FPGAs in Big Data

On the transparent and efficient acceleration of big data frameworks

by

Bob Luppes

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday July 28, 2021 at 1:00 PM.

Student number:	4370236	
Project duration:	November 1, 2020 – August 1, 2021	
Thesis committee:	Dr. Zaid Al-Ars,	TU Delft, supervisor
	Matthijs Brobbel,	Teratide, supervisor
	Dr. Asterios Katsifodimos,	TU Delft

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

TU Delft

Intentionally left blank

Abstract

The increasing volume and latency requirements of big data impose challenges on the processing capacity of existing computing systems. FPGA accelerators can be leveraged to overcome these challenges, but questions remain as to how these accelerators are best deployed to accelerate big data frameworks. This work investigates how future big data frameworks should be designed such that they facilitate transparent and efficient integration with FPGA accelerators in the context of SQL workloads. Three big data frameworks are accelerated to answer this question. These implementations offload the evaluation of a regular expression filter to Tidre, an FPGA-based regular expression matcher. First, Dremio is accelerated to obtain a 1750x speedup of the filter operator. Second, Dask is accelerated and a speedup of 92x is achieved for the same operator. Lastly, an accelerated version of Dask distributed is implemented. This implementation is deployed in a cluster environment to attain an end-to-end speedup of 3.6x as well as a reduction of the total cost per query by 23%. It is identified that query exploration phases could be used to increase the impact of existing FPGA kernels. In addition, it is found that the batch size of batch-processing frameworks has a significant impact on the performance of FPGA accelerated big data frameworks in distributed setups. Tuning this batch size can increase end-to-end query throughput up to 2.1x. Finally, future big data frameworks should make use of hardware-friendly and language-independent in-memory formats such as Apache Arrow. Correct alignment of the memory buffers of this data format can further increase the throughput of the system by 2.5x.

Preface

This thesis is carried out at the Accelerated Big Data Systems group as part of an internship at Teratide.

I would like to express my thanks to everyone that supported me during the MSc thesis project. Over the course of nine months, I have received excellent guidance from the staff at the Accelerated Big Data Systems group as well as from my colleagues at Teratide. The weekly meetings and numerous Discord calls have been a key motivator and have directly attributed to the success of this thesis.

Special thanks go out to Dr. Zaid Al-Ars, my supervisor at Delft University of Technology, and Matthijs Brobbel, my supervisor at Teratide. Zaid's enthusiasm during all stages of the project has never failed to inspire me. At the same time, Matthijs' in-depth knowledge regarding all facets of the project has proven invaluable. During the full length of these nine months I have not been able to come up with a single problem that cannot be solved by Matthijs or his colleagues.

Lastly, I would like to thank my peers at the Accelerated Big Data Systems group, with whom I have had interesting discussions related to the subject of this work. The same holds true for my family and friends, who were forced to listen to me talking about this project whenever I found the opportunity to do so. Their input has helped shape this thesis as it lies before you today.

Bob Luppes
Den Haag, July 2021

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ABS	Accelerated Big Data Systems
API	Application Programming Interface
AWS	Amazon Web Services
BI	Business Intelligence
CAPI	Coherent Accelerator Processor Interface
CPU	Central Processing Unit
CuDF	CUDA DataFrame
CXL	Compute Express Link
DAG	Directed Acyclic Graph
EC2	Elastic Compute Cloud
ETL	Extract, Transform, Load
FPGA	Field-Programmable Gate Array
GFS	Google File System
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDFS	Hadoop File System
HLS	High-Level Synthesis
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
JIT	Just-in-Time
JNI	Java Native Interface
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
MSc	Master of Science
NIC-QF	Network Interface Card with Query Filter
PhD	Philosophiae Doctor
PoC	Proof of Concept
RDBMS	Relational Database Managemen System
RDD	Resilient Distributed Dataset

RTL Register Transfer Level

SIMD Single Instruction Multiple Data

SLA Service-Level Agreement

SQL Structured Query Language

SSD Solid-State Drive

UTF-8 Unicode Transformation Format - 8-bit

XRT Xilinx Run Time

1 Introduction

1.1. Context

The sheer scale of big data is often emphasized by Laney's *3V model*, representing the volume, velocity, and variety of data [1]. An overview of this model can be seen in Figure 1.1. The volume indicates the amount of data that is generated. It is estimated that 90% of all data worldwide has been created in the last two years alone [2]. The velocity is used to indicate the increasing latency requirements for time-sensitive data, while the variety indicates the wide range in which this data can be encoded.

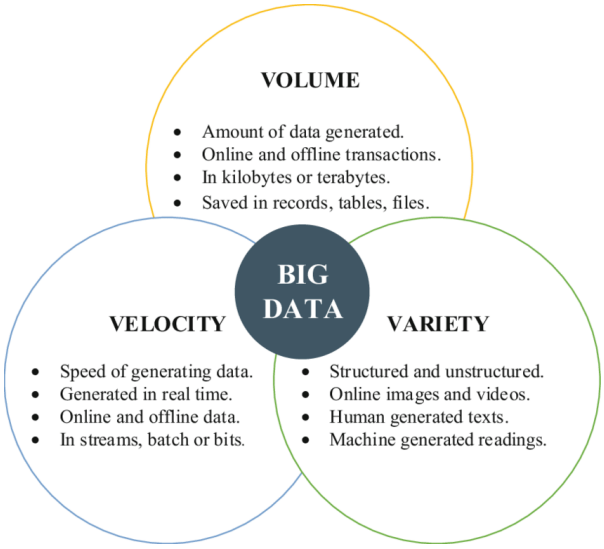


Figure 1.1: Laney's 3V model for big data [3].

These trends impose challenges on how to store and process this data. This work focuses on the latter of these two; processing capacity.

Increasing the processing capacity has been a popular subject of prior research. Examples include CPU optimizations and the use of dedicated hardware accelerators such as GPUs. Another accelerator is the field-programmable gate array or FPGA. These FPGAs consist of a re-configurable fabric that can be programmed to implement custom integrated circuit (IC) designs. This work investigates how these FPGA accelerators can be efficiently deployed to increase processing capacity in a big data context.

For this purpose, it is important to understand how conventional big data applications perform their computations. Because of the enormous variety of big data, there is no one-size-fits-all solution. As a result, the big data landscape is fractured. Figure 1.2 shows a small fragment of the entire big data landscape as identified by Matt Turck. It can be seen that there is a large number of big data frameworks, all specializing in different areas of big data.

Figure 1.2: A fraction of Matt Turck's big data landscape [4].

Different aspects of these frameworks play an important role on the effectiveness of FPGA acceleration. Furthermore, not all frameworks are designed with heterogeneous computing concepts in mind. This complicates FPGA integration. This work therefore investigates which aspects of big data frameworks are important when considering FPGA acceleration. In addition, the impact these aspects have on the eventual performance of the system is investigated.

The project is carried out as part of an internship at Teratide. Teratide is a high-tech spin-off from Delft University of Technology, specialized in the acceleration of high performance analytics application in the context of big data. Their vision aligns with that of the Accelerated Big Data Systems (ABS) group at Delft University of Technology. Both these organisations envision a future big data system that is able to offload computations to different accelerators in a heterogeneous cluster [5].

Nowadays, it is virtually impossible to imagine any field of industry that does not makes use of some form of data analytics. The impact and possibilities of efficient and transparent integration of FPGAs in big data frameworks are therefore limitless. One could distinguish three areas of applications that could substantially benefit from FPGA accelerated big data frameworks; applications with long-running queries, latency-sensitive applications, and applications that aim to achieve a high energy efficiency.

Business intelligence (BI) applications are an example of applications with long-running queries. BI applications aim to support corporate decision making by providing insights into the vast amount of data a company might have acquired. One way of providing these insights is by running SQL workloads, from which the results can be presented on a dashboard or in a report. The runtime of these queries grows with the amount of data on which they operate. Tech companies generally aim for short development cycles, but the length of these cycles is partly governed by the runtime of their BI queries [6]. In these cases, accelerated big data frameworks could allow companies to achieve shorter development cycles.

Second, when considering latency-sensitive applications, one example is a fraud detection system as used by payment processors such as Adyen [7]. These systems are responsible for processing large amounts of payment data in order to filter out fraudulent transactions. These computations are latency-sensitive because in most cases the payment processor has a service-level agreement (SLA) with their customers that obliges them to adhere to certain latency constraints. Effectively, these latency constraints provide an upper bound for the runtime of the processing algorithm. With FPGA acceleration, these systems could therefore perform more compute intensive operations while the original latency constraints are adhered to. These more complex algorithms could result in better fraud detection rates.

Finally, another identified area of impact is the power consumption of datacenters. FPGA accelerators are known for their high energy efficiency, which results from their low operating frequencies as compared to CPUs. In 2018, the world's datacenters consumed an estimated 200 terrawatt hours, which is roughly 70% more than the yearly Dutch energy consumption [8, 9]. FPGA accelerated and energy efficient big data frameworks could reduce this power consumption and hence contribute to the realization of a 'green datacenter'.

1.2. Challenges

Two of the main challenges in integrating FPGA accelerators with big data frameworks are transparency and efficiency.

The acceleration should be transparent for the end user. This means that the user should not be aware of the FPGA acceleration and does not have to tune certain parameters in the framework. This is important because transparent integration lowers the barrier to adopt these technologies. To achieve transparent accel-

eration, the system should autonomously identify where and when certain parts of the computation can be accelerated.

In terms of efficiency, it is important that the acceleration does not introduce additional overheads and/or costs that can lower the overall efficiency of the system.

Sharing data between processes, in this case between the big data framework and the software that interfaces with the FPGA accelerator, typically introduces serialization overheads. Different programming languages have their own definition of how a data structure is laid out in memory. Serialization and deserialization can be used to bridge this gap, but this is paired with significant overheads. An upcoming technology that overcomes this problem is Apache Arrow. Apache Arrow is an in-memory data specification that is language-independent [10]. By making use of Apache Arrow, data can be shared between the big data framework and the software interfacing with the accelerator without introducing serialization overheads. In addition, this memory format is hardware-friendly. This enables data to be copied between the accelerator and the host machine more efficiently.

In terms of cost efficiency, there are two factors that play a role in the context of FPGA accelerators. First, there is the initial cost of developing an FPGA kernel. This is generally more time consuming than software development for CPUs and GPUs, and requires in-depth knowledge about circuit design. Frameworks such as Fletcher exist to reduce this development time and there even exist high-level synthesis (HLS) tools that can synthesize FPGA kernels from C code [11–13]. However, these HLS tools are unable to generate optimized code and thus force the developer to manually optimize the generated code. This optimization in turn requires in-depth knowledge. The amount of effort and specialized knowledge required to develop these kernels is expected to reduce in the future, as frameworks such as Fletcher and HLS tools become more sophisticated.

Lastly, these FPGA accelerators are expensive. Fortunately, the industry standard is to run big data applications in a cloud environment so there is no need for end-users to buy any specialized hardware. Still, cloud instances that have these FPGA accelerators are significantly more expensive than regular compute instances. It is a matter of time before this technology is more widely adopted and these instances become cheaper. This can already be seen at services such as Microsoft Azure, that recently added the NP-instance family which can be rented for a fraction of the regular price using spot pricing [14].

With the emergence of Apache Arrow, frameworks such as Fletcher, and more options for affordable FPGA instances in the cloud, this is the perfect moment to further investigate the integration of FPGAs in big data and to lay the foundation for future research.

1.3. Problem statement and research questions

This work investigates the integration of FPGA accelerators and big data frameworks in the context of SQL workloads. The aims is to provide recommendations for the design of future 'accelerator ready' frameworks. The problem statement is formulated as follows:

How should future big data frameworks be designed such that they facilitate transparent and efficient integration with FPGA accelerators in the context of SQL workloads?

To solve this problem, different research questions are posed:

1. Where in the system should FPGA accelerators be placed in order to accelerate SQL workloads?

2. Which features of big data frameworks are required to facilitate efficient integration and what is their impact on the overall performance of the system?

3. Which parts of the big data framework need to be aware of the acceleration such that the system can be transparently deployed in a heterogeneous setting?

4. Which part of the workload should be accelerated such that the average speedup within the application domain is maximized?

1.4. Contribution

In this work, several software implementations are developed with the goal of answering the research questions as stated above. These development efforts align with the vision of the ABS group and Teratide. The FPGA accelerator, which could be deployed in such a future heterogeneous big data system, is the main object of study. Different FPGA kernels have been developed by MSc and PhD students of the ABS group. These kernels implement file readers, decompression algorithms, and a number of SQL operators [15]. Additionally, previous research has been done regarding the integration of these kernels in Apache Spark, serving as a proof of concept (PoC) for the future system as described above [16].

This work contributes three additional integrations with different big data frameworks. As seen in Figure 1.2, different frameworks have different specializations and it is likely that new challenges arise when integrating FPGA accelerators. The FPGA kernel that is integrated in this work is developed by Teratide and implements a regular expression matcher. First, this kernel is integrated in Dremio, an upcoming big data framework, for a SQL workload featuring a regular expression based filter operator. Second, the same kernel is integrated in Dask, another popular big data framework. Finally, the kernel is integrated with Dask distributed, a separate version of the Dask framework that is designed to run in a cluster setting.

1.5. Thesis outline

Chapter 2 provides all background information necessary to understand the choices that are made in the integration of FPGA accelerators in big data frameworks. In Chapter 3, an overview of the alternative solutions related to three parts of the project is presented. First, the different ways to speed up big data computations are considered. Second, multiple big data frameworks are examined for integration with FPGA accelerators. And lastly, the ways in which these accelerated solutions can be deployed are considered. This is followed by three chapters presenting the implementation, experimental setup, and results of the three frameworks in which FPGA accelerators are integrated. Chapter 4 shows the integration with Dremio, Chapter 5 shows the integration with Dask, and Chapter 6 shows the integration with Dask distributed. Finally, Chapter 7 concludes the project and provides recommendations for future research.

2

Background

This work builds on the work of F. Nonnenmacher, in which transparent FPGA acceleration is implemented in Apache Spark SQL [16]. Nonnenmacher's work introduces multiple software components in order to support the acceleration. At the time of his work, there was no native support for Apache Arrow based recordbatches in Apache Spark. Therefore, a custom implementation of the datareader had to be introduced in order to read parquet files into an Apache Arrow format. Additionally, his work implements the Gandiva execution engine to support SIMD execution on Apache Arrow based recordbatches. Finally, his work integrates a Fletcher based kernel in Apache Spark SQL in order to offload the evaluation of SQL operators to an FPGA accelerator.

Technologies used in the computer engineering field tend to change fast, and indeed a lot of the technologies used in Nonnenmacher's work have undergone changes. There is a trend in big data frameworks to provide native support for columnar and Apache Arrow based in-memory formats. Furthermore, the Gandiva execution engine has become more established within the Apache Arrow project. These recent developments potentially eliminate some of the hurdles encountered in the work of Nonnenmacher. At the same time, the integration of FPGA accelerators in different big data frameworks could potentially give rise to new challenges as seen in Section 1.4.

This chapter provides background information about all involved technologies in order to understand these new challenges that may arise. Section 2.1 provides a high level overview of the general data engineering workflow and the trends in big data. Section 2.2 explains how modern big data frameworks are organized. Section 2.3 presents background information about the Apache Arrow in-memory format, while Section 2.4 provides details on the use of FPGA accelerators in a big data context. Finally, Section 2.5 presents the related work.

2.1. Big data

This section aims to provide a high level overview of how data is typically used in industry and how this is influenced by the trends in big data. This information provides the context for the subsequent background sections.

2.1.1. Data engineering workflow

In order to get a better understanding of the context in which big data frameworks are used, a closer look is taken at the general data engineering workflow. As mentioned in Section 1.1, it is hard to find a field of industry that doesn't use some form of data analytics. Larger companies often have complicated extract, transform, and load (ETL) pipelines. These ETL pipelines can consist of multiple big data frameworks as seen in Figure 1.2. Companies typically direct their engineering effort towards further automating these ETL processes [17–19].

This is however often the situation at larger companies. In general, data engineering encompasses much more than automating ETL pipelines. Mason provides a taxonomy of data science, which describes the different responsibilities of a data engineer [20]. In short, these responsibilities are *obtaining* data, *cleaning* data, *exploring* data, *modeling* data, and *interpreting* data.

A similar decomposition of these responsibilities in the context of artificial intelligence can be found in Rogati's AI hierarchy of needs. This hierarchy is shown in Figure 2.1.

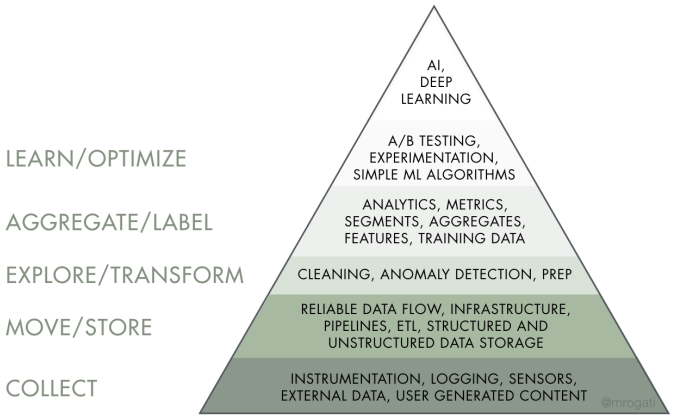


Figure 2.1: The AI hierarchy of needs [21].

This hierarchy displays the different needs of a data-oriented company. In this hierarchy, younger companies typically focus more on the bottom of the pyramid, while more mature companies move towards higher segments.

This aligns with the observation that the needs of data-oriented companies change over time. The process of moving up in this pyramid is described as an iterative process, indicating that data engineers often revisit lower segments of the pyramid after working on a higher segment.

A natural workflow for such an iterative process is to start with processing smaller datasets and gradually move up to larger ones. In early stages of such a project, the computational requirements are typically satisfied by working on a desktop computer or on a powerful laptop. When the computational needs increase, data engineers can move to server environments that run a Jupyter notebook server. There are numerous commercial companies that offer these services such as Google Colaboratory and Deepnote [22, 23]. Apart from normal server nodes running Jupyter kernels, these companies also offer more powerful and GPU accelerated nodes for users that require more compute. Finally, data scientists that need to scale up even further typically move to cluster environments, in which a big data framework can be configured to run in a distributed setting.

2.1.2. Trends in big data computing

The trends in computing systems are considered as to obtain a better understanding of computing in the context of big data. A figure that predicted these trends in computing systems in 2005 by Hofstee can be seen in Figure 2.2.

At that time, single tread performance was stagnating for three reasons [24]. First, Dennard scaling broke down. Originally published in 1974, Dennard introduced a relationship between the size of MOSFET transistors and their power density [25]. Dennard scaling, which is derived from this observation, states that the power density of transistors remains constant as they become smaller. This means that the current through and the voltage over these transistors needs to scale down in accordance with the size of the transistor. Around 2005, this scaling principle broke down as the static power consumption became dominant over the dynamic power consumption of a transistor. Further scaling down the operating voltage increases this static power consumption. This effect is known as the *power wall* and imposes limits on the clock-frequency of CPUs.

Second, instruction-level parallelism (ILP) was deployed in order to achieve small grained parallelism amongst instructions in CPU pipelines. There is however a limited amount of parallelism that can be reached on this granularity for a single core. This effect is known as the *ILP wall*.

Finally, the performance of main memory was not increasing as fast as the performance of the CPU core. This is known as the *memory wall*.

At the same time, Moore's law continued to hold true with the number of transistors on a microchip doubling roughly every 24 months. These performance walls that plagued single core CPUs and the availability

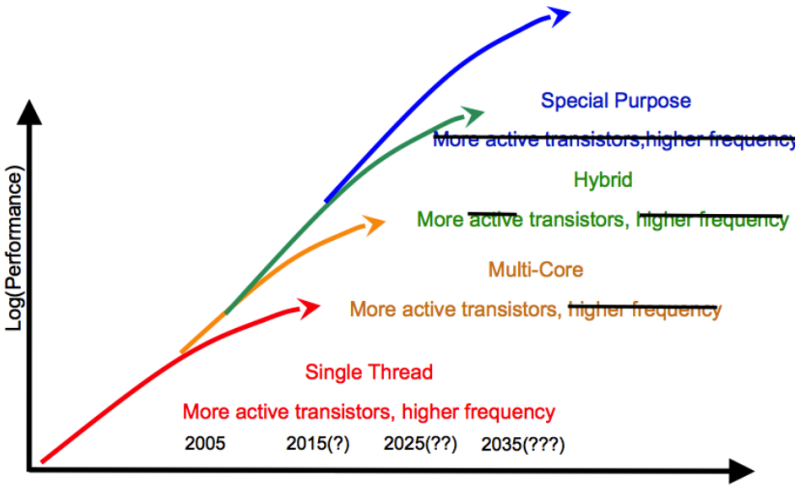


Figure 2.2: Microprocessor trends as predicted in 2005 [24].

of more transistors drove the industry into multi-core CPUs.

This multi-core trend was expected to stagnate at around 2020 because of the power limitations related to more active transistors. A solution was predicted in which hybrid or heterogeneous cores are leveraged. Instead of all transistors being active, they are configured in such a way that they form a set of heterogeneous and specialized compute units. Even this trend is expected to end because of Moore's law breaking down, which imposes limits on the complexity of these specialized compute units on a single microchip. Without more available transistors per chip, specialized and self-contained compute units will need to be realized..

More recently, the lacking performance increase of main memory as identified in the memory wall becomes even more prominent when considering the performance of network and storage. Figure 2.3 shows that the bandwidth of network and storage increases much faster that the bandwidth of main memory. Note the logarithmic y-axis.

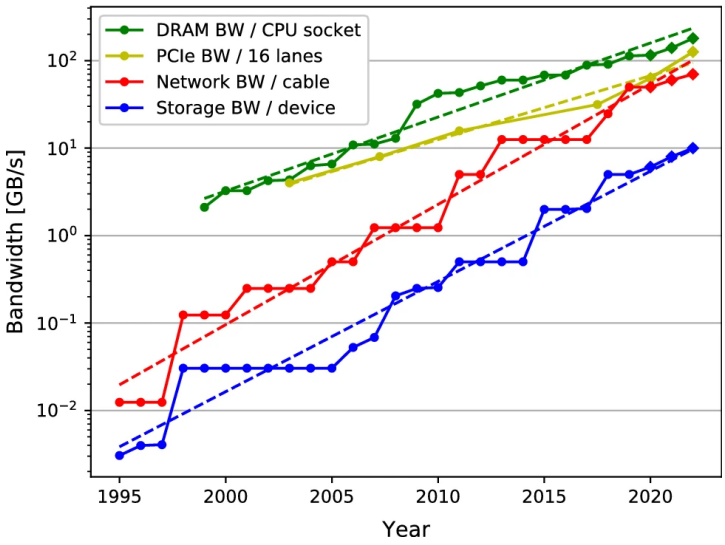


Figure 2.3: Bandwidth trends for DRAM, PCIe, Network, and Storage. DRAM bandwidth is a proxy for CPU throughput [26, 27].

In this figure, the DRAM bandwidth per CPU socket is interpreted as a proxy for CPU throughput [27]. Extrapolating this graph shows that the CPU will form the bottleneck for storage performance in the near future.

All these trends encourage the move towards distributed and heterogeneous compute units.

2.2. Big data architecures

This section takes a closer look at the architecture of the aforementioned big data frameworks. A closer look is taken at the technologies which enable these frameworks to function in order to get a better understanding of why these frameworks are designed in the way that they are.

2.2.1. Distributed file systems

Distributed file systems lie at the foundation of big data frameworks. Popular examples include the Hadoop Distributed File System (HDFS), Google File System (GFS), and cloud blob storage such as Amazon's S3 or Azure Blob Store. The workings of these distributed file systems are explained by exploring the mechanics behind HDFS.

HDFS is the file storage solution of the Hadoop project [28]. HDFS is used in big data settings and is designed to be highly fault-tolerant. This fault-tolerance is very important in big data settings, as the probability of machines failing in large-scale clusters is significant. The focus of HDFS is on batch-processing, meaning that the system is optimized for high throughput data reads instead of low latency [29].

The simplest setup of an HDFS cluster consists of two types of nodes; the *NameNode* and the *DataNode* [30]. The NameNode stores all metadata associated to the files stored on the HDFS cluster. This metadata includes parameters such as the block-size and replication factor, as well as where in the cluster the data of a specific file is stored. The DataNodes handle the actual storage of the files. The NameNode receives regular heartbeats from the DataNodes such that it is able to identify when a node has failed.

As mentioned before, each file is configured with a block-size and a replication factor. Since the size of the files stored on an HDFS cluster is typically in the order of gigabytes or terabytes, a single file can easily exceed the storage capacity of a single DataNode. In order to store the file, it is broken down into multiple blocks, from which the size is configured with the block-size parameter. To enable fault-tolerance, each block is replicated on the cluster. A typical replication factor is equal to three, in which a single block is stored on three *DataNodes*; two blocks are stored on nodes in the same physical rack, while a third block is stored on a node in a different rack. This protects the block in case an entire rack fails.

2.2.2. Compute frameworks

When considering different compute frameworks for big data, one can distinguish three different classes of frameworks; batch-processing frameworks, streaming frameworks, and index-based frameworks. This section examines the workings of batch-processing frameworks, which are built on top of distributed file systems in order to allow the data to be processed. Background information on streaming and index-based big data frameworks is provided in Section 3.3, in which the alternative solution classes are analyzed.

Batch-processing frameworks process the underlying dataset in multiple batches. Aggregating the results of these individual batches yields the result for the entire dataset. Hadoop MapReduce and the more modern Apache Spark are explored in order to gain a better understanding of how these and other batch-processing big data frameworks function.

Hadoop MapReduce Hadoop MapReduce is the most popular implementation of the MapReduce framework, originally proposed by Google [31]. In this framework, users can define *map* and *reduce* functions, which operate on key-value pairs [32]. The map functions produce new key-value pairs from the input, which can be aggregated by use of reduce functions [33]. An example of such an application can be seen in Figure 2.4.

These functions are executed on the DataNodes of the HDFS cluster. This practice brings the computations close to the data, instead of gathering the data near a single node that performs the computations. This reduces the need to transfer the data blocks over the network (known as shuffling). The map functions operate on data on the local node and produce new data on the same local node, this does therefore not require a data shuffle. Reduce functions do require a shuffle, since an aggregation function operates on multiple values with the same key, which can reside on multiple DataNodes.

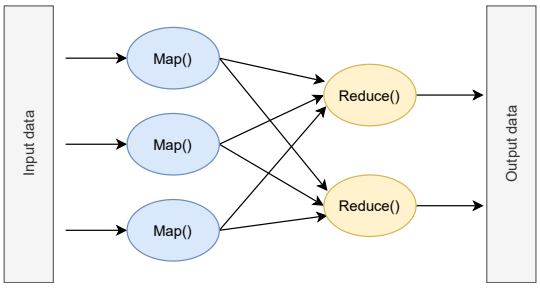


Figure 2.4: Example of an Hadoop MapReduce application with a map function being executed on three DataNodes and a reduce function being executed on two DataNodes. The arrows between the map and reduce functions represent a data shuffle [34].

Apache Spark Apache Spark is a more modern framework as compared to Hadoop MapReduce. One of the most pronounced differences between Apache Spark and Hadoop MapReduce is that Apache Spark enables computations to be performed in-memory. Hadoop MapReduce, on the contrary, stores all intermediate results to disk. This allows Apache Spark to have a much higher throughput. In the case where the dataset on which to operate exceeds the size of memory available on a machine, this advantage diminishes [35].

While the Hadoop ecosystem is designed to be used on low-cost hardware, Apache Spark typically runs on higher-end systems due to its memory requirements.

Apart from this difference in memory usage, Apache Spark provides a higher level API as compared to the relatively simple MapReduce framework. This higher level language enables developers to express complex operations more clearly. This Spark API can be extended through libraries related to machine learning and graph processing, increasing the range of usecases for which the framework can be used.

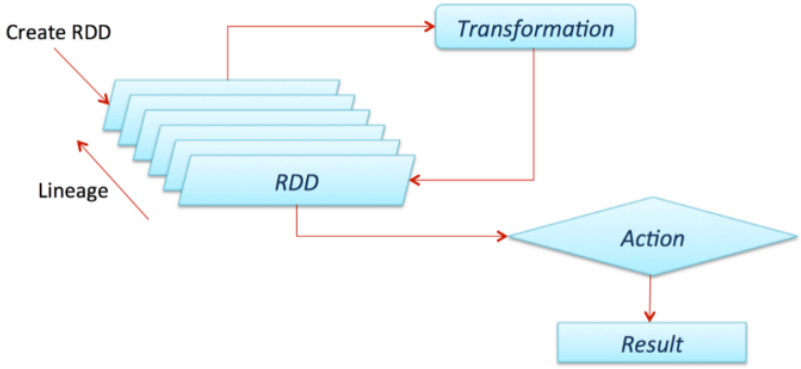


Figure 2.5: Resilient distributed datasets in Apache Spark [36].

Although Spark can operate on data stored in HDFS, Spark is not limited to this file system. Spark makes use of an abstraction called a *resilient distributed dataset* (RDD). This resembles an immutable collection of data stored on multiple nodes in a cluster. The fact that this datastructure is immutable means that performing a transformation on an RDD creates a new one. In this way, all previous versions from which the new RDD is derived are preserved in what is called the *lineage* of the RDD. RDDs in Apache Spark are lazily evaluated, meaning the computation is only performed once the result is needed. Lazy evaluation allows for optimizations at runtime by removing the immediate computation of results.

Operations that trigger the evaluation of an RDD are known as *actions* in Apache Spark. All operations can be divided in two categories; operations with narrow dependencies and operations with wide dependencies. Operations with a narrow dependency do not require a data shuffle, while operations with a wide dependency do. An overview of performing transformations and actions on an RDD is shown in Figure 2.5.

These RDD operations are being tracked in a *directed acyclic graph* (DAG). As the name suggests, edges in this graph have a direction associated with them and the graph does not contain any cycles. An example of what such a DAG looks like can be seen in Figure 2.6. This DAG, together with the initial dataset, can be used to reconstruct intermediate or final results in case of a node failure.

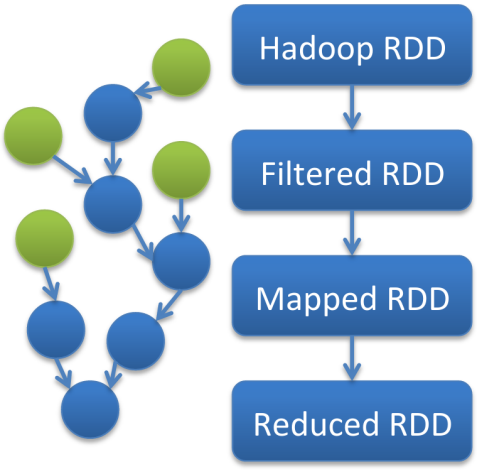


Figure 2.6: Apache Spark's directed acyclic graph [34].

A more complete analysis of these and other batch-processing big data frameworks is performed in Section 3.3.

2.2.3. SQL workloads

In the previous section, it is shown how big data frameworks use different programming paradigms such as MapReduce or higher level APIs to perform operations on large datasets. On a smaller scale, such as *relational database management systems* (RDBMS), the domain-specific SQL language is often used to query datasets. Certain big data frameworks also support the SQL syntax to allow for a more expressive API. Examples are Apache Spark's SQL module and Apache Hive, which provides a SQL-like layer on top of the MapReduce framework [37–39]. Apart from simply parsing the SQL query, these modules perform extra optimization and planning steps in order to efficiently perform the computation. The specific implementation is different for each big data framework, but the paragraphs below provide a general overview of the steps involved.

