, such a data batch is called `ColumnarBatch`. In Arrow terminology, it is called `RecordBatch`. Both terms refer conceptually to a container with columnar data vectors. Section 3.3 describes how both concepts come together at implementation level.

When offloading work to the accelerators, the Scala code iterates over the batches and processes every batch individually with the help of third-party libraries. As Figure 3.2 shows, the computations, considered in this work, can be grouped into three categories. Either they *aggregate* one batch and return a set of values (Figure 3.2a), or they *project* a data batch to a new batch (Figure 3.2b), or finally, they import one data batch from an external source (Figure 3.2c).



(a) Aggregating a `RecordBatch` (b) Projecting a `RecordBatch` to a new one (c) Reading a data from an external source

Figure 3.2: Different categories of computation that are executed by third-party libraries

Chapter 4 presents the concrete integration of different accelerators, whereas this chapter focuses on multiple general, high-level concepts which build an essential basis for these integrations.

3.2. Executing custom code within Spark SQL

As described in Subsection 2.1.3, Spark SQL's Catalyst converts the query into an internal query representation, the so-called *logical plan*. Catalyst optimizes this plan and finally transforms it into a *physical plan*, a combination of *physical operators*. Internally, Spark SQL contains many different physical operators, each of them specifying the execution of one computation step. Combined in the *physical plan*, these operators define the complete execution of the query. The goal of this work is to modify Spark's execution and, therefore, this work replaces Spark's operators with custom implementations.

A physical operator is defined by a class extending `SparkPlan`. This is Spark's internal base class for all *physical operators* and, as described in Subsection 2.1.5, it defines methods for columnar processing. Therefore, all custom *physical operators* implemented in this work, specify the computation by overriding these methods.

Firstly, the method, `supportsColumnar`, has to return `true` to inform Catalyst that this *operator* provides a columnar based computation. Catalyst uses this information to insert additional operators that convert the data when needed.

Secondly, the method, `doExecuteColumnar`, must be implemented to describe the actual computation. Following the volcano model (see Subsection 2.1.4), this method has to call first the `doExecuteColumnar` method of the child node(s). From this call, it receives a `ColumnarBatch` iterator. In Scala, an iterator is a way to access elements of a collection one-by-one [23]. Unlike in a classical list, the elements are evaluated lazily, and the collection never needs to be loaded entirely into memory. The custom implementation can now apply the accelerated computation to the batches received from the child node so that the method

returns an iterator of the resulting `ColumnarBatch`es itself.

By overriding additional methods, it would be possible to define instructions for the Whole Stage Code Generation (see Subsection 2.1.4). However, as previously discussed, the code generation is the most beneficial, when multiple Java operations are called frequently. However, in this work, the computation is forwarded to third-party libraries, which cannot be optimized by the JIT compiler. Furthermore, by iterating over `ColumnarBatch`es instead of rows, the computation is executed less frequently, and the compiler optimization would not have a huge impact. Therefore, this work focuses on other challenges and does not integrate with the Whole Stage Code Generation. Nevertheless, Spark's code generation for the other physical operators is not influenced by this. The code is generated anyway and calls the custom implementations.

The next step is to tell Spark about the custom operator implementations. Therefore, this work implements a `SparkSessionExtension`, that allows adding rules to all phases of the Catalyst optimizer on startup [38]. The first experiments have shown that the least invasive way of modifying the *physical plan* is to use the *pre-columnar transition* phase introduced with the new *columnar processing* feature. At this point, the physical plan is fully created, and default operators can be replaced with custom implementations. For doing so, this work follows the Catalyst tree transformation approach. It iterates through the physical plan and replaces a subtree that matches a specific pattern with a custom operator.

The implementation of this work follows Spark's naming convention. All physical operator classes end with `Exec`, and the rules injected into the Spark session are defined in a class ending with `Extension`.

Figure 3.3 summarizes the procedure of using a custom *physical operator*. Firstly, the `CustomExtension` injects a rule into the *pre columnar transition* phase. Later on, this rule replaces the `FilterExec` with a custom implementation `CustomFilterExec`, extended from `SparkPlan`.

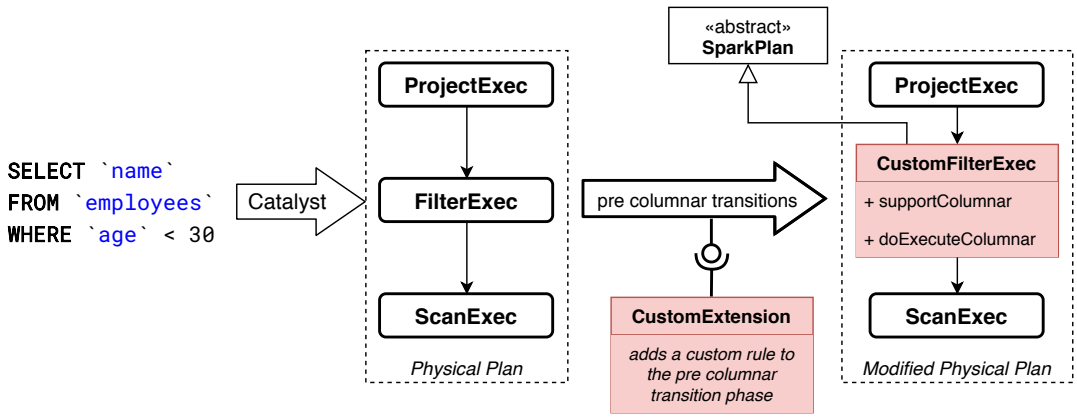


Figure 3.3: Example of replacing a physical operator with a custom implementation

3.3. Arrow-based columnar processing

The idea of this work is to store all data in the Apache Arrow format through the whole computation. In this way, the computation can be accelerated by tools from the Arrow ecosystem, such as Gandiva and Fletcher, without expensive data conversions. As discussed in Subsection 2.1.5, Spark's columnar processing API is currently not based on the Arrow format. Consequently, this work modifies Spark's columnar-processing functionality to use the Arrow format internally.

As shown in Figure 3.4, Spark SQL inserts additional *physical operators* (orange) which transform the memory between the row-based and columnar-based format, when the physical plan contains both, columnar-based and non-columnar-based operators.

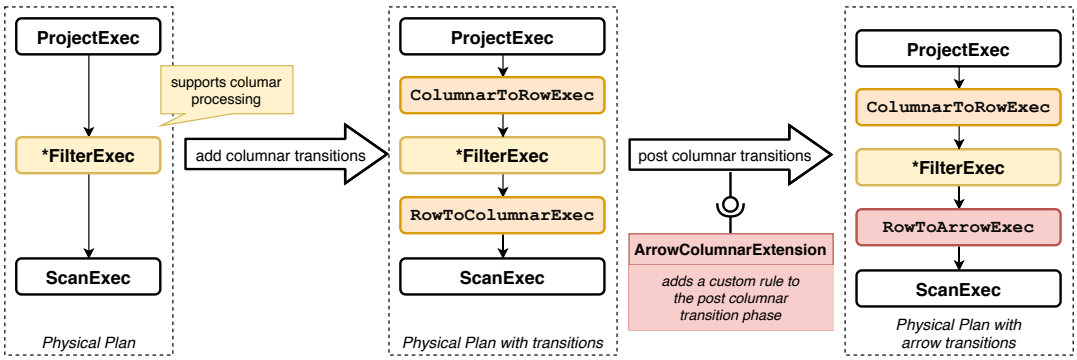


Figure 3.4: Modifying Spark's columnar processing to be based on Apache Arrow

The `ColumnarToRowExec` operator takes the `ColumnarBatch`es as input and maps them to rows. This mapping works independently from the implementation of the underlying `ColumnVector`s and is compatible with data in the Arrow format.

The `RowToColumnarExec` operator works the other way around and transforms rows into `ColumnarBatch`es. Multiple rows are combined and every column of them is stored in an `OnHeapColumnVector`, which is Spark's default `ColumnVector` implementation. Naturally, this data cannot be easily converted into an Arrow array and would require another transformation before processing it with Arrow-based accelerators. Hence, this work replaces this transform operator with a custom implementation (`RowToArrowExec`) that converts the data directly into the Arrow format. These arrays are then wrapped with Spark's internal `ArrowColumnVector` class and can be stored in a `ColumnarBatch`.

Again to modify the physical plan, a custom extension `ArrowColumnarExtension` is used to inject rules into Catalyst. As shown in Figure 3.4, this rule replaces the default `RowToColumnarExec` with the Arrow-based implementation during the *post-columnar transition* phase.

Currently, Spark's public API hides all details related to the Arrow implementation. Therefore, no public methods are available to access the underlying Arrow arrays when processing a `ColumnarBatch`. Consequently, this implementation uses Java's reflection API to access hidden private fields. Naturally, there is no guarantee that this approach will work in future versions of Spark. However, it is sufficient for evaluating the technical feasibility, and hopefully, this *"hack"* will become obsolete as soon as Spark moves its columnar processing entirely to the Arrow format, which is the long-term goal.

3.4. Exchanging Arrow arrays between Java and C++

3.4.1. Overview

Many tools from the Arrow ecosystem are not available as Java libraries. Nevertheless, these tools might provide additional performance improvements and it is worth considering integrating them into Spark. This work illustrates this by integrating Spark with the Arrow Parquet reader and with Fletcher. Both tools only provide APIs in C++ and Python and cannot be called easily from the JVM. This work uses the JNI interface to call C++ libraries from Spark. Thereby, the implementation benefits from the language-agnostic Arrow memory format and avoids expensive data conversion between the two languages.

As described in Section 3.1, the architecture of this implementation also reflects these two

languages. On the one hand, there is the *arrow-processor-native* component, implemented in C++, which interacts with the third-party libraries. On the other hand, there is the *arrow-processor* component, implemented in Scala, which provides a high-level API based on the Java Arrow abstractions. Both components are connected through a JNI based interface and are internally mapping the Java Arrow abstractions to the C++ Arrow abstractions.

Apache Arrow's documentation discusses multiple approaches to exchange data between processes implemented in different languages. This can be done, for example, by using shared-memory, such as the Arrow Plasma Store [44], or by exchanging IPC messages [2]. However, by using JNI this becomes easier. The JNI is Java's standard way to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. Furthermore, the JNI allows the native languages to access the same memory region [22]. As a result, there is no need to copy the data between both components of the implementation. Instead, they can access the same Arrow buffers stored in-memory. When accessing the same objects from Java and native code, another typical challenge is to make sure that the garbage collector is not deleting data still required by the native code. Also, this challenge does not apply to the implementation of this work. The Arrow buffers are already stored off-heap and are not managed by the garbage collector.

As described in Subsection 2.2.4, Gandiva follows a similar structure. It provides a Java library, which delegates all calls to a library implemented in C++. The source code of Gandiva [4], released under the Apache License 2.0, was studied during this work and builds the conceptual basis for this work.

