Master Computer Science

Generic, End-to-end Computation Offloading between Processing and Storage Systems.

Name:	Sebastiaan Alvarez Rodriguez
Student ID:	s1810979
Date:	July 26, 2021
Specialisation:	Advanced Computing and Systems
1st supervisor:	Dr. Alexandru Uta Leiden University
2nd supervisor:	Dr. Carlos Maltzahn University of California, Santa Cruz

Master's Thesis in Computer Science

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

Abstract

Data processing systems in the cloud currently favor disaggregation of compute and storage, so that compute- and storage clusters can scale independently. In modern-day distributed data processing, processing speed is limited by CPUs as shown in several recent studies. Therefore, either storage or networking layers could take over some form of computing from CPUs. In database systems, the concept of 'computation pushdown' is widespread. Here, the idea is to place computation closer to the required data. This concept could also be applied to large-scale data processing Systems, to alleviate the limitation on CPU processing speed in an efficient fashion. Recent work proposed frameworks with a similar goal. However, these frameworks either depend on sophisticated and costly hardware like FPGAs and Smart SSDs, or they are closed-source and not customizable. There exists no *general, customizable* pushdown system, capable of pushing down any computation in large-scale data processing systems.The advent of programmable storage devices opens up the possibility to migrate general (user-defined) computations into the storage layer, consisting of general-purpose hardware.

Closing the gap, in this work, we propose the first design and implementation of an end-to-end computation offloading framework based on Apache Spark and Ceph RADOS. We provide a thorough performance evaluation, and share our experiences with implementing this concept for practitioners.

Contents

1 Introduction

Statistics [1] show that the projected yearly amount of generated data in 2021 exceeds 79 zettabytes, or approximately $216,438,356$ terabyte every day. The yearly amount of generated data follows a strong exponential curve, where we double the amount of generated amount of data every 3 years. This trend is expected to continue in the future, as digitization and data processing become ever more important to industry, academia and government. All of this data in itself is not useful. We use data processing to extract features, patterns, statistics, etcetera from this gigantic amount of raw data for many different purposes. We use distributed data processing systems to process data which is too large or would take a long time to be processed by a single machine.

In modern-day data processing, Distributed Data Processing Systems (DPSs) such as Dask [2], Apache Spark [3] and Snowflake [4] have become pervasive. These systems are used in most domains of science, to power discoveries and breakthroughs in a wide range of wildly varying subjects, from crime fighting [5] to disease outbreak management [6, 7]. Additionally, many industries use distributed data processing systems to optimize production loads and to gain advanced analytical insights in their performance and research, such as CERN [8].

In modern distributed data processing, systems run in on-premise resources as well as cloud infrastructure. In both cases, we need these systems to be as efficient as possible [9]. With cloud infrastructure, we pay per compute cycle, per stored byte and per transmitted byte. More efficient processing directly results in less processing costs. With privately owned datacenters, the same relation holds indirectly. More efficient processing results in lesser power consumption, lesser hardware wear, and lower carbon footprint. Indirectly, increasing processing efficiency lowers processing costs. In this work, we investigate how to improve efficiency for data processing in modern datacenters.

In modern data processing systems, there are two design approaches for mapping computation and storage. First, there is the fully aggregated approach, taken by Hadoop [10] and HDFS [11], improving system performance through locality [12]. Second, there is the fully disaggregated approach, used by cloud-based systems like Databricks or Snowflake, which offload storage to Amazon S3 [13]. This last implementation is currently favored, because doing so comes with a great benefit: it allows to scale computation and storage completely independently [12]. Disaggregated systems also have several inefficiencies, however. The most important inefficiency is that data has to be transported from the storage to the compute layer, and cannot be operated upon locally. In fully disaggregated systems, storage is usually composed of single-functionality sets of machines used with the sole purpose of feeding data through the network to compute clusters [14].

Computational storage is a known concept [15], improving regular storage infrastructure with the integration of some form of compute resources, stepping outside of the traditional storage architecture. The compute resources can either be placed directly with the storage infrastructure, or between the compute and storage layers. The goal is to enable parallel computation and to alleviate constraints on existing compute, memory, storage, and I/O resources across cluster infrastructure. This goal is reached by executing computations on the added compute resources, which are closer to the data on storage hardware than the main compute cluster. An example of an implementation would be the recent release of SmartSSDs by Samsung Semiconductor Inc [15, 16], which are high-performance SSD devices with on-board compute abilities and RAM. In software, SkyhookDM [17] allows practitioners to offload computations to object storage processes, essentially allowing regular storage servers to function as programmable storage systems.

In modern clusters, distributed data processing system performance is limited by the

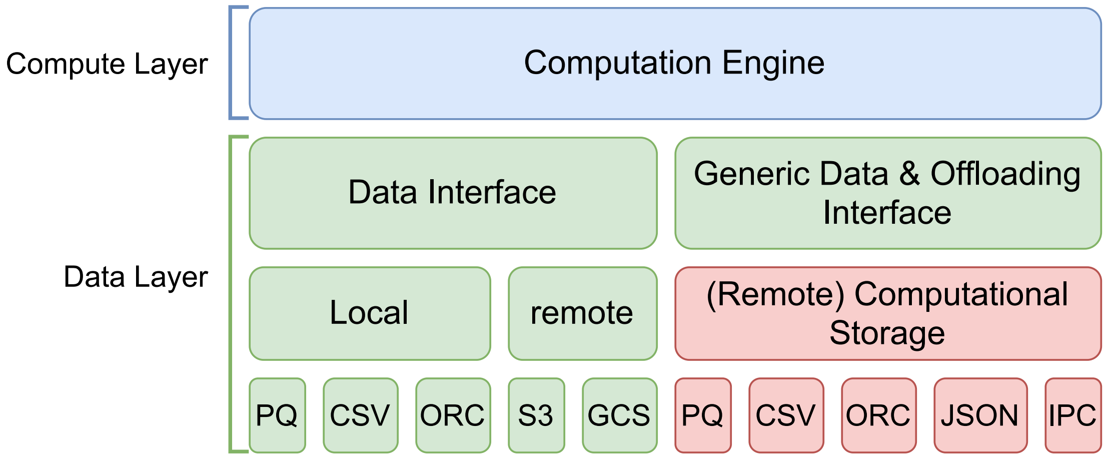


Figure 1: An overview of a typical distributed data processing system, consisting of a *Compute layer* and *Data layer*. On the right side of the *Data Interface* block, we show the addition of programmable storage as a potential alternative for providing data to computation engines, and alleviating their CPU workloads by taking over some computation workloads.

total CPU power of a cluster [18, 19]. Network and storage devices are rapidly increasing in performance, while CPUs are reaching the physical processing speed constraints defined by nature [20]. To alleviate this CPU-boundedness for modern systems, either storage or network infrastructure could help with some form of computing. Computation offloading is at the core of this concept. Cloud providers slowly begin to provide services for computation offloading to storage, noticing the importance and potential benefit for performance. For example, Amazon recently offered S3 Select [21] to its users, which is an improvement on S3 [13] that allows users to offload specific SQL-like queries into S3. An example of a real-world deployment embracing the concept of programmable storage for data-intensive task offloading would be the Alibaba cloud-native relational database PolarDB [22]. Cloud providers also begin to adopt the concept of programmable storage.

Until now, efforts towards computation offloading for distributed data processing systems are limited. Related work always requires expensive, specialized hardware such as SmartSSDs, or only allows offloading a limited subset of computations without options for performance tuning, such as S3 Select's SQL-like queries. Although interesting, these types of systems are either expensive or have very limited applicability due to constraints on computation type. The concept of offloading *generic* computation from distributed data processing systems to general-purpose, *generic* hardware is relatively under-explored field, with much room for improving efficiency when coupling compute clusters with storage systems.

With the advent of programmable storage, one can combine the two design approaches (i.e., co-location and full disaggregation between compute and storage), where we keep the benefits of independent storage and compute scaling of disaggregated systems, while simultaneously providing the locality benefits of aggregated networks. Paramount to the efficiency of this network is the notion of computation offloading, as this principle allows us to offload computations to the storage layer, where the offloaded computations can operate locally on the data. A typical distributed data processing system consists of a compute layer and data layer: the compute layer contains the compute engine, optimization engines etc. The data layer consists of an interface to read data, be that local or remote data. When integrating computation offloading, we essentially need to add an interface that allows to offload computations and read computed-upon data back. Figure 1 depicts this concept.

To allow the clusters to scale independently, we achieve *end-to-end* offloading by leveraging a decentralized object-based storage system: when performing data requests and offloading computations, we directly read and offload these requests to the node containing the data, without the involvement of any centralized forwarding server or other third party. The node containing the data services the request, and returns the resulting data directly to the client. With only direct communication and computation offloading, clusters can scale up without a centralized server choking point, as is common with many storage solutions.

In this work we propose and systematically explore the performance of a *generic* computation offloading framework, which can combine existing computation engines (e.g., Spark) with computation-enabled storage engines (e.g., SkyhookDM). This differs from 'regular' computation offloading approaches in that any computation could be offloaded, as long as it is supported by the chosen distributed data processing system and by the chosen storage backend. While we envision this framework to be able to combine any computation engine with any computation-enabled storage, our prototype focuses on offloading computation from Spark to SkyhookDM.

We provide the first-of-its-kind conceptual design and a practical implementation of this concept, called SparkHook, which uses Apache Spark [3] as distributed data processing system and SkyhookDM [17] as storage backend. We seize this novel opportunity we ourselves created: we experiment with offloading computation expressions directly on generic storage servers. This research is unique and could lead to new datacenter designs, focusing on intelligent cooperation between compute and storage clusters.

In this paper, we aim to answer the following research questions:

RQ1 How can we design and prototype generic end-to-end computation offloading between Distributed Processing Systems and programmable storage?

RQ2 How to assess the performance of SparkHook, our prototype, in a end-to-end fashion?

RQ3 What is the performance of our prototype end-to-end computation offloading framework?

RQ4 What are the lessons learned and how can we make this production-ready, achieving good performance and efficiency?

Contributions The main contributions of this work are:

- The first-of-a-kind exploration of computation offloading through a framework which ships computation from compute clusters to storage clusters. Our initial results are promising, capable of offloading computations effectively. In Section 3, we go over the design of a system applying this. We explain the conceptual issues that arise with computation offloading in Section 6, and we explain how they could be alleviated.

- SparkHook: a novel system that offloads computations from compute- to storage layers. We explain the design and implementation of SparkHook in Section 3. In Section 5, we experiment with offloading computations and show that computations are indeed offloaded. The results are good: computing resources of the storage layer are efficiently leveraged during distributed data processing, alleviating a majority of CPU utilization in the compute cluster.

- An extensible, modular Spark+Ceph [23]+SkyhookDM experimentation framework. We used this framework to obtain all results in a reproducible, systematic manner. Experiments are modular and declaratively defined. The system has many modules that are useful to users of Spark, Ceph, GENI, Prometheus and Grafana. We explain the implementation of the experimentation framework in Section 4.

- Functional, open-source implementations of all frameworks and systems. All our systems are available for everyone to download and use. All systems are practical, well-documented, organized repositories. For availability information, see Section 4.8.

Team Collaboration This research is a cooperation between Leiden University (LU), University of California Santa Cruz (UCSC), and the Institute for Research and Innovation in Software for High-Energy Physics (IRIS-HEP). Our research team consists of Sebastiaan Alvarez Rodriguez (LU), Jayjeet Chakabroty (IRIS-HEP), Ivo Jimenez (UCSC), Alexandru Uta (LU), Carlos Maltzahn (UCSC), Jeff LeFevre (UCSC), Aaron Chu (UCSC), Jianshen Liu (UCSC). In this Thesis, we clearly and accurately distinguish our work and work from the team contributions, using footnotes and terms such as 'the team' versus 'we, us'.