SQL parsing The first step in the execution of a SQL query on a big data framework is to parse the SQL query. Queries are parsed into a format that can be understood by the framework. This output format is typically a tree of logical operators. In the case of Apache Spark SQL, which uses the internal Catalyst SQL optimizer, the query is parsed into Catalyst expressions [37].

Logical optimization The logical operators as produced by the SQL parser are not optimized for execution in a big data setting. Rather, their ordering depends on the way the user of the application has written the SQL query. There are numerous opportunities for optimization. An example of such an unoptimized logical expression tree is shown in Figure 2.7a. In this case, the expression tree is evaluated in the context of Apache Arrow's Catalyst optimizer, but the principles remain the same for any framework that performs optimizations on SQL queries [37]. In Catalyst, which is written in scala, this expression tree can be represented as shown in Listing 2.1.

```
0 Add(Attribute(x), Add(Literal(1), Literal(2)))
```

Listing 2.1: A simple Catalyst expression tree [37].

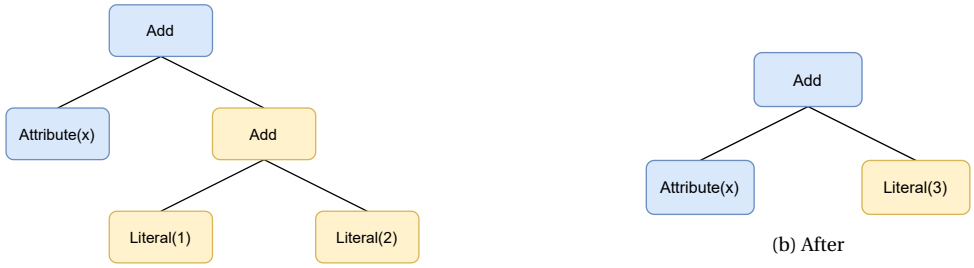
It becomes clear that this expression tree adds a value of $(1+2)$ to an attribute x. since the value of the literals does not depend on the value of attribute x, a more efficient way to produce the same result would be to add the number 3 to attribute x. Catalyst makes use of optimization rules such as the one shown in Listing 2.2.

This rule is passed as an argument to the *transform* method of the aforementioned expression tree and replaces any sum of two literals by a single literal with the summed value. The result of this transformation can be seen in Figure 2.7b.

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

Listing 2.2: A Catalyst optimization rule [37].

More complex optimizations can be made as well. It is seen that some operations in big data frameworks trigger a shuffle, forcing data to be sent over the network between nodes. Since this shuffling of data introduces a significant overhead, it is beneficial to keep the amount of data to be shuffled to a minimum. It is therefore common practice to push down filter operators in order to reduce the amount of data in downstream operators.



(a) Before

(b) After

Figure 2.7: Example of a logical optimization rule in Catalyst [37].

These and other optimization are performed in the *logical optimization* phase. Most modern optimizers, such as Apache Calicte, Apache Spark's Catalyst optimizer, and Orca are based on the Cascades framework [40–42]. These optimizers rewrite query plans during an exploration phase using equivalence rules. Equivalence rules include, but are not limited to, commutativity, associativity, physical implementation rewriting, magic set rewriting, homomorphism, indices, and super operators [41, 43–46].

Physical planning The logical plan that results from the previous phase is merely a tree of logical operations that form the correct result when evaluated. However, these logical operators cannot be evaluated as is, because there is no notion of the actual implementation of the operator. A logical operation can have multiple physical implementations that all lead to the correct result. The *join* operator, for example, can be implemented with a *hash join* as well as a *sort merge join* implementation [47]. The physical planning phase can therefore produce multiple semantically equivalent physical plans.

This physical planning phase can additionally perform rule-based optimizations, similar to the logical planning phase. The difference is that, at this point in the optimization process, the optimizer is aware of the underlying systems on which the plan is to be executed. With certain external storage systems or local file reader implementations, it is possible to push down filter operators into the data reader. These are optimizations that cannot be made without knowing whether the underlying systems supports this.

Optimal plan selection The previous planning phase outputs a collection of plans, as a single logical operator can have multiple physical implementations. Whether this collection exists of a single plan or multiple plans is determined by the operator types in the underlying logical plan and the number of physical implementations that the optimizer considers.

In case there are multiple plans in the collection, a decision has to be made as to which plan gets selected for execution. The optimal plan is selected based on a cost model, which is generally expressed in time units and provides an indication of the total runtime of the physical plan [48, 49]. These cost models might incorporate read and write times, data shuffles, operator processing times, and statistical models of the database.

These cost models are not limited to time units, but can also incorporate other indicators for the effectiveness of the physical plan. One such indicator that is gaining interest is a measure for the energy efficiency of the resulting evaluation of the physical plan [48].

Code generation The selected physical plan can be directly executed in a suitable execution engine. However, depending on the environment on which the big data framework is executed, an additional optimization in the form of code generation can be performed.

The Apache Spark SQL module, which is executed on the Java virtual machine (JVM), generates optimized Java bytecode using quasiquotes [37, 50]. Dremio, a big data framework that is executed on the JVM as well, makes use the low level virtual machine (LLVM) compiler to provide just-in-time (JIT) compilation into highly optimized bytecode that can be executed on the Gandiva execution engine [51]. This Gandiva engine is explained in more detail in the next section.

2.2.4. Cluster setups

When deploying big data frameworks in a cluster environment, two different types of nodes can be distinguished; the *scheduler* node and the *worker* node.

First, the *scheduler* node is used to perform all planning algorithms. In the case of SQL queries, the scheduler node performs all steps as described above. The scheduler node then sends the resulting execution plan to multiple worker nodes.

The worker nodes are used to evaluate the operations in this execution plan. Typically, the worker nodes are implemented on the same physical machines as the DataNodes in a distributed file system. This allows the worker nodes to perform operations with narrow dependencies on local data. The final result, as aggregated on the worker nodes, is then returned to the scheduler node.

2.3. Apache Arrow

The aforementioned Gandiva execution engine does not run on the JVM, but it is instead implemented in C++. In general, there are numerous situations in which a big data application implemented in one language is integrates with an application implemented in a different language. Consider the small fraction of Matt Turck's big data landscape as shown in Figure 1.2, which lists multiple applications that are typically linked together in a pipeline fashion.

The challenge in working with applications implemented in different languages is that each language has its own definition of how a datastructure is laid out in memory. Sharing data between such applications would therefore require serialization of the object into a bytestream, after which the receiving application reconstructs the object using a process called deserialization. This introduces data copies. An example is shown in Figure 2.8. Serialization and deserialization introduces an overhead associated with the data transfers between different applications.

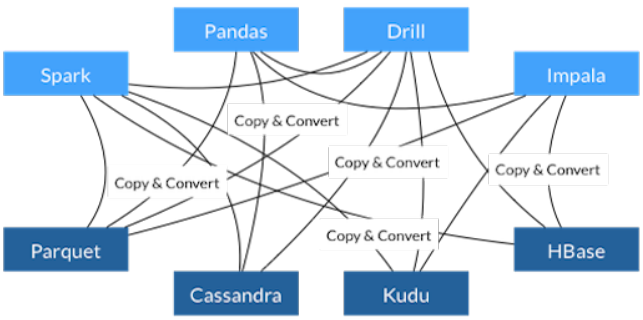


Figure 2.8: Communication between different applications without Apache Arrow [52].

Apache Arrow is a project announced by The Apache Software Foundation in 2016 to address this problem [53]. Apache Arrow is a language-independent in-memory columnar data specification. Because of the language independence, the need for serialization and deserialization is eliminated as shown in Figure 2.9. A column-oriented format is chosen over a row-oriented format because it offers greater performance in big data settings. This columnar format is explored further in the next subsection.

Apart from solving serialization overheads when working with applications in different languages, Apache Arrow is also a hardware-friendly format. When implementing hardware accelerators such as GPUs and FPGAs,

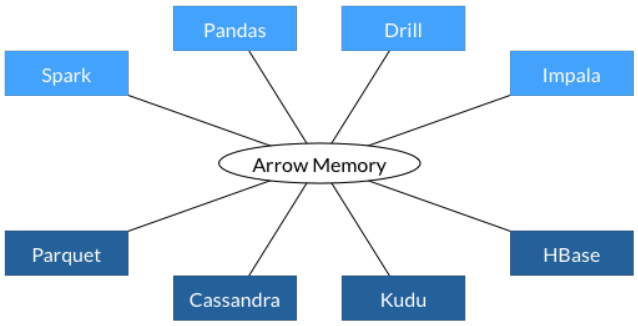


Figure 2.9: Communication between different applications with Apache Arrow [52].

data needs to be copied to and from the accelerator. Apache Arrow enables these copies to be as efficient as possible by minimizing the need for bytes that hold metadata about the datastructure. This metadata is not of interest to the accelerator [54].

Apache Arrow is a rapidly evolving project and as such its version has been increased from 1.0.1 to 4.0.1 during the span of this project. All implementations are based on Apache Arrow version 3.0.0, since this is the most recent release that was available during the development phase of the project. However, all principles used apply to Apache Arrow version 4.0.1 as well.

2.3.1. Columnar data formats

The Apache Arrow specification is used to represent structured and table-like datasets. These datasets consist of fields that can be identified by their associated row and column. Such data can be laid out in memory in two distinct ways; consecutive memory addresses can hold data grouped per row, or the data can be grouped per column. This is shown in Figure 2.10.

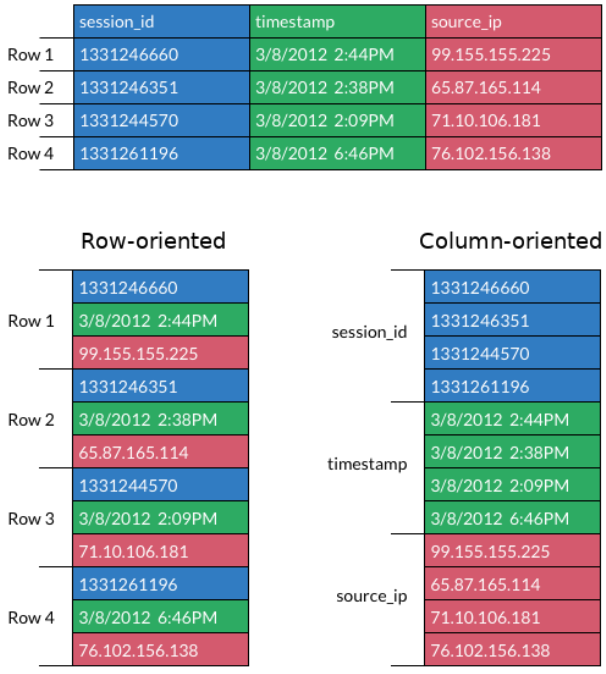


Figure 2.10: Row-oriented versus column-oriented in-memory data formats [52].

The row-oriented format performs well when entire rows of the dataset have to be materialized. Since all

fields in the row are required for this operation, a block of memory can simply be read sequentially in order to obtain the required data. However, typical modern big data applications don't often materialize entire rows from the dataset. Instead they often operate on a small subset of columns. Examples are filter or aggregation operators that target a single field. In this case, there is no need to read all other fields of the row, as they are not used by the operation. If the data were laid out in-memory in a column-oriented format, blocks of memory could still be read sequentially to produce the relevant fields for the operation.

Another advocate for columnar data formats is the additional benefit it has when considering vectorized computing. Vectorized computing can be classified as *single instruction, multiple data* (SIMD), indicating that the same instruction is executed in parallel on a collection of data. This can be achieved by making use of GPU's, which incorporate many stream processors in parallel, or by using Intel's intrinsic AVX instructions, which perform the same operation on an array of data [55, 56]. This programming model fits well with columnar data formats, as entire consecutive sections of memory can be read as the operand of the SIMD instruction.

Lastly, the columnar format contributes to the aforementioned hardware-friendliness of Apache Arrow. Sending buffers with values contained in the same column to an FPGA accelerator eliminates the need for complex decoding algorithms on the FPGA side. Instead, these values can be directly streamed into the functional units as implemented on the accelerator [5].

2.3.2. Specification

As mentioned, the Apache Arrow specification defines a column-oriented in-memory format for table-like datasets. These datasets come with metadata known as a *schema*, which defines the structure of the dataset and the datatypes of the columns. When such a dataset is processed in a batch-processing framework, this is not done at once. Rather, it is split up into multiple *recordbatches*, which are processed individually. A recordbatch holds all fields related to a subset of the dataset. Each column in the recordbatch abstraction stores its data in an Arrow array, which in turn consists of multiple Arrow buffers [57]. These Arrow arrays and Arrow buffers are explained in more detail below.

An Arrow array can store both primitive types as well as more complex nested types. The Arrow array holds different Arrow buffers based on the type of data stored in the array and the associated metadata as specified by the schema. The Arrow buffers are stored sequentially in a memory region. Apache Arrow specifies three types of Arrow buffers; the *validity* buffer, the *offset* buffer, and the *value* buffer.

The validity buffer holds a sequence of i bits to indicate whether the ith record in the recordbatch holds a valid value or whether it holds the *null* value. This validity buffer is only used for fields that are *nullable* as specified by the associated metadata. The offset buffer is used for fields with variable length, such as strings. The ith element in the offset buffer points to the element in the value buffer at which the ith value is located. The length of a value can therefore be obtained by computing the distance between its own offset and the offset of the next element in the offset buffer. Finally, the value buffer holds the actual fixed length values of the column in the recordbatch. An example of the Arrow buffers for a simple recordbatch can be seen in Figure 2.11.
The Apache Arrow specification dictates that all Arrow buffers should be aligned to 64 bytes. This means that the starting address of the Arrow buffers should be a multiple of 64. However, not all language specific Arrow libraries adhere to this part of the specification. The Java Arrow library, for instance, aligns its Arrow buffers to 8 bytes instead of 64.

2.3.3. Gandiva

Gandiva is originally developed as an open-source project by Dremio, an upcoming big data framework, and is donated to the Apache Arrow project in 2018 [58]. Gandiva is designed to efficiently perform operations on Arrow recordbatches. Gandiva consists of two major components; the Gandiva *compiler*, and the Gandiva *execution kernel*. These two components are explained in more detail below. An overview of the system can be seen in Figure 2.12.
The Gandiva compiler is based on the LLVM and is able to JIT compile an expression tree of operations to be performed on an Arrow recordbatch into highly optimized native code [51]. This native code is then evaluated on the Gandiva execution kernel, which in turn consumes and produces Arrow recordbatches. This kernel leverages optimizations such as Intel's AVX based vectorization when the execution environment supports

(a) Schema:

Field A: Float (nullable) Field B: List(Char) Field C: Struct(E: Int16, F: Double)

(b) RecordBatch:

A	B	C
0.5f	"fpga"	(42, 0.125)
0.25f	"fun"	(1337, 0.0)
∅	"!"	(13, 2.7)

(c) Arrow buffers:

Index	Buffers for:					
	Field A		Field B		Field C	
	Validity (bit)	Values (float)	Offsets (int32)	Values (char)	Values E (int16)	Values F (double)
0	1	0.5f	0	f	42	0.125
1	1	0.25f	4	p	1337	0.0
2	0	×	7	g	13	2.7
3			8	a		
4				f		
5				u		
6				n		
7				!		

Figure 2.11: An example of the Apache Arrow in-memory specification [57].

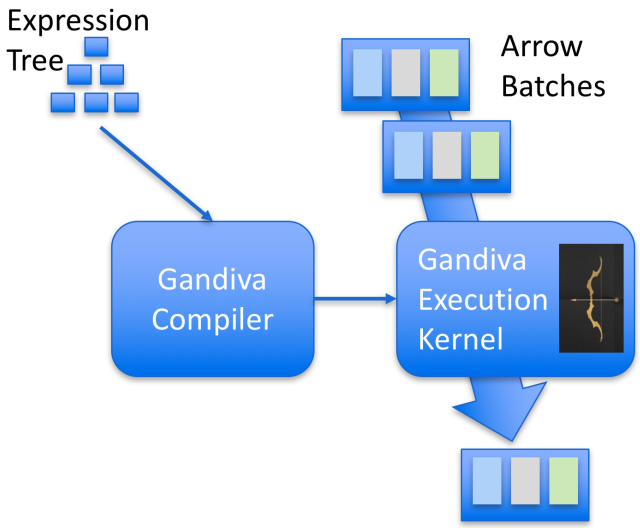


Figure 2.12: Overview of the Gandiva system [51].

this [56]. Another optimization is to use the Arrow validity buffers to determine whether the outcome of an operation results in a null value, resulting in better branch prediction.

2.4. FPGA accelerators

Apart from serialization posing limits on the computational abilities of big data frameworks, Oosterhout et al. found that the *BDBench* and *TPC-DS* benchmarks are CPU bound in Apache Spark [59]. This is contrary to the previous belief that the performance of these industry standard benchmarks is limited by network bandwidth and I/O.

Algorithms that are compute bound can be accelerated by the use of hardware accelerators. The GPU is one such hardware accelerator that can be used for this purpose. However, the programming model of the GPU does not fit all problems that arise in big data applications. Another option is to use FPGA accelerators.

Layered run-time systems, hardware-unfriendly data structures, and serialization overheads are identified as three challenges related to big data systems [57]. The Apache Arrow in-memory format is able to overcome the latter two challenges. The first can potentially be overcome by making use of FPGA accelerators. These accelerators have a greater computational performance as compared to layered run-time systems executing on a CPU [57].

This section briefly explains how an FPGA accelerator works and how it can be used in the context of big

data.

2.4.1. The field-programmable gate array

The field-programmable gate array, or FPGA, is an IC from which the internal logic design can be configured after manufacturing. This allows the programmer to implement different IC designs without having to go through the manufacturing process, which is expensive and time consuming. This reconfiguration of the FPGA is done using a hardware description language, such as *VHDL* or *Verilog*. Data on the FPGA is processed in a dataflow manner.

It is found that developing FPGA kernels is a time-consuming process. Additionally, in-depth knowledge about circuit design in required. Apart from the significant development time associated with the design and implementation of the kernel, an interface between the kernel and the incoming and outgoing data needs to be established.

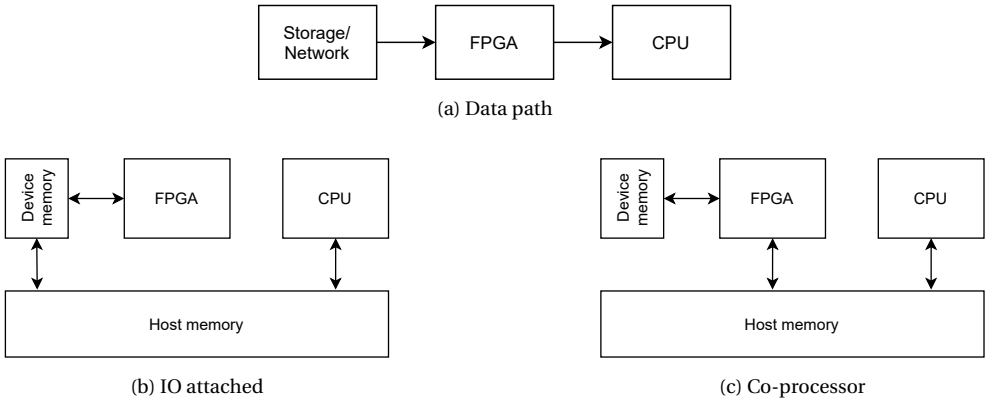
The Fletcher toolchain can be used to automatically generate this interface in the context of Apache Arrow recordbatches [11–13]. This interface is constructed based on the metadata of the underlying recordbatch. Fletcher thereby greatly reduces the development effort required to create an FPGA kernel.

Furthermore, the reuse of existing FPGA kernels is low. It is found that connecting existing hardware components is a labor intensive process. This can be attributed to the low abstraction level of the connections between the components. These connections need to be changed by the hardware developer based on the context in which the hardware component is to be deployed. Tydi provides a high level abstraction for these connections and thereby facilitates kernel reuse [60]. This is done by defining existing hardware components as streamlets, which transform streams of data [57].

Instead of developing a custom kernel, high-level synthesis (HLS) tools could be used instead. These tools typically compile compiled languages into a register transfer level (RTL) hardware description. The resulting design, albeit functional, is not optimized for execution. These designs can be manually optimized, but this process is again time consuming and requires in-depth knowledge about circuit design [47]. These HLS tools do not support the compilation of managed languages [61]. Papadimitriou et al. therefore proposes an extension of TornadoVM that is able to JIT compile Java code to an FPGA bitstream.

2.4.2. FPGAs in big data

When considering ways as to how FPGA accelerators can be integrated into big data systems, one can distinguish three configurations of the FPGA in the computing system. The accelerator can either be placed in the data path between network or storage and the CPU, as seen in Figure 2.13a, or it can be integrated as an additional processor [47, 62, 63]. In this latter case, a further distinction can be made between an IO-attached accelerator, where the FPGA has its own memory space, and a co-processor, in which the FPGA and the CPU communicate through shared memory [47, 64]. These configurations are shown in Figure 2.13b and Figure 2.13c respectively.



(a) Data path

(b) IO attached

(c) Co-processor

Figure 2.13: Different architectural configurations for FPGA accelerators [47].

SQL workloads It is identified in a survey that multi-pass database queries are able to benefit most from FPGA accelerators, because of the high computational intensity of the resulting operators [47]. Addition-

ally, when comparing FPGAs with GPUs, latency-sensitive streaming applications are better suited for FPGA acceleration because of the batch-processing nature of GPUs.

This survey also stresses the importance of good query optimization when considering new accelerators. Query optimizers have been studied extensively in the context of CPU architectures, but the optimization becomes more challenging when FPGA accelerators are considered. In this new situation, the optimizer should take FPGA reconfiguration times into account as well as whether the query is long- or short-running. Not taking these aspects into account when optimizing the query could potentially prove wasteful, as a subobtimal query plan can waste system resources.

2.4.3. Tidre

Tidre is a regular expression matching engine for FPGA accelerators and is developed by Teratide [65]. Tidre is based on VHDRE, a generator for regular expression matcher FPGA kernels [66]. These kernels, as generated by VHDRE, can process batches of UTF-8 strings and can indicate which records in the recordbatch match the given regular expression. This regular expression is hard-coded in the design. Therefore, changing the target regular expression requires resynthesis of the hardware design, which typically takes multiple hours to complete.

Tidre wraps this VHDRE kernel with Fletcher-generated interfaces. Furthermore, it provides a native interface that can be used to upload recordbatches to the accelerator and to download the results.

2.5. Related work

When considering related work, GPU accelerators prove to be of interest. GPU accelerators have been available for a longer amount of time than FPGA accelerators. As a result, there exists a considerable amount of research related to the integration of GPU accelerators in big data frameworks. Apache Spark, as one of the most popular big data frameworks, has been the subject of various integration projects with GPU accelerators [67].

HeteroSpark provides an integration between GPUs and Apache Spark [68]. The acceleration is integrated into the Spark Core and as an effect, the acceleration is made transparent to the developer. The GPUs are leveraged to accelerate compute intensive algorithms such as the ones found in typical machine learning applications. A drawback of HeteroSpark is that serialization and deserialization is required when transmitting data to and from the accelerator, as there is no integration with Apache Arrow. HeteroSpark demonstrates that a speedup of 18x can be achieved for various machine learning applications.

Spark-GPU is another project that integrates GPU accelerators in Apache Spark [69]. Apart from targeting machine learning algorithms, SQL operators can be offloaded to the GPU as well. A task scheduling system is implemented to autonomously select tasks that can be offloaded to the accelerator. The speedup for various machine learning workloads is shown to be 16.13x, while the speedup for various SQL queries is found to be at most 4.83x.

BlazingSQL is a SQL engine built on the RAPIDS stack [70]. A common in-memory data layer is constructed based on the CUDA dataframe (CuDF). This CuDF aims to overcome the same serialization challenges as Apache Arrow. This implementation targets SQL based ETL oprations as used in data preparation stage for machine learning algorithms. It is shown that BlazingSQL can achieve 20-100x speedups as compared to equivelent Apache Spark clusters at price parity.

It is seen that GPUs can be effectively deployed in big data frameworks to achieve high speedups for applications that are related to machine learning. Projects such as Spark-GPU however report diminishing returns when considering pure SQL workloads. FPGA accelerators could potentially be leveraged in these situations to push towards greater speedups.

Nonnenmacher integrates a Fletcher based FPGA kernel into Apache Spark [16]. It is shown that a speedup of up to 13x can be achieved when considering a SQL workload that features a regular expression filter operator. The work of Nonnenmacher contributes to the work of the ABS group, in which an FPGA accelerated big data system is designed [5].

Li et al. propose an FPGA based Network Interface Card with Query Filter (NIC-QF) that is able to perform SQL filtering in the storage nodes of big data frameworks [71]. This approach is able to speed up queries in the TPC-H benchmark suite by $1.9x$.

Ziener et al. propose an FPGA based system that accelerates SQL query processing [72]. Based on the SQL

query, an accelerator pipeline is constructed based on on a set of presynthesized and partially reconfigurable hardware modules. A similar system is presented by Becher et al., in which SQL pipelines are constructed from a library of partially reconfigurable hardware designs [73]. The focus here is on energy efficiency. It is shown that the performance of the system matches high-end database servers, while the energy consumption can be reduced by up to 95%.

Lee et al. propose a SQL acceleration solution based on SmartSSDs; A solid-state drive (SSD) with integrated FPGA fabric [74]. An analysis is performed to show that such a system could potentially accelerate queries in Spark SQL by $3x$, while only consuming 46% of the energy that is required by an equivalent CPU solution.

Apart from the integration between FPGA accelerators and big data frameworks, various works consider the optimization steps required to accelerate SQL workloads. Effectively optimizing FPGA accelerated execution plans requires new cost models that take these accelerators into account in order to be able to select the optimal plan as seen in Section 2.2.

Wang et al. propose a model for SQL operator processing time on FPGA accelerators [75]. Becher et al. identify that more research regarding a novel cost model is required if fast and energy efficient execution is desired in the context of hardware accelerators [76]. Fang et al. identify that query optimization regarding FPGA accelerators is not as well studied as with CPU [47]. SQL engines that consider FPGA accelerators exist but generally do not consider energy efficiency, one of the big advantages of FPGAs, in the optimization step [77, 78].

3

Alternative solutions

As stated in Section 1.3, this project investigates how big data frameworks should be designed such that they can be transparently and efficiently accelerated by use of FPGA accelerators. As seen in Section 2.5, the challenge of accelerating big data frameworks in general has been a popular research subject. In this research, the implementation of FPGA accelerators is not the only identified solution to speed up computations in a big data context. Additionally, the choice of which big data framework to accelerate is not trivial. As seen in Figure 1.2, there are numerous frameworks, all of which specialize in a different area of big data. It is possible that there are certain frameworks that lend themselves better for transparent acceleration. Finally, there are different contexts in which this big data framework can be deployed. As seen in Subsection 2.1.1, it is common for data engineers to run said frameworks on their local machine as well as on a more powerful server node or in a distributed setting on a cluster.

This chapter explores the alternative solutions for each of these three cases and provides multiple proposed solutions based on this analysis.

Section 3.1 elaborates on the method of analysis as used in the evaluation of these alternative solutions. Section 3.2 investigates the different speedup methods in the context of big data. Section 3.3 explores different big data frameworks and identifies the most promising candidates for acceleration. Next, Section 3.4 investigates the different contexts in which these accelerated frameworks can be deployed. Finally, Section 3.5 presents the proposed solutions.

3.1. Method of analysis

This section presents the method of analysis that is used to analyze the alternative solutions in the consecutive sections. Each alternative solution class is scored on several criteria in three categories; *effort*, *performance*, and *impact*.

First, criteria in the effort category are a proxy for the development effort. This effort is an important aspect when accelerating big data applications. It is noted that the integration of accelerator kernels into frameworks is non-trivial and a time consuming process [5]. Special care needs to be taken to make sure all necessary implementations required to answer the research questions as stated in Section 1.3 can be implemented within the time frame of the MSc thesis project.

Second, criteria in the performance category are equally important. These criteria aim to provide a measure for the performance increase as can be achieved with the alternative solution. The criteria are derived from prior research and the theory as provided in Chapter 2.

Finally, criteria in the impact category provide a measure for the potential impact and relevance within the big data community. Building a solution that is extremely fast for specific situations is interesting from an academic perspective. However, if these situations only occur in simulated environments, the impact of the solution in the real world is poor.

It is difficult to give each of the identified alternative solutions an absolute score based on the criteria in these three categories. However, an effort is made to quantize the findings for each of the criteria by means of a score of 1 to 5. The general motivation behind each score is shown below but this might change depending on the context in which it is used.

1. Bad. Completely unsuitable and/or infeasible, would require a major software rewrite or is otherwise a bad solution for the criterion.
2. Not ideal. Requires significant effort or is otherwise a less ideal solution for the criterion.
3. Neutral.
4. Good. Requires some modification of existing software or is otherwise a good fit for the criterion.
5. Excellent. Requires little to no modification of software or is otherwise an excellent fit for the criterion.

3.2. Speedup methods

This section considers the different methods that can be used to achieve a computational speedup in big data settings. Even though the scope of this project is bound to the use of FPGA accelerators, it is good to know how this technology relates to other speedup methods that can be deployed in the same setting. The identified alternative solutions can be seen in Figure 3.1.

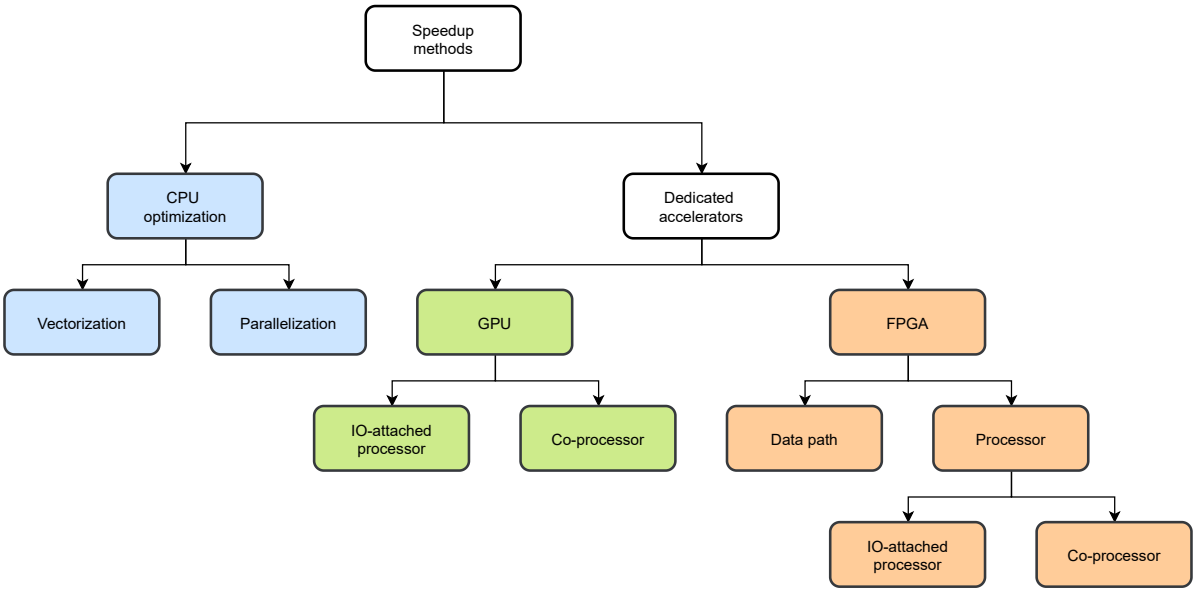


Figure 3.1: Classification tree for speedup methods.