3.4.2. From Java to C++

In this work, Spark coordinates the query execution. It knows about the structure of the data and just calls individual third-party libraries, to compute an Arrow `RecordBatch`. At the time of calling, the arrays are already allocated and initialized. The used third-party libraries follow best practices and consider the arrays as immutable. Hence, they are not modifying any Arrow buffers and just need read access. Typically, the result data of a computation is either a single value which can be directly returned on the JNI method call, or a new `RecordBatch`, which can be transferred back with methods discussed in the following subsections.

As discussed in Subsection 2.2.1, an Arrow `RecordBatch` is a container, described by a schema, that holds multiple Arrow arrays.

For transferring the schema, functionality from the Gandiva library is used. On the Java side, the library provides methods that use Protobuf to serialize the Java schema into a byte array. After passing this array via JNI, methods from the C++ library are used to deserialize it into a C++ schema object.

As previously described, an Arrow array consists of two signed 64bit integers and two or three buffers. The two signed 64-bit integers are storing the length of the array and the number of null elements. They can be passed as a Java long parameter to the JNI method call. For each buffer, the memory address and the size is transferred. This information is enough to access the memory in C++ and to create the related C++ Arrow buffer abstractions. Combining these buffers in the C++ Arrow array abstraction allows accessing the Arrow data with the regular methods of the Arrow API.

3.4.3. From C++ to Java with preallocated buffers

In some cases, the Java side already knows the number of result elements, before calling the native computation methods, e.g. when two arrays are added together to form a new array. The buffers can then be allocated from the Java code and can be sent to C++, following the same approach as transferring data from Java to C++. In this scenario, the C++ code

simply writes the resulting data into the already allocated buffers, which are known and can be accessed from Java.

However, when the resulting array stores values of a variable-width data type, such as strings, the size of the *value buffer* cannot be known before. Gandiva has solved this problem by adding an own buffer implementation `JavaResizableBuffer` which resizes by doing a callback to Java. Because a buffer has to be written in a contiguous memory block, resizing often requires allocation of a new larger buffer and copying the old buffer into the new one. Arrow's Java implementation completely handles this reallocation and returns the memory address of the new buffer to the C++ code.

This work has evaluated this approach and demonstrates the technical feasibility with test-case implementations. For the implementation, functionality from the Gandiva library was used. However, this approach was not suitable for the chosen accelerators. Therefore, in all Spark integrations, the approach described in the next subsection was used.

3.4.4. From C++ to Java by forwarding the allocation to Java.

Some tools, such as the Arrow Parquet reader, are incompatible with preallocated buffers. Instead, these tools use Arrows default approach and call the `MemoryPool` class [4] to allocate memory for buffers, when needed. Unfortunately, the memory allocated in C++ during a JNI call cannot be easily accessed from Java. Furthermore, managing the allocated buffer's lifespan is not easy, because they must remain accessible after the JNI call terminates, so that the data is accessible for the next execution steps. As previously discussed, Spark is responsible for coordinating the whole execution and knows best when the buffers can be freed again. Therefore, this implementation delegates all allocation calls to the Java Arrow library, so that the Java code stays responsible for managing the life-cycle of the Arrow data.

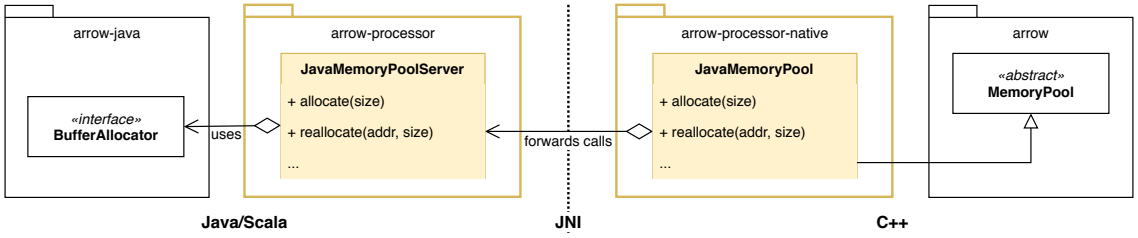


Figure 3.5: Implementation of `MemoryPool` that forwards the allocation to Java.

As Figure 3.5 shows, this was achieved by implementing `JavaMemoryPool`, a custom specialization of the Arrow `MemoryPool` that forwards all allocation requests to a similar Java class `JavaMemoryPoolServer`, by using JNI methods. This Java class then uses the `BufferAllocator` from the Java Arrow library to allocate the needed buffers to off-heap memory. Additionally, the `JavaMemoryPoolServer` stores internally, the allocated Java buffer abstractions, which are then used to create the Arrow array abstractions after the JNI computation terminates.

4

Integration of different accelerators

4.1. Overview

The previous chapter 3 has discussed essential concepts necessary to integrate different tools from the Arrow ecosystem with Apache Spark. This chapter applies these concepts now in practice and gives concrete examples of possible integrations. The integrated tools were chosen to reach the goal of providing a complete data analytics pipeline based on the Arrow format. The present work focuses on integrating the main functionality instead of building a solution that is functionally equivalent to Spark's internal implementation.

The first step of every data analytics pipeline is importing data. For avoiding unnecessary data conversion, it is essential to load the data into the Arrow format directly. The present work has chosen the Parquet file format as input source because it stores the data already in columnar format and is an ideal candidate to read it in columns into the memory. The present work discusses the integration with the *Arrow C++ Dataset* library in Section 4.2. This library is part of the Arrow project and allows demonstrating the data exchange between Java and C++. Alternatively, the Java library Parquet MR would have been another option, which also provides a function to import in the Arrow format [47].

As discussed in Subsection 2.1.4, Spark optimized the CPU and memory usage with Project Tungsten a lot. Nevertheless, due to the limitations of the JVM and its limited columnar processing operators, the SIMD capabilities of modern CPUs cannot be used. Therefore, Section 4.3 describes the integration with Gandiva to execute SIMD-accelerated *filter* and *projection* operations on Arrow data.

At the end of every computation, Spark copies the result from the internal format into Java/Scala objects. Because this copying is a costly operation, the Spark documentation recommends reducing the amount of data and only loading aggregated results. Spark's aggregation implementation does not work well together with the Arrow format. For measuring whole data processing pipelines, the present work provides, in Section 4.4, a simple aggregation that determines the maximum of integer arrays.

Finally, the present work contributes to the vision of the ABS group. Section 4.5 discusses the acceleration with FPGAs by using Fletcher. The implementation of the present work demonstrates the integration based on a simple use case thus creating a valuable foundation for further, deeper integration.

4.2. Importing Parquet files into Arrow format

Internally, Spark implements the importing of files by the physical operator `FileSourceScanExec`. As described in Section 3.2, the implementation of the present work replaces this operator with a custom physical operator, the `ArrowParquetSourceScanExec`, which uses the Arrow C++ Dataset library for reading Parquet Files.

The definition of the computation requires the `doExecuteColumnar` method, which is summarized in Figure 4.1. The implementation uses the metadata (e.g. the input schema), which was determined during the Catalyst optimizations. This metadata is used for initializing the `ParquetReader`, which is a Java abstraction and calls the Arrow C++ Dataset library through JNI. Following the modularization concept introduced in Section 3.1, this class provides an interface based on the Arrow Java abstractions. Precisely, the class implements an operator of `VectorSchemaRoot` objects and allows thereby reading the Parquet file batch-by-batch. Internally, every request to read a new batch is forwarded to the C++ library. The buffers of the returned Arrow `RecordBatch` are passed back through the layers of the implementation. They are mapped to the `VectorSchemaRoot` Java abstraction without transforming or copying any data.

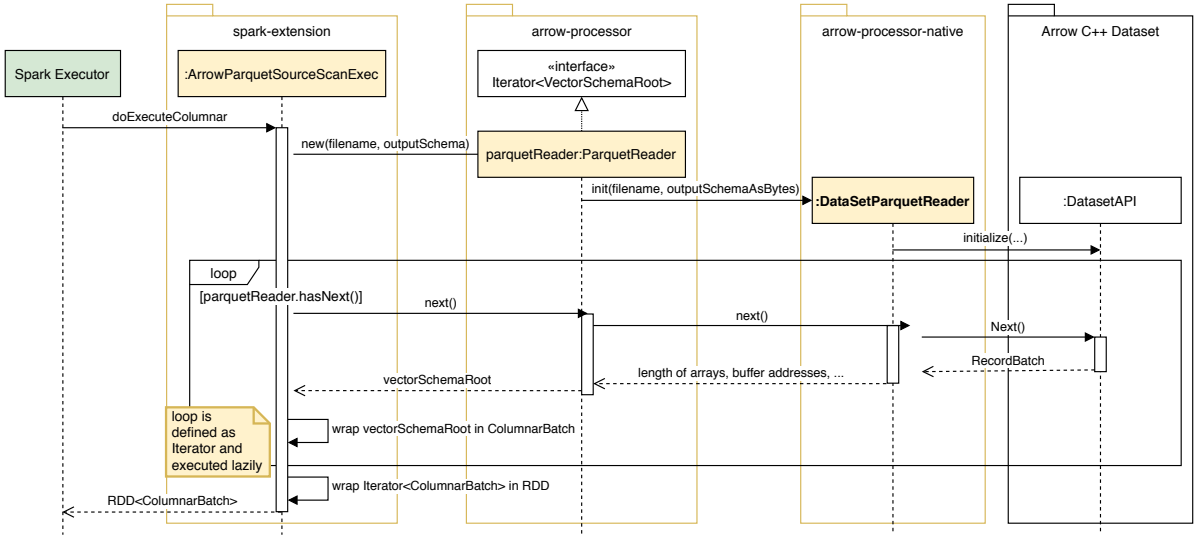


Figure 4.1: Simplified implementation of the custom Parquet reader operator

Following Spark's architecture concept, the iterator is not evaluated, directly. Instead, the implementation applies a transformation to it, which maps the Arrow abstraction to Spark's `ColumnarBatch` container (see Section 3.3). This lazy evaluated iterator approach avoids loading the whole data into memory and allows instead to process one batch through the whole pipeline before reading and processing the next one (volcano model [29]). Finally, the resulting `ColumnarBatch` iterator is wrapped in Spark's `RDD` implementation, which provides functionality to iterate through the data in distributed setups.

Both Spark's Parquet reader and Arrows's C++ Dataset library provide extensive features to read Parquet files. These features include reading multiple files, reading from network shares and optimizations such as predicate pushdown. Generally, both implementations are a good fit and most of Spark's features have an equivalent fit in the Arrow library. The present work does not claim to integrate all of these features and focuses on the basic functionality, which is reading a single file from a local file system. It allows limiting the imported columns but does not allow filtering on the data values (predicate pushdown). Furthermore, it supports uncompressed and snappy-compressed Parquet files.

4.3. Gandiva integration

As introduced in Subsection 2.2.4, Gandiva allows running analytical executions on Arrow data optimized for CPUs with SIMD capabilities. Gandiva takes expressions as input and compiles them with the LLVM compiler at runtime to optimized assembler code. These expressions can be used for two different operations. Either for *filtering* or *projecting* the rows of a `RecordBatch`.