For a comprehensive overview of originality, contributions and credits, see Section Declaration of Originality.

2 Background

In this Section, we provide necessary background information to better understand what distributed data processing systems are, what row- and columnar data formats and types are, and we provide useful information about each framework we use for our system.

2.1 Distributed Data Processing Systems

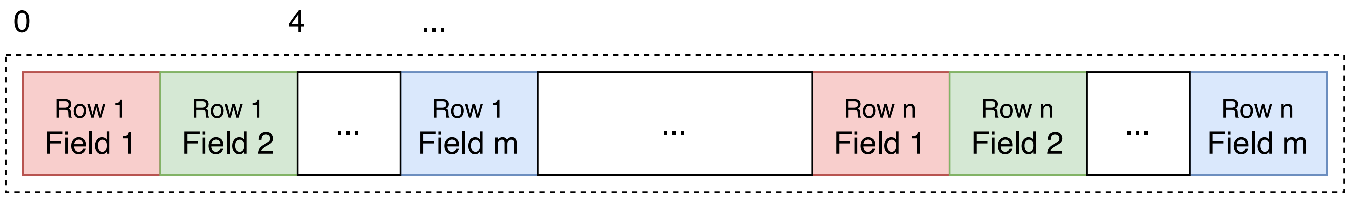
In the most formal sense, a distributed data processing system is a framework or tool that allows a user to execute code to process data in a distributed fashion, either distributed across multiple machines connected through a network or parallelized on a single machine.

In today's world, distributed data processing ecosystems are extensive and touch many application domains, such as stream and event processing [24, 25, 26, 27], distributed machine learning [28], and graph processing [29].

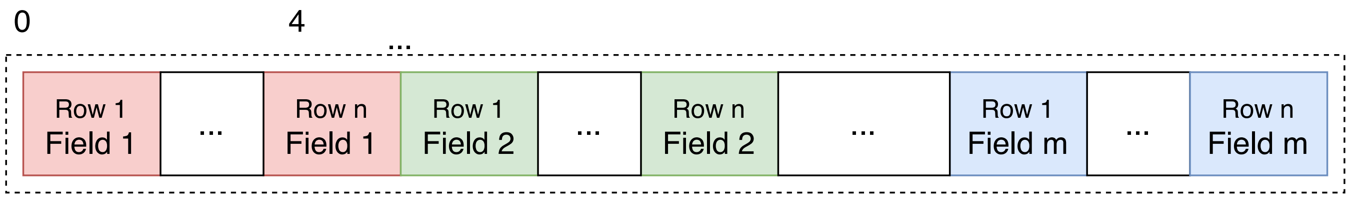
Although this is by no means a requirement, modern distributed data processing systems take over as much work from users as they can. They restrict interfaces to a point where practitioners only need to express their analytical data queries as a series of operations on singular data elements, such as rows of a table, or single nodes of a tree. The used distributed data processing system is in charge of every execution-related task, such as distributing individual tasks, sharing results between cluster nodes, handling fault-tolerance, etc. Additionally, it performs execution-time optimizations, such as distributed reading, query optimization, lazy reading, data caching, and many other common optimizations. The idea behind this principle is that users can and will make mistakes, if they have to implement and control this large amount of features themselves. Because of this responsibility for the execution of user-provided workloads, distributed data processing systems are in an interesting position: They often have a complete overview of all tasks to be executed, of available hardware, and of the data size, location and other useful metadata. Distributed data processing systems use this large amount of insights to extensively optimize all user-specified tasks, using so-called query optimizers [30].

The history of distributed data processing systems started with Apache Hadoop [10] back in the late 2000's. Hadoop was one of the first open-source distributed data processing systems, using the MapReduce paradigm at its core. In MapReduce, users define three phases. In the Map phase, data elements are mapped to key-value pairs. In the Shuffle phase, the key-value pairs are redistributed between nodes based on their element key. In the Reduce phase, the redistributed key-value pairs are aggregated using a function, decreasing the number of elements. Multiple MapReduce programs may be executed after one another, to allow more complex operations.

Hadoop quickly became the most used distributed data processing system around the world, throughout both science and industry. Hadoop had a large community and was improved a lot over the years. In the early 2010's, Apache Spark [3] was created as a successor of Hadoop. Apache Spark is a distributed data processing system, just like Hadoop. It provides more powerful and flexible operations than the static map and reduce operations found in the MapReduce-core of Hadoop. Many improvements came with Spark, such as a state-of-the-art Directed Acyclic Graph (DAG) dynamic job scheduler, a query optimizer, and a physical execution engine. Due to the large improvements, Spark queries are reported to be tens to hundreds of times faster than similar Hadoop queries. Apache Spark is the most used distributed data processing system today. Many other different frameworks are used with Spark at their core, to further optimize the Spark execution environment for highly specialized types of data processing. For example, GraphX [29] brings efficient graph processing to Spark. Mllib [31] extends Spark with machine learning capabilities. Spark SQL brings SQL querying to Spark.



(a) Row compressed storage.



(b) Column compressed storage.

Spark Streaming [32] extends Spark with stream processing (rather than the default batch processing it uses). Apache Kafka [26] implements structured streaming and monitoring on top of Spark Streaming. Apache Storm [25] extends Spark Streaming with computational streaming processing. In this work, we focus on Spark, mainly because it is the most used, most relevant distributed data processing system, and because many frameworks exist with Spark as their core, even further widening applicability and reach of this research.

2.2 Data Formats and Performance Implications

Normally, every distributed data processing system has to read data from some datasource. Internally, these datasources store and provide data in either a row-based format, a column-based format, or a hybrid format. Choosing the right format is essential for performance, because the different format types generally provide quite disparate optimizations. We briefly go over the different formats and their commonly available optimizations and suitable use-cases.

Row-based formats were used since the beginning of computing, and can be categorized in two subcategories. There exist text-based rowwise formats, such as Comma Separated Values (CSV) and JavaScript Object Notation (JSON) formats, and there are binary rowwise formats, such as Apache Avro [33]. Row-based formats store data entries for each row directly after one another in memory, as depicted in Figure 2a. Row-based formats have several benefits, the most important ones being fast row insertions, fast field updates and fast row access. Additionally, we get better cache hit ratios with row-wise storage than with column-wise storage [34] when scanning row-wise through data. In general, it is a good choice to pick row-based formats when computations require to access many entries of a row at once, and when these computations have a high write- and update profile.

When columnar data formats were first researched, this kind of format was quickly disregarded as an efficiently usable format, until, later on, several interesting optimizations and use-cases were found [35]. Reading can be handled very efficiently using columnar formats, provided that only a part of the columns are required [36]. In Figure 2b, note that data entries belonging to the same column are stored directly after each other in memory, forming a chain of columns. Popular columnar formats are the Parquet [37] and Optimized Row Columnar (ORC) [38] formats.

Columnar data formats have multiple read- and compression optimizations available. One of the most important and powerful optimization strategies for columnar data formats is metadata collection. Here, the idea is to split the large columns of data into several partitions, where each partition contains all information of an amount of rows. Each partition keeps metadata about its allocated data, such as "Partition A: integer-type field X has values between -100 and 42". In our example, an accessor wishing to filter data on field X with values higher than 200 can instantly skip reading and comparing all values of partition A. Storing metadata can significantly increase read performance, at the cost of more complicated write operations. There are many design choices when implementing a columnar format, such as the exact metadata entries, the location of the metadata table (header or footer), and the amount of metadata tables (one in total or one per partition).

Column compression is another interesting optimization [34]. We can efficiently apply compression algorithms on columns with frequently repeating values to decrease the storage cost. It is possible to use different algorithms for different columns to maximize compression efficiency. Another optimization property for columnar formats is data co-location, where we store frequently accessed data close together. At read-time, this results in better cache efficiency.

In general, column-based formats are an appropriate choice when doing a lot of filtering and grouping in computations, when accessing few columns of data at once, and when having a low write- and update profile.

Hybrid data formats are still very recent, and try to get the best properties and benefits of columnar and rowwise data formats. One example of a hybrid data format is the Albis [19] storage format. Unlike traditional storage formats, Albis focuses on data access performance at the cost of storage efficiency. It aims to reduce the CPU cost of reading data by using a simple data and metadata format. At the core of Albis is the observation that modern storage and network hardware performance is more efficient than CPU performance, meaning that modern storage formats should shift focus from high data density to better CPU performance-optimization. Hybrid data formats allow for some additional optimization, such as row-based plus column-based partitioning of data.

2.3 Apache Spark

Apache Spark is the de-facto, open-source distributed data processing system, and is written in Scala [39], a JVM language. Apache Spark is the most used distributed data processing system today. Many other frameworks are used with Spark at their core, to further optimize the Spark execution environment for highly specialized types of data processing, such as Mllib [31], GraphX [29], Apache Kafka [26] and Apache Storm [25]. In this work, we focus on Spark, mainly because it is the most used, most relevant distributed data processing system, because it is a quite 'mature' system with very few API-breaking changes in newer releases, and because many frameworks exist with Spark as their core, even further widening applicability and reach of this research.

Spark has a low-level and a high-level data abstraction API. The low-level data abstraction uses Resilient Distributed Datasets (RDDs) to access data in form of partitions. Users can specify operations on rows of data with RDDs. Interaction with Spark used to happen through these RDDs.

Nowadays, most practitioners do not use Spark RDDs directly anymore. Instead, they use the newer high-level API, which adds Dataframes and Datasets to use instead [40]. Dataframes contain series of RDDs and a data schema. Datasets are very much like Dataframes, but are strongly-typed for improved usage. A data schema provides information about the shape and identifiers of the data, such as the (potentially complex) datatype for every column, and column names. Spark forwards this information to Spark Catalyst, the query optimization system of Spark. Catalyst uses the provided schema extensively to optimize distributed execution tasks. Additionally, since Spark knows the shape of the data from the schema, it can make highly compact in-memory representations of the data. With the data shape and identifiers known, Spark allows users to access the data using an SQL engine, called Spark SQL [30]. Dataframes are quite useful to merge differently-shaped data, by looking at columns with common identifiers and types, much like SQL operations left join, join, right join etc. Of course, this functionality can only be used when there exists some common structure between all datasources, however. Finally, users can apply higher-level operations on dataframes than on RDDs, such as groupings and aggregations.

2.4 Apache Arrow and Datasets

Apache Arrow [41] is a framework providing a non-ambiguous, universal data format. The Apache Arrow Dataset API [42], not to be confused with regular Arrow, allows users to make highly efficient, in-memory representations of data, using this format. The Dataset API currently focuses on tabular, potentially larger-than-memory and multi-file datasets. This API supports 'unifying' different data sources, where multiple datasets with potentially different data schemas are represented as a single datasource with a composed schema. It currently has reading implementations for a large amount of different file formats, including the well-known columnar parquet format, and the row-wise CSV format.

The API has developed a lot during the scope of this project, and continues to do so at a high pace. According to the Apache Software Foundation [42], Arrow Datasets will support connecting to databases and to cloud resources like Amazon S3.