Either CPU optimization methods or dedicated accelerators can be leveraged in order to accelerate big data applications. Dedicated accelerators are further subdivided into GPU accelerators and FPGA accelerators. These three main classes of speedup methods are further discussed below.

3.2.1. CPU optimization

This class contains optimizations based on the available CPU hardware.

Vectorization Most modern CPUs offer support for vectorized operations, an example of which is Intel's AVX [56]. These operations perform a single instruction on a vector that holds multiple values. The size of this vector is dependent on the underlying support in hardware.

This allows computations to be accelerated by evaluating operations on multiple values at once. The speedup is determined by the size of the vector.

An example of this acceleration method is the Gandiva execution engine as seen in Section 2.3. The Gandiva engine makes use of vectorized operations to perform computations on a vector of Apache Arrow data.

Parallelization Parallelization can be deployed in the form of multithreading. In multi-core CPUs, these threads can be executed on multiple cores, increasing the effective throughput. It is important to select the correct granularity when implementing a multithreading solution, as an overhead is associated with the creation and the management of multiple threads. The nature of big data allows parallel processing at a high granularity, in which multiple threads operate on separate recordbatches.

Such an implementation is standard in modern big data frameworks such as Apache Spark, Dremio, and Dask [79–81]. An example of how an operation can be performed on multiple threads is the computation of the sum of all values in a dataset. Each thread can be programmed to compute the partial sum of a recordbatch, after which these partial sums are added to obtain the final result.

3.2.2. GPU accelerators

The GPU, or graphics processing unit, is the most widespread hardware accelerator. Originally, GPUs are designed to accelerate graphics processing, but nowadays they can also be deployed for scientific purposes. All modern GPUs can therefore also be classified as a general-purpose graphics processing unit (GPGPU). In short, a GPU encompasses many small and simple processors, enabling it to perform highly parallelized arithmetic operations.

A distinction can be made between GPUs configured as IO-attached processors and GPUs configured as co-processors.

IO-attached processor GPUS that are configured as an IO-attached processor operate in their own memory space, separated from the memory space of the CPU. Sharing data between the host and the GPU accelerator requires the data to be copied between the two memory spaces.

Co-processor GPUs that are configured as a co-processor share their memory space with the CPU. The accelerator can therefore access data in the main memory without the need to copy this data.

An example of such a system is the Intel Graphics Technology, in which a GPU is manufactured on the same die as the CPU. In this case, memory coherence is handled by the hardware on the die.

Another possibility is to connect the GPU accelerator through a coherent IO interface. Examples of such interfaces include IBM's *Coherent Accelerator Processor Interface* (CAPI) or Intel's *Compute Express Link* (CXL) [47]. The compute system has to offer additional hardware support in order to enforce the memory coherence [47].

3.2.3. FPGA accelerators

As seen in Section 2.4, FPGA accelerators can be configured to process streams of data in a dataflow manner. This configuration, based on the usecase, enables greater performance than general purpose solutions. It is identified that the design and implementation of an FPGA kernel is time consuming. Three FPGA classifications are identified based on the placement in the system.

Data path FPGAs in tha data path are placed between network or storage and the CPU. These FPGAs can perform preprocessing operations, such as parquet decompression, which effectively increases the bandwidth between network or storage and the CPU [82].

Examples include the SSD's from Samsung and Xilinx [83]. These SSD's have an FPGA directly attached to the storage unit.

IO-attached processor In this configuration, the FPGA accelerator acts as another processor in parallel with the CPU. The FPGA has it's own memory space, requiring data to be copied between the CPU and the accelerator.

Another drawback of such a system, apart from the required data copies, is the limited capacity of local memory that is available on the FPGA. Because of this limitation, applications that run on FPGA accelerators can potentially only access a fraction of the entire dataset at the same time.

Co-processor In this configuration, the FPGA accelerator acts as a processor and shares it's memory space with the CPU. This eliminates the need for data copies and makes the FPGA accelerator a 'first class citizen' of the system.

Similar to GPU accelerators configured as co-processors, this can be realized by leveraging systems such as CAPI and CXL. Additionally, hybrid microchips exist that integrate FPGA fabric on the same die on which the CPU is implemented.

3.2.4. Comparison

When comparing these alternative solutions, the resulting performance is highly dependent on the system on which the solution is deployed and on the specific usecase that is accelerated. Therefore, the alternative speedup methods are only analyzed based on the required development effort and the expected impact.

The resulting decision making matrix of this analysis can be seen in Figure 3.1.

		Effort		**Impact**	**Total**
		Initial implementation	*Extensibility*	*Availability*	
CPU	Vectorization	3	4	5	**12**
	Parallelization	3	2	5	**10**
GPU	IO-attached	2	4	4	**10**
	Co-processor	2	4	3	**9**
FPGA	Data path	2	1	1	**4**
	IO-attached	1	1	2	**4**
	Co-processor	1	1	1	**3**

Table 3.1: Decision matrix for the speedup methods in the context of big data.

It can be seen that the impact of the CPU-based speedup methods is high. Most modern CPUs are multi-core processors and come with support for vectorized instructions, this makes the availability of these speedup methods very high. The scores for the criteria related to the development effort of these methods is relatively high when compared to the GPU- and FPGA-based speedup methods.

The availability of GPU accelerators is somewhat lower than the CPU speedup methods but still very high. GPU accelerators configured as co-processors have a slightly lower availability due to the additional required hardware support. GPU speedup methods require a little more development effort for the initial implementation because of the challenges associated to integrating hardware accelerators in existing big data frameworks. Writing GPU kernels is however done at a relatively high abstraction level, making extensions to the system straightforward once the integration is complete.

The availability of FPGA accelerators is very low, while the availability of FPGAs in the datapath is even lower. It can be seen that the development effort required for the initial implementation and the extension of existing systems is significant. This can be explained due to the in-depth knowledge about circuit design that is required in order to design and implement FPGA solutions.

As mentioned, the performance of the solution is highly dependent on the underlying system and the specific usecase. It is however perfectly possible to combine these alternative solutions. An example could be a system that offloads certain computations to both GPU and FPGA accelerators, depending on which accelerator is most suitable for that particular computation. At the same time, this system could perform computations on the CPU with vectorized instructions for operations for which there is no accelerator kernel available.

3.3. Big data frameworks

This section investigates different big data frameworks and identifies which frameworks are most suitable for transparent and efficient integration with FPGA accelerators. There is a vast number of different frameworks that all specialize in a different part of big data. In Section 2.2, differences such as in-memory versus on-disk processing and row-based versus column-based formats are identified.

In order to provide a good analysis of the wide selection of frameworks, a high-level classification is derived. This classification does by no means encompass all differences between each framework, but it provides a good starting point for the analysis.

The classification can be seen in Figure 3.2. Big data frameworks are divided into *batch-processing* frameworks, *streaming* frameworks, and *index-based* frameworks. These classifications are further discussed in the sections below. A more in-depth analysis of individual frameworks within these classifications can be found in Appendix A.

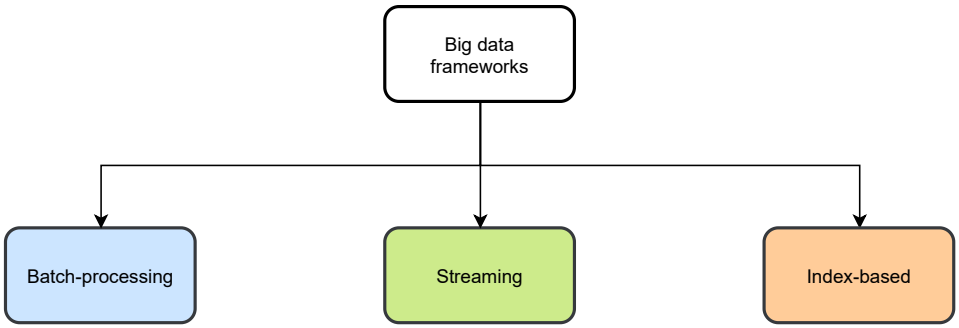


Figure 3.2: Classification tree for big data frameworks.

3.3.1. Batch-processing frameworks

Big data frameworks in this class process their data in batches. For large datasets, multiple batches need to be evaluated in order to process the entire set. These batches can either be in a row-based format or in a column-based format.

This way of processing fits well with distributed file systems as described in Subsection 2.2.1. Multiple batches can be processed in parallel, in which each node in the cluster works on a batch for which the data is present in the local machine until a shuffle is needed. Typically, this batch size is equal to the chunk size of the data stored in the distributed file system. Most modern big data frameworks allow this batch size to be changed, regardless of the chunk size.

Many batch-processing big data frameworks feature additional SQL modules that translate SQL queries into a tree of operations as described in Section 2.2. Here, it is common to see either custom made query optimizers or optimizers based on large open-source projects such as Apache Calcite [40].

3.3.2. Streaming frameworks

Streaming big data frameworks, on the other hand, excel in processing real-time data. Instead of processing large batches, which introduces latency, individual records can be processed as they come in. Alternatively, the data can be processed in micro-batches.

Processing individual records as they come in ensures the lowest possible latency. However, providing fault-tolerance in such a system can be challenging [84].

Processing the data in micro-batches allows for better fault-tolerance. Micro-batches are typically created by convolving an incoming record stream with a sliding window. Processing micro-batches does introduce a higher latency than processing individual records.

3.3.3. Index-based frameworks

Index-based big data frameworks are completely different from batch-processing and streaming frameworks. Instead of processing datasets, the focus is on performing search operations. A typical usecase for an index-

based framework is a string search that queries a large set of text documents.

These frameworks then search the underlying data by means of an inverted index. This inverted index is a datastructure that holds an index for each word in the underlying data. The value associated to this index then lists the positions of all occurrences of the word. This is typically implemented by a *hashmap*, where the index is given by the hash value of a particular word. The time-complexity for a lookup on such a hashmap implementation is $O(1)$, which enables searches to be very fast.

To allow this inverted index search to work, the underlying data first needs to be preprocessed to build the inverted index. This preprocessing might be time consuming, depending on the size of the dataset to be processed and the algorithm that is used to compute the indices.

3.3.4. Comparison

Individual frameworks within these classes are analyzed based on the estimated development effort, the expected performance, and their potential impact. These findings are summarized in a decision matrix which can be found in Table 3.2.

		Effort		**Performance**		**Impact**		**Total**
		Planner Extensibility	*Mem. Structure*	*Mem. Alignment*	*Max. Batch Size*	*Community size*	*Typical applications*	
Batch	Dask (distr.)	4	5	5	5	4	general, machine learning	**23**
	OmniSciDB	5	5	5	5	2	heterogeneous, BI, GIS	**22**
	Spark SQL	4	4	3	5	5	general	**21**
	InfluxDB IOx	4	5	4	5	1	timeseries, streaming	**19**
	Apache Hive	5	3	3	5	3	general, mapreduce	**19**
	Ballista	4	5	4	4	2	WIP, TPC-H	**19**
	Presto	5	1	3	5	4	general	**18**
	Dremio	5	5	3	2	2	preprocessing, data lake engine	**17**
Stream	Spark Streaming	4	4	3	2	4	streaming	**17**
	Apache Kafka	3	3	3	1	5	streaming, latency-sensitive	**15**
Index	Elasticsearch SQL	3	2	3	1	5	(text) search	**14**
	Apache Lucene	1	2	3	1	1	search algorithms	**8**

Table 3.2: Decision matrix for the feasibility of FPGA integration into different big data frameworks. The typical applications column lists typical uses of the frameworks such as business intelligence (BI), geographic information systems (GIS), or industry standard benchmark suites such as TPC-H. In addition, it is listed when a framework is considered a work-in-progress (WIP) at the time of writing.

In general, it can be seen that batch-processing frameworks score higher on the selected criteria than streaming and index-based frameworks. The scoring in the development effort category is high, since most of these batch-processing frameworks offer some form of SQL optimization module. These optimization modules provide a good starting point for integration with FPGA accelerators. One of the selected criteria for the estimated performance is the maximum batch size. Based on prior research, it is found that a high batch size is beneficial when integrating FPGA accelerators configured as IO-attached processors or as co-processors [16]. It is expected that this maximum batch size does not impose significant limitations for integrations with FPGA accelerators in the datapath.

The development effort for streaming frameworks is mostly related to the availability of extensible SQL modules. The maximum batch size of these frameworks is smaller than that of batch-processing framework. A distinction is made between the maximum batch size in frameworks that process individual records and frameworks that process micro-batches.

Finally, index-based big data frameworks score low on criteria in all three categories. These frameworks are already able to evaluate queries very fast by means of their reverse index algorithms. Instead of accelerating the queries, it could be possible to target the preprocessing stage at which the indices are constructed. However, this is outside of the scope of this work.

3.4. Deployment methods

This section analyzes the different deployment options for the FPGA accelerated big data framework. As seen in Subsection 2.1.1, data engineering is an iterative process in which engineers typically start with relatively small datasets on single node systems before moving to more advanced setups. This observation is reflected in the classification of the different deployment methods as seen in Figure 3.3.

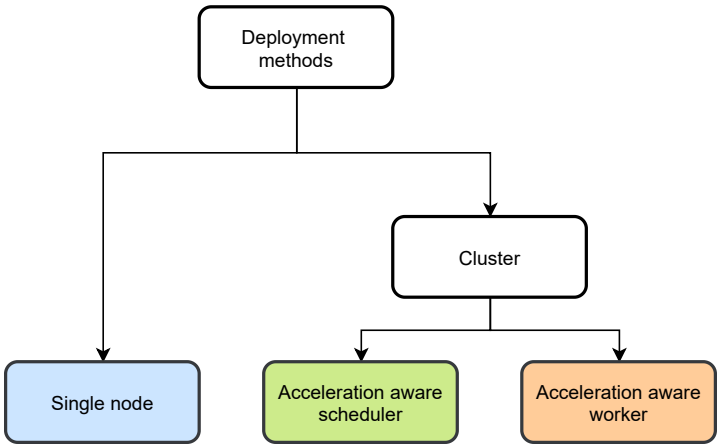


Figure 3.3: Classification tree for deployment methods.

The deployment methods are divided into two main classes; deployment on a single node and deployment in a cluster. The single node class encompasses both workstations such as a desktop or single nodes in a server. When considering deployment in a cluster, either the scheduler node or the worker node can be made aware of the acceleration. Both of these options have their own advantages and disadvantages.

3.4.1. Single node

Due to the iterative nature of data engineering, the single node deployment is the most common deployment of a big data framework. This can be explained by the fact that it is used by companies on all levels of the AI hierarchy of needs as presented in Figure 2.1. As mentioned, this class encompasses both single workstations as well as single nodes in a server. However, since a physical FPGA accelerator is required for the deployment of the accelerated framework, single node setups in a server will most likely be the common case. This holds especially true when considering the ease at which an FPGA instance can be provisioned in popular cloud services [14, 85]. It is not likely that users running a big data framework on their local machine will be able to locally deploy an FPGA accelerated framework.

The planning in terms of which parts of the execution graph are to be accelerated can be performed once per query during the query optimization phase as described in Section 2.2.

Finally, apart from accelerating operators with a narrow dependency, it is also feasible to offload the computation of operators with a wide dependency. Since there is only a single node, no data needs to be shuffled over the network as all chunks are available locally. Allowing operators with a wide dependency to be evaluated on an FPGA accelerator results in more acceleration opportunities.

3.4.2. Acceleration aware scheduler

In contrast to a single node deployment, a cluster deployment has a smaller community size because typically only companies in the higher sections of the AI hierarchy of needs can benefit from such an advanced setup.

When considering a cluster deployment of an accelerated framework, the most straightforward way would be to make the scheduler node aware of the acceleration. This means that the scheduler node performs all necessary transformations of the execution plan in order to offload part of the computation to an FPGA accelerator.

However, if the scheduler node is not aware of the worker node's capabilities, the implication of performing this acceleration planning in the scheduler node means that all worker nodes should be able to execute the new execution graph. Since this new graph contains FPGA accelerated operations, all worker nodes in the cluster should be equipped with an FPGA accelerator. This raises the barrier to adopt such an accelerated framework, as substituting all nodes in an existing compute cluster for FPGA accelerated nodes is costly.

Similar to single node deployments, acceleration aware scheduler deployments are able to perform the acceleration planning once per query in the query optimization phase which runs on the scheduler node.

In contrast to the single node deployment, operators with wide dependencies do require a network shuffle in a cluster deployment. Therefore, the focus is only on accelerating operators with narrow dependencies [5]. This results in less opportunities for acceleration.

3.4.3. Acceleration aware worker

Another option for the deployment of an accelerated big data framework in a cluster setting is to make the worker node aware of the acceleration, instead of the scheduler node. This overcomes the problem of having to substitute all nodes in the cluster by FPGA nodes, because in this case the scheduler node will submit the original execution execution plan to all workers. This lowers the barrier for adoption in industry, because it allows existing nodes in the cluster to remain unchanged when FPGA accelerated worker nodes are added.

Since the scheduler node does not perform the acceleration planning, the accelerated worker node needs to perform this planning for all incoming execution plans. Depending on the implementation of the distributed framework, the scheduler node can send the execution plan to the worker nodes in the form of multiple subtrees of the execution plan. In this case, instead of performing the acceleration planning once on the entire execution plan, it now has to be performed for each subtree of the plan. Having to perform the acceleration planning multiple times for a single query incurs additional overhead when compared to the single node deployment and the acceleration aware scheduler deployment.

Like the acceleration aware scheduler option, the acceleration aware worker is only able to offload the computation of operators with a narrow dependency. Additionally, performing the acceleration planning on subtrees of the execution plan imposes limits on the granularity of the operations that can be accelerated. In the other deployments that were considered, the entire execution plan is guaranteed to be available during the acceleration planning phase. It is therefore possible to offload computations on a higher granularity; for example, targeting a sequence of specific operators. If this sequence spans multiple subtrees of the execution plan, the acceleration planning phase of the worker node is not able to recognize it as a candidate for acceleration. As an effect, there are less opportunities for acceleration when considering an acceleration aware worker deployment.

3.4.4. Comparison

The development effort of implementing the three deployment options as described above is expected to be the same, as the semantics of performing the acceleration planning remain unchanged. The only difference is the location in which they are implemented. For this reason, only criteria related to the expected performance and potential impact of the system are analyzed. The results of this analysis can be seen in Table 3.3.

	Performance		Impact		Total
	Planning efficiency	*Acceleration opportunities*	*Community size*	*Ease of adoption*	
Single node	5	5	5	3	**18**
Accelerated worker	4	3	3	5	**15**
Accelerated scheduler	5	4	3	2	**14**

Table 3.3: Decision matrix for the deployment methods in the context of FPGA accelerated big data frameworks.

When considering the total score of each deployment method, the single node option is the best option with 18 points. This is mainly due to the community size of single node deployments being larger than cluster deployments. Apart from this, it is expected that a single node deployment is able to achieve excellent performance. The only problem with this solution class is that is has a relatively high barrier of adoption in industry. The reason for this is twofold. First, it is expensive and unpractical for users to upgrade their local machine with an FPGA accelerator. Second, for users that run a single node deployment on a server, although it is relatively easy to provision an FPGA instance, the cost associated to this is still significantly higher than that of a normal instance.

When considering cluster deployments, the accelerated worker option is the best option with 15 points, closely followed by the accelerated scheduler option with 14 points. Although the accelerated worker option has the lowest score in terms of acceleration opportunities, it has by far the highest score in terms of adoption.

3.5. Proposed solutions

An important observation is that the speedup methods analyzed in Section 3.2 are not mutually exclusive. An accelerated big data framework that utilizes vectorization in combination with GPU and FPGA acceleration is a perfectly viable solution. The scope of this project is however limited to the implementation of FPGA accelerators, but it is good to know what the strengths of other speedup methods are. The *processor* class solution is chosen for implementation, due to it's slightly higher availability. In addition, it's strengths best fit the acceleration needs of SQL operators. This class is further subdivided into FPGA processors that act as a *co-processor* or *IO-attached processor*. The availability of running as a *co-processor* is heavily dependent on the support of the underlying system such as OpenCAPI [86]. Therefore, a configuration in which the FPGA processor can run as a *co-processor* is preferred but not required.

From the frameworks that are considered for integration, frameworks within the *batch-processing* class best fit acceleration with FPGA accelerators deployed as a processor. Dask and Dask distributed (as the name suggests, this is a version of Dask designed to run in a distributed setting) hold the highest total score of all batch-processing frameworks. However, another bound of this project is that an implementation with Dremio should be provided, as this is described in the original project description. From Section 3.3, it can be seen that the Dremio framework is interesting because of the low development effort required for integration, but the performance and impact leave something to be desired.

Finally, Section 3.4 shows that deploying the accelerated framework as a single node deployment or as an acceleration aware worker in a cluster setting are the two most interesting choices. Both these options have a considerable higher impact than accelerating the scheduler node in a cluster setting. The single node deployment option has the greatest potential performance, while the accelerated worker option has the lowest barrier of adoption in industry.

Based on these options, three different solutions are proposed that all contributed to answering the research questions as stated in Section 1.3. These solutions are briefly presented below. Subsequent chapters elaborate on the implementation and results of these solutions.

All of the proposed solutions make use of FPGA accelerators configured as an IO-attached processor, since the FPGA kernel as deployed on Amazon Web Services (AWS) does not support the co-processor configuration at the time of writing.

Dremio integration Dremio is selected for integration because of the original project description. All data structures in Dremio are stored using the Apache Arrow in-memory format. Additionally, Dremio incorporates the Gandiva execution engine. An implementation that utilizes both CPU optimization methods as well as FPGA accelerators is a good demonstration of multiple big data speedup methods working together. Dremio's internal query optimizer is based on the Apache Calcite framework, which enables optimization rules to be used in other Apache Calcite based frameworks as well. A single node deployment is chosen for Dremio in order to investigate the best performance that can be achieved.

Dask integration Apart from Dremio, Dask is chosen as the second framework for acceleration with FPGA accelerators configured as processors. Both Dask and Dask distributed are identified as the most promising candidates for integration based on the required development effort, expected performance, and potential impact. A single node deployment is chosen, as vanilla Dask is not suitable for deployment in a cluster. Therefore, both the Dask and the Dremio implementations are deployed on a single node. This allows for a good comparison of the effect the different characteristics of these two big data frameworks have on the performance of FPGA integration.

Dask distributed integration Finally, Dask distributed is selected for the third integration. This version of Dask is specifically designed to run in a cluster setting and is therefore the perfect candidate to investigate a cluster deployment with an acceleration aware worker node. Since Dask and Dask distributed are so similar, this allows for a good comparison between the single node and acceleration aware worker deployment options.

4

Dremio integration

Dremio is an open-source data lake engine developed by the commercial company Dremio and written in Java. The developers at Dremio have also created Gandiva, a native execution engine for Apache Arrow data. Additionally, they have contributed to various other modules in the Apache Arrow project. Dremio is a commercial product that aims to accelerate corporate BI queries.

The in-memory format of data in Dremio is completely based on the Apache Arrow format. Together with the incorporated Gandiva engine, this allows Dremio to evaluate queries very fast.

4.1. Implementation

Dremio includes it's own SQL planner which is called *Sabot*. This Sabot planner is based on the open-source Apache Calcite SQL optimizer [40]. The integration of Dremio with FPGA accelerators is mainly concerned with extending this Sabot planner.

An FPGA kernel that can perform regular expression matching is selected for integration. This kernel is able to evaluate regular expression based filter operations, as discussed in Subsection 2.4.3. For this reason, the architectural details of the integration are presented in the context of accelerating filter operators. The details of this implementation can be found below.

4.1.1. Architectural details

Dremio's Sabot planner is extended in two places. First, the *planner package* is extended by introducing an additional FPGA acceleration planning phase. Second, the *operator package* is extended by adding a new operator that is able to offload filter evaluations to a native package by use of the Java Native Interface (JNI). The *native-op* package is implemented in C++ and receives these JNI calls. A Tidre-based filter implementation as well as an RE2-based implementation is included in this package. The Tidre-based evaluation offloads the computation to an FPGA accelerator, while the RE2-based implementation evaluates the regular expression on the CPU using Google's RE2 library [87].

An overview of this system can be seen in Figure 4.1. All individual components are discussed in the following sections.

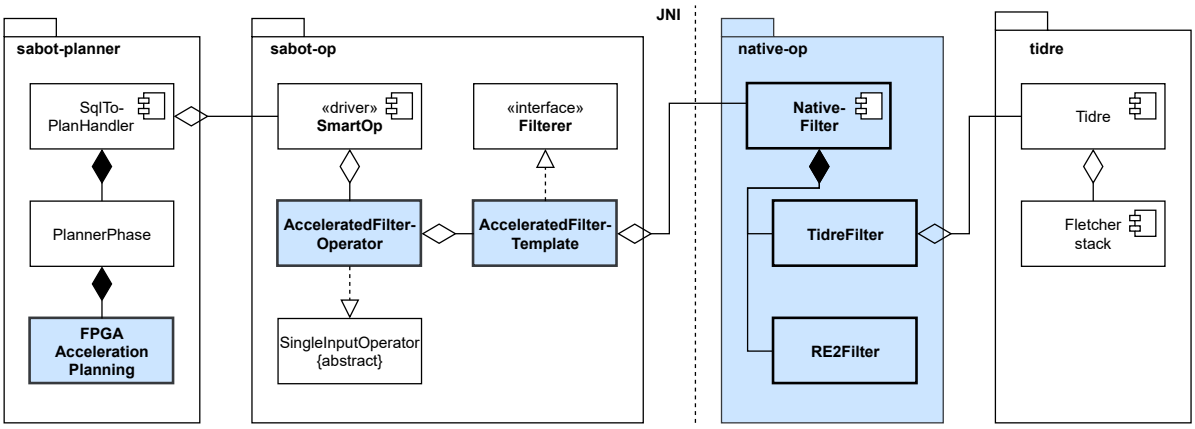


Figure 4.1: Overview of the accelerated version of Dremio. The contributions of this project are shown in blue with bold borders. The diagram is inspired by UML but does not follow all UML practices.

4.1.2. FPGA acceleration planning

Dremio's internal Sabot planner implements a number of planning phases. Each phase transforms the incoming execution tree by use of a number of optimization rules. An overview of the most important planning phases for this integration effort can be seen in Figure 4.2.

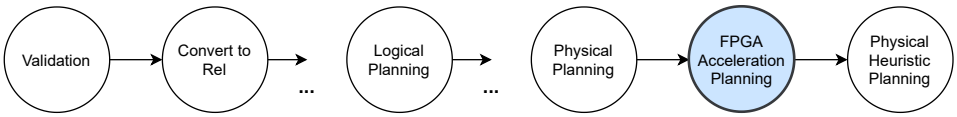


Figure 4.2: Sabot planner phases including the new FPGA acceleration planning phase.

The validation phase validates the query based on the schema of the dataset. In the next phase, the SQL query is parsed and converted to an expression tree of logical operations. This tree is then optimized in a number of logical planning phases. The pre- and post-logical phases are omitted in the figure. These logical planning phases perform optimizations such as filter pushdown and projection merging. The subsequent physical planning phase converts all logical operations in the expression tree to a physical implementation as described in Section 2.2.

The resulting expression tree of physical operators is used as an input for the added FPGA acceleration planning phase. Adding this new phase after the logical and physical planning phases ensures that the framework is able to perform important optimizations. The new planning phase incorporates optimization rules which target physical operators that are semantically equivalent to a given FPGA implementation. Matching physical operators are substituted by an accelerated operator. An example of such an optimization rule that transforms a filter operator to an *AcceleratedFilter* operator can be seen in Listing 4.1.

Finally, the physical heuristic planning phase performs additional heuristic optimizations regarding nested loop joins.

Apache Calcite's *RelOptHelper* is used on line 8 of Listing 4.1 to specify which operators in the incoming execution tree this optimization rule matches on. Additionally, it also provides methods to target a specific sequence of operators. This allows for the construction of more complex optimization rules.

As noted in Section 2.4, developing an efficient FPGA implementation of a SQL operator is labor intensive. In order to increase the impact of such a kernel, operators in the execution plan can be rewritten during the FPGA acceleration planning phase such that they can match the optimization rules in cases where they normally would not. This increases the number of usecases for which a given FPGA kernel can be used. For instance, a *SQL LIKE* based filter operator could be rewritten to a regular expression based filter operator if the predicate can be evaluated on an accelerated regular expression engine. Rewriting these operators to semantically equivalent ones is known as *query exploration* and can result in a set of multiple execution plans that yield the same correct result. As seen in Section 2.2, cost models are required to select the optimal plan from this set.

```
public class AcceleratedFilterPrule extends RelOptRule {

  public static final RelOptRule INSTANCE = new AcceleratedFilterPrule();

  // Match on any filter prel
  private AcceleratedFilterPrule() {

    // More specific rules could be added to match specific filter conditions
    super(RelOptHelper.any(FilterPrel.class), "AcceleratedFilterPrule");

  }

  @Override
  public void onMatch(RelOptRuleCall call) {

    final FilterPrel  filter = (FilterPrel) call.rel(0);

    // Transform the operator to the FPGA based filter operator with the same parameters
    call.transformTo(
        new AcceleratedFilterPrel(
            filter.getCluster(),
            filter.getInput().getTraitSet(),
            filter.getInput(),
            filter.getCondition()
        )
    );

  }

}
```

Listing 4.1: A Sabot optimization rule that targets a filter operator and substitutes an accelerated operator.

4.1.3. Accelerated filter operator

The accelerated physical operator that substitutes the matching physical operators in the FPGA planning phase needs to match with an operator implementation as defined in the Sabot op package. In this case, an accelerated filter operator is added that implements the abstract *SingleInputOperator* class. The actual evaluation of the recordbatch based on the filter predicate is performed in the *AcceleratedFilterTemplate* class, which implements the *Filterer* interface. The vanilla filter template class normally offloads it's computation to the Gandiva engine when this is supported, but the new accelerated class offloads the evaluation to the newly introduced native filter package.

This native filter package has support for evaluation of the recordbatch on an FPGA accelerater through the Tidre package. Additionally, the evaluation can be performed on the CPU by use of the Google RE2 framework, which is a regular expression evaluation framework written in C++.

A sequence diagram of these interactions in the case where the evaluation is performed on an FPGA accelerator can be seen in Figure 4.3.
SmartOp is the driver in the Sabot planner which coordinates the execution of the different operators in the execution plan. This driver additionally checks whether the operator behaves as expected. One example of such a check is to verify whether the schema of the resulting recordbatch matches the expected schema of the next operator in the plan. *SmartOp* calls the *setup()* method of all operators. Normally, this triggers Gandiva code generation, but in case of the accelerated filter operator this is not necessary. The setup method then returns a *VectorContainer*, which holds the Arrow buffers in which the resulting recordbatch is to be written. The setup also sets the state of the operator to *CAN_CONSUME*.