Figure 4.2 shows an example of a query containing filter and projection operations. As shown, both operations are represented by two separate corresponding *physical operators*. The operator `FilterExec` is responsible for filtering rows. Thereby, this physical operator is parameterized by a tree of expressions representing a binary predicate. During the execution, this *operator* filters out all rows where the predicate evaluates to `false`. Whereas, the *projection* operator `ProjectExec` is parameterized with multiple expression trees. When executing it, the results of these expression trees create a new row [58].

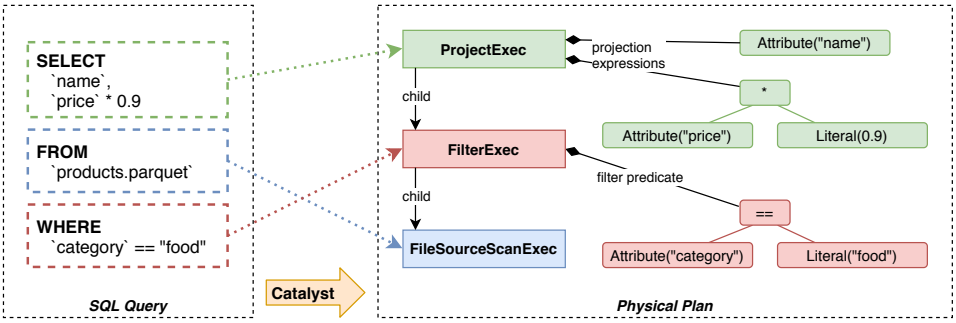


Figure 4.2: Spark SQL Catalyst's representation of filter and projection operations

This shows that the *operators* in Spark are structurally similar to the Gandiva operations, allowing to replace Spark's *physical operators* with custom implementations that use Gandiva for filtering and projection.

For mapping Spark's *filter* and *projection* expression to Gandiva, the implementation of the present work transforms the tree of Spark expressions into a tree of Gandiva expressions. It uses an approach similar to Catalyst's internal transformations. It iterates top-down through Spark's expression tree and uses Scala's pattern matching functionality to replace every node with an equivalent Gandiva representation. The work currently supports expressions containing arithmetic operators, relational operators and data type casts. Nevertheless, the pattern-matching approach is very flexible and can be extended with further functions easily.

Unlike the other accelerators, Gandiva provides a Java library, which internally follows a similar structure as this project and forwards Arrow data also to an underlying native library. The Scala code implementing the custom *physical operator* can directly call Gandiva's Java interface and does not need to handle the Arrow transformation.

As shown in Figure 4.3, the custom implementation `GandivaProjectExec` for projections firstly transforms the projection expressions into Gandiva's format and uses them to instantiate Gandiva's `Projector`. During the instantiation, Gandiva compiles the tree of expressions to SIMD optimized assembly code. After Gandiva is set up, the operator calls the `doExecuteColumnar` method of the child operator, which returns a `ColumnarBatch` iterator. As described previously the iterator is lazy evaluated and not entirely materialized. Also the custom `GandivaProjectExec` implementation follows this approach and defines the processing of every `ColumnarBatch` without executing it instantly. Instead, the processing is embedded in an iterator and is lazily evaluated when requested.

Within the individual processing of a `ColumnarBatch`, firstly, the underlying Arrow data abstractions are extracted. These abstractions are then passed to Gandiva that executes the

projection on it and returns a new `RecordBatch` containing the projected data. Finally, this Arrow structure is mapped to Spark's `ColumnarBatch`.

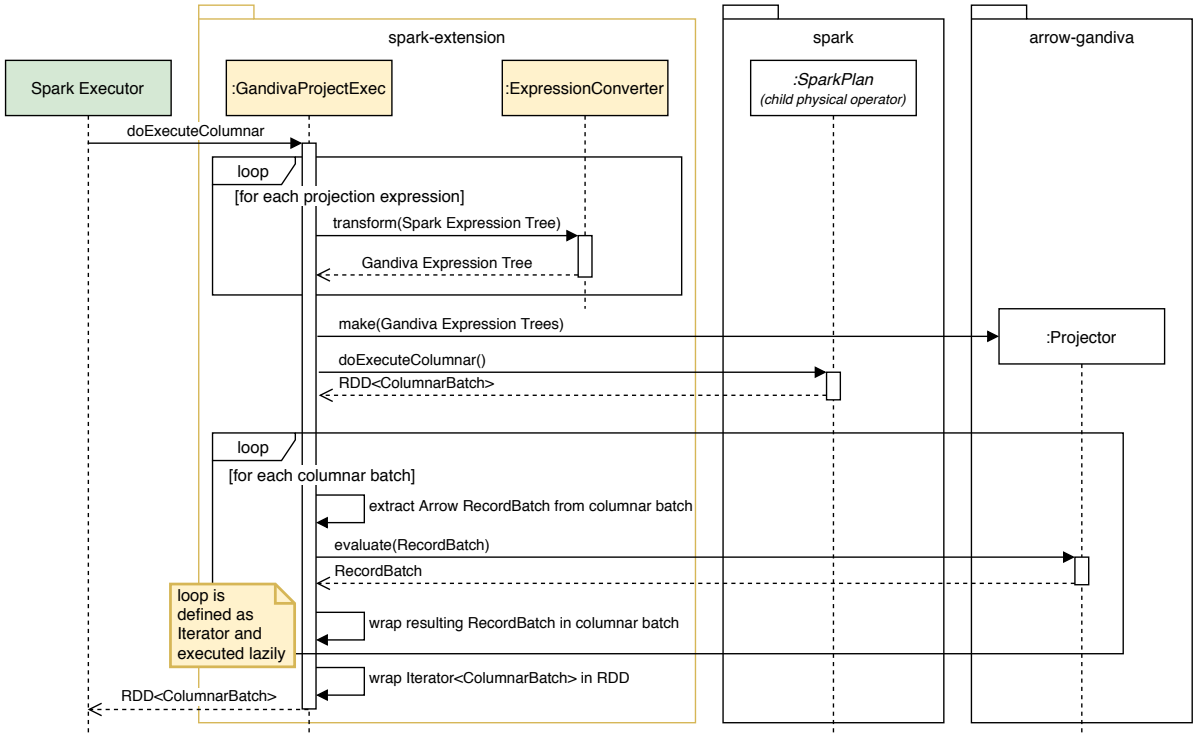


Figure 4.3: Internal computation of custom `GandivaProjectExec` operator

Generally, also the filtering implementation `GandivaFilterExec` follows the same structure. However, instead of returning a `RecordBatch` containing only the rows which evaluate to `true`, Gandiva returns a `SelectionVector`. This vector contains all indexes of the rows not filtered out. When performing a projection directly after filtering, this `SelectionVector` can be passed as an additional parameter and to Gandiva's projection. The projection is then only created for the rows in the `SelectionVector`. Unfortunately, Spark's `ColumnarBatch` representation currently does not support anything similar to a `SelectionVector`. Therefore, this Gandiva-based filter implementation is not fully compatible with other columnar-based operators. Nevertheless, the implementation of the present work has added an alternative for Spark's `ColumnarToRowExec` conversion (see Section 3.3), which only transforms rows listed in the `SelectionVector`, and allows, thereby, combining the filter operation with regular Spark operators.

4.4. Simple max aggregation

Using the Parquet reader with the Gandiva implementation allows running of first data processing use cases entirely based on the Apache Arrow format. However, to continue working with the result data, it is still necessary to store them in a file or to load them with Spark SQL's `collect()` method into the driver's memory. Both operations are quite costly because they are not optimized for Apache Arrow. These operations have a high impact on the execution time and make it hard to evaluate the accelerators' improvements. Therefore, the present work adds an elementary operator that calculates the maximums of an integer Arrow arrays. Using this operator forces Spark to evaluate the full data but reduces the results loaded back to one value per column.

Generally, different from the previously discussed operations, an aggregation operation

does not only operate on exactly one `ColumnarBatch`. Instead, it combines multiple data rows and creates new rows. Thereby, an aggregation is a *wide* Spark *transformation* and requires exchanging data (*shuffling*) between Spark's executors. All the challenges related to shuffling make Spark's aggregation implementation rather complex. Through the different focus of the present work, no custom implementation with a comparable functionality was created. Instead, the present work focuses on a particular, but also essential aggregation used to measure the performance improvements.

As Figure 4.4 shows, Spark SQL creates multiple *physical operators* to define the max aggregation. Firstly, the maximum is calculated for every partition. Afterward, all intermediate results are collected to one executor, and their maximum is calculated. As described before, not many of Spark's operator implementations support columnar processing. Also, the aggregation is no exception and only supports row-based processing. Therefore, Catalyst inserts automatically a `ColumnarToRowExec` operator which converts the batches into rows. The least invasive way of implementing the max aggregation is to replace this transformation operator in the *post columnar transformation* phase with a custom implementation, which maps every `ColumnarBatch` to one row containing the maximum of this batch. As a result, the structure of the data between the operators stays unchanged and the aggregation related operators have not to be changed.

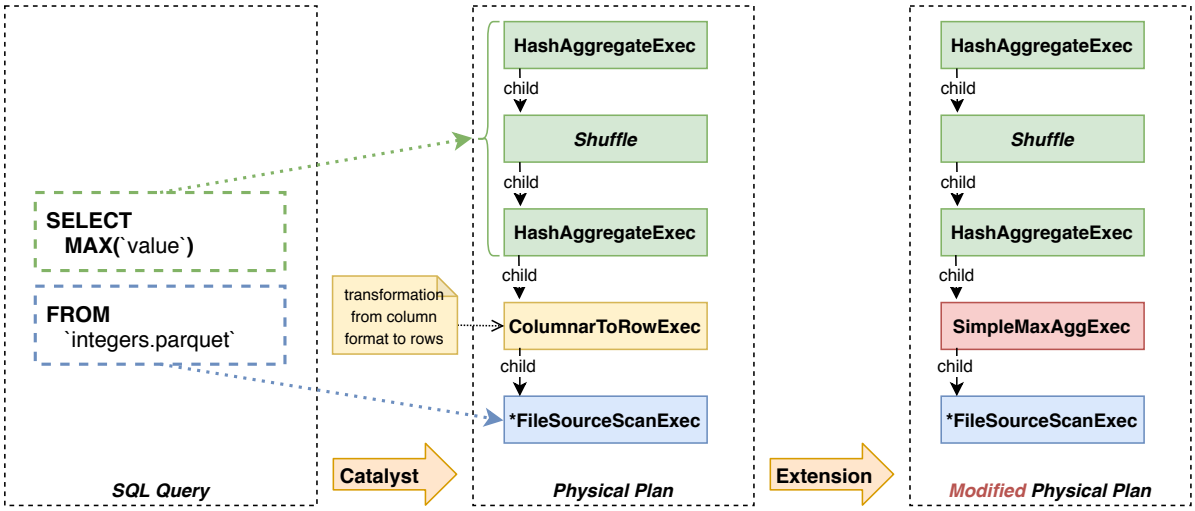


Figure 4.4: Spark SQL's physical plan of a max aggregation and its modification

Internally, the `SimpleMaxAggExec` is simplistic, too. It calls a C++ method that iterates through the Arrow arrays and returns their maximums. Finally, it creates a new row with these maximums for every `ColumnarBatch`. Afterward, Spark's aggregation operators compute the maximums of all `ColumnarBatch` maximums.