The Apache Arrow Dataset API is quite useful for the distributed data processing systems community, as it is capable of reading from and writing to different file formats, cloud storage backends, databases etc. More importantly, it is capable of representing data in a generic, platform-independent and unambiguous manner, and can combine data from different datasources, even when these sources have different data schemes. Finally, the Dataset API can even be leveraged as an interoperability layer between any number of Arrow-enabled applications and frameworks. As such, Arrow Datasets are even more useful for distributed data processing systems, where it is quite common to have complex, composed application stacks in need of data communication and movement. In our work, SparkHook is such a composed application stack, utilizing Spark for computation and Ceph for data storage. We successfully show that Arrow is an efficient interoperability layer between applications on one machine, and even between different machines.

2.5 RADOS & Ceph

Reliable Autonomic Distributed Object Storage (RADOS) [43] is an open-source, distributed object store implementation, created at the University of California at Santa Cruz. Generally, an object store is a storage mechanism where data is split into blobs, or 'objects', of the same size. Each object is given a unique identifier. The complete list of identifiers forming the data is stored as metadata, along with other metadata properties. When the data must be retrieved, object stores reassemble the data by fetching all objects forming the requested data by their unique identifiers. RADOS only differs from this general description in one way: RADOS does not require objects to be all of the same size. Different sizes are allowed.

Ceph [23] is a a unified, distributed storage system. It contains several projects to build distributed storage systems, such as the Ceph Storage Cluster framework. Ceph Storage Cluster is the foundation for all Ceph deployments, and based on RADOS. It has interfaces to uniquely

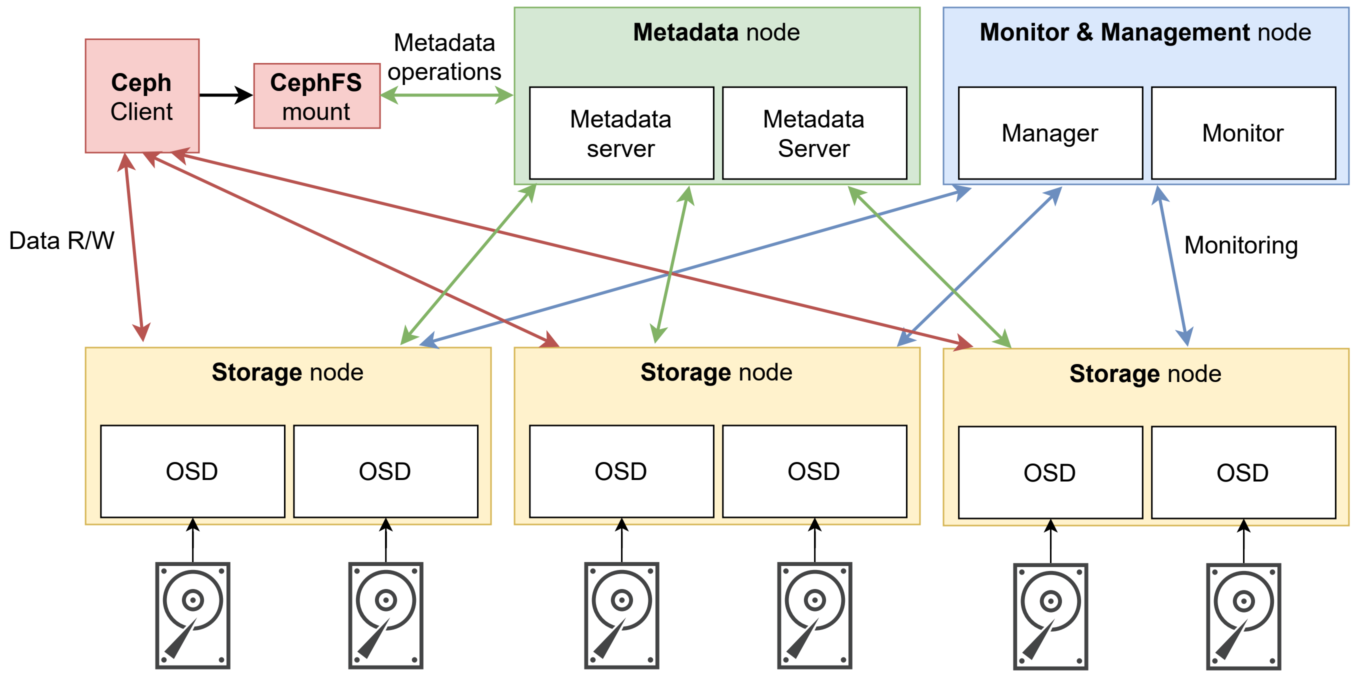


Figure 3: A typical Ceph cluster, consisting of Object Storage Devices (OSDs), Monitors, Managers, and Metadata servers. Interaction between the cluster and a Ceph node with mounted Ceph FileSystem (CephFS) is shown.

deliver object, block and file storage in a unified system. Based on these interfaces, Ceph provides AWS S3, block and POSIX interfaces to interact with the cluster. In Figure 3, we display a typical Ceph cluster. Interestingly, we see that there is no central node as can be found in many other cluster architectures. Instead, OSDs and Monitors communicate directly with each other. Additionally, OSDs service clients directly, without the regular centralized request forward & dispatch. These features bring great scalability improvements and avoid any single point of failure that could bring down the entire system.

The Ceph Storage Cluster implementation provides daemons for so-called Object Storage Devices (OSDs), Monitors (MONs) and Managers (MGRs). OSD daemons implement object storage on a node. MON daemons maintain, among other things, cluster state maps. These maps contain critical state information for the Ceph cluster. MGR daemons provide additional monitoring, and run alongside MONs. The Ceph Filesystem interface also includes a distributed Metadata service (MDS). MDS daemons are responsible for metadata storage for the Ceph File System (CephFS). They implement the POSIX file system interface, map directories and files to objects and coordinate parallel access by file system clients.

In this work, we use the Ceph Storage Cluster as our storage backend for several reasons. First and most important of all, Ceph Storage Cluster enables users to define their *own* storage interface, using the libRADOS API. This impressive piece of engineering opens possibilities to us to define interfaces with functions accepting computation requests. Additionally, Ceph features a class-loading system that allows users to load custom code directly into OSDs. It allows access to the same internal APIs as libRADOS. With classloading, SparkHook could, and actually does execute code remotely on the OSDs of a storage cluster. Another important aspect is that computations are offloaded to objects rather than servers, and objects are dynamically mapped to servers. Should a server fail, then the computations can be completed on a replica of the object on another server. Finally, we use Ceph because it is widely used in industry, and because team members are familiar with Ceph/RADOS.

2.6 SkyhookDM

Normally, a storage system provides simple object-, block-, and file storage interface to applications. The applications cannot access the inner data management functionality of the storage system. Applications that require similar functionality are forced to reimplement large amounts of code, while the required functionality can be found inside the inner layers of the storage system interfaces. To mitigate this issue, *programmable storage systems* expose internal subsystem abstractions as interfaces, such that higher-level services can be created through composition.

Skyhook Data Management (SkyhookDM) [17] implements a programmable storage system interface for the Ceph Object Storage interface, to provide data management functionality directly inside the storage layer. It integrates with Ceph through a plugin for the class-loading system for RADOS, which we previously described. SkyhookDM allows users to grow and shrink their data storage and processing needs by offloading computation and other data management tasks to the storage layer. The goal is to reduce CPU, memory, IO, and network traffic on the client side, by executing queries inside the storage layer. This framework perfectly aligns with the goal of the project: to offload computation from a compute engine to storage layers. Because of this, SkyhookDM forms an integral part of SparkHook's storage clusters.

2.7 Previous Work

Before this Master Research Project (MRP), the authors of this work worked on an Introductory Research Project (IRP). This work is quite related to this work, and forms an essential basis upon which we expand.

The goal of the Introductory Research Project was to make a case for separating compute- and data ingestion layers for distributed data processing systems. We made that case by implementing Arrow-Spark [44], a Generic Data Interface for Apache Spark [3] with the Apache Arrow Dataset API [42]. With Arrow-Spark, Spark is able to read all data from Arrow in a on-par, sometimes dramatically more efficient way than Spark does natively. The generality provided by our implementation becomes plain when looking at an example: If a researcher or developer publishes a new data format with read- and write implementation in Arrow, then Arrow-Spark does not have to be updated in any significant way, and Spark can read from this new format without any additional implementation work. This way, Spark benefits from using Arrow as a more efficient, generic data ingestion layer, and Spark's maintainers could focus on their original goal, which is to create a efficient compute- and analytics engine.

In this work (the MRP), we greatly improved and expanded on Arrow-Spark, implementing entirely different and new functionality. As described above, Arrow-Spark *was* a Generic Data Interface Connector. For this work, we added Computation Aggregation and Generic Computation Offloading functionalities to Arrow-Spark. Computation Aggregation functionality allows Arrow-Spark to obtain a collection of computation expressions from the Spark Core Compute Engine. With the Computation Offloading functionality, we *transparently* translate and offload these computation expressions to Apache Arrow. Doing so opens up exciting new possibilities to offload computations into our Generic Data Ingestion Layer. In this work, we seize this opportunity we ourselves created: we experiment with offloading computation expressions directly on Generic Storage Devices. This research is unprecedented and could lead to new datacenter designs, focusing on intelligent cooperation between compute and storage clusters.

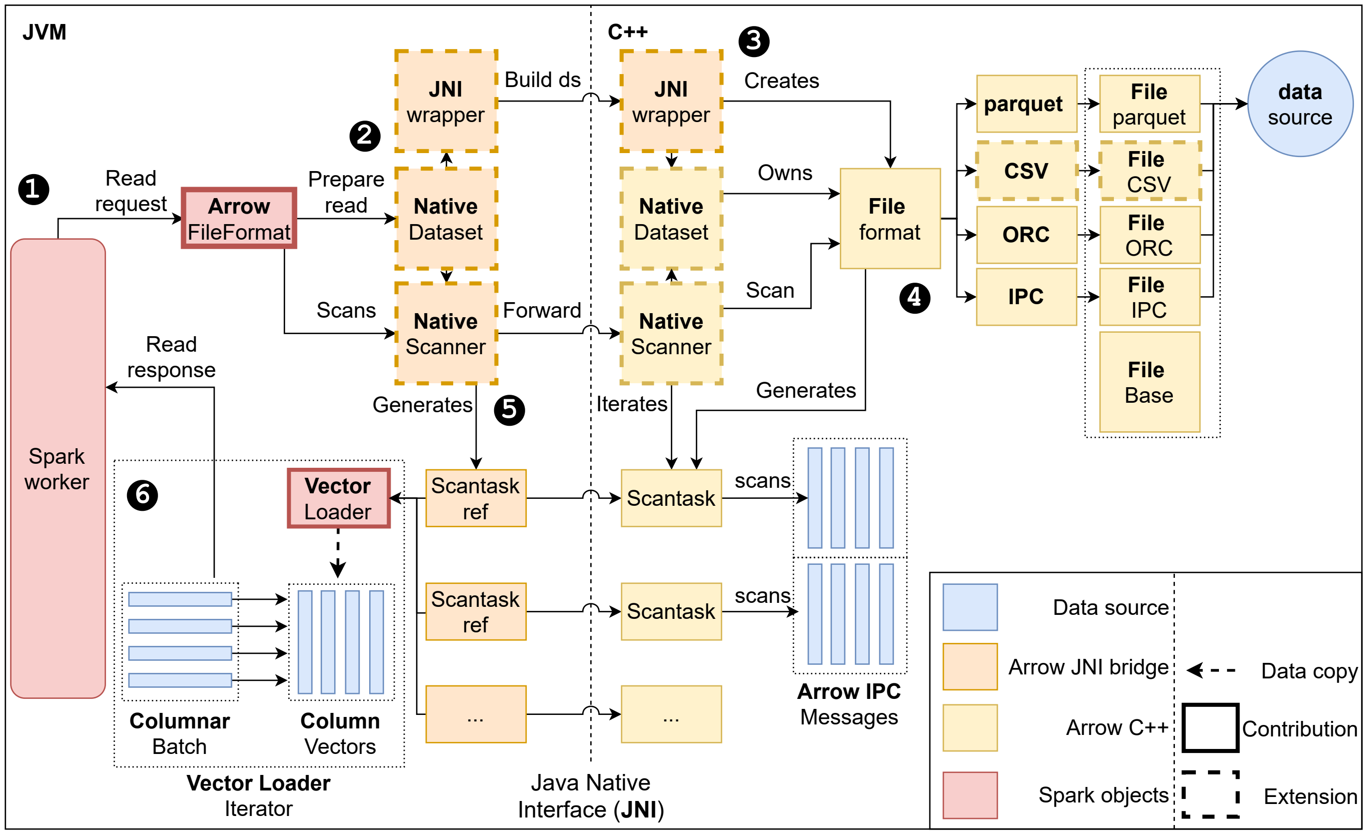


Figure 4: Arrow-Spark design overview, integrating Arrow (right) and Spark (left) using the Java Native Interface (JNI).