Tidre evaluation After the accelerated filter operator is set up, SmartOp repeatedly calls the *consumeData()* method until the state of the accelerated filter operator no longer is set to *CAN_CONSUME*. This method is where the filter evaluation happens. The *filterBatch()* method is called on the accelerated filter template, which invokes a call to *filterBatchSV()* or *filterBatchNoSV()* depending on whether the incoming recordbatch has a selection vector specified. This selection vector is a vector that holds indices of records in the recordbatch that matched an upstream filter. In this case, it is assumed that there are no upstream filters and

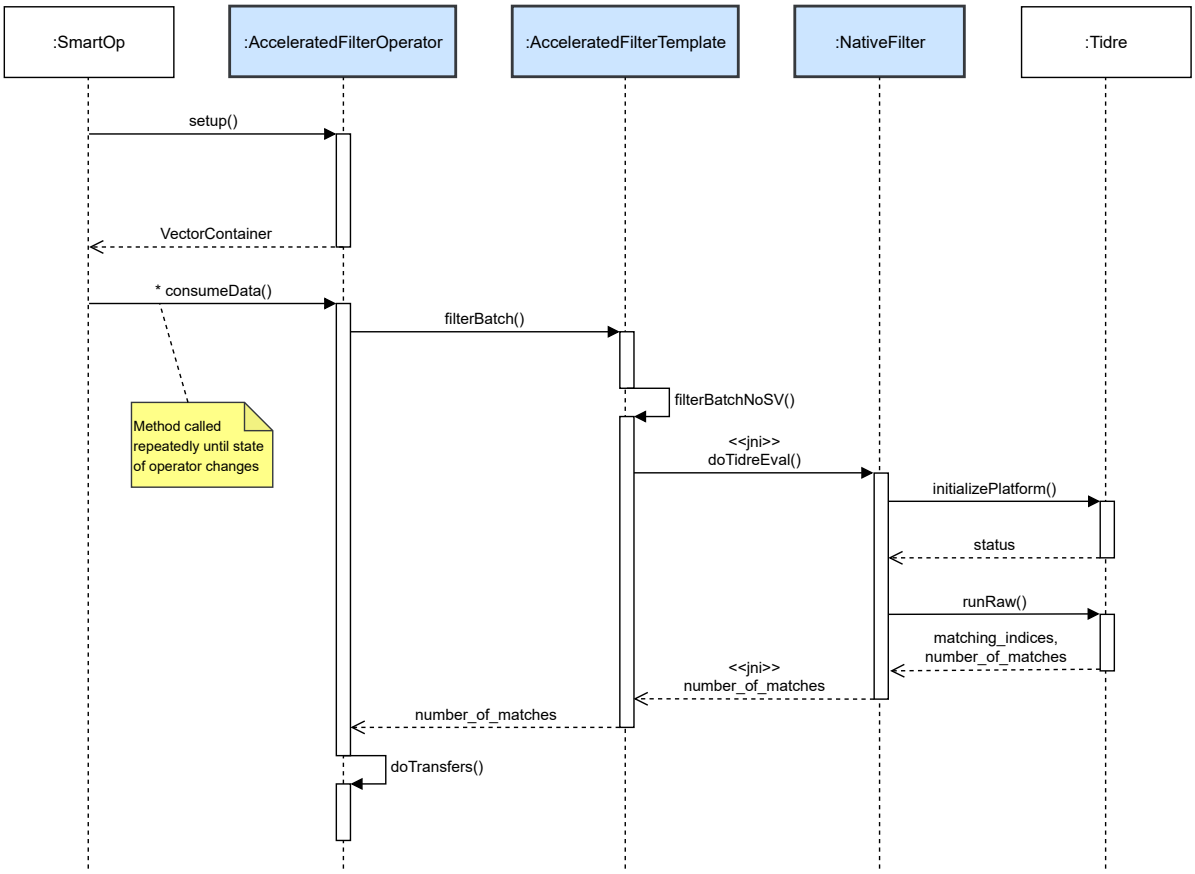


Figure 4.3: Sequence diagram illustrating the interactions of the accelerated operator in Dremio.

the accelerated filter template invokes the *filterBatchNoSV()* method. This assumption hold true as the selected usecase, which is further explained in Subsection 4.2.1, only contains a single filter operator. The *filterBatchNoSV()* method offloads the evaluation to the native filter package by calling the *doTidreEval()* method through the JNI. Pointers to the Arrow buffers of the input recordbatch and outgoing selection vector are passed as arguments. The *doTidreEval()* method only returns the number of matches, as the indices of records that match the filter predicate are written to the Arrow buffer of the outgoing selection vector. Since the native filter package already holds a pointer to this Arrow buffer, this does not need to be returned by the *doTidreEval()* function.

The native filter package in turn initializes the Fletcher platform through the Tidre package and offloads the evaluation by invocation of the *runRaw()* method.

RE2 evaluation In case the evaluation is to be performed using the RE2 library, the accelerated filter template invokes the *doRE2Eval()* method instead of the *doTidreEval()* method.

Unlike evaluation using the Tidre kernel, in which the regular expression that is to be evaluated is hard-coded in the FPGA design, the RE2 libary offers more flexibility here. In this case, the regular expression string is passed as an additional argument in the *doRE2Eval()* method. The native function then pre-compiles this regular expression in the RE2 library and evaluates the records in the incoming recordbatch.

4.2. Experimental setup

All experiments are performed on the Elastic Compute Cloud (EC2) of AWS and are run on an *f1.2xlarge* instance (FPGA accelerated machine), unless stated otherwise. This machine type has 8 virtual cores and features the high frequency Intel Xeon E5-2686 v4 (Broadwell) processor [88]. In addition, these instances are equipped with a 16nm Xilinx UltraScale Plus FPGA. The *f1.2xlarge* machine type has 122 GiB memory. The Centos 7 based *FPGA developer AMI* is used for the machine image.

The codebase of the accelerated version of Dremio can be found on GitHub [89]. This repository contains

a detailed installation guide with instructions on how to deploy the framework on AWS.

Two simple benchmarks are written to assess the performance of the accelerated version of the framework. These benchmarks are based on the regular expression usecase as described below.

4.2.1. Regular expression usecase

A usecase is selected where a regular expression filter is applied to a dataset containing tweet-like strings. Apart from these tweet-like strings, the dataset also contains a column of integer values, which can be used for various aggregation operations. The schema of this dataset can be seen in Listing 4.2.

```
{
    "value_column":   Int32,
    "string_column":  List(Char)
}
```

Listing 4.2: Schema of the regular expression usecase dataset.

The query for this usecase can be seen in Listing 4.3. This query features a sum aggregation and a regular expression filter.

```
SELECT SUM("value_column") FROM "tweet-like-strings.parquet" WHERE REGEXP_LIKE("string_column", '.*[tT][eE][rR][aA][tT][iI][dD][eE][ \t\n]+[dD][iI][vV][iI][nN][gG][ \t\n]+([sS][uU][bB])+[sS][uU][rR][fF][aA][cC][eE].*')
```

Listing 4.3: Chosen SQL query for the regular expression usecase.

4.2.2. Dataset

The dataset is generated by means of a Python script, which can be found on GitHub [90]. This generator is able to generate datasets with a variable number of records. The schema of the dataset can be seen in Listing 4.2. Fields in the *value_column* contain a randomly generated integer between 1 and 100. Fields in the *string_column* contain strings of 100 characters consisting of upper- and lowercase ASCII characters, digits, and whitespaces. Of these strings, 5% match the selected regular expression as seen in Listing 4.3. The content of these matching strings is generated using the Xeger library and padded to 100 characters [91]. The remaining 95% of strings are randomly generated. The resulting dataset is stored in a parquet file.

4.2.3. Input size benchmark

This is the first benchmark constructed for the regular expression usecase. The benchmark tests how the system behaves for a varying number of input records. The number of records in the dataset range from 1,000 records up to 512,000 records, increasing by factors of two. The batch size is kept constant at 64,000 records, the maximum value in the case of Dremio.

This benchmark therefore spans two interesting regions; the input datasets of 1,000 records up to 64,000 records can be evaluated in a single recordbatch, while the datasets of 128,000 up to 512,000 records have to be evaluated in multiple.

4.2.4. Batch size benchmark

This benchmark tests how the system behaves for a varying number of records in the recordbatch. The input size is kept constant at 32,000 records, while the batch size varies from 1,000 to 64,000 records, increasing by factors of two.

Therefore, the smaller batch sizes require the dataset to be evaluated in multiple batches, while the largest batch sizes of 32,000 and 64,000 records allow the dataset to be evaluated in a single batch. The largest batch size is interesting as it is bigger than the input size and is therefore not expected to behave different compared to the batch size of 32,000 records.

4.2.5. Measurement setup

The first run for each configuration in an experiment is discarded in order to mitigate cache warming effects. After this first run, ten consecutive runs are performed and the measured runtimes are averaged to obtain the final result for that configuration. The experiments measure the runtime of the filter operator, as well as the total query runtime.

4.3. Results

The results in terms of the input size and batch size benchmarks are presented in this section. Additionally, an experiment regarding query exploration is considered.

4.3.1. Accelerating the query

The query as seen in Listing 4.3 is executed on the accelerated version of Dremio. The FPGA acceleration planning phase transforms the physical execution plan by substituting the filter operator with the accelerated version. This can be seen in Figure 4.4. As described in Section 4.1, this accelerated operator can either offload the evaluation of the filter to the RE2 library or to an FPGA accelerator through the Tidre package.

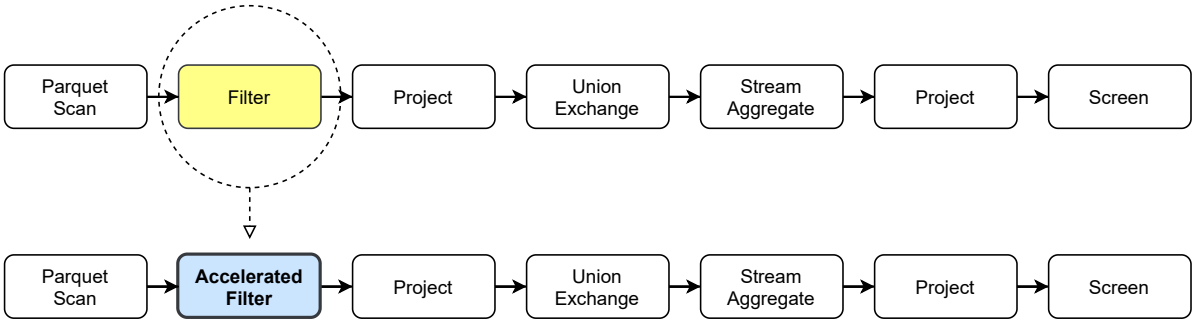


Figure 4.4: Execution plan transformation for the regular expression usecase in Dremio.

4.3.2. RE2 acceleration

This section shows the results of accelerating Dremio with the RE2 library, which implements a regular expression engine optimized for the CPU.

It was found that performing the vanilla Dremio benchmarks on an *f1.2xlarge* instance type would induce costs out of budget for this project. This is due to the significantly higher runtimes of the vanilla Dremio implementation as compared to the accelerated versions of the framework, as well as the high cost of the *f1.2xlarge* instance type itself. Therefore, the *r4.2xlarge* instance type is used for benchmarks performed on the vanilla Dremio implementation. These machines feature the same processor as the *f1.2xlarge* machine types. Instead of 122 GiB memory, these machines have 61 GiB installed. All benchmarks as performed on the RE2 accelerated version of Dremio are run on an *f1.2xlarge* instance as described in Section 4.2.

The results in this section are presented per benchmark as described in Subsection 4.2.3 and Subsection 4.2.4.

Input size benchmark The runtimes of the parquet scan and filter operators for the input size benchmark on vanilla Dremio and the RE2 accelerated version of Dremio can be seen in Figure 4.5. The x-axis shows the number of records in the input file, while the y-axis shows the operator runtime in seconds.
It can be seen that the runtime of the parquet scan operators remains roughly the same for both the vanilla Dremio and RE2 accelerated version of Dremio. This is expected since the implementation of the parquet scan operator is left unchanged. This differs from the work of Nonnenmacher in which the parquet scan operator had to be changed in order to convert the data into an Arrow format. This resulted in higher parquet read times for accelerated version of the framework [16].

Looking at the filter operators, it can be seen that the filter operator in the RE2 accelerated implementation is $\sim 66x$ faster than the filter operator in the vanilla Dremio implementation. The figure also shows the linear relationship between the filter runtime and number of input records, indicating a time complexity of $O(n)$. For low input sizes this linearity breaks down for the RE2 based filter operator. It is expected that this occurs due to the overhead associated with the JNI calls.

Batch size benchmark The runtimes of the parquet scan and filter operators for the batch size benchmark on vanilla Dremio and the RE2 accelerated version of Dremio can be seen in Figure 4.6. The x-axis shows the number of records in the recordbatch, while the y-axis shows the operator runtime in seconds.
Again, it can be seen that the runtimes of the parquet scan operators are roughly the same. The differences between the two are slightly more distinct than in Figure 4.5, but these differences remain in the order of 10^{-2}.

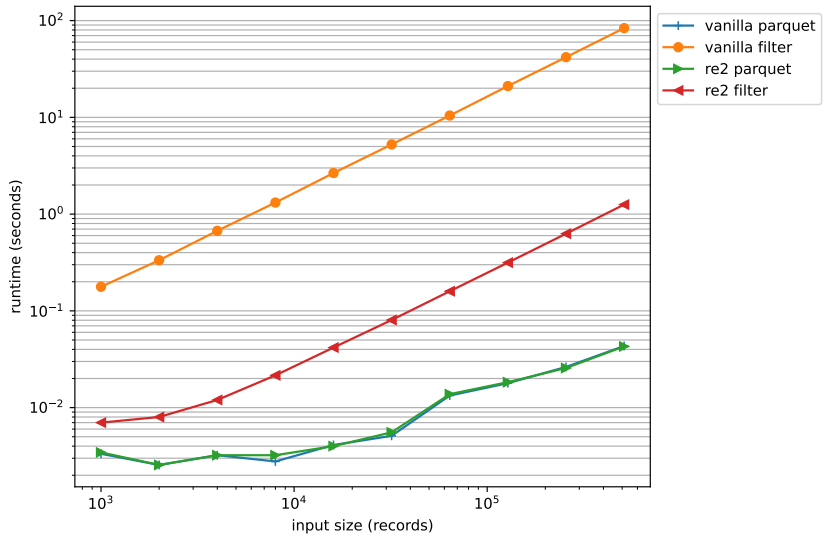


Figure 4.5: Parquet scan and filter operator runtimes for the input size benchmark on vanilla Dremio and the RE2 accelerated version of Dremio. The batch size is set to 64,000 records.

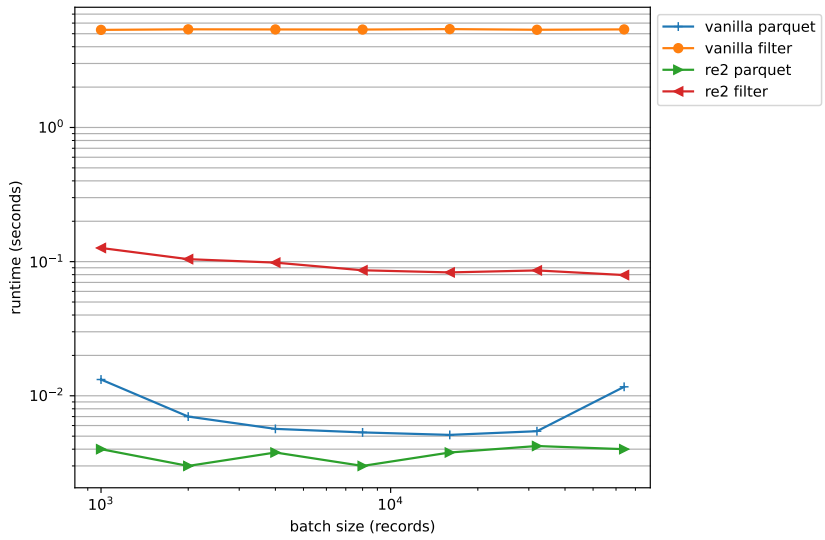


Figure 4.6: Parquet scan and filter operator runtimes for the batch size benchmark on vanilla Dremio and the RE2 accelerated version of Dremio. Input size is set to 32,000 records.

As expected, the runtime of the vanilla Dremio filter operator remains constant for all batch sizes. In the case of the RE2 accelerated version of Dremio, the runtime of the filter operator is reduced by ~ 37% by increasing the batch size from 1,000 records to 64,000 records. This is likely due to an overhead associated with the invocation of the native filter operator through the JNI, which has to be done more often for smaller batch sizes.

Contrary to the expectations, increasing the batch size above 32,000 records further reduces the runtime of the RE2 based filter operator. It is possible that there are still deviations in the recorded runtimes after averaging 10 consecutive runs.

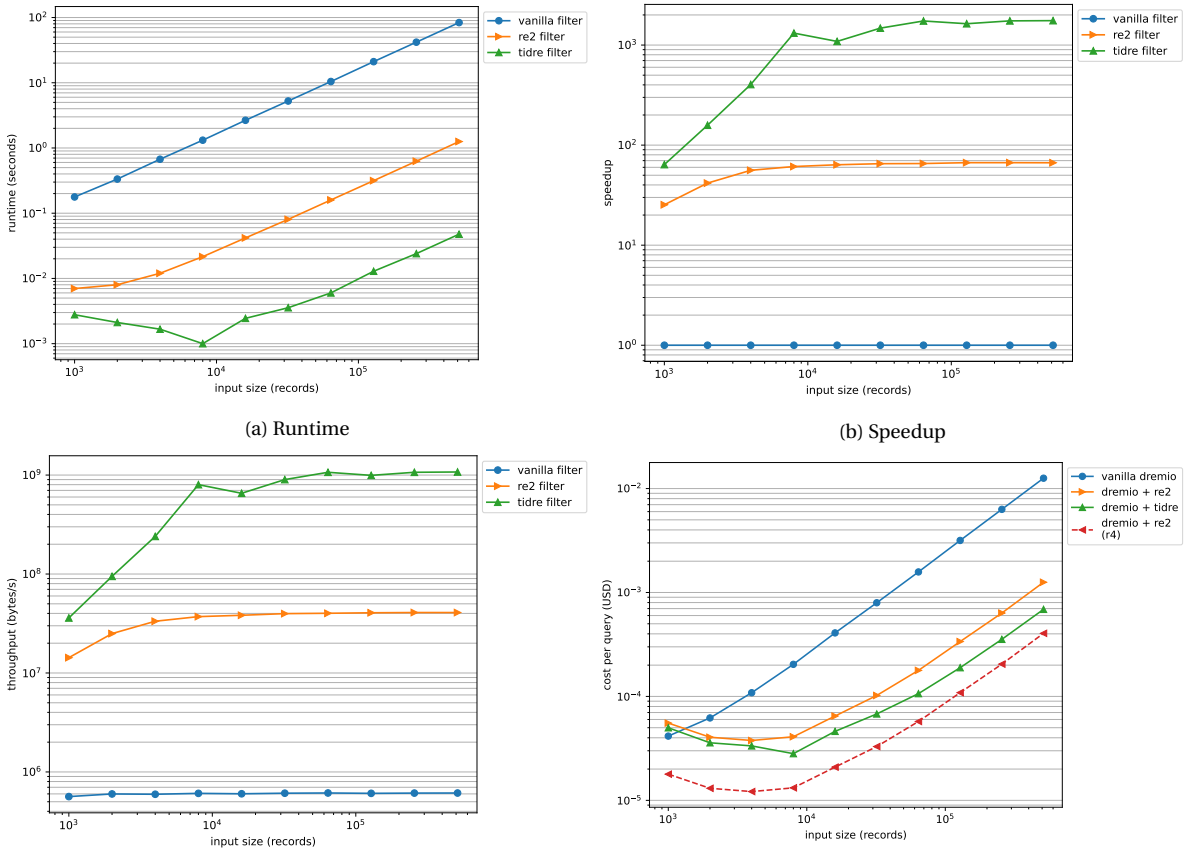
4.3.3. Tidre acceleration

In this section, the results of accelerating Dremio with Tidre are presented. These results are compared with both the vanilla Dremio implementation and the RE2 accelerated implementation. All measurements related to the Tidre accelerated version of Dremio are performed on an *f1.2xlarge* machine type as described in

Section 4.2.

Input size benchmark The results of the input size benchmark for vanilla Dremio, the RE2 accelerated version of Dremio, and the Tidre accelerated version of Dremio can be seen in Figure 4.7. The filter operator runtime, speedup relative to the vanilla operator, throughput, and cost per query can be seen in Figure 4.7a, Figure 4.7b, Figure 4.7c, Figure 4.7d respectively.

The speedup and throughput are computed based on the measured runtime of the filter operator. The cost per query is computed based on the runtime of the entire query.



(a) Runtime

(b) Speedup

(c) Throughput

(d) Cost per query

Figure 4.7: Filter runtime, speedup, throughput, and cost per query for the input size benchmark on both the RE2 and Tidre accelerated versions of Dremio. The batch size is set to 64,000 records.

From Figure 4.7a, it can be seen that the runtime of the filter operator in the Tidre accelerated version of Dremio is roughly three orders of magnitude lower than the runtime of the filter operator in the vanilla version of Dremio. When compared to the RE2 accelerated version of Dremio, the runtime of this Tidre based filter operator is roughly one order of magnitude lower.

Interestingly enough, the runtime of the Tidre based filter operator decreases for increasing input sizes up to 8,000 records. It was found that the measurements in this region contained several outliers, resulting in this strange behaviour. The raw data of these measurements can be found in Appendix B. It is suspected that the timer implementation as found in Dremio's *OperatorStats* package, which is used to perform these measurements, is not suited for measuring operators with such low runtimes. As applications in a big data context typically involve greater datasets than input files with a mere 8,000 records, this inaccuracy for lower input sizes is considered acceptable. When considering the trend of the filter operator runtimes for input sizes above 8,000 records, there appears to be a linear relation between the runtime of the operator and the input size.

Figure 4.7b shows the speedup of the filter operators as compared to the filter operator in the vanilla Dremio implementation. It can be seen that the Tidre based filter operator provides a speedup of ~ $1750x$ as compared to the vanilla implementation. This is an additional speedup of ~ $26x$ compared to the RE2 based filter implementation.

The throughput as computed from the measured runtime of these filter operators can be seen in Figure 4.7c. It can be seen that the vanilla Dremio filter only reaches a throughput of ~ $600kB/s$. This throughput is very low and signifies the great difficulty Dremio has in evaluating the massive regular expression as shown in Listing 4.3. The RE2 based filter operator reaches ~ $40MB/s$, while the Tidre based filter operator reaches as throughput of ~ $1GB/s$.

Finally, Figure 4.7d shows the cost per query for the input benchmark. This cost is shown in USD. The cost is derived by interpolating the hourly costs of the used instance types. The resulting cost per second is then multiplied by the average runtime of the total query.

Again, outliers were found in the measurements of the total query runtimes. This is also the case for measurements performed with the RE2 accelerated version of Dremio. Similar to the measurements of the individual filter operators, these outliers only occur for input sizes up to 8,000 records.

Apart from this lower region of the plot, which is plagued by these outliers, queries performed on the RE2 and Tidre accelerated versions of Dremio are always cheaper than queries performed on the vanilla version. Since the RE2 accelerated version is run on an *f1.2xlarge* instance without requiring an FPGA accelerator, it could also be deployed on an *r4.2xlarge* machine. A fourth line is added that projects the cost per query of the RE2 accelerated version of Dremio if it were to be run on this cheaper machine type. The assumption is made that switching to this machine type does not affect the performance of the RE2 accelerated framework.

Batch size benchmark The results of the batch size benchmark for vanilla Dremio, the RE2 accelerated version of Dremio, and the Tidre accelerated version of Dremio can be seen in Figure 4.8. The filter operator runtime, speedup relative to the vanilla operator, throughput, and cost per query can be seen in Figure 4.8a, Figure 4.8b, Figure 4.8c, and Figure 4.8d respectively.

Again, the speedup and throughput are computed based on the measured runtime of the filter operator. The cost per query is computed based on the runtime of the entire query.

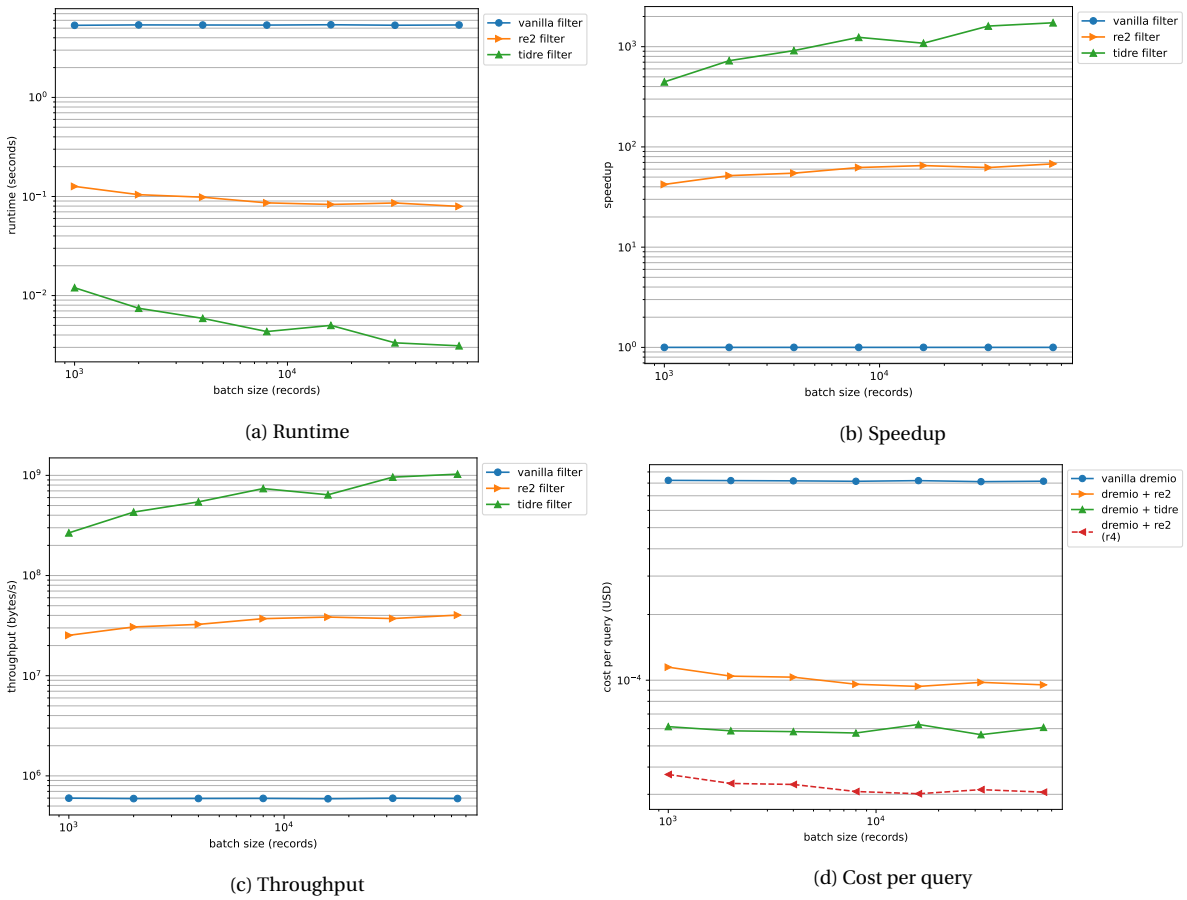
Figure 4.8a shows the runtime of the filter operators. For the RE2 accelerated version of the framework, it was seen that increasing the batch size up to 64,000 records decreases the filter operator runtime by ~ 37%. In the case of the Tidre based filter operator, this effect is more significant and the runtime of the filter operator is almost reduced by an order of magnitude. Apart from the overhead associated to setting up multiple JNI calls, there is an overhead associated to copying data to and from the FPGA accelerator. Smaller batch sizes result in more data copies to and from the accelerator, as there are more recordbatches to evaluate.

The speedup as compared to the vanilla filter operator can be seen in Figure 4.8b. It can be seen that increasing the batch size from 1,000 records to 64,000 records increases the speedup of the Tidre based filter operator from ~ $450x$ to ~ $1750x$. The line showing the speedup of the Tidre based filter operator for increasing batch sizes is steeper than the line showing the speedup of the RE2 based filter operator. This can be explained by the fact that increasing the batch size not only mitigates the overhead associated to the JNI calls, but it also reduces the overhead associated to the data copies to and from the accelerator.

The throughput of the filter operators can be seen in Figure 4.8c. The throughput of filter operator in the vanilla framework remains constant at ~ $600kB/s$ and is not affected by the batch size. The RE2 based filter operator is able to achieve a throughput of ~ $40MB/s$, while the Tidre based filter operator reaches a throughput of ~ $1GB/s$ for the largest batch size. Increasing the batch size for the Tidre based filter operator from 1,000 records to 64,000 records yields an increase in throughput of ~ $2.6x$.

From Figure 4.8d, it can be seen that the batch size has a smaller effect on the cost per query of the Tidre based filter operator as compared to the RE2 based filter operator. This does not align with the expectations, as the runtime of the Tidre filter shows greater decrease for increasing batch size.

It is expected that this can be accounted to the Dremio framework not being able to measure the total execution time of very short running queries accurately. This aligns with the observations made for the input



(a) Runtime

(b) Speedup

(c) Throughput

(d) Cost per query

Figure 4.8: Filter runtime, speedup, throughput, and cost per query for the batch size benchmark on both the RE2 and Tidre accelerated versions of Dremio. The input size is set to 32,000 records.

size benchmark.

4.3.4. Optimizer exploration

In order to increase the impact a given FPGA kernel has, the number of usecases in which it can be deployed should be maximized. This can be achieved in an optimization exploration phase, in which an execution plan is rewritten such that it contains the sequence of target operators of the FPGA kernel.
In this experiment, an additional rule is added to the FPGA acceleration phase that targets all *SQL LIKE* filter operators from which the filter predicate only consists of character sequences and wildcards. This operator is then rewritten as a regular expression filter.

The filter operator as resulting from the query shown in Listing 4.4 is rewritten to a regular expression based filter. The resulting regular expression can be seen in Listing 4.5. The data generator as described in Section 4.2 is configured to produce matching records for this regular expression with a match frequency of 5%.

```
0 SELECT SUM("value_column") FROM "tweet-like-strings.parquet" WHERE "string_column" LIKE '%Taxi%Taxi%'
```

Listing 4.4: SQL query that makes use of the SQL LIKE operator instead of the regular expression based filter operator.

```
0 ".*Taxi.*Taxi.*"
```

Listing 4.5: Equivalent regular expression for the SQL LIKE expression.

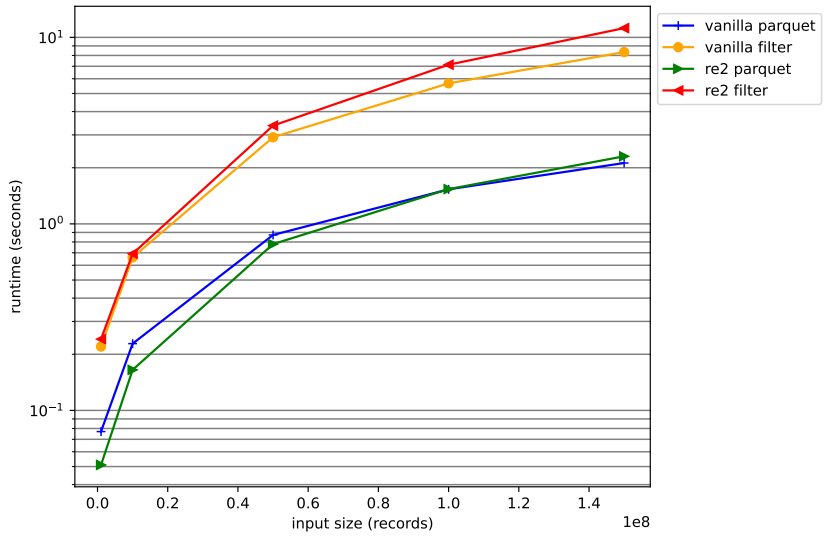


Figure 4.9: Optimizer exploration SQL LIKE to regex

The result of evaluating the resulting execution plan on the RE2 accelerated version of Dremio can be seen in Figure 4.9.
This figure only includes results for the acceleration with the RE2 framework, as there was no FPGA kernel available for this particular regular expression usecase at the time of writing. In this experiment, the vanilla implementation of the *SQL LIKE* filter is much faster than the vanilla regular expression filter implementation. It can be seen that the total query runtime for the RE2 accelerated version of Dremio is slightly higher than that of the vanilla Dremio version. This difference can be explained by the overhead associated to setting up the required JNI calls.