4.5. Fletcher

The last accelerator integrated during the present work is a significant step towards the vision of *heterogeneous computing*. By using Fletcher, a vendor-agnostic FPGA accelerator framework, this integration offloads parts of the computation to FPGAs.

An FPGA is a programmable device that allows configuring the internal circuit. This circuit can be optimized using different parallelism techniques and can perform specific tasks much faster than a CPU designed for generic tasks. The execution logic of an FPGA is specified using a hardware description language (HDL) which describes the digital circuit. Generally, implementing such a circuit requires in-depth knowledge related to digital electronic designs. In

layman's terms, this makes the development of a HAF for an FPGA much more expensive than the development of equivalent software programs. Additionally, there are different compilers available that generate the HDL design from programming languages or SQL. However, these automatically generated designs are often much slower than manually written designs [25].

Nevertheless, even converting the HDL design into the physical circuit layout (*Logic Synthesis*) is a time-consuming process and can take up to several hours. Therefore, as opposed to Gandiva, for FPGA accelerators, it is not beneficial to automatically generate the hardware designs at runtime. Instead, the vision of the ABS group is to store the hardware-accelerated functions in a repository and use them to accelerate parts of data analytics queries. The analysis of Spark SQL's capabilities showed that Spark SQL's internal representation is generally detailed enough for querying such a repository. However, checking if Spark's internal logical plan can be partly substituted with a HAFs requires nontrivial tree comparison and was not further elaborated in the present work.

Instead, the present work demonstrates the technical feasibility of such an integration based on a specific use case. This use case uses the publicly available Chicago Taxi data set [40], which includes all taxi trips operated in Chicago from 2013 to the present with additional metadata such as the trip length. The use case reads the data from a Parquet file and calculates the total duration of all trips operated by the company "Blue Ribbon Taxi Association Inc."

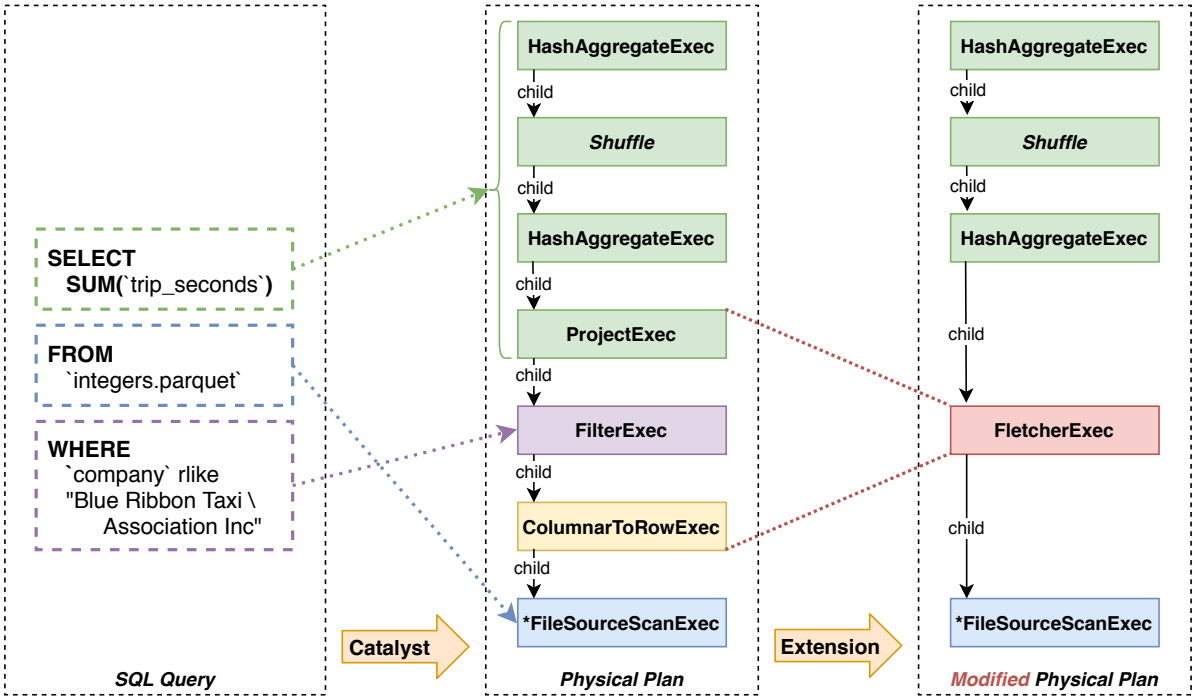


Figure 4.5: Spark SQL's physical plan of the Fletcher use case and its modification

Thus, the execution is accelerated by a Fletcher HAF, which takes a `RecordBatch` as input and computes the "Blue Ribbon Taxi Association Inc's" total trip duration of this batch. This HAF is based on a regex and allows generally different patterns as well. Therefore, also the SQL query shown in Figure 4.5 is based on a regex match.

Internally, this query is mapped to a physical plan containing the Parquet file reading, the memory transformation, the regex-based filtering, the projection to the relevant 'company' field and, finally, the sum-based aggregation including a shuffle phase. The present work replaces now, a part of this query with a custom implementation `FletcherExec`, which in-

ternally calls Fletcher's HAF. Similar to the simple max aggregation, this replacement also includes the memory transformation. Thereby, the custom implementation is again reducing every `ColumnarBatch` to exactly one row containing the intermediate sum. Afterward, these rows are regularly handled by Spark and added together with Spark's default aggregation mechanism.

Figure 4.6 shows the implementation of `FletcherExec`. As already mentioned, this *operator* also converts `ColumnarBatch`es into rows. Therefore, the `doExecute` method is overwritten. When entering the method, firstly, the Fletcher run-time is set up. Afterward, the child operator is called. For this use case, this is the `ArrowParquetSourceScanExec`. This operator returns an iterator of `ColumnarBatches`. As before, to allow batch-by-batch evaluation, this iterator is not directly processed but mapped to a new iterator. This new iterator applies a mapping function to every `ColumnarBatch`. This mapping function passes the underlying Arrow buffers, through the different layers, to the Fletcher run-time, which loads them and executes the previously configured HAF. The intermediate sum returned from Fletcher is then finally written into a row, so that the final method returns an RDD containing an iterator of rows.

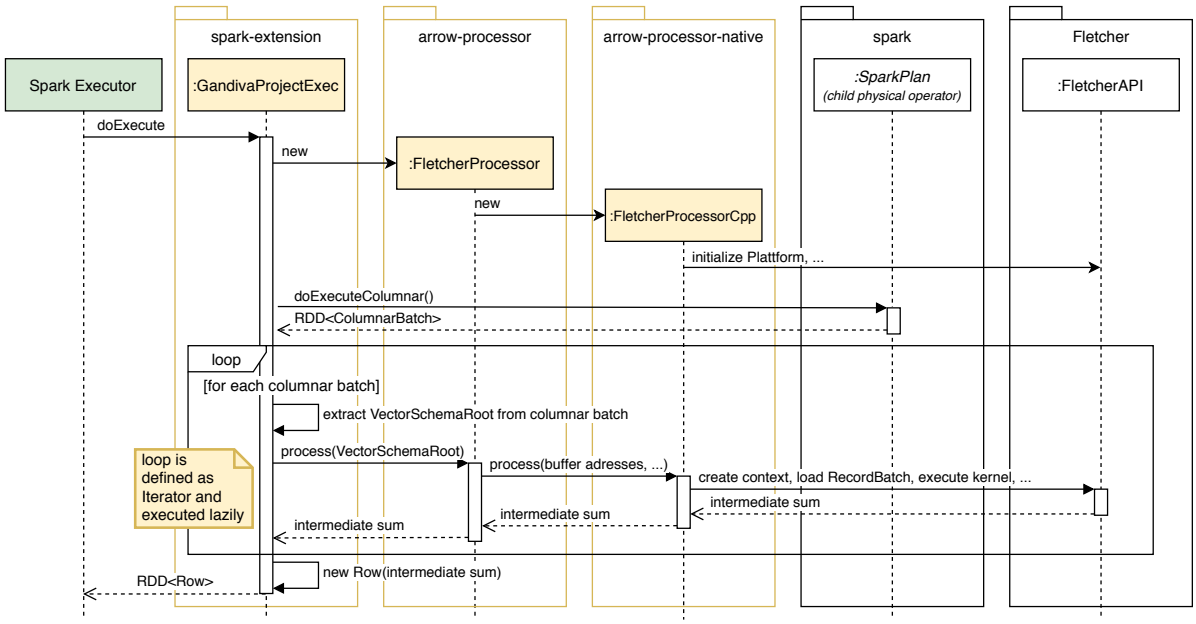


Figure 4.6: Implementation of the `FletcherExec` operator

A particular challenge has arisen related to the recommended Arrow buffer 64-byte alignment. Fletcher requires all buffers to be aligned on 64-byte, but the Java implementation only supports 8-byte alignment. Therefore, Arrow data initialized by the Java API cannot be processed easily with Fletcher. Luckily, in the taxi data use case, the data processed by Fletcher is initialized by the C++ Dataset Parquet file reader. As described in Section 3.4.4, this implementation uses a call-back mechanism to allocate buffers in Java. The API used for this is a thin layer around the Netty Buffer API [53], which can be configured to allocate 64-byte aligned buffers. Unfortunately, this configuration is not sufficient for Arrow arrays created with the Java API. Therefore, the Fletcher accelerator is, for example, not fully compatible with the Gandiva accelerator. However, the implemented approach is suitable for the taxi use case and allows evaluating performance improvements on it.

5 Evaluation

5.1. Setup

When executing code on the JVM, the code is optimized by the JIT compiler during run-time. Generally, many of these optimizations are based on different code heuristics and, therefore, the JIT compiler can optimize the code better the more often it executes it [50].

This effect can also be seen when executing a Spark query multiple times. Figure 5.1 shows an example of this on the use case of determining the maximum value of 500 million integers stored in a Parquet file. As shown, especially, the first iterations are slow, and it takes roughly ten iterations until the execution time stabilizes. Naturally, even then, the execution times vary, because of effects that cannot be controlled, such as background processes of the OS.