2.8 Arrow-Spark

As previously mentioned, we created Arrow-Spark in our Introductory Research Project, and expand on it in this work. Before we can visit the new features in Section 3, we explain the basics.

Arrow-Spark provides an efficient interface between Apache Spark and all Arrow-enabled data sources and formats. Spark is in charge of execution, and Arrow provides the data, using its in-memory columnar formats. In Figure 4, we give a more detailed overview of how we access data through Arrow. Spark is a JVM-based distributed data processing system, whereas the Arrow Dataset API [42] is only implemented in C++ and Python, but not Java. To enable communication, we created a bridge implementation. We briefly visit each stage of the system:

❶ **Reads.** Data transmission is initiated by a read request coming from Spark. Read requests arrive at the *Arrow FileFormat* datasource.

❷ **JVM Dataset.** The *Arrow FileFormat* constructs a JVM-based *Dataset* interface to read data through JNI, using the Arrow Dataset C++ API. The JVM *Dataset* interface forwards all calls through the *JNI wrapper* to C++. The Arrow *Dataset* API interface is constructed on the C++-side, and a reference UUID is passed to the JVM interface counterpart. Through this JVM interface, the *Arrow FileFormat* passes required information about datasources, and initiates data scanning (reading) using a *scanner*.

❸ **Arrow C++ Dataset API.** On the C++ side, a native *Dataset* instance is created. On creation, it picks a *FileFormat*, depending on the type of data to be read. All fileformats have a common parent, *File Base*.

➍ **Data transmission.** The Dataset instance, containing given fileformat, is scanned. This produces a stream of scantasks. Each scantask is to scan a horizontal slice (i.e. an offset and length of every column) of the data.

➎ **Arrow IPC.** Each data batch is placed in memory as an Arrow IPC [45] message, a columnar data format. The address to access each message is forwarded to the JVM, and stored in a *Scantask* reference. Notice that here we make only one additional data copy.

➏ **Conversion.** Each Arrow IPC message is converted to an array of Spark-readable *column vectors*. Because Spark operators exchange row-wise data, we convert the *column vectors* to a row-wise representation by wrapping the vectors in a *ColumnarBatch*, which wraps columns and allows row-wise data access on them. Finally, this batch is returned to Spark. Spark keeps reading batches until the data is exhausted.

3 RQ1: Design and SparkHook Prototype

Offloading computations from distributed data processing systems is a complex process, and requires a clear design. Enforcing the requirement that the system allows for *generic* offloading, from any distributed data processing system to any storage backend, adds even more complexity. In this Section, we discuss different suitable design strategies, and explain our design strategy for offloading computations, using Apache Spark as distributed data processing system and Ceph as our storage backend. This Section answers Research Question 1 (RQ1): How to assess the performance of SparkHook, our prototype, in a end-to-end fashion?

3.1 Fundamental Design Requirements

In order to offload computations, there are many fundamental requirements for the involved distributed data processing system and storage backend to allow for data offloading:

1. We require a system that collects (or aggregates) computations from the distributed data processing system. We need this requirement, to later have computations to push down.

2. The storage backend must have a computation offloading interface, to accept aggregated computations in-order. We need this requirement, so we can offload computations to the storage backend. The computation ordering is important, since computations do not need to be commutative. By ensuring the interface maintains computation provider ordering, we ensure consistent and correct computation ordering in the storage backend.

3. There has to be a common representation of computation expressions between the distributed data processing system and the storage backend. Computations must be provided to the storage backend in this representation. We need this requirement to ensure that computations are interpretable in the storage backend, and retain their meaning when executed.

4. The chosen storage backend must present data in a way that the distributed data processing system can read. We need this requirement to ensure that the distributed data processing system can accept the data after applying computations on it.

5. The chosen storage backend must be able to execute computations. We need this requirement, since we cannot offload computations to the storage backend otherwise.

This set of requirements to design a computation offloading system is both minimal and complete: We cannot remove any requirement and generate only valid designs, and we generate only valid designs using the current requirement set, since it enforces computations are aggregated in the distributed data processing system, can be pushed down and executed in the storage layer, and results can be read by the distributed data processing system. This is exactly what computation offloading requires to be possible and effective.

Additionally, our goal is to make such a system **modular**—one should be able to individually swap our distributed data processing system and storage backend modules, for other similarly behaving systems, and still have a fully functioning computation offloading system. To design a *modular* offloading system, we require even more:

1. There must be a standard computation expression framework. Each distributed data processing system computation aggregation system must provide computations using the standard computation expression framework. We need this requirement so we can

pushdown computations from any distributed data processing system to any storage backend.

2. There must be a standard computation offloading interface between distributed data processing systems and storage backends, which accepts aggregated standard computation expressions from distributed data processing systems and forwards them to storage backends. We need this requirement to be able to independently switch distributed data processing systems and storage backends.

3. There must be a standard data interface between distributed data processing systems and storage backends for reading and writing. We need this requirement to be able to independently switch distributed data processing systems and storage backends.

With these additional requirements, the requirement set to design modular computation offloading systems is again minimal and complete: We cannot remove any requirement and generate only valid designs, and we generate only valid designs using the current requirement set, since it enforces pushed down computations are represented in a standard manner, these standard computations are accepted using a standard computation offloading interface, and the resulting data is also represented in a standard manner, and accepted in a standard data interface. Combined, these three requirements ensure that both distributed data processing systems and storage backend components can be individually swapped, since all communication between the modules is standardized. This means that our extra rules make the requirement set complete. The set is also minimal, since removing any of the three rules would make designs possible with non-standard inter-module communication, meaning we would end up with non-modular systems.

3.2 High-level Design

When all Design Requirements are satisfied, pushing down operations consists of several steps:

1. (*Optional*) Use the generic data interface to write data.

2. Obtain computations from the distributed data processing system.

3. Translate computations from distributed data processing system-specific to generic expressions.

4. Push computations from the distributed data processing system to the generic offloading interface.

5. Push computations from the generic offloading interface to the storage backend and execute them.

6. Send the data from the storage backend to the client.

7. Read the data in from the distributed data processing system, through the generic data interface.

Technically, the generic computation offloading interface and the generic data interface could be completely separate systems. This is impractical, however, as we would need a way inside the distributed data processing system to find out to which exact storage node the computations were pushed to, and send our read requests there. This is not always feasible, as some storage systems use load-balancing techniques, meaning similar requests could end up at different storage nodes. Additionally, most storage systems work using a simple request-and-respond system, which makes delayed computations and result reading requests impractical. In fact, not all storage systems maintain state information about requests, making separate computation and result reading a burden to implement.

Instead, we propose a simpler design, where a computation offloading- and a data request are combined into one, single request. It simplifies the requirements for the storage system, since it can now be stateless: It 'only' has to read the data, execute provided computations on it, and return the results directly. This design is the one we used to create SparkHook, our prototype system implementation.

3.3 Design for Apache Spark, Ceph Storage Cluster

For our prototype, SparkHook, we are interested in offloading computations from Apache Spark [3] to the Ceph Storage Cluster [23] system. In order to do so, we use the Apache Arrow Dataset API [41] as a mediator. For our design plan, we need generic data- and offloading interfaces. The Arrow Dataset API forms the basis of both, since the Arrow Dataset API has a generic data interface and a generic computation expression framework. We define our computation offloading framework using its generic computation expression framework. On a high level, there are six essential steps to offload computations:

1. Obtain computations from Spark.
2. Translate computations from Spark- to Arrow-understandable expressions.
3. Push operations from Spark to the Arrow Dataset API.
4. Push operations from the Arrow Dataset API to Ceph Object Storage Devices (OSDs) and execute them.
5. Send the data from the OSDs to the client.
6. Read the data in Spark, through the Arrow Dataset API.

We implemented SparkHook to follow these steps. An alert reader might notice that these steps are in chronological order, and form a full request cycle: We start at the requester of data (Spark), flowing through the Generic Offloading- and Data Interfaces (Arrow), to get to the data provider (Ceph OSDs). Next, we follow the response data stream from the data provider back through all layers, to form a full circle.

For the rest of this Section, we explain the design of SparkHook, following the lifetime of a request in a chronological order, as described by the steps above.

3.4 Obtaining Computation

We implemented a Computation Aggregation framework, which aggregates computations from Apache Spark that are 'useful to pushdown'. We define a computation to be 'useful to pushdown' when a pushdown alleviates client CPU hardware utilization, since that is the main goal of this work. At this point, we implemented support for all kinds of filtering and projection operations. A filtering operation is an operation where we only keep data *rows* which match a predicate. A projection operation is an operation where we only keep data *columns* which match a predicate.

Filter operations are always useful to pushdown, for several reasons. First of all, pushing down filter operations to the storage layer always alleviates CPU work on the client-side, filtering being a CPU-intensive operation. Additionally, Apache Arrow has much more efficient filtering implementations than Spark does, meaning that offloading filtering is always beneficial for overall performance. Finally, we can skip sending filtered-out data by pushing down the filtering operations to the OSDs, where the data resides. The filtered-out data not being sent further alleviates client CPU utilization. This reduces transmission time and cost considerably when we filter out a large part of the stored data. Projection operations are also always useful to pushdown, just like filters, and for equivalent reasons.

Spark exposes all computations specified by users through so-called query plans. A query plan is an iterable series of nodes, where each node represents a single operator to be applied to the elements in a present state, and where each subsequent operator changes the previous state.

To obtain filters and projections, we query the Spark query plan at runtime in the '*preparation stage*', before we need to read data, using the Spark Datasource API. Using the Spark Datasource API allows us to obtain optimized filters and projections from the query plan. SparkHook stores the filter and projection computations in a 'Computation Aggregator'. In Figure 5, we depict the process of aggregating computations and pushing them down to Arrow. The Aggregator stores a mapping from computation type to computation. This way, we can lookup specific kinds of computations (i.e. filters, projections) in constant time, and eliminate doubly specified predicates on the fly. We store computations as either unordered or ordered computations. With unordered computations, it does not matter when we apply the computation, as long as it happens before we start reading data. With ordered computations, we strictly maintain insertion order, as otherwise side-effects could occur.

After all computations have been aggregated, we broadcast the Computation Aggregator to all Spark workers. When the broadcast completes, Spark will start the '*computation stage*', and starts reading the data in all worker nodes. Because we broadcasted our Aggregator, all worker nodes can query the computations to pushdown, in an efficient, fully cached manner.