The parquet scan operators for both versions of the framework differ slightly, but this difference is exaggerated by the logarithmic scale of the y-axis.

Even though the RE2 based filter operator has a slightly higher runtime than the filter operator in the vanilla framework, acceleration on an FPGA accelerator could achieve additional speedup. However, as the runtime of the filter operator in the vanilla framework is in the same order of magnitude as that of the RE2 based filter operator, acceleration with an FPGA is expected to yield diminished returns as compared to the regular expression usecase.

4.4. Preliminary conclusion

Dremio's regular expression query is not optimized for the given regular expression usecase. In such a situation, moving to a highly optimized native implementation of the specific operator can already show a significant performance increase. In the case where the evaluation of a regular expression filter is offloaded to the RE2 framework, a speedup of $\sim 66x$ can be achieved. FPGA accelerators can achieve an additional speedup of $\sim 26x$, resulting in a total speedup of $\sim 1750x$ as compared to the vanilla framework.

Increasing the batch size up to 64,000 records in a single node setup increases the throughput of the Tidre accelerated version of the framework by up to $2.6x$. This can be attributed to a reduced overhead associated with memory copies to and from the accelerator as well as a reduced overhead associated with setting up JNI calls. Other frameworks need to be considered in order to draw conclusions regarding batch sizes greater than 64,000 records.

Finally, it is possible to rewrite the execution plan during an exploration phase such that the impact of a given FPGA kernel is increased. In practice, the performance gain for these new usecases is expected to be lower than the gain for the original target usecase of the FPGA kernel.

Accelerated kernels are typically written for compute intensive operators. Similar operators that can be rewritten to these target operators of the FPGA kernel do not need to have an equally high computational in-

tensity. As the evaluation of these operators on a CPU is performed faster than their computationally intense counterparts, the performance gain as achieved by FPGA acceleration is diminished.

5

Dask integration

Dask is another big data framework and is written in Python [81]. Dask is very flexible and includes the *DataFrame API*, which is based around the popular Pandas framework [92]. The underlying data is stored based on the Apache Arrow in-memory format. Unlike the Arrow Java API, all Arrow buffers as allocated by the Arrow Python API are aligned to 64 bytes. Another difference with Dremio is that Dask allows the batch size to be set to an arbitrary number of records, where Dremio is limited to a mere 64,000 records per recordbatch. Integration with Dask is expected to have a high impact in the big data community, as the userbase of Dask is significantly greater than that of Dremio.

Dask SQL is a module developed to parse and optimize SQL queries such that they can be planned and executed on the Dask engine. At the time of writing however, this module is still in development and does not support the full SQL syntax. Additionally, it is not optimized, resulting in execution plans from which the performance is not comparable to Dremio. An alternative is to map SQL queries to the Pandas API, which is fully supported in Dask. This latter approach is chosen over the Dask SQL module.

5.1. Implementation

The system architecture of Dask differs from Dremio. The most significant differences are the representation of the execution plan and the way in which this plan can be optimized.

The execution plan in Dask is called a *task graph* and is stored in a Python dictionary. Each operation in the graph has an associated key in this dictionary. The value that is associated to that key holds information regarding the operator itself. This includes its operands, as well as the keys of all operators it depends on.

Optimization and manipulation of this task graph can be done through Dask's optimization package. This package allows the task graph to be transformed using simple rewrite rules. At the time of writing, this optimization engine is however not as far developed as Apache Calcite's optimization engine. Transforming simple task graphs that result from toy examples in Dask presents no difficulties. However, moving to more complex task graphs can be complicated. These complex graphs can be the result of performing operations on a parquet file with multiple chunks. The resulting graph is known in Dask as a *high-level graph*. High-level graphs provide an additional layer of abstraction on top of a task graph, such that all operations in the graph are grouped by the chunk of data on which they operate. The optimization package does not directly support the transformation of high-level graphs. Since these graphs are implemented using a Python dictionary, normal Python operations can be used to manipulate the high-level graph.

As a result, the implementation of the accelerated version of Dask differs from the implementation of the accelerated version of Dremio in these places. Where possible, implementations that can remain the same are left unchanged. The following subsections describe the design and implementation of the accelerated version of Dask.

5.1.1. Architectural details

The implementation of the accelerated version of Dask follows a similar structure as the implementation of the accelerated version of Dremio. First, an FPGA acceleration planning stage is introduced. This planning

stage is implemented in a different way than the acceleration planning phase in Dremio, as it is not based on Apache Calcite optimization rules. Instead, it makes use of Dask's own optimization module.

Second, different accelerated operators are implemented. These operators are used by the FPGA acceleration planning stage to offload the computation of filter operators to Tidre as well as the RE2 library. A distinction is made between operators with Arrow buffers aligned to 64 bytes and operators with unaligned Arrow buffers.

Furthermore, the *native-op package* remains largely the same as in the accelerated version of Dremio. The only difference is the added functionality to convert a selection vector, as returned by the Tidre interface, to a bitmap. Lastly, instead of using the JNI to invoke this native package, Python bindings are used. This is achieved by means of the Pybind11 library [93].

These additions can be seen in Figure 5.1. Individual components are further discussed below.

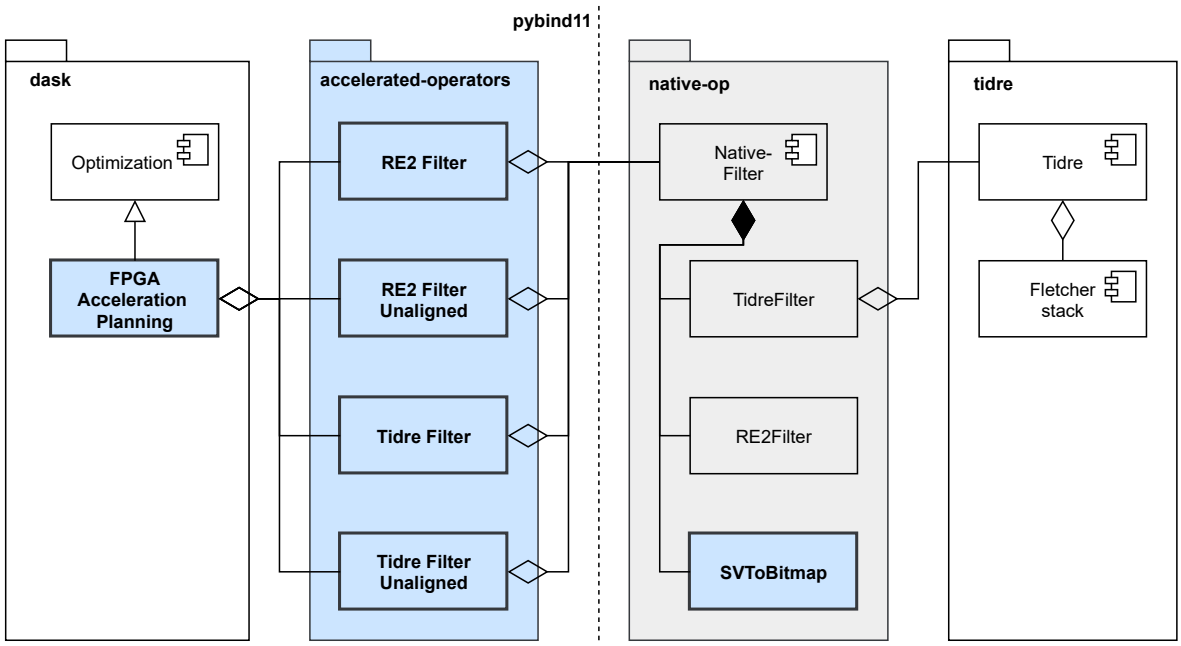


Figure 5.1: Overview of the accelerated version of Dask. The contributions of this project are shown in blue with bold borders. Contributions that remain unchanged as compared to the accelerated version of Dremio are grayed out. The diagram is inspired by UML but does not follow all UML practices.

5.1.2. FPGA acceleration planning

As mentioned, the datastructure of a task graph in Dask differs from that in Dremio. The most important dependency of the FPGA acceleration planning stage is the *SubgraphCallable* class. Each operator in the task graph holds a *SubgraphCallable* which defines how the operator is to be executed. The structure of this class, in pseudocode, can be seen in Listing 5.1. This pseudocode lists an instance of the *SubgraphCallable* object for a *string match* operator in Dask.

On line 1, the *dsk* attribute holds a dictionary that defines the execution of the string match operator. This string match operator is used to perform the regular expression filtering in Dask. On line 3 and 4, it can be seen that the functions *apply()* and *apply_and_enforce()* are specified. These are funtions in the Dask utils and Dask core packages respectively that handle the execution of most operators. Line 5 specifies which indices of the input tuple are to be passed as arguments to the operator. The input tuple itself is given on line 14, indicated by the *inkeys* attribute. The interesting part of this *SubgraphCallable* is shown on lines 6 to 12. Here, the specific function that is to be executed and the metadata of the result can be seen.

When accelerating this part of the execution plan, the metadata of the result remains unchanged. The function that produces this result is however substituted for the accelerated function.

```
subgraph_callable:
    dsk = {
        'str-match-7357e89bd7a48e889528f4dcac2a81e6': (
            <function apply at 0x7fe38b29a5e0>,
            <function apply_and_enforce at 0x7fe370df2a60>,
            ['_0', '_1', '_2', '_3', '_4'],
            (
                <class 'dict'>,
                [
                    ['_func', <function Accessor._delegate_method at 0x7fe370e44700>],
                    ['_meta', Series([], Name: string, dtype: bool)]
                ]
            )
        )
    }
    inkeys = ('_0', '_1', '_2', '_3', '_4', '_5')
    name = 'subgraph_callable'
    outkey = 'str-match-7357e89bd7a48e889528f4dcac2a81e6'
```

Listing 5.1: Pseudocode of the str-match operator's SubgraphCallable.

The FPGA acceleration planning stage therefore unwraps the *SubgraphCallable* object of the operation to be accelerated and substitutes the *_func* variable for the accelerated operator.

5.1.3. Selection vector to bitmap

Another difference between the Dask and Dremio frameworks is the way in which records that match the filter predicate are recorded. Where Dremio makes use of a selection vector that holds all indices of the matching records, Dask works with a bitmap. This bitmap holds a boolean value for each record that speciefies whether it matches the predicate.

By default, the Tidre framework returns a selection vector. This is a more efficient way of transferring the matching records between the accelerator and the host machine in case the fraction of matching records is limited to a small amount (a sparse matching recordbatch). When all records in the recordbatch match the filter predicate (a dense matching recordbatch), it is more efficient to transfer a bitmap instead of a vector that holds all indices in the recordbatch. As seen in Section 4.2, only 5% of the records in the dataset of the regular expression usecase match the filter predicate. Therefore, transferring a selection vector between accelerator and host machine is a solid choice.

However, for Dask to function correctly, this selection vector needs to be converted to a bitmap. This is done in the native-filter package on the CPU. A simple algorithm is constructed that loops over all matching indices and builds the equivalent bitmap. Special care needs to be taken to determine the byte and bit position within this byte that corresponds with the matching index. The resulting algorithm can be seen in Listing 5.2.

```
int matching_index;
int sv_byte;     // The byte in the selection vector buffer
int sv_bit;      // The specific bit within this byte that corresponds to the matching index

// Loop over all matching indices as returned by the Tidre framework
for (int i = 0; i < number_of_matches; i++) {
    matching_index = matching_indices[i];

    // Compute the corresponding byte and bit in the selection vector
    sv_byte = floor((float) matching_index / (float) 8);
    sv_bit = matching_index % 8;

    // Set this bit to 1 to indicate a matching record
    out_values[sv_byte] |= 1UL << sv_bit;
}
```

Listing 5.2: Simple algorithm to convert a selection vector to a bitmap.

The computation of the equivalent byte and bit of the matching index on lines 9 and 10 is independent of the number of indices in the selection vector. It can be seen that the time-complexity of this algorithm is therefore equal to $O(n)$, where n indicates the number of matching records. With this linear time-complexity,

it is assumed that this conversion is favorable to constructing the bitmap in hardware and introducing a larger memory transfer between accelerator and host. The statistics of the underlying dataset determine whether this assumption holds true or not. Finding the statistical details that govern which solution is optimal is left for future research.

5.1.4. Accelerated operators

Four accelerated operators are implemented in the accelerated version of Dask. Each of these operators performs the evaluation of a regular expression filter. Similar to which is the case in the accelerated version of Dremio, a distinction is made between evaluation using Google's RE2 library and evaluation using Tidre. Additionally, separate implementations are provided for filter operators with unaligned Arrow buffers

RE2 evaluation Similar to the accelerated version of Dremio, the RE2 library is used for the evaluation of the regular expression on the CPU. The regular expression string is passed as an argument to the accelerated operator and is forwarded to the native operator through the Pybind11 package.

The RE2 accelerated filter operators are mainly concerned with unwrapping the incoming Pandas object such that its underlying Arrow buffers are exposed. The pointers to these buffers are then passed as arguments to the native operator.

By default, unlike Arrow buffers as allocated with Arrow's Java API, Arrow buffers that are allocated using the Python API are aligned to 64 bytes. In order to investigate the effect that this alignment difference has on the performance of the resulting system, an unaligned version of the operator is implemented in the accelerated version of Dask. The unaligned Arrow buffers are obtained by prepending some garbage data to the the incoming Pandas object. This garbage data is then discarded by taking a slice of the underlying Arrow buffers that only contain the original data. In this way, the data of the Arrow buffers is conserved while the starting memory address is offset by the length of the garbage data. It was found that this conversion takes less than 1ms. However, the duration of this conversion has no effect on the measured duration of the filter operator, as it is performed outside of the timed code.

After evaluation on the CPU using the RE2 library, the resulting Arrow buffer is wrapped in a Pandas object and returned to Dask.

Tidre evaluation Just like the RE2 accelerated operators, the Tidre operators unwrap the incoming Pandas object and extract the pointers to the underlying Arrow buffers. These pointers are passed to the native operator such that they can be evaluated using the Tidre framework.

The unaligned version of this operator changes the alignment of the Arrow buffers in a similar way as to how it is done in the RE2 accelerated operator.

After evaluation of the regular expression on the Tidre framework, the resulting Arrow buffer is wrapped in a Pandas object and returned to Dask.

The sequence diagram that illustrates the evaluation of the string match operator using this Tidre filter implementation can be seen in Figure 5.2.

The workings of the *NativeFilter* and *Tidre* packages are identical to their counterparts in the acccelerated verion of Dremio. As mentioned, the *TidreFilter* operator wraps the resulting Arrow buffer into a Pandas object. This is done using the *maybe_wrap_pandas()* method. The *apply_and_enforce()* method in Dask core then calls the *_rename()* method to enforce the metadata of the resulting dataframe.

5.2. Experimental setup

The experimental setup is largely the same as in Section 4.2. This section is used to describe the differences between these two setups.

The codebase of the accelerated version of Dask can be found on GitHub [94]. This repository contains a detailed installation guide with instructions on how to deploy this framework on AWS.

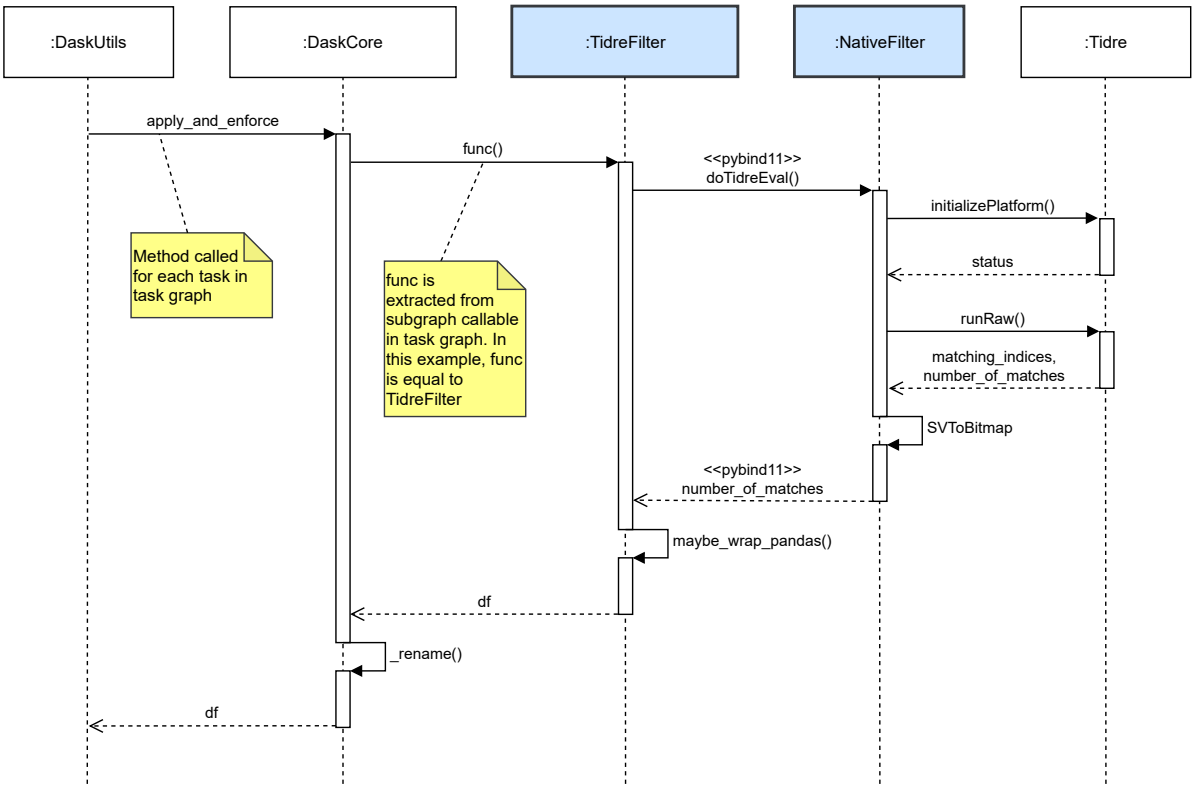


Figure 5.2: Sequence diagram illustrating the interactions of the accelerated operator in Dask.

5.2.1. Regular expression usecase

The selected regular expression usecase remains the same as in Subsection 4.2.1. In the case of Dask, the SQL query is ported to the Pandas API. The result can be seen in Listing 5.3.

```
0  regex = '
       .*[tT][eE][rR][aA][tT][iI][dD][eE][ \t\n]+
2      [dD][iI][vV][iI][nN][gG][ \t\n]+
       ([sS][uU][bB])+[sS][uU][rR][fF][aA][cC][eE].*
4  '
   res = df[df["string"].str.match(regex)]["value"].sum()
```

Listing 5.3: Chosen SQL query ported to the Pandas API.

In this code listing, the *df* variable on line 5 holds the dataframe that results from Dask's *read_parquet()* method.

5.2.2. Input size benchmark

The parameters of the input size benchmark for the accelerated version of Dask span a larger range. The input sizes range from 1,000 records to 4,096,000 records, increasing by factors of two. The range is widened such that it accomodates Dask's support for larger batch sizes. This batch size is set to 1,024,000 records.

The input size benchmark therefore spans two interesting regions; the input datasets of 1,000 records up to 1,024,000 records can be evaluated in a single recordbatch, while the input datasets of 2,048,000 up to 4,096,000 records have to be evaluated in multiple.

5.2.3. Batch size benchmark

The configurations of the batch size benchmark are changed such that it includes greater batch sizes. These batch sizes range from 64,000 records up to 8,192,000 records per recordbatch. The input size is kept constant to 4,096,000 records.

Therefore, the smaller batch sizes require the dataset to be evaluated in multiple recordbatches, while the largest batch sizes of 4,096,000 and 8,192,000 records allow the dataset to be evaluated in a single recordbatch.

The largest batch size is interesting, as it is bigger than the input size and is therefore not expected to behave different as compared to the batch size of 4,096,000 records.

5.2.4. Measurement setup

The first run for each configuration in an experiment is discarded in order to mitigate cache warming effects. After this first run, ten consecutive runs are performed and the measured runtimes are averaged to obtain the final result for that configuration. The experiments measure the runtime of the filter operator.

5.3. Results

The results in terms of the input size and batch size benchmarks are presented in the following subsections.

5.3.1. Accelerating the query

The new FPGA acceleration planning stage is used to transform the task graph in the accelerated version of Dask. This planning stage substitutes the string match operator for its accelerated version. The original task graph and the accelerated task graph can be seen in Figure 5.3.
The *accelerated str-match* operator can either offload the evaluation to the RE2 based filter implementation or to the Tidre based filter implementation.

5.3.2. RE2 acceleration

This section shows the results of accelerating Dask with the RE2 library. Both the 64 byte aligned version as well as the unaligned version of the operator are considered. All measurements are performed on an *f1.2xlarge* instance.

Input size benchmark Dask's vanilla implementation of the string match operator is based on the implementation in Pandas, which in turn uses Secret Labs' regular expression engine in CPython [95]. This implementation is very similar to the Google RE2 implementation. Figure 5.4 shows the results for the input size benchmark. The x-axis shows the number of records in the input dataset, while the y-axis shows the runtime in seconds.
This is a big difference as compared to the results of the RE2 acceleration in the accelerated version of Dremio. In this case, using the RE2 library does not yield lower runtimes as compared to the vanilla filter implementation. It can be seen that for input sizes of 1,000 and 2,000 records, the runtime of the RE2 accelerated is slightly higher than for the vanilla filter operator. It is expected that this is the case due to an overhead associated with the invocation of the native operator package through the Pybind11 library. For low input sizes, this overhead could be significant when compared to the runtime of the filter evaluation. For higher input sizes, this evaluation becomes dominant over the aforementioned overhead.

Batch size benchmark The results of the batch size benchmark can be seen in Figure 5.5. The x-axis shows the number of records in the recordbatch, while the y-axis shows the runtime in seconds.

For this large input size of 4096,000 records, it becomes clear that the RE2 accelerated version of the filter operator is only able to reduce the runtime by $\sim 6\%$. As expected, increasing the batch size does not have a significant impact on the runtime of the filter operator.

5.3.3. Tidre acceleration

The results of accelerating Dask with Tidre are presented in this section. These results are compared with both the vanilla Dask implementation as well as the RE2 accelerated implementation. All measurements are performed on an *f1.2xlarge* machine type conform the measurement setup in Section 4.2.

Input size benchmark The results of the input size benchmark for vanilla Dask, the RE2 accelerated version of Dask, and the Tidre accelerated version of Dask can be seen in Figure 5.6. The runtime of the filter operator, speedup relative to the vanilla operator, throughput, and cost per operation can be seen in Figure 5.6a, Figure 5.6b, Figure 5.6c, and Figure 5.6d respectively.

The speedup, throughput, and cost per operation are computed based on the measured runtime of the filter operator.

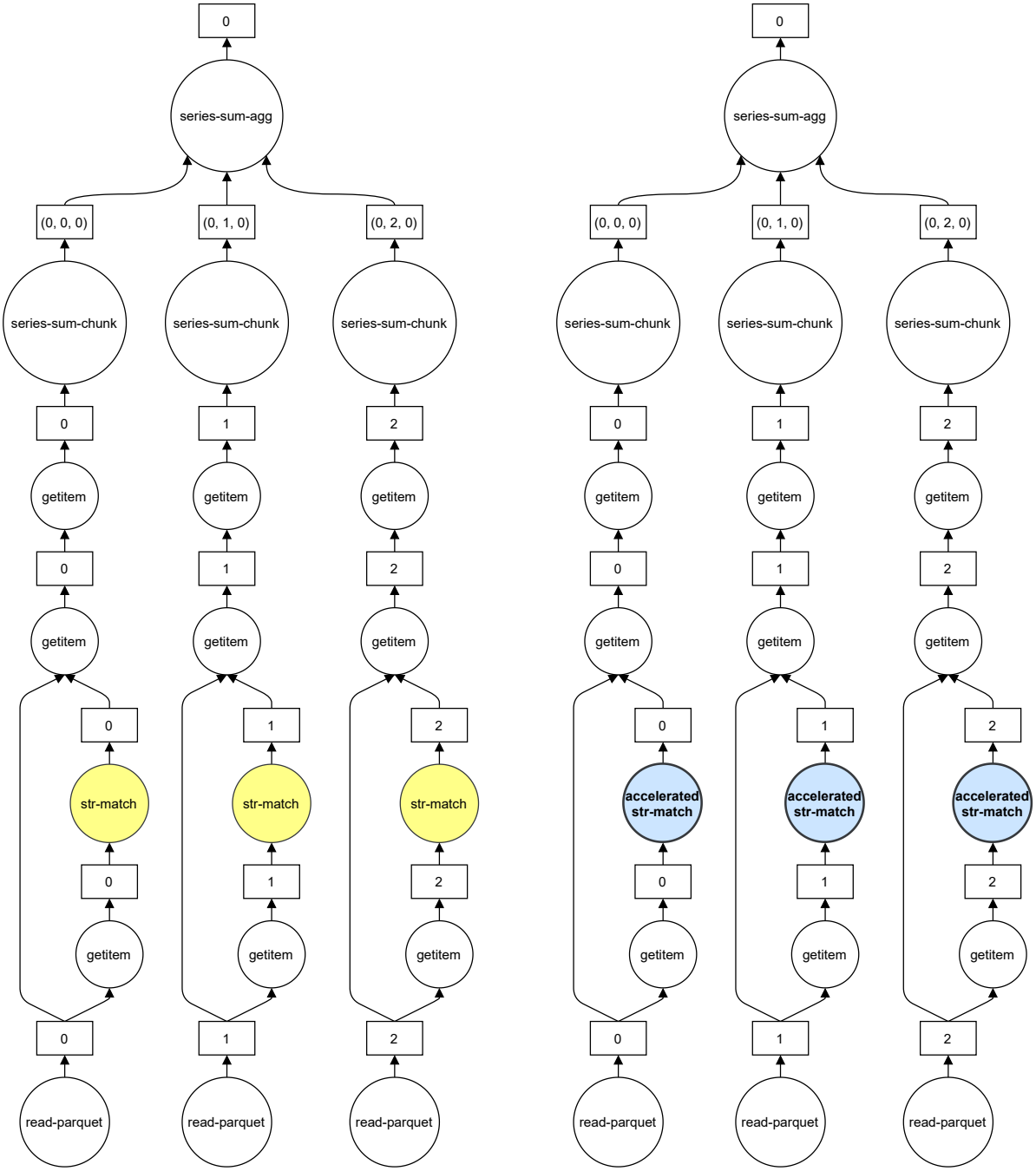


Figure 5.3: Task graph for the regular expression usecase in Dask. In this example, the input size is set to 3,000,000 records with a parquet chunksize of 1,000,000 records.

From Figure 5.6a, it can be seen that the runtime of the Tidre based filter operator is roughly two orders of magnitude lower than the runtime of the vanilla filter and RE2 based filter operators. Furthermore, it can be seen that the 64 byte aligned version of the Tidre based filter operator is $\sim 2.5x$ faster as compared to the unaligned operator.

A dashed line is added to the figure to signify the runtime of the Tidre based filter operator when a single record is processed. This provides an indication of the overhead associated with offloading the evaluation of the filter operator to the FPGA accelerator. Therefore, this line can be seen as a proxy for the absolute minimum runtime that is achievable with this implementation.

The speedup of the filter operators as compared to the filter operator in the vanilla Dask framework is shown

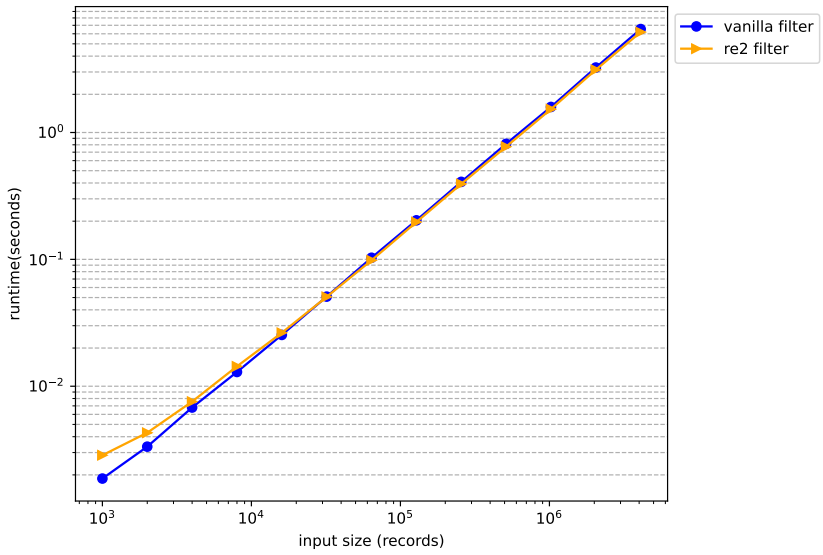


Figure 5.4: Filter operator runtime for the input size benchmark on the RE2 accelerated version of Dask. The batch size is set to 1,024,000 records.