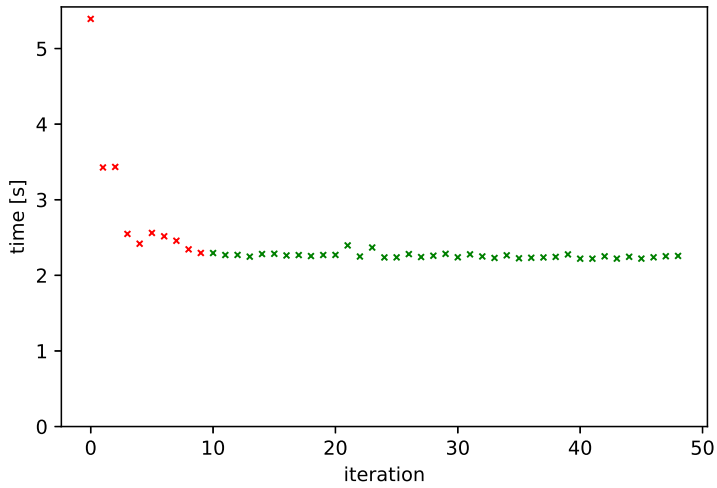


Figure 5.1: Changes to execution time when executing a Spark query multiple times. Query was executed on vanilla Spark and determines the maximum from 500 million integers stored in a Parquet file

Therefore, the measurements executed in the present work always start with a warm-up phase consisting of 20 iterations to give the JIT compiler the time to optimize the execution. Afterward, another 20 iterations are executed, and their average value is measured. To avoid defining these iterations manually, Java Microbenchmark Harness (JMH) is used. This OpenJDK project supports setting up environments for performance tests and allows configuring

the execution with annotations [50].

For determining the execution time, the present work relies on Spark's internal metrics [17]. Additionally, new metrics have been added to measure the execution of the custom executors. Unfortunately, this only works for columnar-based operators and not for Spark's internal row-based operators. Processing a single row does not take much time, and measuring this would add a high overhead.

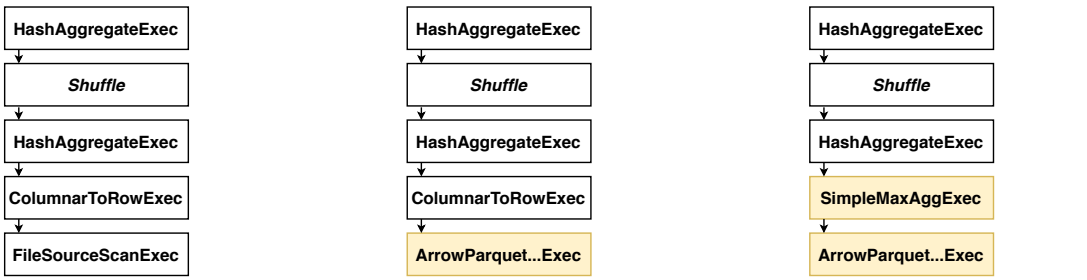
Except for the Fletcher-related use case, all measurements are executed on a Macbook Pro Early 2015 (Intel Core i5, 2.4 GHz, GB RAM) and the oracle JVM in version `1.8.0_241`. This work is only interested in the performance improvements of a single executor. Therefore, Spark is started in local mode using 1 core and 4GB memory for the execution. All files loaded by Spark, are stored on the internal SSD. If not mentioned differently, the evaluation uses a batch size of 100,000 elements.

5.2. Parquet reading

Firstly, the present work compares the custom Arrow-based Parquet reading implementation to Spark's default implementation. For this evaluation, a Parquet file with 500 million rows containing 3 integers (*x1*, *x2*, and *x3*) is used. To force Spark to process all data but avoid loading everything into memory, the following query returns the maximum of every row.

```
1 SELECT MAX(`x1`), MAX(`x2`), MAX(`x3`) FROM parquet.`500-million-triples.parquet`
```

This query has been executed on three differently configured Spark instances that map the query to different physical plans. The first configuration is the unmodified *Vanilla Spark*, which executes the physical plan shown in Figure 5.2a. Secondly, Spark uses the `ArrowParquetExtension` implemented in the present work, which creates a physical plan (Figure 5.2b) containing the custom operator loading the Parquet file into the Arrow format. Finally, the third executed configuration additionally uses the custom max aggregation thus replacing Spark's column into row transformation (see Figure 5.2c).



(a) Vanilla Spark (b) Custom Arrow parquet reading (c) Custom Arrow parquet reading with simple max aggregation

Figure 5.2: Parquet reading scenario: Physical plans of the different Spark configurations

The measured execution times are summarized in Figure 5.3. Thanks to Spark's internal metrics, it is possible to measure the time taken for reading the Parquet file. Evaluating these metrics reveals that Spark's default implementation is slightly faster (approximately 10%) then the custom implementation calling Arrow's C++ Dataset API. Nevertheless, Arrow's C++ Dataset API was not chosen because it promised to be faster. Instead, it allows the creation of pipelines entirely based on Arrow, and the current PoC implementation has to live with this trade-off.

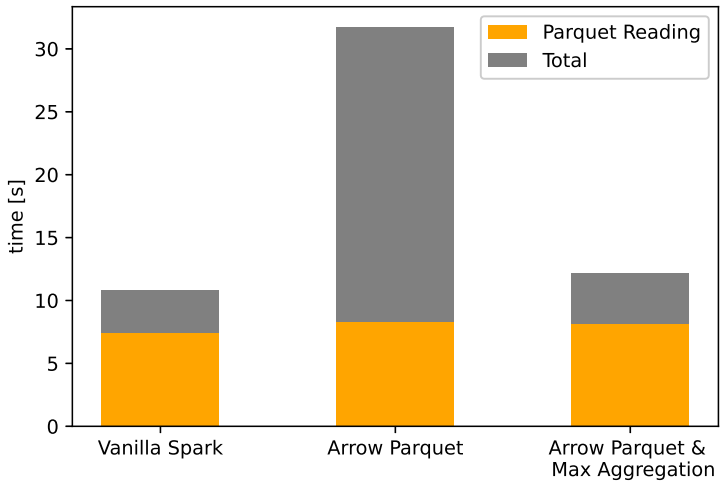


Figure 5.3: Comparing the execution of the Arrow-based Parquet reader with Vanilla Spark (importing 500 million integer triples with a batch size of 100,000 rows)

Furthermore, it is especially noticeable that determining the maximum value from the data in the Arrow format is a long-running computation. Consequently, the physical plan using the custom Arrow Parquet Reader takes three times longer than the Vanilla Spark. This demonstrates that converting the Arrow-based columnar format back to Spark's rows is a very expensive operation. In contrast, converting the columnar data produced by Spark's default Parquet Reader is highly optimized. With the improvement suggested in [37], Spark's Whole Stage Code Generation (see Subsection 2.1.4) generates specific code for reading data from Spark's internal columnar storage, which enables the JIT compiler to apply optimizations. Indeed, the custom max aggregation improved the performance of aggregating the Arrow data dramatically. Nevertheless, it was not sufficient to compete with Vanilla Spark's implementation and is still 1.1 times slower.

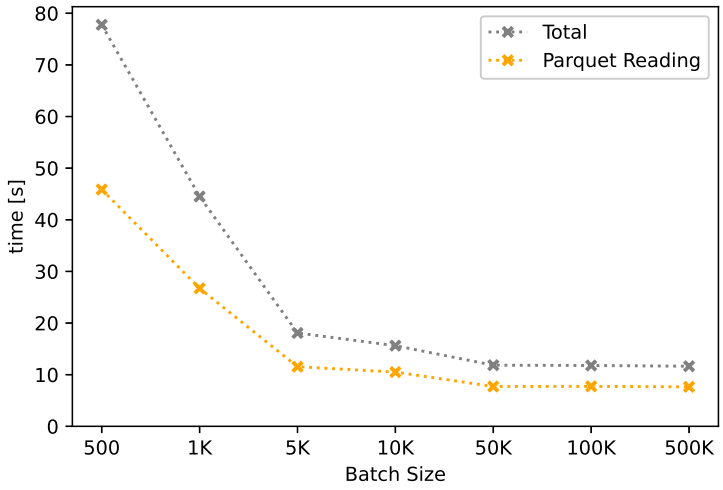


Figure 5.4: Effect of changing the batch size on the Arrow-based Parquet reader (incl. max aggregation) (importing 500 million integer triples)

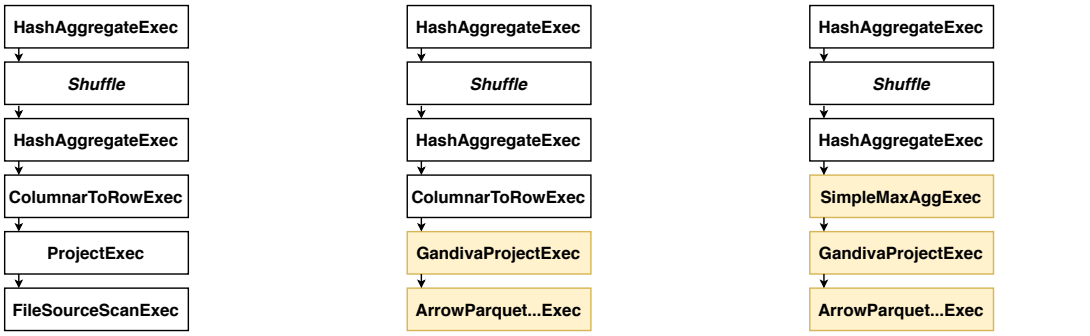
Finally, the present work elaborates on the impact of the `ColumnarBatch` size. Therefore, the execution of the third Spark configuration (Arrow Parquet reading, max aggregation) has been parameterized with the `ColumnarBatch` size. The results are shown in Figure 5.4. As expected, the execution is very slow for small batches, requiring calling the accelerator more often and having a high coordination overhead. With an increasing batch size, which requires more memory, it becomes faster and converges to the fastest possible execution. The Figure shows that the execution time of the batch size of 50.000 elements is very close to this limit and further increases of the batch size barely have a measurable impact on the execution.

5.3. Gandiva

For demonstrating the performance improvements of Gandiva, again, three different Spark configurations are compared. These are (1) Vanilla Spark, (2) Spark with the Arrow Parquet reader and the Gandiva accelerator, and finally (3) Spark with the Arrow Parquet reader, the Gandiva accelerator and the columnar-based max aggregation. Additionally to the previous query, the new query also includes a projection operation. The query reads a file containing 50 million records of ten integers. These ten integers are summed up, and their maximum is returned. The SQL query is defined as follow:

```
SELECT
  MAX(`x1` + `x2` + `x3` + `x4` + `x5` + `x6` + `x7` + `x8` + `x9` + `x10`)
FROM
  parquet.`50-million-10-ints.parquet`
```

Internally, these three configurations map the query into different physical plans, that are shown in Figure 5.5.



(a) Vanilla Spark

(b) Arrow parquet reading & Gandiva

(c) Arrow parquet, Gandiva & max aggregation

Figure 5.5: Gandiva scenario: Physical plans of the different Spark configurations

When evaluating the results shown in Figure 5.6, firstly, the same effects as before can be seen. The Arrow Parquet reading is significantly slower than Spark's internal implementation. Again aggregating the results in configuration (2) requires the transformation from the Arrow-based format into Spark's rows, which is a very costly operation.