Finally, note that every other existing Spark computation is available by searching the Spark query plan, and could be added to our Aggregator easily.

3.5 Computation Expression Translation in Apache Spark

Once we have the computations to pushdown and start the computation stage in Spark, we need to pushdown the computation to Arrow. As one might imagine, pushing down computation expressions from Spark into Arrow is not simple, since Spark computation expressions are implemented in Scala, while the Arrow Dataset Interface resides in C++. Only pushing down the expressions from Scala to C++ is not enough, however. In addition, it is required to translate the expressions from Spark-understandable computation expressions to a common format for Spark and RADOS, with equivalent meaning.

Since quite recently (at the time of writing), Arrow 4.0.0 became available, which incorporates a general computation expression framework alongside its general in-memory datasets. By pushing down computations and translating to Arrow computation expressions, we ensure our framework's offloading capabilities are truly generic. By using Arrow's general expressions, we could swap Spark for any other computation engine, define a translation expression step, and then SparkHook would be able to offload computations again for this new computation engine, just as with Spark. Of course, the swapped-in computation engine needs to have an Arrow Dataset API interface, such as our implementation for Arrow-Spark. Additionally, by using the new Arrow computation expression framework, the computations can be pushed transparently to the storage backend, through its Arrow interface.

Projection computation expressions simply consist of the column names to project. All

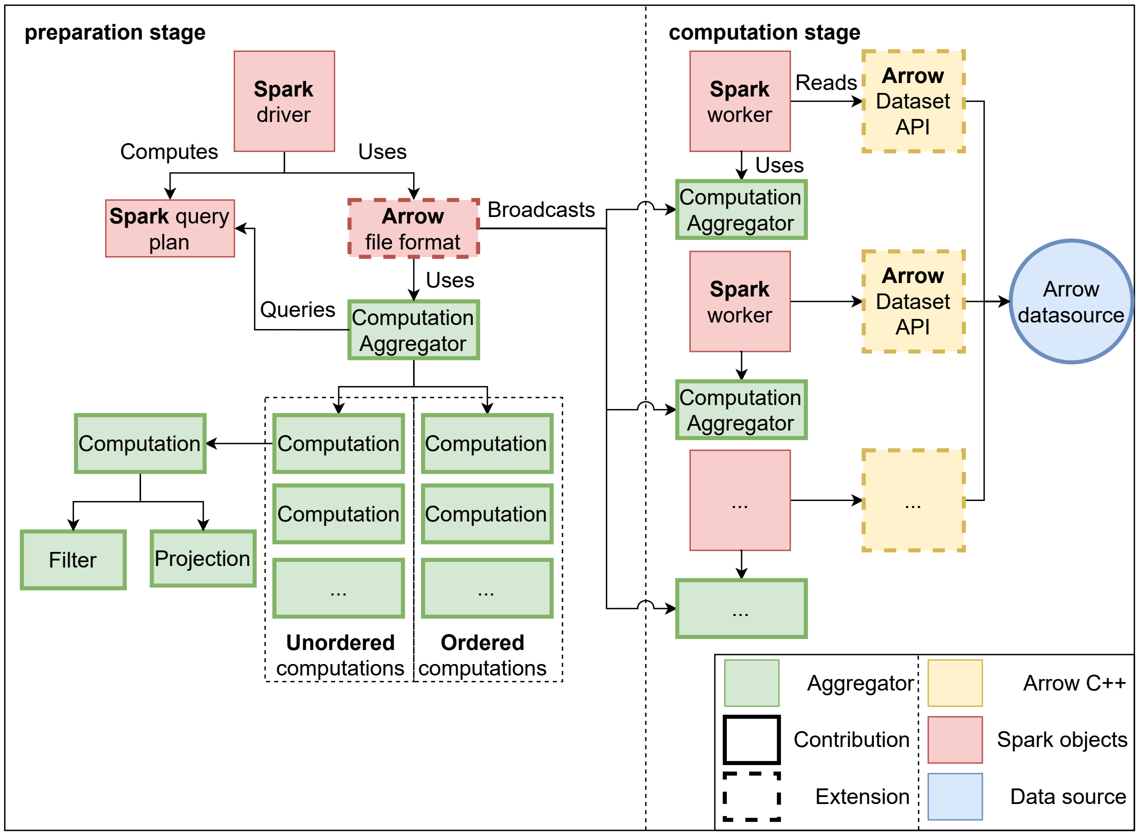


Figure 5: Computation aggregation and broadcasting process. Computations are aggregated in Spark's preparation stage, transmitted to all Spark workers, and used during the computation stage to pushdown stored operations.

column names not specified are skipped when reading. For this simple mechanism, we simply pushdown the container with column names from Spark, inside the Java Virtual Machine (JVM), to the Arrow Dataset API, residing in C++. On the C++ side, we can immediately use the projection without any translation, as Spark and Arrow evaluate the expression equivalently.

For filter predicates, a more complex translation protocol is required. Two example filter predicates could be

```
age > 30 and donation >= 100
date >= 2021-06-26 and notnull(name)
```

In code, these filters are represented as trees with either zero, one or two children per node. At the start of translation, we simply chain together all individual filters using and-operations, such that one single filter predicate remains, with one tree to translate. We use Google Protobuf [46] to transport the predicate expression from the JVM to C++. Protobuf efficiently translates each node in the expression tree to a number of bytes in the JVM, such that we end up with a byte array after evaluating the entire filter predicate. We skip any subfilters with operators or operands in Spark that have no equivalent in Arrow. These parts of the predicates are evaluated by Spark and not pushed down, to guarantee a valid program execution. After we use Protobuf to get a bytearray, we simply pushdown this array to the Arrow Dataset API in C++. There, we recursively evaluate the byte array with Protobuf, and build an expression tree with Arrow Dataset API computation expressions.

Offloading for other computations than filters and projections are currently not implemented. They could be added easily, however. As long as the chosen distributed data processing system is able to collect them and represent them using the generic computation expression framework from Arrow, any computation could be offloaded. If the chosen programmable storage backend is capable of executing the offloaded computation, the offloading system works.

3.6 Operation Pushdown from Spark to Arrow

Apache Spark (core) is implemented in Scala, and there is no Arrow Dataset implementation written in any JVM-compatible language. The Apache Arrow Dataset API [42] is not to be confused with main Arrow library, for which a Java stub implementation exists [47]. To get computation expressions and data from one side to the other, we use a Java Native Interface (JNI) bridge. There are two major situations where we communicate between Spark and the Arrow Dataset API, namely when we read data in Spark through Arrow, and when we pushdown aggregated computations. We require this bridge to be extremely efficient, since all data passes through this interface.

Practitioners using Spark with Arrow to read data are currently bound to a very small set of features. To use the pyarrow-dataset (Python wrappers around the C++ Arrow dataset API) implementation with PySpark (Python wrapper over Spark), one needs to implement these explicitly through the PySpark program, unlike our approach, which is transparent to the programmer. Then, the Python bridge between Spark (JVM) and Arrow (C++) adds a highly inefficient link in applications, and a large functionality limitation. PySpark requires to convert the pyarrow dataset tables to *pandas* [48] data, a PySpark-readable format. This conversion cancels Spark's lazy reading, and requires materializing the entire dataset into memory. We experimented with reading performance with a PySpark+pyarrow setup, and found it was consistently 30 to 50 times slower than Arrow-Spark, with a growing performance difference when increasing dataset sizes. Additionally, due to eager materialization dataset size is limited by RAM capacity. By implementing SparkHook for core Spark (JVM), we circumvent all aforementioned inefficiencies and shortcomings. We can therefore use SparkHook with all programming languages that Spark supports, maintaining maximum applicability to practitioners.

The design choice for implementing SparkHook in core Spark has one drawback, however: the Apache Arrow Dataset API implementation is in native C++, and not in a Java Virtual Machine (JVM)-compatible language. To access the Dataset API from the JVM, we extensively use a Java Native Interface (JNI) implementation [49], based on the Intel Optimized Analytics Platform project [50]. Even though we could have chosen other languages for which there exists an Arrow Dataset implementation, we decided to use C++, because of its native-execution performance. Moreover, the Arrow Dataset API *core* is programmed in C++. Using any other language with Arrow bindings would add additional overhead. Even though the bridge between the JVM and native code brings a slight time penalty, we were able to minimize it by limiting data copies (see Figure 6 for the data copies that our interface implements). Additionally, C and C++ are the best-supported languages for interfacing with the JVM, through the JNI.

In Figure 6, we show the process of pushing down computations.

❶ **Reads.** Just as with Arrow-Spark, data transmission is initiated by a read request coming from Spark. Read requests arrive at the *Arrow FileFormat* datasource. With SparkHook, The *Arrow FileFormat* queries the computations to pushdown in the broadcasted *Computation Aggregator*.

❷ **JVM Dataset.** Like Arrow-Spark, the *Arrow FileFormat* constructs a JVM-based *Dataset* interface to read data through JNI. To start reading data, we build a *Native Scanner* and execute the *ScanTask references* it produces. With SparkHook, we add computations to pushdown to the scan options of the *Native Scanner*.

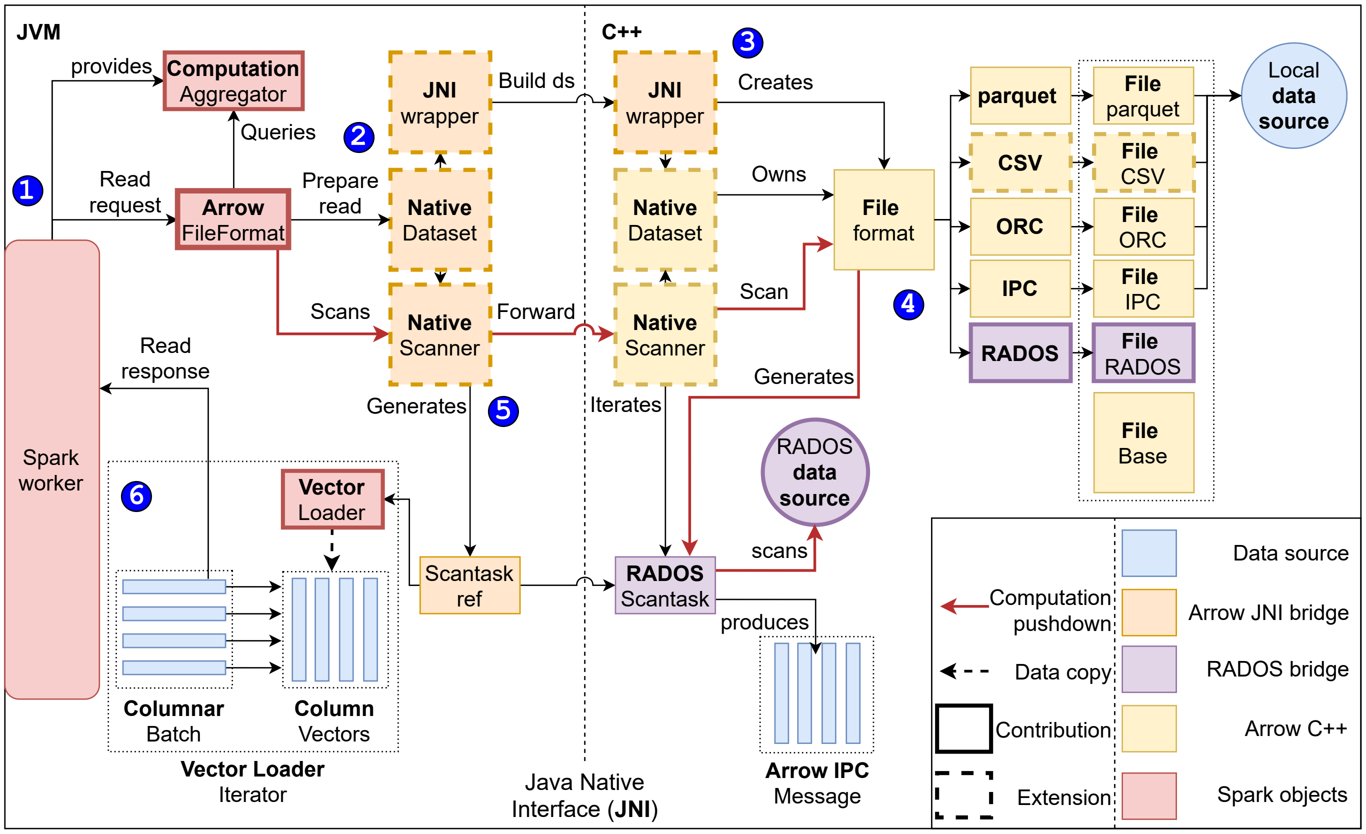


Figure 6: Design overview focusing on the **frontend** of SparkHook, running Spark (left) and communicating with the Arrow Dataset API (right) through our Java Native Interface (JNI). Note that the general structure of this frontend is distantly similar to our overview of our previous work, Arrow-Spark (Figure 4).