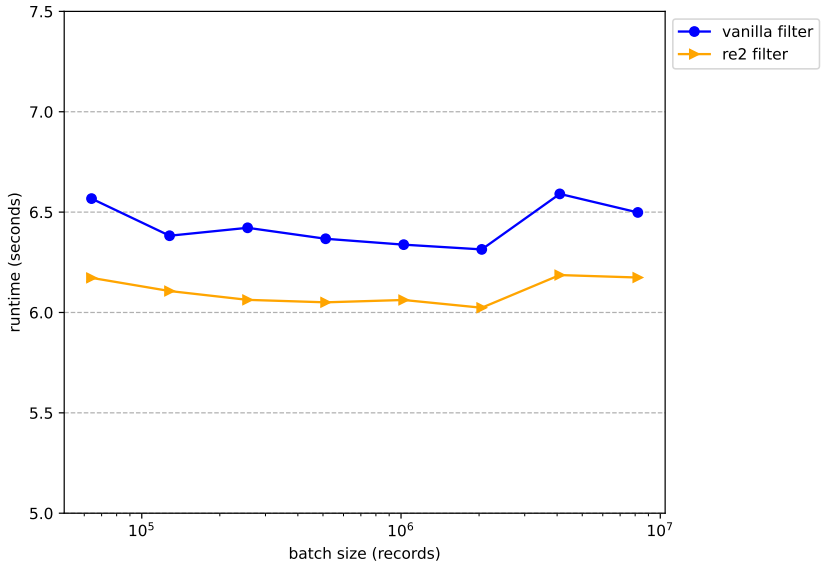
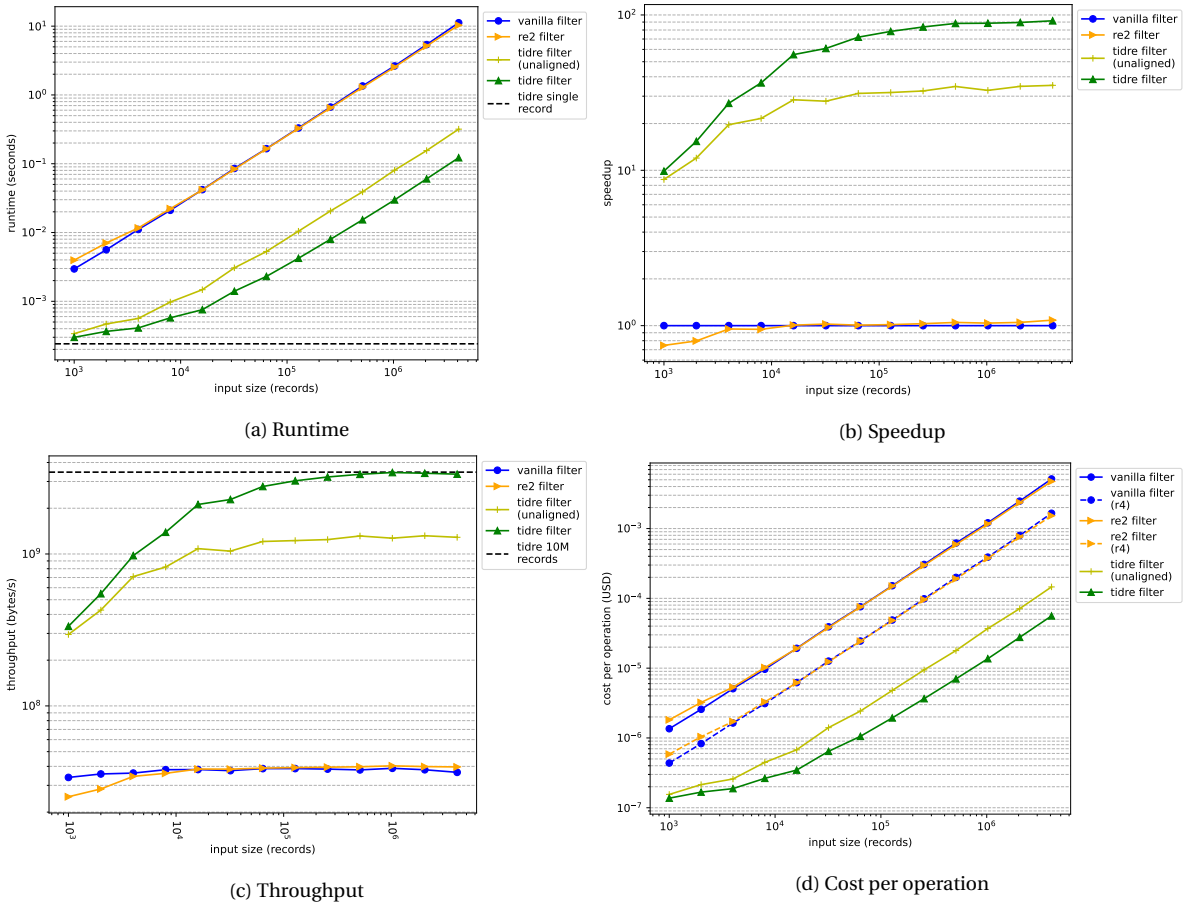


Figure 5.5: Filter operator runtime for the batch size benchmark on the RE2 accelerated version of Dask. The input size is set to 4096,000 records.

in Figure 5.6b. As seen in the previous section, the RE2 based filter operator has larger runtimes for small input sizes as compared to the vanilla filter. This is reflected in this figure, as the speedup of this RE2 based filter operator is smaller than one for input sizes under 16,000 records.

The Tidre based filter operator achieves a speedup of ~ $92x$ as compared to the vanilla filter implementation. The unaligned version of this Tidre based filter operator achieves a speedup of ~ $35x$. It can be seen that this speedup increases for increasing input sizes up to roughly 512,000 records, after which it remains relatively stable. It is therefore suspected that an input size of 512,000 records is the configuration for this usecase at which the speedup related to the evaluation time of the filter operator dominates the overhead associated to memory copies to and from the accelerator.

The throughput of these filter operators can be seen in Figure 5.6c. The vanilla filter implementation and the RE2 based filter implementation are limited to a throughput of ~ $35-40 MB/s$ respectively. The Tidre



(a) Runtime

(b) Speedup

(c) Throughput

(d) Cost per operation

Figure 5.6: Filter operator runtime, speedup, throughput, and cost for the input size benchmark on the Tidre accelerated version of Dask. The batch size is set to 1,024,000 records.

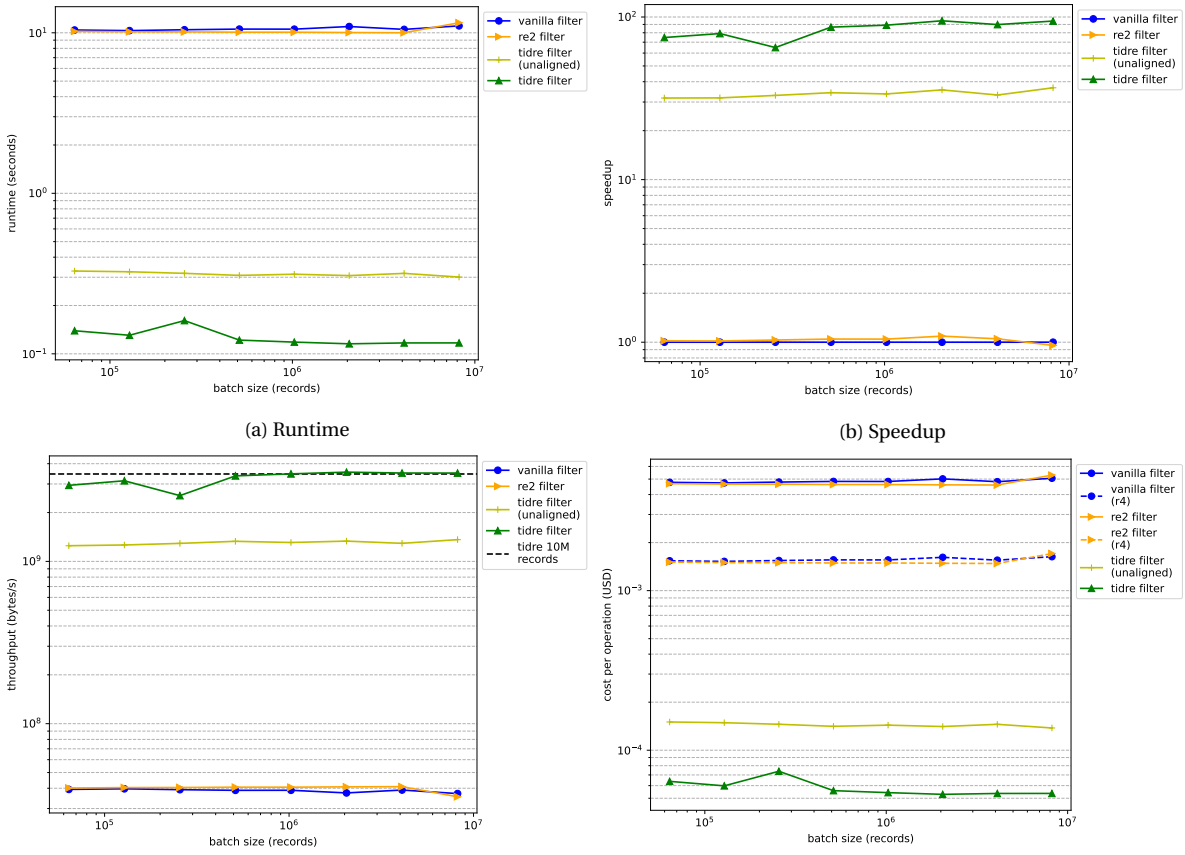
based filter operator however reaches a throughput of $\sim 3.4 GB/s$, while the unaligned version of this operator reaches $\sim 1.3 GB/s$.

A dashed line is added to the figure to indicate the throughput of the Tidre based filter operator if an dataset with 10,000,000 records is processed in a single recordbatch. This line is seen as a proxy for the maximum achievable throughput of the Tidre based filter operator. It can be seen that with a batch size of 1,024,000 records and an input size of equal size, the Tidre based filter operator is able to reach this maximum throughput.

The cost per operation can be seen in Figure 5.6d. All measurements are run on an *f1.2xlarge* instance type. Projections are added that estimate the resulting cost per operation of running the vanilla filter implementation and the RE2 based filter implementation on an *r4.2xlarge* instance.

It can be seen that the cost per operation for both the aligned and unaligned versions of the Tidre based filter operator is always lower than that of the vanilla based filter operator. This remains true regardless of whether the vanilla implementation is run on an *f1.2xlarge* or on the more economical *r4.2xlarge* machine type.

Batch size benchmark The results of the batch size benchmark for vanilla Dask, the RE2 accelerated version of Dask, and the Tidre accelerated version of Dask can be seen in Figure 5.7. The runtime of the filter operator, speedup relative to the vanilla operator, throughput, and cost per operation can be seen in Figure 5.7a, Figure 5.7b, Figure 5.7c, and Figure 5.7d respectively. The filter operator runtime and throughput of the Tidre accelerated implementations are shown again in Figure 5.8. This figure does not contain the runtime and throughput of the other implementations of this filter operator. The speedup, throughput, and cost per operation are computed based on the measured runtime of the filter operator.



(a) Runtime

(b) Speedup

(c) Throughput

(d) Cost per operation

Figure 5.7: Filter operator runtime, speedup, throughput, and cost for the batch size benchmark on the Tidre accelerated version of Dask. The input size is set to 4,096,000 records.

The runtime of the filter operators can be seen in Figure 5.7a. It can be seen that the runtime of the Tidre based filter operator is roughly 2 orders of magnitude lower than the runtime of the vanilla filter implementation. This hold true regardless of the chosen batch size. It can be seen that the runtimes of both the aligned and unaligned Tidre operators only decreases marginally for increasing batch sizes. In the case of the aligned operator, increasing the batch size from 64,000 to 512,000 records, the runtime is decreased by ~ 12%. For batch sizes above 512,000 records, the reduction is negligible.

This can be seen more clearly in Figure 5.8a, in which only the runtimes of the Tidre based filter implementations are shown. A dashed line is added that shows the runtime of the aligned Tidre based filter operator when an input dataset of a single record is processed.

A peak can be seen at the runtime of the aligned version of the Tidre based filter operator for a batch size of 256,000 records. It is expected that this can be attributed to an outlier in the measurements. However, this cannot be confirmed, as only the average runtime of the 10 consecutive runs is recorded.

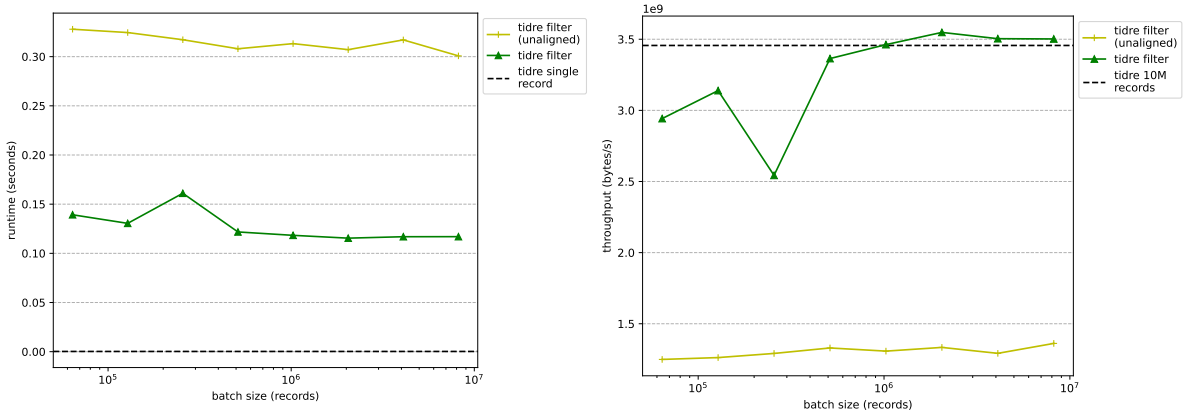
The speedup of these filter operators can be seen in Figure 5.7b. As expected after the discussion of the last figure, it can be seen that the speedup of the Tidre based filter operators only increases marginally for increasing batch sizes. For batch sizes above 512,000 records, this speedup remains more or less stable.

This same conclusion can be drawn when considering the throughput of these filter operators in Figure 5.7c and Figure 5.8b.

Again, a dashed line is added that acts as a proxy for the maximum achievable throughput. This throughput is achieved by evaluating a dataset containing 10,000,000 records in a single recordbatch. It can be seen that the 64 byte aligned version of the Tidre based filter operator exceeds this estimated maximum throughput of $\sim 3.4 GB/s$ for batch sizes greater than or equal to 512,000 records.

The fact that this estimate for the maximum throughput is exceeded by a small amount could indicate that after 10 runs, there are still fluctuations in the runtime of the filter operator. Potentially, more consecutive runs are require to obtain a better estimate for the maximum achievable throughput of the Tidre based filter operator.

Finally, Figure 5.7d shows the cost per operation for the filter operators. As seen in the input size benchmark, both the aligned and unaligned versions of the Tidre based filter operator have a lower cost per operation than the vanilla filter implementation. This cost per operation reduces marginally for increasing batch sizes.



(a) Runtime

(b) Throughput

Figure 5.8: Tidre filter operator throughput for the batch size benchmark on the Tidre accelerated version of Dask. The input size is set to 4,096,000 records.

5.4. Preliminary conclusion

Operators in the vanilla framework that are already optimized for performance do not benefit from moving to a native implementation such as the Google RE2 library for regular expression operators. In these cases, FPGA accelerators can be leveraged to increase the performance of the system. For the regular expression usecase, FPGA acceleration is able to achieve a speedup of the filter operator of $\sim 92x$ as compared to the vanilla filter implementation.

Memory alignment of the underlying Arrow buffers is an important parameter and can be attributed to roughly $2.5x$ of the aforementioned speedup. It is therefore important to align the Arrow buffers of operators that are to be accelerated on an FPGA to 64 bytes.

When the runtime for different batch sizes is considered, only a marginal decrease is found for increasing batch sizes above 64,000 records. This decrease is significantly lower when compared to the experiments as performed on the accelerated version of Dremio, in which increasing the batch size from 1,000 records up to 64,000 records decreases the runtime of the Tidre accelerated filter operator by $\sim$ 37%. In this implementation, it is found that further increasing the batch size to 512,000 records only yields a additional decrease in runtime of $\sim$ 12%. It is therefore concluded that increasing the batch size for this usecase above 64,000 records yields diminishing returns.

Lastly, it is found that the maximum throughput is achieved when the input size is equal to or larger than the batch size. However, increasing the batch size above 512,000 records does not yield additional throughput. It is expected that this threshold is highly dependent on the size of the data that is to be copied to and from the accelerator per recordbatch, as well as on the computational complexity of the evaluation of the filter operation. This threshold can therefore change for different usecases.

6

Dask distributed integration

Dask distributed is a separate version of Dask that is specifically designed for distributed computing. According to its documentation, Dask distributed extends the Dask API such that it can be deployed in moderate size compute clusters [96]. All characteristics of the underlying Dask framework remain the same, which makes Dask distributed an excellent candidate for acceleration.

The most significant differences with vanilla Dask are the implementation of the scheduler and worker classes. The new scheduler implementation is capable of distributed scheduling and accounts for data locality. The distributed worker classes support peer-to-peer data sharing, which allows them to resolve data shuffles without the need for central coordination.

6.1. Implementation

Dask distributed is a version of Dask that can be run on a cluster. This provides the potential for added compute power and paralellization, but at the same time it also adds additional complexity. To orchestrate these computations on a distributed cluster, different types of nodes are defined. The *client* node is controlled by the end-user and is used for all interactions with the cluster. The client communicates with the scheduler node, which keeps track of all *worker* nodes in the cluster. These worker nodes are held in an abstraction known as the *worker pool*. The scheduler performs the planning and optimization steps for the queries it receives from the client node. The resulting task graph is then distributed amongst the workers in its worker pool.

The accelerated version of this framework is deployed as an acceleration aware worker setup, as identified in Section 3.4. This differs from the single node deployments as used for the accelerated version of Dremio and the accelerated version of vanilla Dask. The following sections elaborate on the design and implementation of the accelerated version of Dask distributed.

6.1.1. Architectural details

An *accelerated worker* node is implemented in order to leverage FPGA accelerators within the Dask cluster. From the scheduler's point of view, the behaviour of this new worker implementation is identical to that of the original workers in the framework. The acceleration is therefore transparent to the scheduler node.

The contributions that are made in order to realize the accelerated version of Dask distributed can be seen in Figure 6.1. The *AcceleratedWorker* class is the only new implementation. This class depends on the existing packages as implemented in the accelerated version of vanilla Dask. The subsection below describes the implementation of this new *AcceleratedWorker* class.

6.1.2. Accelerated worker

The accelerated worker implementation is the only new contribution required to accelerate Dask distributed. The FPGA acceleration planning stage and the accelerated operator implementations can be imported from the accelerated version of vanilla Dask.

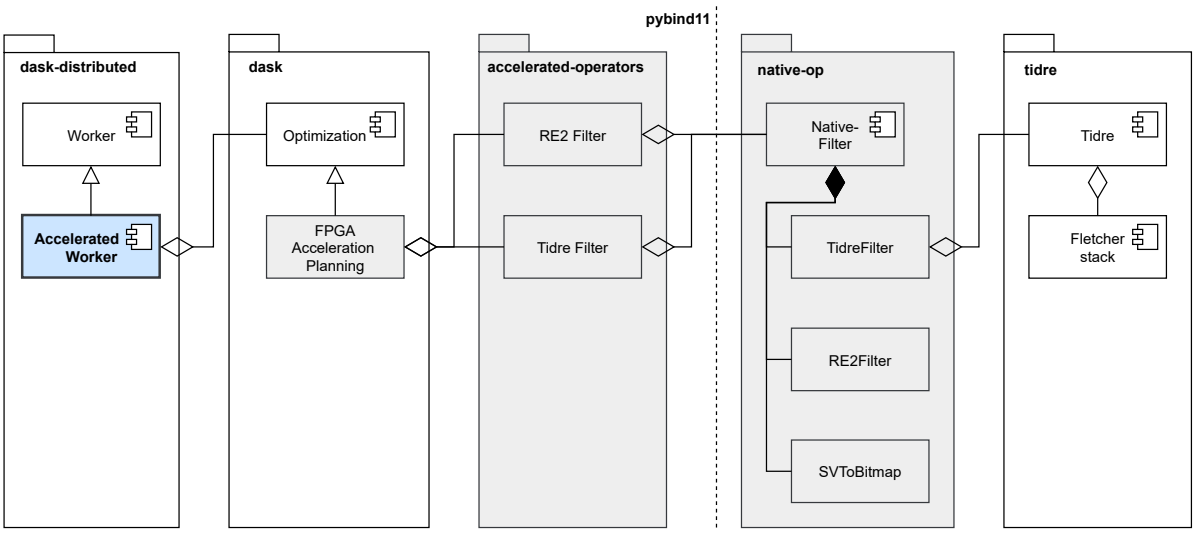


Figure 6.1: Overview of the accelerated version of Dask distributed. The contributions of this project are shown in blue with bold borders. Contributions that remain unchanged as compared to the accelerated version of vanilla Dask are grayed out. The diagram is inspired by UML but does not follow all UML practices.

This new accelerated worker is designed to be run in the worker pool of a Dask distributed cluster, as can be seen in Figure 6.2.

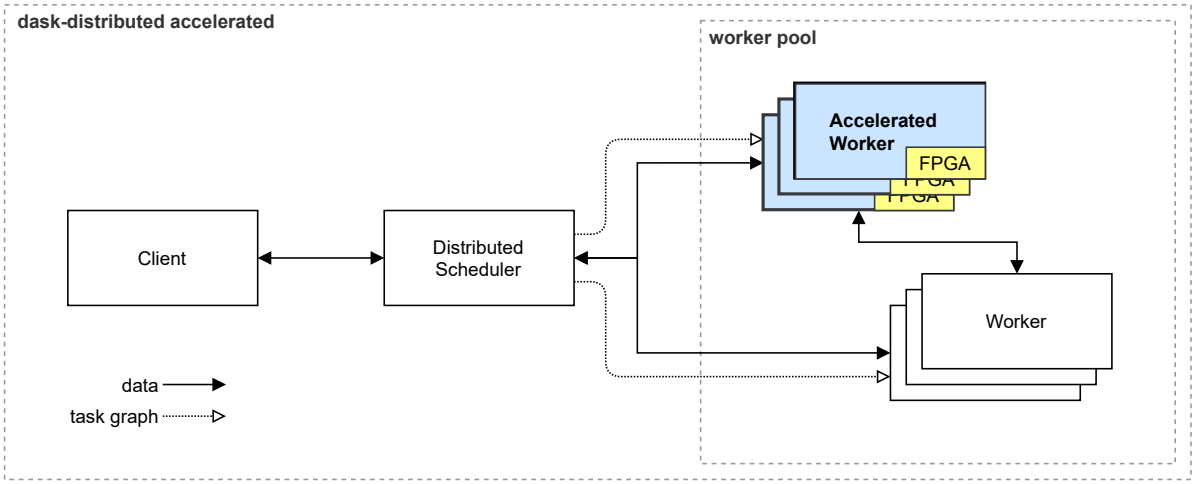


Figure 6.2: System architecture of Dask distributed including the accelerated worker nodes.

The client node can submit a dataset to the cluster via the scheduler, such that it is scattered over the worker pool. Additionally, this connection is used to submit queries. The scheduler plans these queries and sends the resulting task graph to the worker nodes in the worker pool. This task graph is not send as a whole, but rather, individual tasks are submitted to the workers.

The worker nodes then execute these tasks, which can be seen as subgraphs of the original task graph. The worker nodes send data between each other in order to satisfy the missing dependencies of these tasks. The accelerated worker node implementation performs the FPGA acceleration stage on all incoming subgraphs of the task graph. Therefore, only accelerated worker nodes require an FPGA accelerator to be installed, as the vanilla worker implementations execute the original task subgraphs without any accelerated operators.

The *AcceleratedWorker* class extends the default *Worker* class in Dask distributed. The accelerated worker overrides the *add_task()* method of the parent class. This method is invoked every time the scheduler submits a new task to the worker.

The new implementation of the *add_task()* method checks to see if any key in the newly submitted sub-

graph matches an operation for which there is an accelerated version. If a key matches, the underlying *_func* variable is changed to the implementation of the specific accelerated function. This is done using the same implementation of the FPGA acceleration planning stage as used in the accelerated version of vanilla Dask. The parent implementation of the *add_task()* method is then called with the new accelerated subgraph as its argument.

6.2. Experimental setup

The experimental setup remains largely the same as the setup in Section 5.2. The regular expression usecase and selected dataset remain unchanged. The differences between the two setups are described below.

The codebase of the accelerated version of Dask distributed can be found on GitHub [97]. This repository contains a detailed installation guide with instructions on how to deploy this framework on AWS.

6.2.1. Cluster setup

Unlike the previous two implementations which are deployed on a single node, the accelerated version of Dask distributed is deployed on a cluster. All worker nodes in the cluster, both vanilla workers and accelerated workers, run on an *f1.2xlarge* instance. The scheduler is deployed on an *m4.2xlarge* instance.

The cluster that is provisioned for these experiments contains three worker nodes in the worker pool. Therefore, there are four configurations of this cluster; each with a different fraction of accelerated workers in the pool. Because of the size of the worker pool is set to three, there can be 0, 1, 2, or 3 accelerated workers in the pool. In each configuration, a number of vanilla workers is added such that the total number of workers in the pool is equal to three.

Each benchmark is run for all configurations of this cluster.

6.2.2. Input size benchmark

The parameters of the input size benchmark for the accelerated version of Dask distributed range from 256,000 records to 4,096,000 records, increasing by factors of two. The batch size is fixed to 1,024,000 records.

This benchmark therefore spans two interesting regions; the input datasets of 256,000 records up to 1,024,000 records can be evaluated in a single recordbatch, while the datasets of 2,048,000 up to 4,096,000 records have to be evaluated in multiple.

6.2.3. Batch size benchmark

The configurations of the batch size benchmark are left unchanged. These batch sizes range from 64,000 records up to 8,192,000 records per recordbatch. The input size is kept constant to 4,096,000 records.

Therefore, the smaller batch sizes require the dataset to be evaluated in multiple batches, while the largest batch sizes of 4,096,000 and 8,192,000 records allow the dataset to be evaluated in a single batch. The largest batch size is interesting, since it is bigger than the input size and is therefore not expected to behave different as compared to the batch size of 4,096,000 records.

6.2.4. Measurement setup

Similar to the previous experimental setups, the measurements from the first run are discarded to mitigate cache warming effects. Next, 9 consecutive are performed after which the average of the measurements is recorded.

The input dataset is scattered over the worker pool before running these experiments, this ensures that the input data is available to all worker nodes before the query is submitted. All measurements record the total query runtime, as opposed to the runtime of the filter operator.

6.3. Results

This section presents the results as obtained for the accelerated version of Dask distributed. The performed benchmarks only consider acceleration by means of the Tidre based filter operator, as it is found that the RE2 based filter operator does not yield additional performance in Dask.

6.3.1. Accelerating the query

The query is accelerated by manipulating individual tasks as they are submitted to the accelerated worker nodes. This can be seen in Figure 6.3. Only the function attribute of the task is changed to match the new accelerated implementation of the operator.

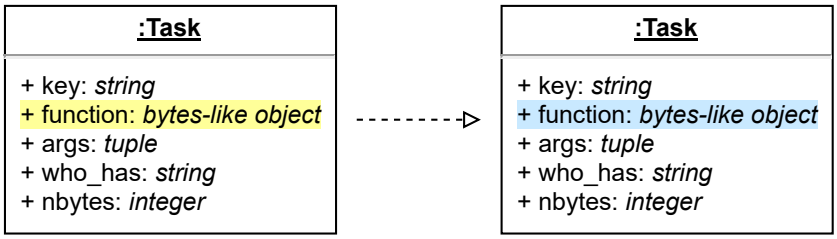


Figure 6.3: Task transformation for the regular expression usecase in Dask distributed.

To accelerate the regular expression usecase, the incoming string match task is targeted by the FPGA acceleration planning stage. The function attribute is substituted for the Tidre accelerated version of the string match operator.

6.3.2. Tidre acceleration

The results from the input size benchmark and the batch size benchmark are discussed below. As is specified in Section 6.2, these benchmarks measure the total query runtime on the Dask distributed cluster.

Input size benchmark The results of the input size benchmark for the four cluster configurations of the accelerated version of Dask distributed can be seen in Figure 6.4. The total query runtime, speedup relative to the cluster configuration without any accelerated workers, throughput, and cost per query can be seen in Figure 6.4a, Figure 6.4b, Figure 6.4c, and Figure 6.4d respectively.

The total query runtime for the four configurations can be seen in Figure 6.4a. It can be seen that the configuration with 3 accelerated workers achieves a runtime that is $\sim 3.5x$ lower as compared to the runtime of the configuration with 0 accelerated workers. This is confirmed in Figure 6.4b.

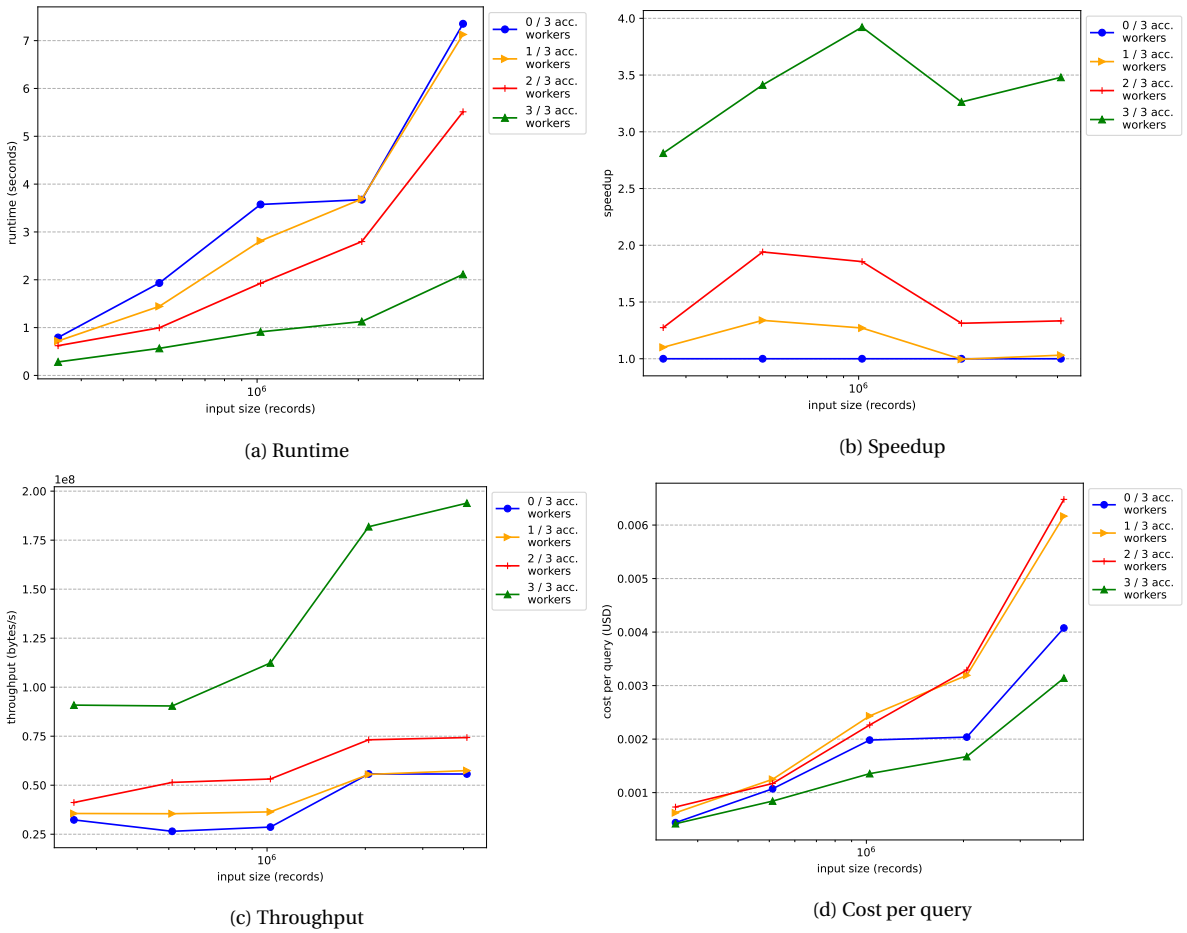
Furthermore, it can be seen that only substituting a fraction of the vanilla workers in the worker pool by accelerated workers yields diminishing returns. Input sizes up to 1,024,000 records can be evaluated in a single recordbatch. Since the distributed scheduler node makes use of round-robin scheduling, this recordbatch is in turn evaluated on a different worker in the pool for the first 3 repeats of the experiment. This cycle is repeated for successive runs. As these results show the average performance of the system for 9 consecutive runs, the recordbatch is evaluated on an accelerated worker in 33% of the cases. This is reflected in Figure 6.4a and Figure 6.4b, in which the configurations with 1 and 2 accelerated workers show an average performance increase.

Input sizes greater than the batch size of 1,024,000 records yield lower speedups. This can be explained by the fact that one of the recordbatches to be evaluated can be scheduled on a vanilla worker implementation. For multiple recordbatches, this is always the case for the configuration with only a single accelerated worker. When the accelerated worker is done processing its assigned recordbatch, the vanilla worker is still processing. It is therefore stated that the performance in the accelerated Dask distributed cluster is bound by the fraction of non-accelerated workers.

Figure 6.4c shows the throughput of this system. It can be seen that the configuration with 3 accelerated workers reaches a throughput of $\sim 190 MB/s$ for an input size of 4,096,000 records.

At this input size, the configuration with 0 accelerated workers reaches a throughput of $\sim 55 MB/s$. This holds true as well for the configuration with a single accelerated worker. The configuration with 2 accelerated workers reaches the slightly higher throughput of $\sim 75 MB/s$.