Unfortunately, it is not possible to measure the time taken for calculating the sum of the ten integers in Vanilla Spark, because it is a row-based computation and does not allow to measure the execution efficiently. Therefore, it is only possible to compare the whole computation without the Parquet reading with each other. This does include not only the projection but also the aggregation. It can be seen that Gandiva and the maximum aggregation are executed 1.27 times faster than the computation of Vanilla Spark (see red arrows on Figure 5.6). This

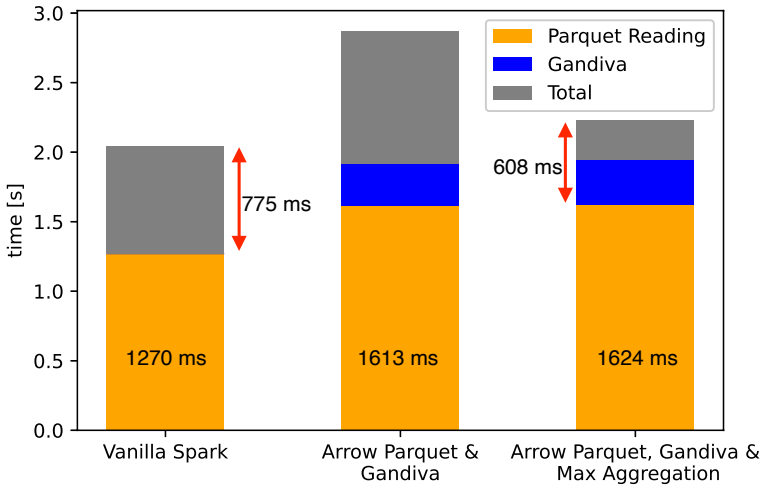


Figure 5.6: Comparing the execution of Gandiva-accelerated Spark with Vanilla Spark (calculating 50 million times the sum of 10 integers with a batch size of 100,000 rows)

experiment demonstrates that Spark's computation can be accelerated by using SIMD capabilities of modern GPUs. This improvement is not sufficient to overcome the slower Parquet reading. Nevertheless, the results demonstrate how important it is to avoid costly IO operations and the importance of efficient data aggregation. Moreover, it makes clear that queries with more complex computations also have a higher potential for acceleration.

Analyzing the impact of the `ColumnarBatch` size reveals similar results as before. Especially small batches lead to a much slower execution. Choosing a batch size of 50.000 elements leads to an execution close to the limit, and increasing the size barely affects the computation.

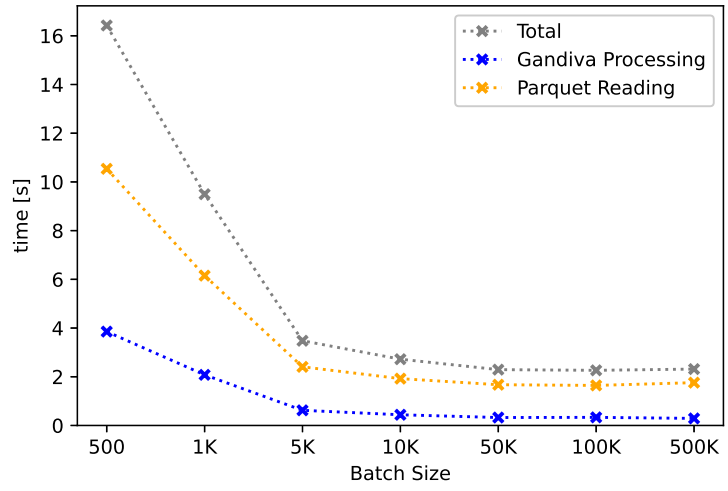


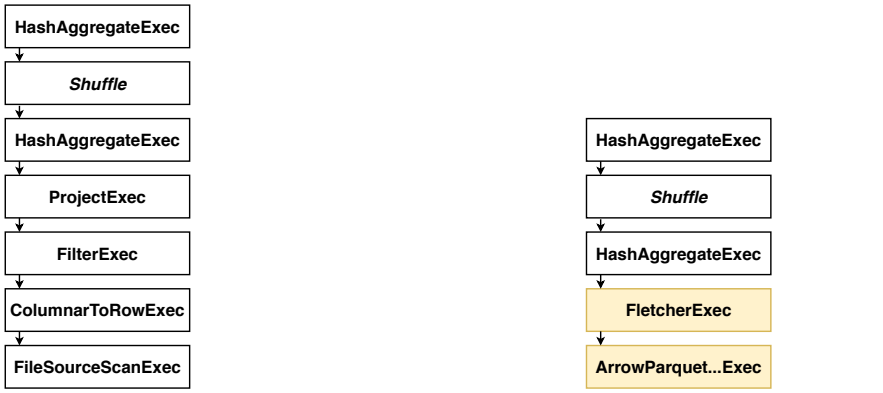
Figure 5.7: Effect of changing the batch size on Gandiva-accelerated Spark (incl. max aggregation) (calculating 50 million times the sum of 10 integers)

5.4. Fletcher

As distinguished from the previous experiments, the Fletcher scenario is not based on random data sets. As described in Section 4.5, it uses the Chicago taxi data [40] and determines the total duration of all trips operated by the *Blue Ribbon Taxi Association Inc* company. The following query defines this scenario.

```
SELECT
  SUM(`trip_seconds`)
FROM
  parquet.`chicago-taxi.parquet`
WHERE
  `company` rlike "Blue Ribbon Taxi Association Inc"
```

Similarly, as before, this experiment compares the execution of Spark's default physical plan (Figure 5.8a) with the Fletcher-accelerated one (Figure 5.8b).



(a) Vanilla Spark (b) Arrow Parquet Reading & Fletcher

Figure 5.8: Fletcher scenario: Physical plans of the different Spark configurations

The Fletcher experiments were conducted on a POWER9 system with two CPUs (44 cores), 128 GB memory, and an AlphaData ADM-PCIE-9H7 FPGA accelerator card. In this Setup, Spark was executed in local mode with 16 GB memory and a `ColumnarBatch` size of 1 million records was chosen.

Figure 5.9 shows the results of comparing the Fletcher-accelerated Spark version with Vanilla Spark on reading data sets of different size. First of all, this use case, which includes reading string data, reveals the weaknesses of the Arrow Parquet reading implementation even more clearly than before. While Vanilla Spark was able to import the parquet file containing 150 million entries (about 130MB) within 250ms, the Arrow implementation needed 9.6 seconds. This work could not to find a satisfying explanation for this enormous difference. However, it could show that the problem is not directly related to the Spark integration or the memory allocation in Java. A separate C++ program just importing the file with the *Arrow C++ Dataset API* resulted in a similar importing time.

Nevertheless, the FPGA-accelerated Spark configuration could increase the performance of the other processing operations enormously. For the data set with 150 million record, the remaining processing steps could be executed more than 13 times faster (see red arrows on Figure 5.9). This improvement is slightly smaller for the other data sets, but sufficient to compensate the much slower Parquet file import. In total, the accelerated variant was more than two times faster than the default implementation for all data set sizes. The FPGA's

parallel execution capabilities can clearly explain this improvement compared to the single-threaded Spark CPU executor. Nevertheless, the current implementation of the HAF is also only using a small part of the FPGA's capacity. This experiment demonstrates that Spark can be accelerated with different computing hardware, but it leaves the problem of using the hardware at full capacity for further work.

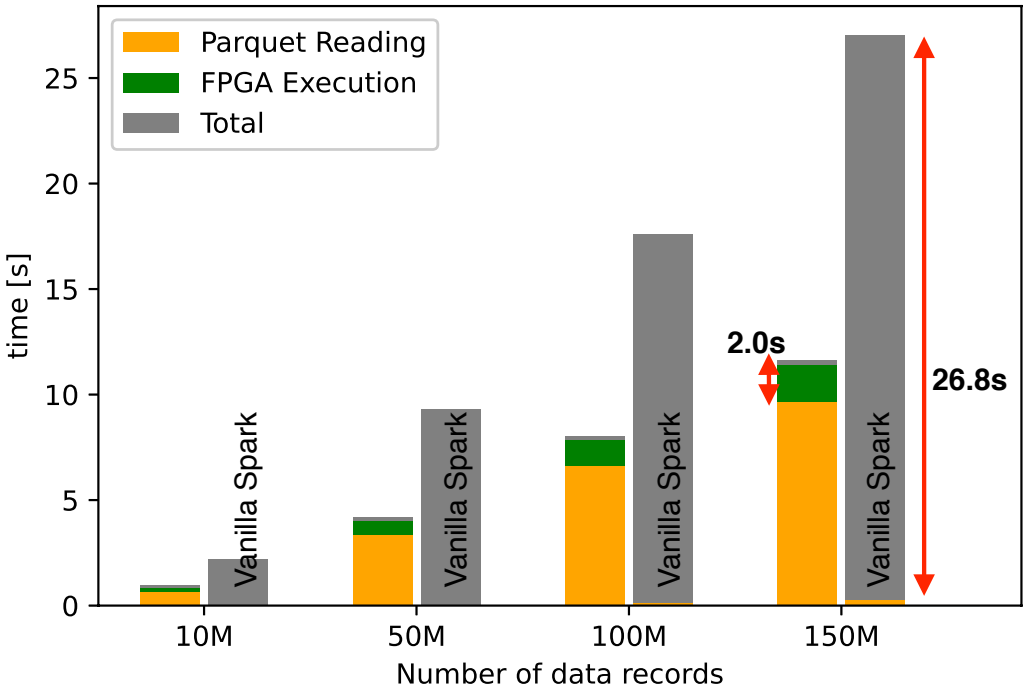


Figure 5.9: Comparing the execution of Fletcher-accelerated Spark to Vanilla Spark (Chicago taxi data use-case presented in Section 4.5 using a batch size of 1 million rows)

6

Conclusions and future work

6.1. Conclusions

The research question *"Can Spark SQL's internal structural information of the query be used to accelerate the query execution by offloading work to hardware accelerators based on Apache Arrow?"*, formulated in Section 1.3, has set the primary purpose of the present work to evaluate the technical feasibility of accelerating Spark SQL transparently. Firstly, the present work analyzed the involved technologies, such as Apache Spark, Apache Arrow, and Fletcher. It elaborated recent features supporting heterogeneous computing and used these insights to create an architecture and concepts for integrating different tools. During the work, several custom accelerators were implemented to validate the developed concepts and to discover further challenges. This successful PoC implementation was evaluated with regard to first performance improvements and to gain further insights on necessary preconditions for strong acceleration.

The first subquestion, *"Does Spark SQL provide sufficient extension points that allow the provision of different hardware accelerators?"* was answered by analyzing Spark SQL deeply and presenting its concepts. It has been shown that especially the query optimizer Catalyst plays an important role. Internally, it processes the query defined by the user and creates precise execution instructions. Additionally, it is designed and built very modifiable and provides an extensive API to manipulate the query optimization process. With the new columnar processing function, which perfectly integrates into Catalyst's APIs, Spark took a significant step towards heterogeneous computing and laid an essential basis for columnar accelerators. The architecture developed within the present work (Section 3.2) highly relies on Catalyst's API. It uses the API to define custom accelerators (physical operators) and uses the rule-based Spark extensions to replace Spark's default operators. This demonstrates the extensive possibilities to modify Spark SQL and shows that Spark SQL's API fulfills all necessary preconditions for the present work and allows extending Spark with custom implementations calling hardware accelerators.