3 **Arrow C++ Dataset API.** On the C++ side, a native *Dataset* instance is created. On creation, we pick the *RadosParquetFileFormat* as underlying format.

The other stages (4-6) are explained below, when we visit their respective stages in the pushdown process.

3.7 Operation Pushdown from Arrow to OSDs

When the operations are pushed from Spark to the Arrow Dataset API through the JNI bridge, alongside our read requests, we need to pushdown these operations. The computations and request data should be transmitted to the Ceph OSD node which contains the data for given read request. [1]

4 **Data transmission.** The computation is pushed down from Spark through the *Scanner* interface, in the scan options. Once the scanner starts scanning the *RadosParquetFileFormat*, this format will produce exactly one specialized *RADOS Scantasks* instead of an iterable amount of regular fileformat *Scantasks*. Only 1 *Scantask* is produced, because the backend OSD is stateless, and executes the computations on all data in one go. As a consequence, the

[1]The feature described in this Subsection (operation pushdown from Arrow to OSDs) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

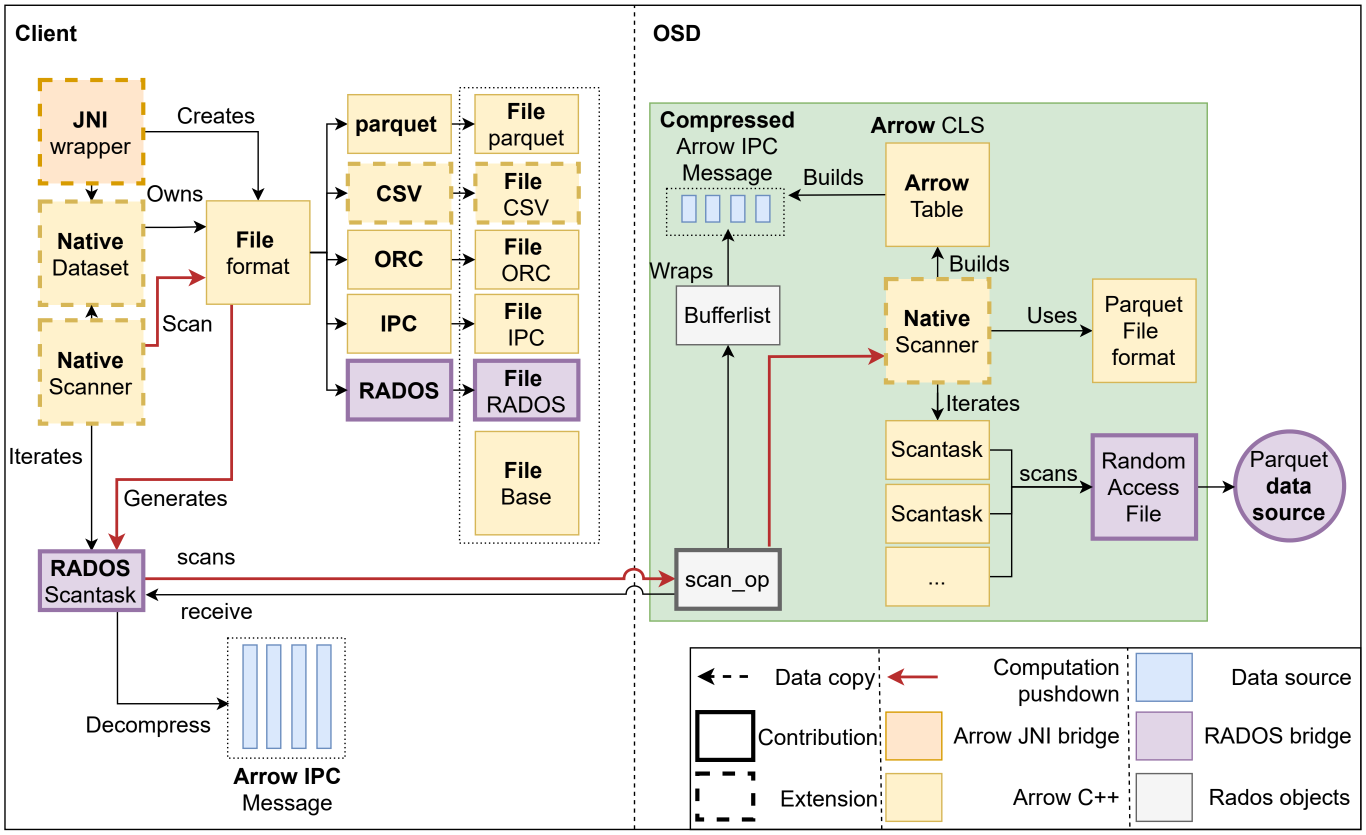


Figure 7: Design overview focusing on the **backend** of SparkHook, showing the communication between the client node (left) and a OSD node (right). Note that both sides use the Apache Arrow Dataset API to get data.

backend has to send all data back at once, and we get the entire object in-memory. Since all data will already get loaded in client memory once we receive data, we have no reason to split scanning the data in multiple lazy tasks. This specialized *Scantask* handles getting the data, and pushing down computations.

When the single *RADOS ScanTask* is executed, it connects to the Ceph [23] cluster, using configurable usernames, access keys, and data pools. We call a remote scanning function inside the RADOS cluster, which is exposed through the SkyhookDM Programmable Storage system [17], using SkyhookDM's class-loading system. With this remote procedure call, we pass along all computations to be handled on the storage node, as well as the the scan request, in a serialized bufferlist. SparkHook finds the *index node number* (inode number) for a requested filename using Ceph's MetaData Servers (MDSs). The inode is translated into a RADOS Object Identifier, using a simple one-to-one mapping from inodes to object identifiers. With the object identifier, *libRADOS* determines on which Object Storage Device (OSD) the object is stored. On that device, the procedure call is executed for the given object. Every device can deserialize the Arrow objects with ease, because the SkyhookDM-exposed scanning functions on the OSDs have a copy of the same Arrow Dataset API library as our client machines have. Once deserialized, the procedure call can start executing our pushed down computations while reading batches of data.

In Figure 7, the class-loaded code on the OSDs uses the Apache Arrow Dataset API to read the object, using a similar strategy as the client. Each object is 128MB in size (the maximum allowed object size in RADOS), and inside each object is a full parquet file, including

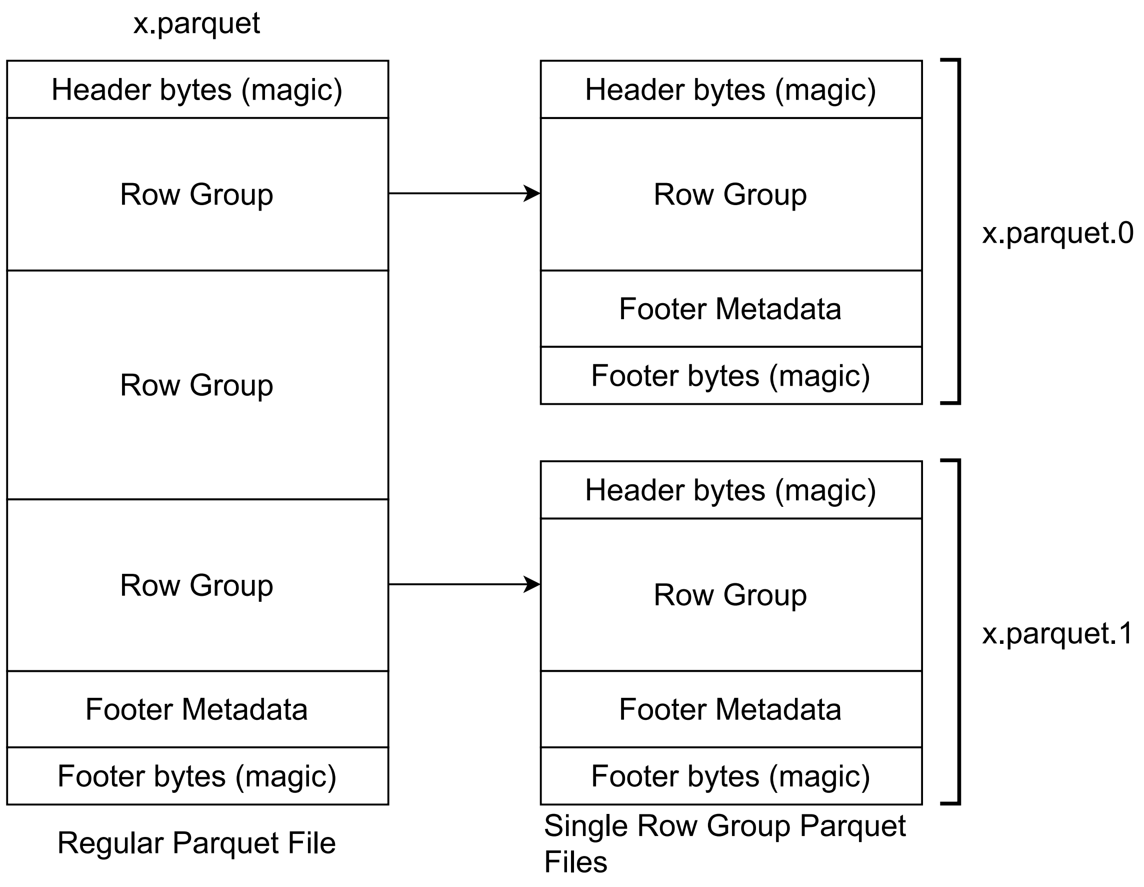


Figure 8: File mapping to objects: each Parquet file is split into a number of files, containing a single row group and metadata fields. Each file is stored in a single RADOS object.

header and footers. We always use the Parquet *FileFormat* in the OSD to read this data from the objects.

Notice that in Figure 7, the computation pushdown stops once it reaches the *Native Scanner*. This is because we implemented pushdown support for filter- and project computations, and both of them are handled by the *Native Scanner*. The *native Scanner* has options to set a filter expression and projection expression.