The cost of running a query on this cluster can be seen in Figure 6.4d. This cost includes the cost of the worker nodes as well as the cost of the scheduler node. The assumption is made that the vanilla worker implementations can be run on an *r4.2xlarge* instance without suffering a decrease in performance.



(a) Runtime

(b) Speedup

(c) Throughput

(d) Cost per query

Figure 6.4: Total query runtime, speedup, throughput, and cost per query for the input size benchmark on the accelerated version of Dask distributed. The batch size is set to 1,024,000 records.

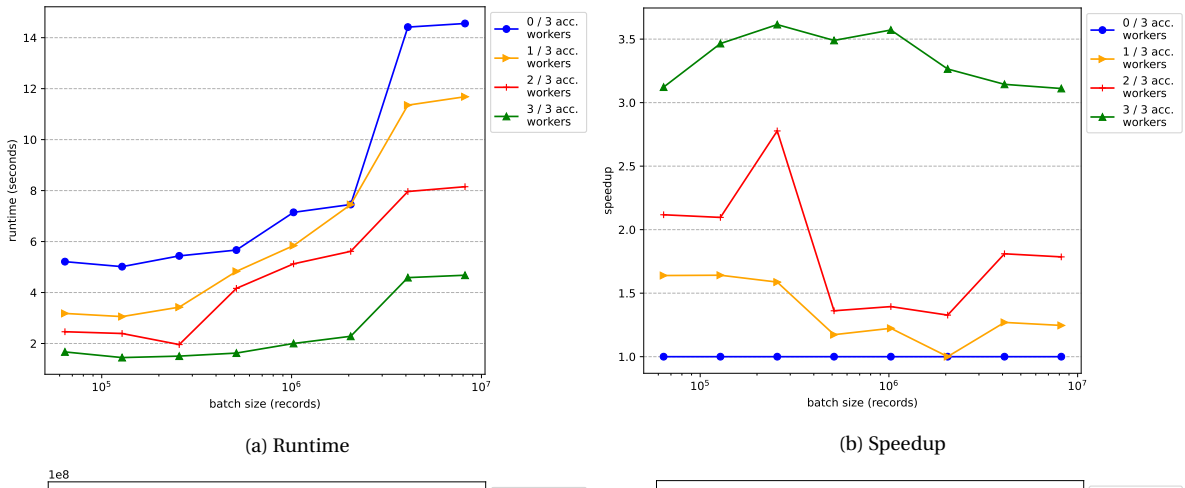
It can be seen that the configurations with 1 and 2 accelerated workers in the worker pool are more expensive than the configuration with 0 accelerated workers for all input sizes. The marginal speedup as achieved with these configurations does not outweigh the increased costs of the *f1.2xlarge* instance type.

It is observed that the configuration with 3 accelerated workers is slightly cheaper than the configuration of 0 accelerated workers for input sizes above 256,000 records. At an input size of 4,096,000 records, this configuration is roughly 25% less expensive.

Batch size benchmark The results of the batch size benchmark for the four cluster configurations of the accelerated version of Dask distributed can be seen in Figure 6.5. The total query runtime, speedup relative to the cluster configuration without any accelerated workers, throughput, and cost per query can be seen in Figure 6.5a, Figure 6.5b, Figure 6.5c, and Figure 6.5d, respectively.

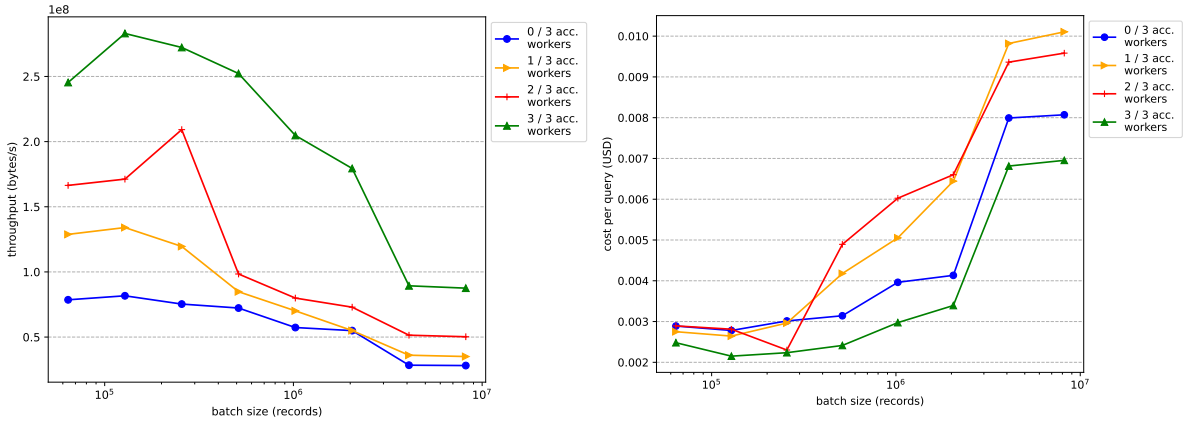
The runtimes of the four configurations can be seen in Figure 6.5a. Interestingly enough, increasing the batch size yields greater runtimes for all configurations of the cluster. Larger batch sizes increase the granularity at which operations can be parallelized over the worker pool. When the two largest batch sizes of 4,096,000 and 8,192,000 records are considered, the dataset is evaluated in a single batch. As an effect, only one worker in the worker pool is able to perform this evaluation. This completely nullifies the parallel computing capabilities of the distributed environment in which the framework is deployed. Furthermore, when only a fraction of the worker pool is accelerated, the entire worker pool can be kept waiting on the result as computed by a non-accelerated worker. Large batch sizes increase the evaluation time of the operator on these non-accelerated worker implementations.

These effects outweigh the positive impact greater batch sizes have on the overhead associated with data copies to and from the accelerator and the overhead associated to Pybind11 invocations.



(a) Runtime

(b) Speedup



(c) Throughput

(d) Cost per query

Figure 6.5: Total query runtime, speedup, throughput, and cost per query for the batch size benchmark on the accelerated version of Dask distributed. The input size is set to 4,096,000 records.

The speedup of these configurations as compared to the configuration without any accelerated workers can be seen in Figure 6.5b. The same observation as made for the input size benchmark can be made here. The speedup of the query is bound by the number of non-accelerated workers in the worker pool.

The configuration with 3 accelerated workers reaches its maximum speedup of $\sim 3.6x$ for a batch size of 256,000 records. This appears to be the optimal batch size as well for the configuration with 2 accelerated workers. However, increasing the batch size for the configuration with a single accelerated worker does not yield an increase in performance.

It is possible that the peak at a batch size of 256,000 records for the configuration with 2 accelerated workers is caused by an outlier in the measurement data. Unfortunately, this cannot be confirmed as only the average runtime of the consecutive runs is recorded.

From Figure 6.5c, it can be observed that the maximum throughput for the configuration with three accelerated workers is reached at a batch size of 128,000 records. At this batch size, a throughput of $\sim 280MB/s$ is reached. Increasing this batch size to 8,192,000 records drastically reduces this throughput down to $\sim 90MB/s$. Therefore, tuning this batch size to the optimal value can increase the throughput of the system by up to $\sim 2.1x$.

At this batch size, the configuration without any accelerated workers reaches a throughput of $\sim 70MB/s$, the configuration with a single accelerated worker reaches $\sim 140MB/s$, while the configuration with 2 accelerated workers reaches $\sim 170MB/s$.

Finally, the total cost per query for these configurations is seen in Figure 6.5d. As expected after the ob-

servations from the previous figures, increasing the batch size results in a higher cost per query for all configurations. For batch sizes of 512,000 and higher, the configurations with 1 and 2 accelerated workers become more expensive as compared to the configuration without any accelerated workers. For batch sizes under 512,000 records, these configurations yield a marginally lower cost per query.

The configuration with three accelerated workers is cheaper for all batch sizes when compared to the configuration without any accelerated workers. The lowest total cost per query for this configuration is found at a batch size of 128,000 records. This cost per query is roughly 23% lower as compared to the configuration without any accelerated workers.

6.4. Preliminary conclusion

It is found that the speedup is bound by the fraction of non-accelerated workers in the worker pool. Only accelerating a fraction of the workers yield diminishing returns as compared to accelerating the entire worker pool.

Furthermore, the speedup of the entire query is much lower when compared to the speedup of the filter operator as seen in the acceleration of vanilla Dask. This can be attributed to parquet read operators, which take up a significant portion of the runtime of the query. Additionally, data shuffles pose a significant overhead in cluster setups and are not present in single node deployments.

From the previous two implementations, it is found that increasing the batch size leads to in increase in performance. These previous implementations shown that increasing the batch size above 64,000 records yields diminishing returns, while the optimum performance of the system is reached at a batch size equal to or greater than 512,000 records. However, this does not hold true when considering a cluster deployment.

In a cluster deployment, increasing the batch size increases the granularity at which operations can be parallelized on the cluster. Additionally, larger batch sizes can increase the evaluation time on the fraction of non-accelerated workers in a worker pool, resulting in larger end-to-end runtimes. For all tested configurations of the cluster, increasing the batch size above 512,000 records leads to a decrease in performance. For the configuration in which all workers in the pool are accelerated, the maximum throughput is achieved at a batch size of 128,000 records. The maximum speedup as compared to the configuration without any accelerated workers is achieved at a batch size of 512,000 records. It is expected that these thresholds change depending on the underlying statistics of the usecase.

For batch sizes above 512,000 records, it only makes sense to accelerate the entire worker pool from an economical point of view. However, at lower batch sizes, only accelerating a fraction of the worker pool leads to a minor decrease of the total cost per query. A cost reduction of roughly 23% is obtained when the entire worker pool is accelerated and the batch size is set to 128,000 records.

7

Conclusions and recommendations

7.1. Conclusions

This work set out to find an answer to the following problem statement. *How should future big data frameworks be designed such that they facilitate transparent and efficient integration with FPGA accelerators in the context of SQL workloads?*

Three different implementations of FPGA accelerated big data frameworks are presented in this work. The results obtained from these integration efforts are used to answer the research questions as derived from the problem statement. The conclusions can be found in the paragraphs below.

Where in the system should FPGA accelerators be placed in order to accelerate SQL workloads? It is identified that FPGA accelerators can either be placed in the data path or attached as a processor. The latter, FPGA accelerators attached as a processor, is best suited for the acceleration of individual SQL operators in an existing big data framework. A distinction is made between FPGA accelerators configured as an IO-attached processor and FPGA accelerators configured as a co-processor. From these two configurations, FPGA accelerators configured as a co-processor are preferable, but this configuration is only available if the underlying compute system supports this.

This work integrates FPGA accelerators configured as an IO-attached processor into three batch-processing big data frameworks. It is found that such an acceleration is able to achieve a speedup of $1750x$ for a regular expression filter operator as implemented in Dremio. In Dask, this acceleration produces a speedup of $92x$ for the same regular expression filter operator. In the latter case, this speedup of the filter operator yields an end-to-end speedup of $3.6x$ when the runtime of the entire query is considered in a distributed setup. This leads to a reduction of the cost per query of 23%.

Additionally, it is found that reading data from disk takes up a significant portion of the runtime of a SQL workload. In this case, FPGA accelerators can be placed in between storage and the CPU. These accelerators could perform algorithms such as parquet decompression, projection, or filtering to reduce the runtime of these operations. Such an implementation is however not provided in the current work and poses an interesting opportunity for future research.

Which features of big data frameworks are required to facilitate efficient integration and what is their impact on the overall performance of the system? Two absolute requirements for the efficient integration of FPGA accelerator in big data frameworks are the presence of a flexible API for the manipulation of execution plans and the use of a hardware-friendly and language-independent in-memory data format such as Apache Arrow.

Furthermore, the batch size of a batch-processing framework is an important parameter which should be set by the developer or the end-user. Some big data frameworks impose limits on the maximum value of this batch size, but this could lead to suboptimal solutions.

The optimal batch size is heavily dependent on the deployment of the big data framework and the specific workload that is to be evaluated. It is found that the regular expression usecase as presented in this

work is best evaluated with a batch size of at least 512,000 records when the accelerated big data framework is deployed on a single node in a compute cluster. On the contrary, when the framework is deployed in a distributed setup, a trade-off related to this batch size is found. Considering the overhead related to memory copies to and from the accelerator, as well as the overhead associated to the invocation of the native interface that communicates with this accelerator, larger batch sizes are beneficial. At the same time, when considering the granularity at which operations can be parallelized on the cluster, as well as idle time of accelerated workers resulting from larger evaluation times of the fraction of non-accelerated workers in the worker pool, lower batch sizes are preferred. For the regular expression usecase as evaluated on an accelerated framework deployed in a distributed setup, a batch size of 128,000 records is found to be optimal.

Additionally, for this usecase as evaluated in a distributed setup, tuning this batch size can result in an increase of end-to-end query throughput of up to $2.1x$. When the throughput of the individual SQL operator is considered in a single node deployment, this tuning of the batch size is accountable for an increase in throughput of up to $2.6x$.

Finally, all memory buffers of the hardware-friendly in-memory format should be aligned to 64 bytes. Misalignment of these buffers can decrease the throughput of a specific SQL operator by $2.5x$.

Which parts of the big data framework need to be aware of the acceleration such that the system can be transparently deployed in a heterogeneous setting? This research question cannot be satisfied with a single answer. There are multiple system architectures that facilitate the transparent deployment of an accelerated big data system. Each of these architectures has its own advantages and disadvantages depending on the deployment context.

For frameworks deployed on a single node in a compute cluster, the scheduler class needs to be aware of the acceleration. This ensures that the acceleration planning is performed in an efficient way, as this planning needs to be performed only once.

In addition, this solution achieves the greatest number of acceleration opportunities. This observation is supported by two findings. First, it is guaranteed that the entire execution plan is available during the acceleration planning phase, allowing for the acceleration of sequences of operators spanning multiple subgraphs of this execution plan. Second, both operators with a narrow dependency as well as operators with a wide dependency can be accelerated, as data shuffles do not occur on single node deployments.

Big data frameworks deployed in a distributed environment can be accelerated by two distinct approaches. Either the scheduler node or the worker node can be made aware of the acceleration. The optimal implementation depends on the specific usecase and underlying statistics of the dataset, as well as a trade-off between speedup and incurred cost.

System architectures that make the scheduler node aware of the acceleration achieve the same planning efficiency as the single node deployment as discussed above. In this case, only operators with narrow dependencies are accelerated. However, the resulting number of acceleration opportunities remains higher than is the case for implementations that perform the acceleration in the worker node. This can be attributed to the guarantee that the entire execution plan is available at the start of the acceleration planning phase.

A drawback of this architecture is that all workers in the worker pool are required to have an FPGA accelerator installed unless the scheduler node is aware of the capabilities of all workers in the worker pool. This can impose challenges on the ease of deployment and the resulting cost of such a system. If the scheduler node is aware of the different capabilities of the workers in the pool, different versions of the execution plan are to be established and distributed.

System architectures that make the worker node aware of the acceleration achieve a slightly lower planning efficiency as compared to the previous system architecture. This can be attributed to the fact that this planning is now performed in parallel on each worker node. Furthermore, when the execution plan is submitted to the worker nodes as multiple subgraphs, each worker node has to perform the planning for each incoming subgraph.

When considering the number of acceleration opportunities, this is lower as compared to the other available system architectures. Performing the acceleration planning on subgraphs of the original execution plan makes it impossible to accelerate sequences of operators that span multiple subgraphs.

It is found that the acceleration is bound by the fraction of non-accelerated workers in the worker pool. Depending on the specific usecase, statistics of the dataset, and configured batch size, such a configuration with a partially accelerated worker pool can induce greater costs per query as compared to a non-accelerated worker pool. In these cases, the attained speedup does not outweigh the increased cost of the added FPGA instances. Configurations in which the entire worker pool consists of accelerated workers are found to reduce the cost per query for the regular expression usecase.

Which part of the workload should be accelerated such that the average speedup within the application domain is maximized? The design and implementation of an FPGA kernel is found to be time consuming. It is therefore important to maximize the impact a given kernel has on queries within an application domain.

Each FPGA kernel targets a specific sequence of operators. In order to maximize the impact of the kernel, different queries within the application domain should be rewritten in such a way that they feature this specific sequence. Such a rewrite of the execution plan can be performed during the exploration phase of the query optimizer.

It is likely that the acceleration of these semantically equivalent sequences of operators yields lower performance improvements. Hence, it is necessary to evaluate whether the accelerated execution plan is expected to perform better than the original execution plan. This can be achieved through the use of cost-based query optimization, utilizing FPGA-specific cost models to evaluate the different execution plans. This remains an open challenge for future research.

7.2. Recommendations

Multiple opportunities for potential performance improvements are identified over the course of this project. These opportunities are either outside of the scope of this work or they do not fit within the allocated timeframe of the MSc thesis project. As such, they are not acted upon. Instead, recommendations for future work are provided. These recommendations can be found below.

- The acceleration efforts in this work make use of FPGA accelerators configured as processors. This proves to be effective for the acceleration of SQL operators. However, as it is found that the speedup of these SQL workloads is partly limited by the significant runtimes of file reader implementations, FPGA accelerators in the data path could be deployed alongside the implemented FPGA processors.

  Such a system could make use of SmartSSDs, which integrate FPGA fabric into the package of the storage medium [83]. The overall performance of the system could be increased by the implementation of parquet decompression, projection, and filter algorithms on these accelerators.

- The regular expression usecase as constructed in this work is not used in industry. Therefore, the benchmarks that are derived from this usecase are not fully representative for real world problems. A more conventional and industry-recongnized set of benchmarks is the TPC-H benchmark suite [98]. This benchmark suite provides a set of business oriented decision support queries.

  An integration effort is performed regarding the integration of an existing FPGA kernel that targets the SQL operators resulting from TPC-H query 1 into Dremio. This integration is not completed due to time limitations. Therefore, benchmarking of the accelerated systems by use of an industry-standard benchmark suite is left for future research.

- The present work does not implement a fully functional query exploration phase in which execution plans are rewritten to accommodate FPGA acceleration. Instead, a simple PoC implementation is realized. This implementation could be extended by means of cost-based query optimization.

  In addition, FPGA-specific cost models could be considered that take performance indicators related to energy efficiency into account. This research direction could pave the way towards the realization of 'green datacenters' as described in Section 1.1.

- The implementations as presented in this work only consider the acceleration of batch-processing big data frameworks. More research could be done regarding the integration of FPGA accelerators into streaming big data frameworks. Cluster deployments of these streaming frameworks could benefit from FPGA accelerators directly attached to the network interface. This is another realization of FPGA accelerators deployed in the data path and could be accomplished with the use of SmartNICs.

- The current implementation of the accelerated operator initializes the Tidre platform for each recordbatch evaluation. As this platform only needs to be initialized once, this initialization could be moved to a setup method. Only invoking this method once per operator instead of once per recordbatch is expected to yield additional performance.

  This improvement could be made in a future and more general implementation of the accelerated operator. Completely basing this new operator on Apache Calcite and Apache Arrow could make it useful for a wide range of big data frameworks. In addition to the invocation of the platform initialization in the setup method, generalizations could be made regarding the *consume data* and *output data* methods. These generalizations should enable the acceleration of different sequences of SQL operators without the need for manual interference from the developer.

- The implementation of the FPGA accelerated regular expression matcher returns a selection vector containing the indices of matching records in the recordbatch. The kernel has built-in support for the computation of a bitmap instead of a selection vector.

  A distinction is made between sparse matching recordbatches and dense matching recordbatches. Based on these underlying statistics of the dataset, the accelerated operator should configure the FPGA kernel to either return a selection vector or a bitmap. If this returned format does not match the expected format in the big data framework, an additional conversion is required at the software side. A cost model could be developed to identify which format to use in a specific context.

- At the time of writing, the FPGA kernel as deployed on AWS did not offer support for the co-processor configuration. However, this could change as AWS F1 instances now support the Xilinx Run Time (XRT), which enables FPGA kernels to directly access main memory. How this XRT support could be leveraged to enable to deployment of big data frameworks accelerated with FPGA co-processors is left for future research.

- The regular expression matcher kernel as used in this work is used to target one of many existing SQL operators. In order to fully realize the future big data framework as envisioned by Teratide and the ABS group, a complete library containing the FPGA bitstreams for these SQL operators is required.

  The design and implementation of these kernels is an ongoing effort and as such is listed as future research.

Bibliography

[1] Doug Laney. Deja vvvu: Gartner's original "volume-velocity-variety" definition of big data. URL https://community.aiim.org/blogs/doug-laney/2012/08/25/deja-vvvu-gartners-original-volume-velocity-variety-definition-of-big-data. accessed on 2021-06-08.

[2] Zaid Al-Ars Delft University of Technology. Sbd lecture 1: The new age of big data, Sep 2020. Supercomputing for Big Data.

[3] Tianqi Yu and Xianbin Wang. *Real-Time Data Analytics in Internet of Things Systems*, pages 1–28. 01 2020. ISBN 978-981-4585-87-3. doi: 10.1007/978-981-4585-87-3_38-1.

[4] Matt Turck. Resilience and vibrancy: The 2020 data ai landscape. URL https://mattturck.com/data2020/. accessed on 2021-06-08.

[5] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H Peter Hofstee. Fpga acceleration for big data analytics: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, 2021.

[6] Eric Ries. *The lean startup : how constant innovation creates radically successful businesses*. Portfolio Penguin, London; New York, 2011. ISBN 9780670921607 0670921602. URL http://www.amazon.de/The-Lean-Startup-Innovation-Successful/dp/0670921602/ref=sr_1_2?ie=UTF8&qid=1396199893&sr=8-2&keywords=eric+ries.

[7] Adyen. Adyen | the payments platform built for growth. URL https://www.adyen.com. accessed on 2021-06-27.

[8] Nicola Jones. How to stop data centres from gobbling up the world's electricity. URL https://www.nature.com/articles/d41586-018-06610-y. accessed on 2021-07-05.

[9] International Energy Agency. The netherlands - key energy statistics, 2018. URL https://www.iea.org/countries/the-netherlands. accessed on 2021-07-05.

[10] The Apache Software Foundation. Apache arrow | apache arrow, . URL https://arrow.apache.org/. accessed on 2021-02-09.

[11] ABS TU Delft. abs-tudelft/fletcher: Fletcher: A framework to integrate fpga accelerators with apache arrow on github. URL https://github.com/abs-tudelft/fletcher. accessed on 2021-02-09.

[12] Johan Peltenburg, Jeroen Van Straten, Lars Wijtemans, Lars Van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277. IEEE, 2019.

[13] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H Peter Hofstee, Zaid Al-Ars, C Hochberger, B Nelson, A Koch, R Woods, and P Diniz. Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In *ARC*, pages 32–47, 2019.

[14] Microsoft. Linux virtual machines pricing. URL https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. accessed on 2021-06-08.

[15] Johan Peltenburg, Lars TJ van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H Peter Hofstee. Battling the cpu bottleneck in apache parquet to arrow conversion using fpga. In *2020 international conference on Field-Programmable technology (ICFPT)*, pages 281–286. IEEE, 2020.

[16] F. Nonnenmacher. Transparently accelerating spark sql code on computing hardware. Master's thesis, Delft University of Technology, the Netherlands, 2020.

[17] Robert Chang. A beginner's guide to data engineering — part i, . URL https://medium.com/@rchang/a-beginners-guide-to-data-engineering-part-i-4227c5c457d7. accessed on 2021-06-09.

[18] Robert Chang. A beginner's guide to data engineering — part ii, . URL https://medium.com/@rchang/a-beginners-guide-to-data-engineering-part-ii-47c4e7cbda71. accessed on 2021-06-09.

[19] Robert Chang. A beginner's guide to data engineering — the series finale, . URL https://medium.com/@rchang/a-beginners-guide-to-data-engineering-the-series-finale-2cc92ff14b0. accessed on 2021-06-09.

[20] Hilary Mason. A taxonomy of data science. URL http://www.dataists.com/2010/09/a-taxonomy-of-data-science/. accessed on 2021-06-09.

[21] Monica Rogati. The ai hierarchy of needs. URL https://hackernoon.com/the-ai-hierarchy-of-needs-18f111fcc007. accessed on 2021-06-09.

[22] Google Inc. Welcome to colaboratory, . URL https://colab.research.google.com7. accessed on 2021-06-17.

[23] Deepnote. Deepnote - data science notebooks for teams. URL https://deepnote.com/. accessed on 2021-06-17.

[24] Zaid Al-Ars Delft University of Technology. Acs lecture 1: Trends in computing systems, Sep 2020. Advanced Computing Systems.

[25] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[26] Y.T.B. Mulder. Feeding high-bandwidth streaming-based fpga accelerators. Master's thesis, Delft University of Technology, the Netherlands, 2018.

[27] Fritz Kruger. Cpu bandwidth – the worrisome 2020 trend. URL https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/. accessed on 2021-06-25.

[28] The Apache Software Foundation. Apache hadoop, . URL https://hadoop.apache.org/. accessed on 2021-06-17.

[29] The Apache Software Foundation. Hdfs architecture guide, . URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. accessed on 2021-06-17.

[30] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[32] The Apache Software Foundation. Apache hadoop 3.3.1 - mapreduce tutorial, . URL https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. accessed on 2021-06-17.

[33] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proc. VLDB Endow.*, 5(12):2014–2015, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367562. URL https://doi.org/10.14778/2367502.2367562.

[34] Zaid Al-Ars Delft University of Technology. Sbd lecture 2: Apache spark: High-performance big data framework, Sep 2020. Supercomputing for Big Data.

[35] G. Jevtic. Hadoop vs spark – detailed comparison. URL https://phoenixnap.com/kb/hadoop-vs-spark. accessed on 2021-06-17.

[36] Barbatunde Towards Data Science. Introduction to apache spark with scala. URL https://towardsdatascience.com/introduction-to-apache-spark-with-scala-ed31d8300fe4. accessed on 2021-06-22.

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

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[39] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1773–1786, 2019.

[40] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, 2018.

[41] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2014.

[42] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[43] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: Proving query rewrites with univalent sql semantics. *ACM SIGPLAN Notices*, 52(6):510–524, 2017.

[44] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. *arXiv preprint arXiv:1802.02229*, 2018.

[45] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 631–646, 2017.

[46] Jyoti Leeka and Kaushik Rajan. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment*, 13(3):348–361, 2019.

[47] J. Hidders J. Lee H. P. Hofstee J. Fang, Y. T. B. Mulder. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29:33–59, 2020.

[48] R. Singh M. Sharma, G. Singh. A review of different cost-based distributed query optimizers. *Progress in Artificial Intelligence*, 8:45–62, 2019.

[49] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*, volume 2. Springer, 1999.

[50] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala. Technical report, 2013.

[51] Ravindra Pindikura. Introducing the gandiva initiative for apache arrow. URL https://www.dremio.com/announcing-gandiva-initiative-for-apache-arrow. accessed on 2021-06-18.

[52] The Apache Software Foundation. Apache arrow overview, . URL https://arrow.apache.org/overview/. accessed on 2021-06-20.

[53] The Apache Software Foundation. The apache® software foundation announces apache arrow™ as a top-level project, . URL https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces87. accessed on 2021-06-20.

[54] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, Zaid Al-Ars, and H Peter Hofstee. Generating high-performance fpga accelerator designs for big data analytics with fletcher and apache arrow. *Journal of Signal Processing Systems*, 93(5):565–586, 2021.

[55] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[56] Intel. Intel intrinsics guide. URL https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX_512. accessed on 2021-06-20.

[57] J. W. Peltenburg. *Methods for Efficient Integration of FPGA Accelerators with Big Data Systems*. PhD thesis, Delft University of Technology, 2020.

[58] Jacques Nadeau. Gandiva: A llvm-based analytical expression compiler for apache arrow. URL https://arrow.apache.org/blog/2018/12/05/gandiva-donation/. accessed on 2021-06-21.

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

[60] Johan Peltenburg, Jeroen Van Straten, Matthijs Brobbel, Zaid Al-Ars, and H Peter Hofstee. Tydi: an open specification for complex data structures over hardware streams. *IEEE Micro*, 40(4):120–130, 2020.

[61] A. Stratikopoulos F.S. Zakkak C. Kotselidis M. Papadimitriou, J. Fumero. Transparent compiler and runtime specializations for accelerating managed languages on fpgas. *The Art, Science, and Engineering of Programming*, 5(2), 11 2020.

[62] Rene Mueller and Jens Teubner. Fpga: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 999–1004, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585512. doi: 10.1145/1559845.1559965. URL https://doi.org/10.1145/1559845.1559965.

[63] Rene Mueller and Jens Teubner. Fpgas: A new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, page 721–723, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589459. doi: 10.1145/1739041.1739137. URL https://doi.org/10.1145/1739041.1739137.

[64] Z. István. The glass half full: Using programmable hardware accelerators in analytics. *IEEE Data Eng. Bull.*, 42:49–60, 2019.

[65] Teratide. teratide/tidre-demo: Demo application for the tidre regular expression accelerator, . URL https://github.com/teratide/tidre-demo. accessed on 2021-02-18.

[66] Teratide. vhdre: a vhdl regex matcher generator, . URL https://github.com/abs-tudelft/vhdre. accessed on 2021-07-05.

[67] Dieudonne Manzi and David Tompkins. Exploring gpu acceleration of apache spark. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 222–223. IEEE, 2016.

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

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

[70] Alexander Ocsa. Sql for gpu data frames in rapids accelerating end-to-end data science workflows using gpus. In *LatinX in AI Research at ICML 2019*, 2019.

[71] Ying Li, Jinyu Zhan, Wei Jiang, Junting Wu, and Jianping Zhu. An fpga based network interface card with query filter for storage nodes of big data systems. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 556–561. IEEE, 2020.

[72] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. Fpga-based dynamically reconfigurable sql query processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(4):1–24, 2016.

[73] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[74] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. Smartssd: Fpga accelerated near-storage data analytics on ssd. *IEEE Computer architecture letters*, 19(2):110–113, 2020.

[75] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. Relational query processing on opencl-based fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10. IEEE, 2016.

[76] Andreas Becher, BG Lekshmi, David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. Integration of fpgas in database management systems: challenges and opportunities. *Datenbank-Spektrum*, 18(3):145–156, 2018.

[77] Andreas Becher, Achim Herrmann, Stefan Wildermann, and Jürgen Teich. Reprovide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. *BTW 2019–Workshopband*, 2019.

[78] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218. IEEE, 2017.

[79] The Apache Software Foundation. Apache spark - unified analytics engine for big data, . URL `https://spark.apache.org/`. accessed on 2021-02-16.

[80] Dremio. Introduction - dremio, 2020. URL `https://docs.dremio.com/`. accessed on 2020-11-06.

[81] Dask. Dask: Scalable analytics in python, . URL `https://dask.org/`. accessed on 2021-07-16.

[82] Jian Fang, Jianyu Chen, Zaid Al-Ars, Peter Hofstee, and Jan Hidders. Work-in-progress: A high-bandwidth snappy decompressor in reconfigurable logic. In *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–2. IEEE, 2018.

[83] Xilinx. The first and only adaptive computational storage platform. URL `https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html`. accessed on 2021-07-02.

[84] Zaid Al-Ars Delft University of Technology. Sbd lecture 3: Spark libraries and apache kafka, Sep 2020. Supercomputing for Big Data.

[85] Amazon. Amazon ec2 f1 instances. URL `https://aws.amazon.com/ec2/instance-types/f1/`. accessed on 2021-06-29.

[86] OpenCAPI Consortium. Opencapi consortium: Official site. URL `https://opencapi.org/`. accessed on 2021-06-30.

[87] Google. google/re2: Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. it is a c++ library. URL `https://github.com/google/re2`. accessed on 2021-02-18.