Besides Spark, Apache Arrow is a central component that connects all tools used in the present work. Its columnar in-memory format and the language-specific abstraction allow exchanging data without copying overhead. The idea of the present work is based on using Apache Arrow for exchanging data efficiently between Spark and tools from the Arrow ecosystem, such as Gandiva and Fletcher. Validating this idea led to the second subquestion, *"How mature is Spark's columnar processing function and is it compatible with the Apache Arrow memory format?"*. The present work discussed that Spark's columnar processing functionality does not yet fully integrate with Apache Arrow. Although Spark contains Arrow-based implementations, the underlying Arrow structures are not part of the public API and can only be

extracted by accessing hidden fields. Moreover, when Spark converts rows into the columnar format, it uses an internal format. But, as the concept of the present work (Section 3.3) discusses, these restrictions can be overcome and enable Spark to manage its data in the Arrow format. However, the performance evaluations showed a disadvantage, which remained. Spark's columnar-to-row conversion is optimized for the internal format, and using the Arrow format leads to significantly slower conversions.

Based on the Apache Arrow format, the present work integrated hardware-accelerators and libraries from the Arrow ecosystem with Spark. Firstly, this was the *Arrow C++ Dataset* component. It allows importing Parquet files directly into the Arrow format and enables the present work to execute full data analytics pipelines without having to transform the data. Moreover, this integration demonstrates how Arrow data imported in C++ can be made available to Spark implemented in Scala/Java. Secondly, the present work provides an integration that conceptually allows executing all of Spark's *projection* and *filter* operations on Gandiva. By converting Spark's operations into Gandiva's expression trees, these operations are compiled into assembly code optimized for CPUs with SIMD capabilities and are accelerating Spark's internal computation. Finally, the present work provides a Fletcher integration as a prototype, allowing offload computational work to FPGAs. While implementing these integrations, the present work also has revealed differences between the tools. Gandiva's filter operation is returning a vector containing all indexes of the selected rows, which does not find a conceptional equivalent in Spark and is therefore not fully compatible. Furthermore, Fletcher requires Arrows suggested 64-byte alignment. However, Arrow's Java library only aligns the buffers on 8 bytes and makes it difficult to send data to Fletcher.

Finally, to answer the third subquestion, *"Which performance improvements can be identified and what are potential bottlenecks?"*, the present work conducted experiments on different use cases. These experiments have shown that transforming the data between the Arrow format, and Spark's row-based format are costly operations. Therefore, the best results can be achieved when these transformations are avoided and the whole data analytics pipeline manages the data in the Arrow format. It became clear that aggregations are common operations and an Arrow-based implementation is required to avoid transferring the data to Spark's format first. Furthermore, the performance evaluations have revealed that the Parquet reading implementation is significantly slower than Spark's implementation, making it hard for Arrow-based computations to compete with Vanilla Spark. Nevertheless, the performance experiments have demonstrated that Spark can be significantly accelerated by using hardware accelerators. Not considering the Parquet reading operations, the Gandiva-accelerated computation was up to 1.27 times faster and the Fletcher-accelerated version could compute the data of the Chicago taxi use case more than 13 times faster.

6.2. Further work

The present work has focused on elaborating on the technical feasibility of integrating Spark with the Apache Arrow ecosystem and, in particular, with Fletcher. This feasibility has been demonstrated, being an important step towards the presented vision of heterogeneous computing based on Apache Arrow. During the evaluations and implementations done in the present work, many new questions and challenges arose, which could not be answered in favor of focusing on the main purpose of the present work. Therefore, the present work creates a basis for further implementations and additional integrations with other work of the community.

As the evaluation has shown, the Arrow-based Parquet reading function, developed in the present work, is significantly slower than Spark's default implementation and has a significant impact on the overall execution. To overcome this limitation, it is necessary to analyze this issue further and evaluate other implementations such as Parquet Mr [47], which provides

a Java-based implementation to import Parquet files into the Arrow format. Moreover, the implementation of the present work focuses on the basic functionality. To reach feature parity with Spark's implementation, it is necessary to integrate additional features of the Arrow C++ Dataset library such as predicate push-down and to read from network shares.

An essential result of the present work is that the conversion between Apache Arrow and Spark's internal row format should be avoided. For being able to accelerate a wide variety of use cases, it is necessary to provide Arrow-based implementations for most of Spark's physical operators. To be pointed out is the aggregation operator, which is an essential operation for many data analytics pipelines and also enables the execution of the TPC-H benchmarks [52]. The TPC-H benchmark is a standardized database benchmark consisting of business oriented ad-hoc queries and concurrent data modifications. This benchmark has been defined to have broad industry-wide relevance and allows a better evaluation of the performance improvements and a comparison with other available data analytics systems.

However, implementing columnar-based versions for all operators is an enormous task due to Spark's extensive features. Therefore, it is essential to exploit potential synergies with other publicly available work. Particularly Intel's *"Spark Native SQL Engine"* project [36], introduced in Section 2.4, is an interesting candidate. At the time of writing, this project was still in an early stage, but similarly, as the present work, it uses Spark's columnar processing feature and has integrated an Arrow Parquet reader, Gandiva and further columnar processing operators. To follow the vision of heterogeneous computing and making Fletcher accessible for broader data analytics, this project is an interesting candidate and should be evaluated further.

The present work has focused on accelerating a single executor node and does not discuss challenges related to distributed Setups. Therefore, challenges such as using the available resources efficiently, choosing a physical plan based on the available resources and distributing the work in a heterogeneous cluster are not addressed and remain open for further work.

This work forms the basis for the ABS group's vision, presented in Section 4.5, to accelerate Spark queries with Fletcher-based HAFs without losing the usability of Spark's powerful API. The key idea is to choose a matching HAF automatically from a repository and integrate it into the execution to accelerate the execution as much as possible. This approach requires a tree comparison between Spark's logical plan and the HAFs. These additional steps necessary to achieve this vision have not been considered in the present work and remain open for further work.

With version 3.0.0, Spark has announced the Adaptive Query Execution feature [62], which gathers statistics at runtime and uses them to adapt the execution dynamically. Integrating with this function might be a valuable addition to the greater vision of the presented work. Using this feature might allow switching dynamically to a different accelerator based on the insights gathered from the first executions.

A

Measurement Results

This appendix lists the exact measured values from the experiments conducted in Chapter 5. The implementations of the experiments can be found in the repository[1] containing the implementation of the present work.

A.1. Parquet Reading

The experiments conducted in Section 5.2 compare the Arrow-based Parquet reader, implemented in the present work, with Vanilla Spark. They were executed on Spark started in local mode using 4 GB running on a Macbook Pro Early 2015 (Intel Core i5, 2.4 GHz, GB RAM). The following tables show the average value of 20 executions after another 20 warm-up executions

	Vanilla Spark	**Arrow Parquet Reader**	**Arrow Parquet & Max Aggregation**
Parquet reading	7.5 s	8.4 s	8.2 s
Total	10.8 s	31.8 s	12.1 s

Table A.1: Comparing the execution of the Arrow-based Parquet reader with Vanilla Spark (importing 500 million int triples with a batch size of 100,000 rows)

batch size	**500**	**1K**	**5K**	**10K**	**50K**	**100K**	**500K**
Parquet reading	45.8 s	26.7 s	11.5 s	10.5 s	7.7 s	7.7 s	7.6 s
Total	77.7 s	44.5 s	18.1 s	15.6 s	11.8 s	11.7 s	11.6 s

Table A.2: Effect of changing the batch size on the Arrow-based Parquet reader (incl. max aggregation) (importing 500 million integer triples)

A.2. Gandiva

Section 5.3 shows the experiments to evaluate the performance of the Gandiva accelerator. The experiments were conducted on the same setup as the parquet reading and the following tables show the measured values for the comparison with Vanilla Spark and the effect of the batch size:

[1] https://github.com/fnonnenmacher/spark-arrow-accelerated#performance-tests

	Vanilla Spark	Gandiva	Gandiva & Max Aggregation
Parquet reading	1.27 s	1.61 s	1.62 s
Gandiva (sum)	-	0.31 s	0.32 s
Total	2.05 s	2.87 s	2.23 s

Table A.3: Comparing the execution of Gandiva-accelerated Spark with Vanilla Spark (calculating 50 million times the sum of 10 integers with a batch size of 100,000 rows)

batch size	500	1K	5K	10K	50K	100K	500K
Parquet reading	10.54 s	6.15 s	2.41 s	1.92 s	1.67 s	1.64 s	1.76 s
Gandiva (sum)	3.85 s	2.08 s	0.62 s	0.44 s	0.33 s	0.33 s	0.29 s
Total	16.43 s	9.50 s	3.48 s	2.72 s	2.29 s	2.26 s	2.32 s

Table A.4: Effect of changing the batch size on Gandiva-accelerated Spark (incl. max aggregation) (calculating 50 million times the sum of 10 integers)

A.3. Fletcher

The Fletcher related experiments of Section 5.4 were conducted on a POWER9 system with two CPUs (44 cores), 128 GB memory, and an AlphaData ADM-PCIE-9H7 FPGA accelerator card. In this Setup, Spark was executed in local mode with 16 GB memory and a configured batch size of 1 million records.

	10M	50M	100M	150M
[Vanilla] Parquet reading	0.02 s	0.04 s	0.11 s	0.25 s
[Vanilla] Total	2.22 s	9.31 s	17.59 s	27.06 s
[Fletcher] Parquet reading	0.69 s	3.36 s	6.64 s	9.65 s
[Fletcher] FPGA execution	0.15 s	0.68 s	1.21 s	1.77 s
[Fletcher] Total	0.98 s	4.20 s	8.03 s	11.62 s

Table A.5: Comparing the execution of Fletcher-accelerated Spark to Vanilla Spark (Chicago taxi data use-case using a batch size of 1 million rows with different data set sizes)

Bibliography

[1] Michael Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797.

[2] Apache Arrow. *Apache Arrow Docs v0.17.1*. 2020. URL: https://arrow.apache.org/docs/index.html (visited on 07/01/2020).

[3] Apache Arrow. *Apache Arrow homepage*. URL: https://arrow.apache.org/ (visited on 06/30/2020).

[4] Apache Arrow. *Apache Arrow on Github (v0.17.1)*. May 2020. URL: https://github.com/apache/arrow/tree/apache-arrow-0.17.1 (visited on 07/13/2020).

[5] Apache Arrow. *Gandiva: A LLVM-based Analytical Expression Compiler for Apache Arrow*. Dec. 2018. URL: https://arrow.apache.org/blog/2018/12/05/gandiva-donation/ (visited on 07/02/2020).

[6] Himani Bansal. *Why Industries Running behind Spark — “50 Reasons why spark is important”*. Feb. 2020. URL: https://medium.com/javarevisited/why-industries-running-behind-spark-50-reasons-why-spark-is-important-83473477b41d (visited on 07/20/2020).

[7] Bigstream. *Hyperacceleration With Bigstream Technology*. URL: https://blog.bigstream.co/resources/hyper-acceleration-with-bigstream-technology (visited on 08/04/2020).

[8] Priyabrata Biswas. *Towards Data Science: Introduction to FPGA and its Architecture*. 2019. URL: https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c (visited on 07/20/2020).