3.8 Read Data in OSD & Compute

Parquet has become the de-facto file format for popular data processing systems like Spark and Hadoop, since it is very efficient in storing and accessing data. Since Parquet files are often multiple gigabytes in size, a standard way to store Parquet files is to store them in blocks as in HDFS [51, 11], where a typical block size is 128MB [52]. While writing Parquet files to HDFS, each row group is stored in a single block to prevent reading across multiple blocks when accessing a single row group. We aim to follow a similar file layout for storing Parquet files in Ceph so that every row group is self-contained within an object. [2]

With SparkHook, a Parquet file with R row groups is split into R Parquet files. Each file

[2]The feature described in this Subsection (reading data in OSDs & compute) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

contains data from a single row group. The footer metadata and schema of the original data are appended to all files, so we do not lose the opportunity to optimize queries on the data. Figure 8 shows the split file design. Each file is stored in a RADOS object. With this design, the main benefits are:

- Each RADOS object is completely autonomous, as it contains a valid parquet file.
- Any existing parquet reader implementation can interact with a RADOS object.
- When reading data and offloading computations, there is always 1 request for an object. No row groups can span multiple objects, and every object has a copy of the metadata table.
- It is simple to lookup an object, because all RADOS objects have the same size, making it easy to calculate object offsets.

.

The main drawbacks are:

- This strategy requires users to store their data in a specific way, in files with single row groups, with a constant size.
- Each object contains a redundant copy of the metadata, which includes the schema, and redundant magic header and footer bytes.

Note however, that the size of the redundant information does not scale linearly with the data. An extremely complex data layout commonly is several hundred kilobytes. Metadata for a single parquet row group is commonly less than 16 KiB, and the header and footer bytes are not even 1 KiB. In total, the amount of non-data is less than 1% of the total object size.

Because of the chosen file-to-object mapping approach the Arrow CLS Object is able to read the requested fragment transparently, using its default *ParquetFileFormat*, as depicted in Figure 7.

We execute computations when reading the data, to apply filters, projects and other read-time optimizations. We currently support only filters and projections. However, we will support other kinds of computations, and these other kinds of computations should be executed on the produced *Arrow Table*.

3.9 Send Data from OSD to Client

Once we have read all data, and applied all of the received computations on it, we must return the data from the Storage Device to the client machine. [3] Apache Arrow has implemented functionality to construct a contiguous buffer from a `Table` with data for transmission between machines, and read it back on the receiving end. The benefit of using this strategy is that we do not have to implement much, as Arrow already provides us most of the required implementation. Additionally, when serializing the data to a contiguous buffer, our bufferlist construction consists of wrapping a single buffer, which is quite easy to implement.

The drawback of this serialization method is its speed. After measuring, and mailing with the core Arrow developers, we found that sender-side serialization consumes approximately

[3] The feature described in this Subsection (sending data fromOSD to client) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

50% of the total transmission time. The Arrow `Table` serialization system performs quite a lot of `memcopy` calls to get the data into one contiguous buffer. Approximately 50% to 99% of the serialization time goes directly into `memcopies`. So, significant amounts of time are spent in serialization and serialization functions suboptimal. This is a major performance bottleneck.

For future work, we plan to implement a new serialization system. This new system would not move data to one contiguous buffer on the client, instead sending the buffers as raw data over the network. This brings an end to the need to use memcopies when serializing. Essentially, this serialization is only needed because we require to interpret the data twice. We discuss the fundamental inevitability of double interpretation, and provide practical information to review the cost of interpreting the data twice in Section 6.1.

An entirely different factor to take into account is the transmission size of data. By default, Arrow translates data to Arrow Serialization and InterProcess Communication (IPC) [41] format. This data format has a very large memory footprint, IPC being optimized for speed rather than for data transmission. Network interconnects would easily become overburdened by the data size to send using this format. There are several options to fix the in-memory data size.

Firstly, we can apply compression methods on the bulky Arrow IPC output. There are several compression methods available in Arrow. A good compression method for sending data must be fast, light on the CPU, while still producing a notable data size reduction. We experimented with the ZSTD and LZ4 frame formats. LZ4 reduces the data size by approximately 4.0 times, and ZSD by approximately 10.0 times. ZSTD is a more 'heavy' compression algorithm than LZ4, however, and compression takes approximately twice as long as with LZ4. SparkHook supports both LZ4 and ZSTD compression, via a toggleable switch. We recommend users to use LZ4 on high-bandwidth, stable interconnects, and ZSTD when there is only limited network bandwidth available.

Another strategy is to write to an other format than Arrow IPC. This other format should be fast to write to and read from, like IPC, but be more compact, such as e.g. parquet. We tried to use parquet, but found out that it consumes to much time to write to and read from. Writing to a different format than IPC remains a viable strategy for future work.

3.10 Reading data in Spark, through Arrow

Once the client receives the Arrow IPC from the backend, it uncompresses the Arrow IPC data, and wraps *RecordBatch* structures around the data. Now, the data, with all pushed down computations applied to it, is back inside the client machine which made the request. We immediately split out the data into batches, and return them. From the viewpoint of the Apache Arrow Dataset API, this is all we must do. The data can correctly be read from C++.

The last step is to make the data in Arrow (on the C++ side) available to Spark (inside the JVM), which is (still) waiting on the data. Arrow-Spark[44] provides in this matter.

Scanning a RADOS file format produces an iterable vector of *RecordBatches*, which is standard behaviour for all standard Arrow Dataset API format implementations. Unlike most other implementations, however, scanning a RADOS file format produces only one recordbatch. As discussed before, lazy scanning has no purpose for us, as SparkHook receives scanned data in a single piece of data from the backend.

There are three main ways of providing the data from our recordbatch to the JVM from C++. The most simple method is to use some intermediate storage, write the data to it, and pass the identifiers to the JVM. The JVM can read from the intermediate storage, and it then has the data again. There are several drawbacks to this way of working, the most important ones performance-related. Depending on the chosen intermediate storage, the writing and/or reading is very slow. More importantly, we would have to completely reinterpret all data on the JVM side, crippling the performance of this method.

Alternatively, we could serialize the data using serialization frameworks like Google Protobuf, as we did before to offload filter computation predicates from the JVM to C++. Although this strategy is more efficient than using intermediate files, there are still some drawbacks. Mainly, we need to copy all the data we will read to Google Protobuf bytearrays, and copying data is slow, as we saw with the Arrow Serialization implementations. Moreover, we would need to convert the data from Arrow types to Protobuf, and from Protobuf to Spark types, which is a very CPU intensive task.

The strategy we use with Arrow-Spark is the most efficient one we could think of:

❺ **Arrow IPC.** On the JVM side, we execute the *RADOS Scantask* reference. The scantask connects to the RADOS cluster to pushdown computations and fetch the data. We built a custom memory manager in Arrow, which forwards memory allocation and deletion calls to Spark, essentially creating a shared memory section between Spark and Arrow. While receiving the data, SparkHook allocates and places the data in memory shared between the JVM and C++. Since we call Arrow from Spark, Arrow lives in the same memory space and has no trouble accessing the virtual memory from Spark processes. When fetching data in Arrow, we store all recordbatches in a custom, Spark-allocated shared memory space. We pass only a pointer to the allocated data over the JNI bridge, completely circumventing all memory allocations and copies.

❻ **Conversion.** This stage is exactly equivalent to Arrow-Spark. Back at the Spark side, we have to read the data from Arrow-understandable to Spark-understandable types. We achieve this by loading the columns in Column Vectors, as depicted in Figure 6. Column Vectors implement all the interfaces Spark needs to read and write to the columns. This Arrow-to-Spark conversion cannot be circumvented, regardless of the chosen method transferring data from Arrow to Spark, as there are fundamental differences in the way Arrow represents data and the way Spark represents data, e.g. rowwise vs columnar, JVM composed objects vs C++ composed objects etc.

4 RQ2: Performance, Reproducibility, Measuring for SparkHook

The implementation presented in this work can be divided in two categories. First, there is the implementation of SparkHook, a system which offloads computation into the storage layer. The other category contains a large amount of implementations, tools, modules and frameworks which we created to automate and guide experimentation, result gathering, system monitoring etcetera.

Deploying an experimentation environment for SparkHook requires many different setup steps: Nodes need to be allocated, Spark, RADOS, Ceph and Arrow need to be installed on the right nodes, mountpoints need to be created, data must be deployed, and finally, the executable must be deployed with *spark-submit*. After each experiment, the results have to be fetched from the right nodes. Managing all these setups by hand is suboptimal and greatly impedes experiment reproducibility.

To address this issue, we automated every stage, using (mostly) platform-independent, user-friendly, well-documented frameworks, modules and commandline interfaces. This made our research progress much faster, and allowed us to perform experiments with SparkHook in an automated fashion. The tools were created for Python3 [53], a popular scripting- and programming language developed for its ease of operation. They can all be installed using PIP Installs Packages (Python3-PIP) [54], the default package manager of Python3. We chose for Python3 and Python3-PIP, as these tools are familiar to a large number of scientists and industry users. We want to highlight that finishing this project would *not* have been possible without the large amount of repositories we describe here.

We expect that these tools we created will be quite useful to fellow researchers, and regular industry practitioners, using Spark, RADOS and Ceph. Their base implementations are capable of handling most normal use-cases they were designed to address, and these functionalities can easily be expanded through the use of plugins, custom configurations etcetera.

In this Section, we explain more about this set of automation frameworks. This Section answers RQ2: how to assess the performance of SparkHook, our prototype, in a end-to-end fashion? Throughout this Section, we refer to Figure 9, which depicts a design overview of our experimentation framework and all relevant modules for experimentation guidance and monitoring.

4.1 Resources and Allocation

In order to experiment, we need to have nodes to run on. We built a generic reservation framework, called *MetaReserve*, implemented as an installable Python PIP [54] module. By itself, it only provides all interfaces needed to make reservations.

We perform our experiments on *GENI* [55], an open infrastructure for at-scale networking and distributed systems research, from the U.S. National Science Foundation (NSF) [56]. We created *MetaReserve-GENI* to seamlessly integrate GENI node allocation into our frameworks. Both *MetaReserve MetaReserve-GENI* are depicted in Figure 9.

The GENI infrastructure is composed of several cluster sites and heterogeneous hardware. Individual nodes can be reserved on a cluster site, and network interconnects can be defined between them. In order to reserve, one has to use a graphical user interface on a website, or alternatively, use the Python library. We had to use the Python library for automatising purposes. Sadly, the Python library is severely out of date.