[88] Amazon. Amazon ec2 instance types. URL `https://aws.amazon.com/ec2/instance-types/`. accessed on 2021-07-02.

[89] Bob Luppes Teratide. Dremio accelerated, . URL https://github.com/teratide/dremio-accelerated. accessed on 2021-07-01.

[90] Bob Luppes Teratide. Data generator, . URL https://github.com/teratide/data-generator. accessed on 2021-07-01.

[91] Colm O'Connor. xeger 0.3.5. URL https://pypi.org/project/xeger/. accessed on 2021-07-04.

[92] Pydata.org. Pandas. URL https://pandas.pydata.org/. accessed on 2021-07-22.

[93] Pybind. pybind11 — seamless operability between c++11 and python. URL https://github.com/pybind/pybind11. accessed on 2021-07-15.

[94] Bob Luppes Teratide. Dask accelerated, . URL https://github.com/teratide/dask-accelerated. accessed on 2021-07-01.

[95] Secret Labs. Secret labs' regular expression engine. URL https://github.com/python/cpython/blob/main/Modules/sre.h. accessed on 2021-07-15.

[96] Dask. Dask distributed, . URL https://distributed.dask.org/en/latest/. accessed on 2021-02-16.

[97] Bob Luppes Teratide. Dask distributed accelerated, . URL https://github.com/teratide/dask-accelerated/tree/worker-optimization. accessed on 2021-07-01.

[98] TPC. Tpc-h vesion 2 and version 3. URL http://www.tpc.org/tpch/. accessed on 2021-07-17.

[99] The Pandas Development Team. Comparison with sql - pandas 1.2.2 documentation. URL https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html. accessed on 2021-02-17.

[100] OmniSci Inc. Accelerated analytics platform | omnisci, . URL https://www.omnisci.com/. accessed on 2021-02-16.

[101] Influxdata. Influxdata/influxdb_iox: Pronounced (influxdb eye-ox), short for iron oxide. this is the new core of influxdb written in rust on top of apache arrow. URL https://github.com/influxdata/influxdb_iox. accessed on 2021-02-16.

[102] The Apache Software Foundation. Datafusion: A rust-native query engine for apache arrow | apache arrow, . URL https://arrow.apache.org/blog/2019/02/04/datafusion-donation/. accessed on 2021-02-16.

[103] The Apache Software Foundation. apache/hive: Apache hive, . URL https://github.com/apache/hive. accessed on 2021-02-18.

[104] Ballista Compute. Ballista: Distributed compute platform. URL https://github.com/ballista-compute/ballista. accessed on 2021-07-17.

[105] PrestoDB. prestodb/presto: The official home of the presto distributed sql query engine for big data, . URL https://github.com/prestodb/presto. accessed on 2021-02-18.

[106] PrestoDB. Apache arrow connector - issue 12201, . URL https://github.com/prestodb/presto/issues/12201. accessed on 2021-02-18.

[107] FASTDATAio. Homepage - fastdataio. URL https://fastdata.io/. accessed on 2021-02-16.

[108] TileDB Inc. Tiledb | tiledb, . URL https://tiledb.com/. accessed on 2021-02-16.

[109] Turbodbc. Turbodbc - turbocharged database access for data scientists - turbodbc latest documentation. URL https://turbodbc.readthedocs.io/en/latest/. accessed on 2021-02-16.

[110] Cylon. Cylon | cylon. URL https://cylondata.org/. accessed on 2021-02-16.

[111] Kinetica. Kinetica docs home - kinetica documentation 7.1.1 documentation. URL https://www.kinetica.com/docs/. accessed on 2021-02-18.

[112] The Apache Software Foundation. Apache kafka, . URL https://kafka.apache.org/documentation/. accessed on 2021-06-29.

[113] Neha Narkhede. Introducing ksql: Streaming sql for apache kafka. URL https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/. accessed on 2021-06-29.

[114] Elasticsearch B.V. Elasticsearch sql: Query elasticsearch indices with sql | elastic. URL https://www.elastic.co/what-is/elasticsearch-sql. accessed on 2021-02-16.

A

Analysis of big data frameworks

A.1. Batch-processing frameworks

This section provides the analysis of individual batch-processing big data frameworks. Several frameworks are analyzes but were not found to be suitable for acceleration. Therefore, a distinction is made between frameworks that are and frameworks that are not considered for acceleration.

A.1.1. Considered for acceleration

Dask and Dask distributed Dask has no internal parser and planner for SQL queries [81]. Instead, the Pandas API is used to perform all SQL-like operations [99]. The execution plan is represented as a *Task Graph* in Dask, which can be modified using rewrite rules in the Dask API. Pyarrow is supported when accessing parquet files. By default, buffers allocated by the Python API of Apache Arrow are 64-byte aligned. Batch size is referred to as the chunk size in Dask and can be set to an arbitrary number.

In addition, Dask distributed is a separate version of this framework specifically designed to run in distributed setups. The scheduler and worker implementations are modified to support these deployments. All other characteristics of the underlying framework remain unchanged.

Dask has 8k stars, 1.2k forks, 237 watchers, and 82 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

OmniSciDB OmniSciDB, formerly MapD Core, is a SQL based database engine that is designed to run on hybrid CPU/GPU systems [100]. All query parsing and optimization is offloaded to Apache Calcite, which can be easily extended by adding new planning phases and optimizer rules. Apache Arrow is used as the internal memory format. OmniSciDB has a custom *Data Manager* impelementation which is used to allocate memory buffers. These memory addresses are 512-byte aligned. Columns are divided into fragments and chunks, which can be set to an arbitrary number. Together, these parameters determine the batch size, which by default is set to 1GB.

Typical application domains include business intelligence tools (BI) and geographic information systems (GIS).

OmniSciDB has 2.4k stars, 352 forks, 144 watchers, and 23 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Spark SQL Spark SQL makes use of the catalyst optimizer, which can be easily extended by adding new optimization rules. At the time of Nonnenmacher's work, Apache Arrow is not the default memory structure, but there is support for columnar data formats [16]. The JVM aligns all memory addresses to 8 bytes. By default the batch size is set to 10,000 records, but this can be increased manually and is uncapped.

Apache Spark, which incorporates Spark SQL, has 28.9k stars, 23.4k forks, 2.1k watchers, and 225 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

InfluxDB IOx InfluxDB IOx uses DataFusion as the in-memory query engine [101, 102]. SQL queries are planned and optimized, but no separate planner stages are identified. Apache Arrow is used as the default in-memory structure. The Rust API of Apache Arrow provides methods that allow users to specify their own memory alignment constraints. The default batch size is set to 1,000 records but it seems that this can be increased manually.

InfluxDB IOx has 522 stars, 46 forks, 40 watchers, and 6 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Apache Hive As identified in Section 2.2, Apache Hive provides a SQL-like layer over the MapReduce framework [38, 39]. In the newest version of Apache Hive, SQL queries can be evaluated on Apache Hadoop MapReduce, Apache Tez, or Apache Spark [103]. Apache Hive incorporates the Apache Calcite query optimizer, which can be easily extended by adding new optimization rules. There appears to be support for Apache Arrow, but it is not the default in-memory structure. The maximum batch size in Apache Hive is set to 2,147,483,647 records.

Apache Hive has 3.6k stars, 3.4k forks, 327 watchers, and 94 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Ballista Ballista is a distributed compute engine based on Apache arrow and DataFusion [102, 104]. The project is a work in progress, but there is support to run TCP-H queries 1, 3, 5, 6, 10, and 12. As is the case for InfluxDB IOx, the Rust API of Apache Arrow allows users to specify their own memory alignment constraints. The maximum batch size is hard coded to 32,768 records at various locations in the framework, but this could be increased manually.

Ballista has 2.1k stars, 126 forks, 72 watchers, and 3 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Presto Presto is an open-source distributed SQL query engine [105]. Presto includes a full-fledged query planner and optimizer. This optimizer can be extended by adding optimization rules. The underlying in-memory structure is managed by the JVM, which by default aligns memory addresses to 8 bytes. There is an open issue regarding Apache Arrow support in Presto. At the time of writing, nobody has been assigned to this issue [106]. By default, the batch size is set to 1,000 records, but this can be increased manually.

Presto has 11.7k stars, 4k forks, 889 watchers, and 125 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Dremio Dremio's internal query engine, the Sabot engine, is based on Apache Calcite. It is therefore highly extendable and features multiple planning phases. Apache Arrow is used as the default memory structure. The java API of Apache Arrow aligns memory addresses on 8 bytes. The maximum batch size for narrow recordbatches (records with up to 100 fields) is set to 64,000 records. This maximum batch size is even lower for wide recordbatches (records with over 100 fields).

Dremio has 849 stars, 263 forks, 99 watchers, and 27 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

A.1.2. Not considered for acceleration

Fastdata.io Fastdata.io is an Apache Arrow based extension to Apache Spark that enables evaluation on GPU accelerators [107]. At the time of writing, their code is not open-source and thus cannot be evaluated.

TileDB TileDB is a storage engine for multi-dimensional arrays that uses the Apache Arrow in-memory data format [108]. It does not include a query parser or optimizer, instead it relies on tools such as Apache Spark and Dask to query their underlying storage engine. TileDB could potentially benefit from an accelerator in the data path. As TileDB does not include a query engine, it is not considered for acceleration.

Turbodbc Turbodbc is a Python module that can access external databases over ODBC [109]. It does not include a query parser or optimizer and as such is not considered for acceleration.

Cylon Cylon is a library for processing structured data, written in C++ [110]. Python and Java language interfaces are provided. Cylon uses Apache Arrow as the in-memory structure. The C++ API of Apache Arrow allows buffers to be allocated with an alignment of 64 bytes. Cylon merely provides implementations of relational operators and can therefore not be used as a standalone big data framework. Cylon is therefore not considered for acceleration.

Kinetica Kinetica is a database engine with support for *many-core devices* such as GPUs [111]. The Kinetica framework is however not open-source and is therefore not considered for acceleration.

A.2. Streaming frameworks

Two streaming big data frameworks are considered for acceleration. A brief analysis of their characteristics can be found below.

Spark Streaming Spark Streaming is a module that extends the Spark Core in order to support stream processing. It does so by evaluating micro-batches. These micro-batches are obtained by convolving a record stream with a sliding window. All characteristics of the underlying Spark Core remain unchanged.

Spark Streaming is contained in the official Spark repository on Github. As not all Spark users are expected to make use of the Spark Streaming module, it's community size is expected to be smaller than or equal to the community size of Apache Spark.

Apache Kafka Streaming SQL engines exist for Apache Kafka such as KSQL [112, 113]. KSQL differs from database SQL engines in the sense that it runs *continuous* queries on streams of data. SQL queries are translated to transformations in Apache Kafka by use of a custom SQL parser. Apache Kafka uses a standardized binary format that differs from Apache Arrow. It is not known whether this format is hardware-friendly like Apache Arrow.

Apache Kafka has 19.3k stars, 10.2k forks, 1.1k watchers, and 908 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

A.3. Index-based frameworks

Finally, two index-based big data frameworks are considered for acceleration. A brief analysis of their characteristics can be found below.

Elasticsearch SQL Elasticsearch SQL is an index-based search engine that provides a SQL plugin [114]. This plugin parses SQL queries and translates them to operations in the Elasticsearch framework, which requires all underlying data to be indexed. Since SQL queries are not the main focus, the planner is implemented in a very simple way and does not feature multiple extendible stages. The framework uses JSON as an underlying data format. Memory addresses are aligned to 8 bytes by default in the JVM. Batch size is not a thing in Elasticsearch SQL since there are no records to be processed. Instead, all data is retrieved by performing a reverse lookup of a given index.

It is identified that this is not a computationally heavy operation that could benefit from FPGA acceleration. The framework could potentially benefit from an accelerator in the data path during the indexing phase, but this is out of scope for this project.

Elasticsearch, which incorporates the Elasticsearch SQL plugin, has 53.9k stars, 19.3k forks, 2.8k watchers, and 415 open pull-requests on Github. The sum of these numbers is used as a proxy for the community size of this big data framework.

Apache Lucene Apache Lucene is another index-based big data framework. Lucene implements various algorithms that are based on reversed index lookups. There is no support for parsing and optimizing SQL queries. Just like Elasticsearch, JSON is used as an underlying data format.

Apache Lucene has 337 stars, 186 forks, 39 watchers, and 32 open pull-requests on GitHub. The sum of these numbers is used as a proxy for the community size of this big data framework.

B

Measurement data

B.1. Dremio integration

This section provides the raw measurement data of the benchmarks as performed on vanilla Dremio, the RE2 accelerated version of Dremio, and the Tidre accelerated version of Dremio. Separate data is presented for each repeat of the experiments. Only the runtime of the operators and the runtime of the total query are shown, as the throughput, speedup, and cost can be derived from these figures.

B.1.1. Input size benchmark

vanilla Dremio

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.005	0.005	0.003	0.004	0.003	0.003	0.003	0.004	0.003	0.002
2,000	0.003	0.004	0.003	0.003	0.002	0.003	0.002	0.002	0.002	0.002
4,000	0.003	0.003	0.004	0.003	0.002	0.003	0.004	0.002	0.004	0.004
8,000	0.006	0.003	0.003	0.003	0.004	0.003	0.003	0.002	0.002	0.002
16,000	0.009	0.005	0.004	0.004	0.005	0.004	0.003	0.005	0.004	0.003
32,000	0.005	0.007	0.005	0.005	0.005	0.004	0.004	0.005	0.005	0.006
64,000	0.014	0.013	0.013	0.014	0.013	0.013	0.014	0.013	0.013	0.014
128,000	0.018	0.018	0.018	0.018	0.018	0.018	0.017	0.018	0.018	0.018
256,000	0.027	0.027	0.026	0.026	0.027	0.026	0.026	0.026	0.026	0.027
512,000	0.044	0.043	0.043	0.043	0.043	0.043	0.045	0.043	0.043	0.044

Table B.1: Parquet scan operator runtimes in seconds for the input size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.166	0.166	0.164	0.164	0.173	0.167	0.164	0.257	0.179	0.163
2,000	0.343	0.349	0.346	0.344	0.328	0.327	0.327	0.329	0.327	0.327
4,000	0.653	0.67	0.739	0.653	0.652	0.652	0.705	0.667	0.66	0.652
8,000	1.313	1.301	1.306	1.417	1.306	1.304	1.304	1.305	1.303	1.303
16,000	2.616	2.711	2.605	2.606	2.642	2.606	2.621	2.62	2.859	2.659
32,000	5.218	5.225	5.213	5.232	5.207	5.207	5.209	5.368	5.353	5.216
64,000	10.43	10.415	10.416	10.434	10.424	10.418	10.43	10.422	10.602	10.431
128,000	20.822	20.834	20.819	20.834	21.375	22.033	20.82	21.119	20.842	20.831
256,000	41.678	42.43	41.65	41.751	42.065	41.762	41.688	42.237	41.67	41.66
512,000	83.305	83.97	83.324	83.881	83.346	83.69	83.959	83.493	83.504	83.39

Table B.2: Filter operator runtimes in seconds for the input size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	296	297	294	289	279	257	262	343	261	231
2,000	432	432	437	415	396	478	406	405	401	398
4,000	735	735	800	714	712	709	769	723	733	715
8,000	1373	1373	1366	1482	1373	1364	1362	1362	1359	1360
16,000	2829	2829	2712	2701	2739	2702	2715	2714	2950	2749
32,000	5368	5368	5362	5368	5348	5340	5345	5499	5481	5422
64,000	10642	10642	10636	10660	10657	10638	10714	10640	10824	10652
128,000	21225	21225	21217	21229	21822	22433	21212	21511	21269	21228
256,000	43175	43175	42403	42550	42810	42552	42435	42982	42410	42403
512,000	85453	85453	84761	85314	84813	85199	85405	84965	84969	84840

Table B.3: Total query runtimes in milliseconds for the input size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

RE2 accelerated version of Dremio

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.004	0.005	0.003	0.003	0.003	0.004	0.004	0.003	0.004	0.002
2,000	0.003	0.003	0.002	0.002	0.003	0.002	0.003	0.003	0.002	0.003
4,000	0.004	0.003	0.004	0.003	0.003	0.004	0.003	0.003	0.003	0.003
8,000	0.005	0.004	0.004	0.004	0.003	0.003	0.003	0.002	0.003	0.003
16,000	0.01	0.004	0.004	0.005	0.004	0.004	0.004	0.004	0.004	0.003
32,000	0.005	0.006	0.005	0.005	0.005	0.005	0.007	0.006	0.006	0.005
64,000	0.013	0.014	0.014	0.014	0.013	0.013	0.014	0.014	0.013	0.015
128,000	0.018	0.018	0.019	0.018	0.018	0.017	0.019	0.017	0.018	0.021
256,000	0.027	0.027	0.026	0.025	0.027	0.025	0.025	0.025	0.026	0.025
512,000	0.044	0.043	0.042	0.041	0.046	0.042	0.043	0.044	0.043	0.042

Table B.4: Parquet scan operator runtimes in seconds for the input size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.006	0.006	0.004	0.004	0.004	0.006	0.007	0.006	0.022	0.004
2,000	0.007	0.007	0.007	0.007	0.011	0.007	0.007	0.008	0.007	0.011
4,000	0.011	0.011	0.011	0.012	0.011	0.016	0.012	0.011	0.012	0.012
8,000	0.022	0.021	0.023	0.021	0.021	0.022	0.022	0.021	0.021	0.022
16,000	0.043	0.041	0.041	0.042	0.045	0.042	0.041	0.042	0.041	0.041
32,000	0.08	0.082	0.08	0.08	0.079	0.081	0.082	0.08	0.08	0.08
64,000	0.158	0.159	0.16	0.161	0.159	0.16	0.159	0.159	0.16	0.159
128,000	0.316	0.314	0.314	0.32	0.314	0.314	0.314	0.315	0.318	0.315
256,000	0.629	0.625	0.625	0.625	0.633	0.626	0.625	0.626	0.633	0.626
512,000	1.253	1.249	1.248	1.252	1.271	1.252	1.251	1.261	1.254	1.256

Table B.5: Filter operator runtimes in seconds for the input size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	142	144	117	135	96	114	99	103	174	83
2,000	82	82	82	78	105	104	92	90	80	89
4,000	74	74	75	90	76	100	80	80	86	87
8,000	91	91	100	92	80	85	82	83	94	95
16,000	141	141	152	150	142	150	135	137	133	132
32,000	222	222	219	231	219	224	225	225	222	220
64,000	391	391	383	400	382	383	395	386	387	386
128,000	799	799	711	718	711	706	715	710	715	771
256,000	1369	1369	1377	1377	1378	1373	1387	1384	1449	1376
512,000	2700	2700	2707	2821	2731	2737	2826	2708	2698	2756

Table B.6: Total query runtimes in milliseconds for the input size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

Tidre accelerated version of Dremio

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.003	0.005	0.003	0.003	0.003	0.003	0.002	0.003	0.002	0.002
2,000	0.004	0.003	0.002	0.002	0.003	0.002	0.002	0.002	0.002	0.003
4,000	0.004	0.003	0.004	0.003	0.003	0.002	0.003	0.004	0.002	0.003
8,000	0.003	0.003	0.003	0.003	0.004	0.004	0.003	0.002	0.002	0.002
16,000	0.008	0.004	0.004	0.003	0.003	0.005	0.005	0.004	0.004	0.004
32,000	0.005	0.005	0.006	0.005	0.005	0.007	0.006	0.004	0.006	0.004
64,000	0.013	0.013	0.013	0.013	0.015	0.014	0.013	0.013	0.013	0.013
128,000	0.02	0.021	0.019	0.017	0.017	0.017	0.018	0.017	0.017	0.017
256,000	0.025	0.025	0.025	0.026	0.026	0.025	0.025	0.026	0.026	0.026
512,000	0.042	0.058	0.041	0.043	0.04	0.041	0.041	0.042	0.044	0.043

Table B.7: Parquet scan operator runtimes in seconds for the input size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	0.001	0.001	0.003	0.001	0.001	0.009	0.001	0.007	0.001	0.001
2,000	0.001	0.001	0.001	0.002	0.001	0.001	0.005	0.003	0.003	0.002
4,000	0.001	0.001	0.001	0.001	0.005	0.001	0.001	0.001	0.001	0.003
8,000	0.004	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
16,000	0.002	0.002	0.002	0.006	0.002	0.002	0.002	0.002	0.002	0.002
32,000	0.003	0.008	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
64,000	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006
128,000	0.012	0.012	0.014	0.013	0.012	0.012	0.012	0.017	0.012	0.012
256,000	0.024	0.024	0.024	0.024	0.024	0.024	0.024	0.024	0.024	0.024
512,000	0.048	0.048	0.047	0.047	0.047	0.052	0.047	0.047	0.047	0.047

Table B.8: Filter operator runtimes in seconds for the input size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

input size	1	2	3	4	5	6	7	8	9	10
1,000	131	131	143	99	88	116	106	111	83	81
2,000	91	91	86	67	82	87	68	68	71	70
4,000	72	72	70	81	79	61	80	79	73	62
8,000	64	64	59	57	61	62	75	57	56	59
16,000	96	96	111	103	93	105	99	94	104	105
32,000	145	145	141	146	139	174	182	137	136	135
64,000	232	232	231	236	236	232	228	224	239	231
128,000	407	407	412	405	407	402	406	411	468	398
256,000	759	759	762	769	768	764	762	851	773	761
512,000	1507	1507	1472	1473	1471	1522	1536	1489	1549	1491

Table B.9: Total query runtimes in milliseconds for the input size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

B.1.2. Batch size benchmark

Vanilla Dremio

batch size	1	2	3	4	5	6	7	8	9	10
1,000	0.021	0.021	0.021	0.014	0.011	0.011	0.011	0.01	0.01	0.01
2,000	0.008	0.008	0.007	0.007	0.007	0.007	0.007	0.007	0.006	0.007
4,000	0.006	0.006	0.005	0.006	0.005	0.006	0.006	0.006	0.006	0.005
8,000	0.006	0.007	0.005	0.005	0.005	0.005	0.006	0.005	0.005	0.005
16,000	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.006	0.005	0.005
32,000	0.005	0.005	0.007	0.005	0.007	0.005	0.005	0.005	0.005	0.005
64,000	0.005	0.005	0.007	0.005	0.005	0.005	0.005	0.005	0.063	0.005

Table B.10: Parquet scan operator runtimes in seconds for the batch size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	5.469	5.342	5.467	5.331	5.405	5.285	5.289	5.296	5.333	5.322
2,000	5.315	5.282	5.282	5.604	5.352	5.337	5.333	5.559	5.444	5.336
4,000	5.33	5.324	5.322	5.371	5.683	5.319	5.359	5.316	5.387	5.341
8,000	5.327	5.442	5.518	5.322	5.316	5.314	5.344	5.406	5.343	5.325
16,000	5.319	5.317	5.361	5.465	5.42	5.326	5.387	5.322	5.618	5.497
32,000	5.332	5.316	5.626	5.318	5.316	5.315	5.317	5.324	5.314	5.314
64,000	5.321	5.368	5.323	5.322	5.864	5.318	5.315	5.316	5.315	5.318

Table B.11: Filter operator runtimes in seconds for the batch size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	5545	5624	5738	5545	5630	5502	5547	5486	5523	5505
2,000	5489	5466	5455	5795	5526	5519	5504	5720	5609	5499
4,000	5488	5495	5490	5529	5846	5493	5519	5483	5600	5491
8,000	5478	5598	5674	5473	5467	5462	5504	5557	5493	5472
16,000	5466	5463	5520	5661	5567	5470	5537	5466	5768	5650
32,000	5476	5463	5804	5472	5475	5457	5456	5467	5451	5453
64,000	5457	5514	5461	5465	6002	5455	5453	5455	5512	5454

Table B.12: Total query runtimes in milliseconds for the batch size benchmark on vanilla Dremio. The measurement data is shown for 10 consecutive repeats.

RE2 accelerated version of Dremio

batch size	1	2	3	4	5	6	7	8	9	10
1,000	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
2,000	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
4,000	0.003	0.004	0.004	0.003	0.003	0.004	0.003	0.006	0.004	0.003
8,000	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
16,000	0.004	0.004	0.004	0.004	0.004	0.004	0.003	0.003	0.004	0.004
32,000	0.006	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.006	0.004
64,000	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004

Table B.13: Parquet scan operator runtimes in seconds for the batch size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	0.127	0.126	0.127	0.126	0.126	0.126	0.126	0.129	0.126	0.126
2,000	0.104	0.104	0.103	0.106	0.104	0.106	0.104	0.104	0.104	0.104
4,000	0.093	0.093	0.092	0.092	0.092	0.094	0.093	0.144	0.092	0.093
8,000	0.086	0.086	0.086	0.088	0.086	0.086	0.086	0.086	0.086	0.086
16,000	0.084	0.083	0.082	0.083	0.083	0.083	0.083	0.083	0.085	0.083
32,000	0.085	0.081	0.081	0.08	0.081	0.08	0.08	0.081	0.126	0.084
64,000	0.079	0.079	0.079	0.079	0.079	0.08	0.08	0.079	0.079	0.081

Table B.14: Filter operator runtimes in seconds for the batch size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	251	250	251	249	249	252	251	254	248	248
2,000	225	228	226	230	228	229	226	226	233	226
4,000	216	217	216	219	213	216	214	311	216	215
8,000	212	208	207	212	212	206	210	210	207	207
16,000	206	202	203	204	208	204	204	203	206	203
32,000	206	200	200	202	201	201	201	203	289	230
64,000	202	200	204	257	211	200	202	202	198	201

Table B.15: Total query runtimes in milliseconds for the batch size benchmark on the RE2 accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

Tidre accelerated version of Dremio

batch size	1	2	3	4	5	6	7	8	9	10
1,000	0.004	0.004	0.005	0.003	0.004	0.004	0.004	0.004	0.004	0.003
2,000	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
4,000	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
8,000	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.005
16,000	0.005	0.005	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
32,000	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
64,000	0.004	0.004	0.006	0.006	0.004	0.004	0.004	0.004	0.004	0.004

Table B.16: Parquet scan operator runtimes in seconds for the batch size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	0.012	0.012	0.013	0.012	0.012	0.012	0.012	0.012	0.012	0.011
2,000	0.007	0.007	0.008	0.008	0.008	0.008	0.007	0.007	0.007	0.007
4,000	0.005	0.006	0.005	0.005	0.005	0.011	0.005	0.005	0.006	0.005
8,000	0.004	0.004	0.004	0.004	0.004	0.004	0.005	0.004	0.005	0.005
16,000	0.004	0.004	0.004	0.004	0.004	0.009	0.004	0.008	0.004	0.004
32,000	0.003	0.003	0.003	0.003	0.004	0.003	0.003	0.004	0.004	0.003
64,000	0.003	0.003	0.003	0.003	0.004	0.003	0.003	0.003	0.003	0.003

Table B.17: Filter operator runtimes in seconds for the batch size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

batch size	1	2	3	4	5	6	7	8	9	10
1,000	136	132	136	130	132	134	134	133	140	130
2,000	126	130	128	127	127	128	129	127	126	131
4,000	127	123	129	128	125	130	126	132	125	123
8,000	123	134	122	128	126	122	122	124	123	126
16,000	156	153	124	178	121	127	122	135	129	123
32,000	123	122	122	123	121	120	123	124	125	125
64,000	130	122	163	153	142	122	121	127	124	122

Table B.18: Total query runtimes in milliseconds for the batch size benchmark on the Tidre accelerated version of Dremio. The measurement data is shown for 10 consecutive repeats.

B.2. Dask integration

This section provides the measurement data of the benchmarks as performed on vanilla Dask, the RE2 accelerated version of Dask, and the Tidre accelerated version of Dask. The measurement values for each repeat of the experiment are averaged before the result is recorded. As such, only the average runtime per configuration of the benchmark is shown.

B.2.1. Input size benchmark

input size	vanilla filter	re2 filter	tidre filter (unaligned)	tidre filter (aligned)
1,000	0.002956151	0.003961324	0.000338633	0.000299215
2,000	0.005610624	0.007038831	0.000467936	0.000365177
4,000	0.011072556	0.011637210	0.000563383	0.000410079
8,000	0.021036148	0.022220611	0.000975131	0.000575462
16,000	0.041970411	0.041655460	0.001475413	0.000755469
32,000	0.085360527	0.083416541	0.003065347	0.001402139
64,000	0.165530045	0.164292891	0.005296150	0.002300977
128,000	0.330947081	0.325950702	0.010447104	0.004223585
256,000	0.667097171	0.647831678	0.020582278	0.007971366
512,000	1.350455919	1.287713050	0.038978179	0.015311320
1,024,000	2.636155525	2.539052804	0.080622911	0.029798428
2,048,000	5.390320936	5.136881192	0.155350526	0.060254335
4,096,000	11.20614918	10.32821965	0.318155924	0.122184117

Table B.19: Filter operator runtimes in seconds for the input size benchmark on vanilla Dask, the RE2 accelerated version of Dask, and the Tidre accelerated version of Dask.

B.2.2. Batch size benchmark

batch size	vanilla filter	re2 filter	tidre filter (unaligned)	tidre filter (aligned)
64,000	10.40990583	10.20908530	0.327896753	0.139211813
128,000	10.32164494	10.14421232	0.324502706	0.130489905
256,000	10.44996277	10.13163725	0.317209084	0.161064863
512,000	10.54750529	10.08743151	0.308020035	0.121752500
1,024,000	10.53775946	10.08665307	0.313249429	0.118306795
2,048,000	10.94178239	10.03863088	0.307125488	0.115442832
4,096,000	10.50310659	10.01158491	0.317032019	0.116899013
8,192,000	11.04190683	11.55125530	0.300794839	0.116959571

Table B.20: Filter operator runtimes in seconds for the batch size benchmark on vanilla Dask, the RE2 accelerated version of Dask, and the Tidre accelerated version of Dask.

B.3. Dask distributed integration

This section presents the measurement data of the benchmarks as performed on Dask distributed. These experiments records the end-to-end runtime of the query.

B.3.1. Input size benchmark

input size	0 / 3 acc. workers	1 / 3 acc. workers	2 / 3 acc. workers	3 / 3 acc. workers
256,000	0.792335669	0.719644268	0.621766328	0.281761566
512,000	1.932520270	1.443918704	0.995650529	0.566291173
1,024,000	3.574651559	2.812757293	1.925983707	0.911388675
2,048,000	3.674423217	3.690255522	2.799145738	1.126245737
4,096,000	7.352140386	7.128950277	5.511878490	2.112696448

Table B.21: Total query runtimes in seconds for the input size benchmark on the accelerated version of Dask distributed. A worker pool with a size of three is configured. Four configurations with different fractions of accelerated workers in the pool are considered.

B.3.2. Batch size benchmark

batch size	0 / 3 acc. workers	1 / 3 acc. workers	2 / 3 acc. workers	3 / 3 acc. workers
64,000	5.211608012	3.179878354	2.461477398	1.669256130
128,000	5.015094598	3.055605371	2.392320315	1.447363575
256,000	5.437679171	3.425900141	1.957973798	1.504485567
512,000	5.666631658	4.829833467	4.162257154	1.623860279
1,024,000	7.146155277	5.840875784	5.125004649	2.000774780
2,048,000	7.453825195	7.449797232	5.615672389	2.283123970
4,096,000	14.41515847	11.35234797	7.963922739	4.584929863
8,192,000	14.55684224	11.68263487	8.152046481	4.678944428

Table B.22: Total query runtimes in seconds for the batch size benchmark on the accelerated version of Dask distributed. A worker pool with a size of three is configured. Four configurations with different fractions of accelerated workers in the pool are considered.