[9] Rajesh Bordawekar. *Accelerating Spark workloads using GPUs O’Reilly*. Aug. 2016. URL: https://www.oreilly.com/content/accelerating-spark-workloads-using-gpus/ (visited on 08/04/2020).

[10] Rajesh Bordawekar. *Nvidia GTC Silicon Valley 2016: Accelerating Spark Workloads Using GPUs*. 2016. URL: https://on-demand-gtc.gputechconf.com/gtcnew/sessionview.php?sessionName=s6280-accelerating+spark+workloads+using+gpus (visited on 08/04/2020).

[11] Weiting Chen Calvin Hung. *Spark+AI Summit 2020: Accelerating Spark SQL Workloads to 50X Performance with Apache Arrow-Based FPGA Accelerators*. 2020. URL: https://databricks.com/session_na20/accelerating-spark-sql-workloads-to-50x-performance-with-apache-arrow-based-fpga-accelerators (visited on 08/04/2020).

[12] Cloudera. *Cloudera Documentation v6.3.x : Predicate Pushdown in Parquet*. URL: https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/cdh_ig_predicate_pushdown_parquet.html (visited on 07/09/2020).

[13] Falcon Computing. *Kestrel Runtime - Big Data application scheduling and load balancing*. URL: https://www.falconcomputing.com/falcon-kestrel-runtime/ (visited on 08/04/2020).

[14] Databricks. *Glossary - Tungsten*. URL: https://databricks.com/glossary/tungsten (visited on 07/10/2020).

[15] Databricks. *What is Apache Spark*. URL: https://databricks.com/spark/about (visited on 06/28/2020).

[16] Quora Discussion. *Why is Apache Spark popular among data scientists?* 2018. URL: https://www.quora.com/Why-is-Apache-Spark-popular-among-data-scientists (visited on 07/20/2020).

[17] Apache Spark 3.0.0 Documentation. *Monitoring*. June 2020. URL: https://spark.apache.org/docs/latest/monitoring.html (visited on 07/28/2020).

[18] Apache Spark 3.0.0 Documentation. *PySpark Usage Guide for Pandas with Apache Arrow*. 2020. URL: https://spark.apache.org/docs/3.0.0/sql-pyspark-pandas-with-arrow.html (visited on 07/13/2020).

[19] Apache Spark 3.0.0 Documentation. *RDD Programming Guide*. June 2020. URL: http://spark.apache.org/docs/3.0.0/rdd-programming-guide.html (visited on 07/08/2020).

[20] Apache Spark 3.0.0 Documentation. *Spark SQL and DataFrames*. June 2020. URL: https://spark.apache.org/docs/3.0.0/sql-programming-guide.html (visited on 07/07/2020).

[21] LLVM 10 Documentation. *Auto-Vectorization in LLVM*. 2020. URL: https://llvm.org/docs/Vectorizers.html (visited on 07/02/2020).

[22] Oracle Java Documentation. *Java Native Interface Specification*. 2014. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html (visited on 07/23/2020).

[23] Scala Documentation. *Iterators*. 2020. URL: https://docs.scala-lang.org/overviews/collections-2.13/iterators.html (visited on 07/23/2020).

[24] Daniel Eaton. *Turning big data challenges into opportunities with FPGA-accelerated computing*. 2018. URL: https://www.datacenterdynamics.com/en/opinions/turning-big-data-challenges-opportunities-fpga-accelerated-computing/ (visited on 07/20/2020).

[25] Jian Fang et al. “In-memory database acceleration on FPGAs: a survey”. In: *The VLDB Journal* 29.1 (Jan. 2020), pp. 33–59. ISSN: 0949-877X. DOI: 10.1007/s00778-019-00581-w.

[26] Jian Fang et al. “Refine and Recycle: A Method to Increase Decompression Parallelism”. In: July 2019, pp. 272–280. DOI: 10.1109/ASAP.2019.00017.

[27] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[28] The Apache Software Foundation. *Apache® Software Foundation announces Apache Arrow™ as a Top-Level Project*. Feb. 2016. URL: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces87 (visited on 06/30/2020).

[29] G. Graefe. “Volcano— An Extensible and Parallel Query Evaluation System”. In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. ISSN: 1041-4347. DOI: 10.1109/69.273032.

[30] Accelerated Big Data Systems Group. *Fletcher on Github*. 2020. URL: https://github.com/abs-tudelft/fletcher (visited on 07/07/2020).

[31] Stavros Harizopoulos, Daniel Abadi, and Peter Boncz. *Column-Oriented Database Systems*. 2009. URL: http://www.cs.umd.edu/~abadi/talks/Column_Store_Tutorial_VLDB09.pdf (visited on 07/11/2020).

[32] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[33] Joost Hoozemans et al. “VLIW-Based FPGA Computation Fabric with Streaming Memory Hierarchy for Medical Imaging Applications”. In: Mar. 2017, pp. 36–43. DOI: 10.1007/978-3-319-56258-2_4.

[34] Ernst Joachim Houtgast et al. “Hardware Acceleration of BWA-MEM Genomic Short Read Mapping with Longer Read Length”. In: *Computational Biology and Chemistry* 75 (Jan. 2018). DOI: 10.1016/j.compbiolchem.2018.03.024.

[35] Muhuan Huang et al. “Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC ’16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, pp. 456–469. ISBN: 9781450345255. DOI: 10.1145/2987550.2987569.

[36] Intel. *Spark Native SQL Engine - Optimized Analytics Package for Spark Platform*. 2020. URL: https://github.com/Intel-bigdata/OAP/tree/master/oap-native-sql (visited on 08/03/2020).

[37] Kazuaki Ishizaki. “Analyzing and Optimizing Java Code Generation for Apache Spark Query Plan”. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE ’19. Mumbai, India: Association for Computing Machinery, 2019, pp. 91–102. ISBN: 9781450362399. DOI: 10.1145/3297663.3310300.

[38] Spark 3.0.0 JavaDoc. *SparkSessionExtensions*. 2020. URL: http://spark.apache.org/docs/3.0.0/api/java/org/apache/spark/sql/SparkSessionExtensions.html (visited on 07/23/2020).

[39] Dr. Chris Kachris. *FPGA Acceleration of Apache Spark on the Cloud, Instantly*. 2018. URL: https://bigdataconference.lt/2018/wp-content/uploads/2018/12/Machine-Learning-Acceleration-using-FPGAs-in-the-Cloud-by-Christophoros-Kachris-min.pdf (visited on 08/04/2020).

[40] Kaggle. *Chicago Taxi Trips*. 2020. URL: https://www.kaggle.com/chicago/chicago-taxi-trips-bq (visited on 07/27/2020).

[41] Romeo Kienzler. *Mastering Apache Spark 2.x: Scalable analytics faster than ever*. Second. Birmingham, UK: Packt Publishing Ltd, July 2017. ISBN: 9781786462749.

[42] Peilong Li, Ning Zhang, and Yu Cao. “HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms”. In: Aug. 2015. DOI: 10.1109/NAS.2015.7255222.

[43] Boston Limited. *What Is GPU Computing?* URL: https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx (visited on 07/20/2020).

[44] Philipp Moritz and Robert Nishihara. *Plasma In-Memory Object Store | Apache Arrow*. Aug. 2017. URL: https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/ (visited on 07/23/2020).

[45] NVIDIA. *Spark RAPIDS plugin - accelerate Apache Spark with GPUs*. 2020. URL: https://github.com/NVIDIA/spark-rapids (visited on 08/04/2020).

[46] Apache Parquet. *Apache Parquet Documentation*. URL: https://parquet.apache.org/documentation/latest/ (visited on 07/02/2020).

[47] Apache Parquet. *Apache Parquet MR on Github*. 2019. URL: https://github.com/apache/parquet-mr/tree/apache-parquet-1.11.0 (visited on 07/24/2020).

[48] J. Peltenburg et al. “Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 270–277. DOI: 10.1109/FPL.2019.00051.

[49] Johan Peltenburg, Shanshan Ren, and Zaid Al-Ars. “Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm”. In: Dec. 2016, pp. 758–762. DOI: 10.1109/BIBM.2016.7822616.

[50] Kevin Peters. *codecentric AG Blog: Performance measurement with Java Microbenchmark Harness*. 2017. URL: https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/ (visited on 07/28/2020).

[51] Ravindra Pindikura. *Dremio - Introducing the Gandiva Initiative for Apache Arrow*. June 2018. URL: https://www.dremio.com/announcing-gandiva-initiative-for-apache-arrow/ (visited on 07/02/2020).

[52] Meikel Poess and Chris Floyd. “New TPC Benchmarks for Decision Support and Web Commerce”. In: *SIGMOD Rec.* 29.4 (Dec. 2000), pp. 64–71. ISSN: 0163-5808. DOI: 10.1145/369275.369291.

[53] The Netty project. *Docs: Buffer API*. URL: https://netty.io/wiki/using-as-a-generic-library.html (visited on 07/14/2020).

[54] RAPIDS. *Open GPU Data Science*. URL: https://rapids.ai/about.html (visited on 08/04/2020).

[55] Jason Lowe Robert Evans. *Spark+AI Summit 2020: Deep Dive into GPU Support in Apache Spark 3.x*. July 2020. URL: https://databricks.com/de/session_na20/deep-dive-into-gpu-support-in-apache-spark-3-x (visited on 08/04/2020).

[56] Josh Rosen. *Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal*. 2015. URL: https://databricks.com/de/session/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal (visited on 07/20/2020).

[57] Davies Liu Sameer Agarwal and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop - The Databricks Blog*. May 2016. URL: https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html (visited on 07/11/2020).

[58] Apache Spark. *Github - Apache Spark source code (v3.0.0)*. June 2020. URL: https://github.com/apache/spark/tree/v3.0.0 (visited on 07/13/2020).

[59] Apache Spark. *Spark Issue Tracker*. URL: https://issues.apache.org/jira/browse/SPARK (visited on 07/09/2020).

[60] Apache Spark. *Spark Release 3.0.0*. June 2020. URL: https://spark.apache.org/releases/spark-release-3-0-0.html (visited on 07/14/2020).

[61] Statista. *Digital Economy Compass 2019*. 2019. URL: https://cdn.statcdn.com/download/pdf/DigitalEconomyCompass2019.pdf (visited on 06/26/2020).

[62] Herman van Hövell Wenchen Fan and MaryAnn Xue. *Databricks: How to Speed up SQL Queries with Adaptive Query Execution*. May 2020. URL: https://databricks.com/de/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html (visited on 08/03/2020).

[63] Alex Woodie. *A Decade Later, Apache Spark Still Going Strong*. 2019. URL: https://www.datanami.com/2019/03/08/a-decade-later-apache-spark-still-going-strong (visited on 07/20/2020).

[64] Reynold Xin and Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. Apr. 2015. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (visited on 07/10/2020).

[65] Y. Yuan et al. “Spark-GPU: An accelerated in-memory data processing engine on clusters”. In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, pp. 273–283. DOI: 10.1109/BigData.2016.7840613.

[66] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, 2012, p. 2.