When reserving with GENI, we first need to define a so-called '*slice*', which is a common resource management component. A slice contains ownership information, members, a slice expiration date, and most importantly, one or more so-called '*slivers*'. A sliver is an actual,

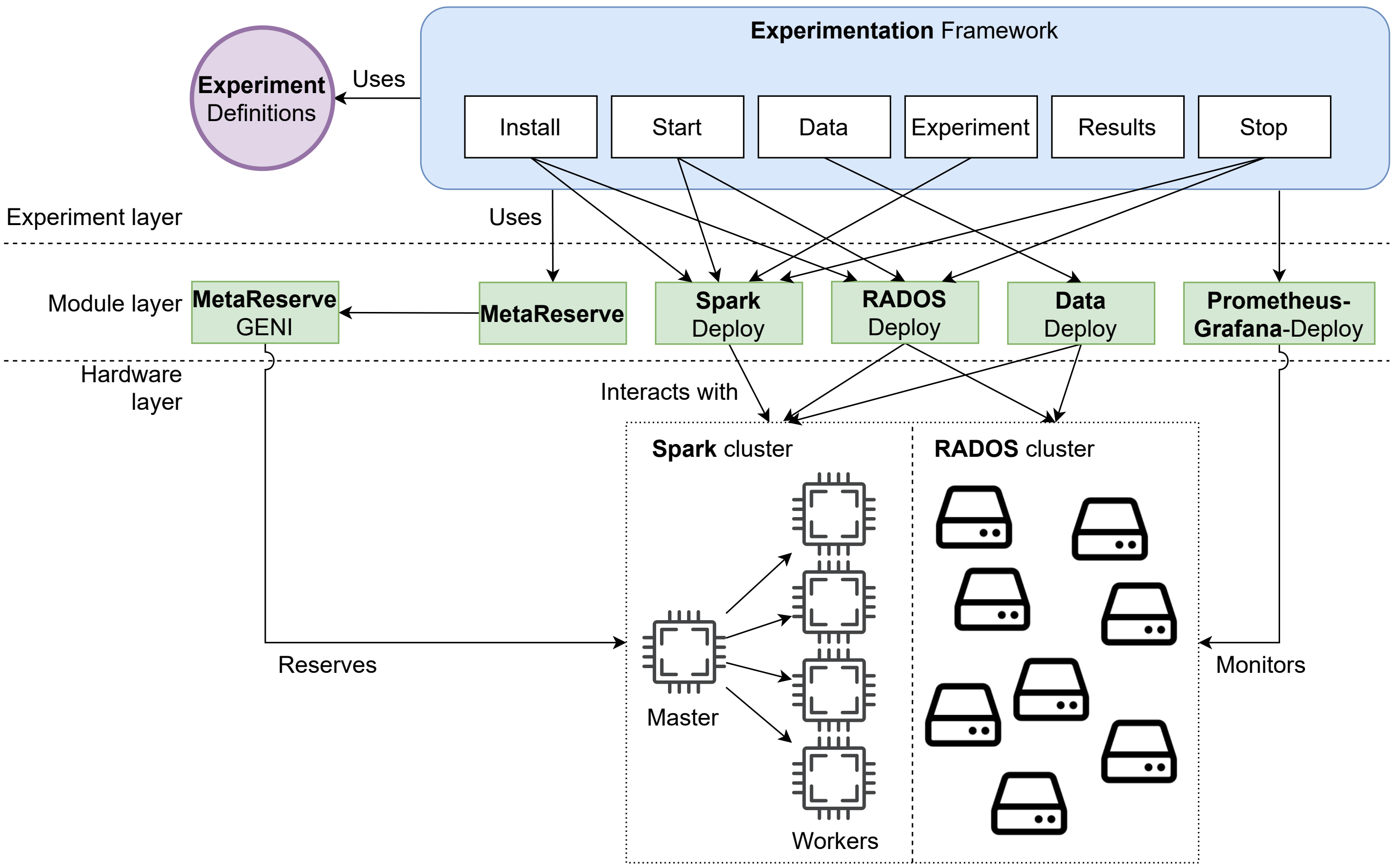


Figure 9: Design overview focusing on the experimentation framework for SparkHook, highlighting interactions between experiment stages (top), available modules (middle) and the cluster hardware (bottom).

named, hardware reservation. It has information about every node and network that was reserved, another expiration date, connection information, and resource status information. Using sliver names, we request connection- and resource status information in the Python library.

During this project, we found that GENI is plagued by bugs. The most problematic bugs were all related to allocation. One problem was that sometimes, GENI adds a node to a reservation that never comes online. Other encountered problems are sudden node failure, node interconnect outage, and unreasonably slow global network interconnect bandwidth performance. *MetaReserve-GENI* aims to standardize node allocation procedures with GENI in order to minimize these problems, at times even correcting wrong behaviour.

4.2 Spark

To quickly and securely interact with Spark, we created *Spark-Deploy*.

A Spark installation consists of a number of Spark workers, and a Spark master node. Users supply Java Virtual Machine (JVM)- and Python applications to Spark. Spark then executes the main function inside these applications in a process, which is called the 'Driver Process'. The Spark driver has the definition of the computation that we are interested in, and defines the data to execute this computation on.

The node on which Spark executes the driver depends on the deployment strategy. We can either deploy in Cluster or Client mode. With Cluster mode, the application is moved to one of the workers, and the driver process is started there. In Client mode, the driver process is launched locally.

By itself, *Spark-Deploy* is a Remote Procedure Call (RPC) framework, which installs, runs, stops, and uninstalls Spark with all dependencies in a parallel, distributed fashion. The local machine serves as the primary node, and all nodes we interact with are secondary nodes. We use the Remoto [57] Python RPC (pyRPC) project for our RPC. We generate and interpret sources at runtime to perform the actions we require. After interpreting generated sources on remote nodes, we can execute our generated code remotely. The primary node controls all secondaries, and guides distributed installation, boot, stop, and removal operations. *Spark-Deploy* has functionality to submit Spark applications in a secure fashion to remote clusters. All options from spark-submit of the chosen Spark distribution are forwarded transparently to the remote master node. *Spark-Deploy* transfers the applications to be deployed to the remote master node. The experimentation framework uses *Spark-Deploy* for all Spark cluster interactions. *Spark-Deploy* is depicted in Figure 9.

4.3 Ceph, RADOS, Arrow

As depicted in Figure 9, there exist two clusters to manage: a Spark cluster to handle distributed data processing, and a RADOS cluster with programmable storage capabilities to provide data and execute offloaded computations. Reliable Autonomic Distributed Object Storage (RADOS) provides the object store implementation that Ceph extends with filesystem- and block storage modules. We need to compile and install Arrow as a generic data interface library for RADOS, such that we can read objects from the RADOS object store using Arrow. To orchestrate storage clusters, we created *RADOS-Deploy*.

RADOS and Ceph work with daemons. Four types of daemons exist:

- object Storage Devices (OSDs): OSD daemons control object storage on a node.
- Monitors (MONs): MON daemons control, among other things, cluster state maps. These maps contain critical state information for the Ceph cluster.
- Managers (MGRs): MGR daemons provide additional monitoring, and run alongside MONs.
- Metadata Servers (MDSs): MDS daemons are responsible for metadata storage for the Ceph File System (CephFS). They allow users to efficiently execute queries relying on metadata.

RADOS-Deploy is a Remote Procedure Call (RPC) framework, which installs, runs, stops, and uninstalls RADOS, Ceph, and Arrow with all dependencies in a parallel, distributed fashion. The RPC network basis is similar to *Spark-Deploy*. In both systems, the machine we execute on locally is the primary node, and all nodes we interact with are secondary nodes. Additionally, both systems use Remoto [57] as base RPC project.

The system is capable of deploying storage clusters based on *Memstore*, an in-memory storage system, and *Bluestore*, a storage-device backed storage system.

With *RADOS-Deploy*, users can directly specify what daemons need to run on every node in the cluster, making RADOS clusters completely customizable.

4.4 Data Deployment

We created *Data-Deploy*, a plugin-based data deployment framework and commandline tool. Each plugin deploys the given data sources on a given cluster.

Depending on many factors, such as network topology, access patterns, latency, local network availability, network filesystem (NFS) target destinations, different ways of deploying some deployment strategies are much more efficient than others.

Additionally, we sometimes want to duplicate data. E.g. we would want to make 1023 copies of every file in our 1 GB dataset, to inflate it to 1 TB. First inflating and then sending causes poor performance in terms of locality, as the network will be used much more due to the inflation. To mitigate this problem, our tool supports passing copy multipliers, to inflate the data on the remote nodes.

Finally, there exist situations where one wants to inflate datasets size to very large sizes, while there is not enough storage space on the entire cluster to support these sizes. This can happen e.g. when one wants to deploy a dataset of 1 GB, inflate it to 2 TB on a cluster with 128 GB storage space. Making copies to inflate the dataset size obviously will (under most circumstances) not work here, as this will (normally) inflate the dataset to the point where it does not fit within storage anymore. We solved this problem by adding a link multiplier. When set, every file receives receives a number of hardlinks to inflate the dataset size. Hardlinks make the data appear larger, much like copying data sources. However, hardlinks do not inflate the stored dataset size, but simply add another link to an existing inode.

We primarily decided to implement this framework because our experiments require a very specific way of deploying data sources on Ceph, using special extended attributes of the Ceph Filesystem (CephFS) to create objects of a constant size and fill them with our data. *Data-Deploy* is depicted in Figure 9.

4.5 Experimentation Guidance

Even with all individual automation frameworks, there are still several commands to execute by hand to run an experiment, and more are needed between experiments. Especially when the cluster composition changes between experiments, we commonly require to install and start more Spark nodes, or more Ceph daemons, which requires issuing lots of different commands. With this in mind, we would need to supply lengthy descriptions and large task lists to enable researchers to experiment with SparkHook.

To completely automate *all* experiments, and to make *all* experiments repeatable, we created *Spark-Arrow-Experiments*. This declarative experiment framework uses *all* of the aforementioned frameworks, tools and modules to prepare a cluster for an experiment, execute it, aggregate results, and prepare the cluster for the next experiment.

To define an experiment, users primarily implement an execution interface, and optionally use a configuration, in the Python programming language [53]. In their experiment declaration, users can override generic parameters, e.g. the amount of Spark nodes to use, and add very specific options, e.g. setting the Ceph stripe size. These parameters are accessible by framework functions in different experimentation stages, once the experiment is first in line to execute.

Spark-Arrow-Experiments's execution interface has several stages:

1. Distribute nodes to groups. Here, we distribute allocated nodes in groups, such that each group has enough nodes available.

2. Installation of Spark, other dependencies.

3. Start Spark, other systems.

4. Data generation.

5. Data deployment.

6. Experiment execution.

7. Result aggregation.

8. Stop Spark, other systems.

The stages are also executed in this order. *Spark-Arrow-Experiments*, with its stages, is depicted in Figure 9.

For every stage, the user registers an amount of functions to carry out the stage tasks. Users can declare their own callable function to execute on that stage, or use some of the functions we implemented for experimentation. Some stages allow for one callable function only, such as the install/start/stop Spark stages, while other stages allow to set multiple functions, such as the install/start/stop others. With this strategy, users can define experiments in an unconstrained manner, fully utilizing the capabilities of our modules.

We implemented many default stage functions, and use an intelligent optimization system: We order experiments in a way that node distributions remain equivalent as long as possible, so no systems have to be halted and rebooted in an equivalent configuration later on. Additionally, we take into account experiments could require the same data, so we can coallocate them and skip redeploying data.

4.6 Resource Monitoring

Resource monitoring greatly helped us develop SparkHook. It allows one to check the main hardware utilization and bottlenecks during framework execution, and verify whether computations got offloaded from the Compute cluster to Storage cluster correctly. We use Prometheus [58] and Grafana [59], two popular tools from the cloud computing community. We developed a small tool, called *Prometheus-Grafana-Deploy*, to automatically install, configure, start and stop Prometheus and Grafana.

Prometheus monitors hardware on all nodes. Every node sends its statistics to the Prometheus admin node, where all statistics are aggregated. During development and experimentation, we mainly used hardware statistics such as CPU utilization, network interface usage, and storage device usage.

Grafana is installed on one node only. It uses the Prometheus admin as a datasource. To visualize data, Grafana needs a dashboard. Our tool has a plugin module to generate Grafana dashboards as simple importable JSON objects. Using dashboards, we can review system performance at a glance. We depict this framework in Figure 9.

4.7 Implementation Statistics

To give an impression of the time required for this project, we present an overview of manhours spent and lines of code for our subprojects in Tables 1,2. Here, we see that large amounts of time were spent in implementing SparkHook. However, note that also the Experiment Guidance projects consumed considerable time.

An alert reader might notice that the amount of experiment guidance code is almost twice as much as SparkHook code, even though relatively more time was invested in SparkHook. One reason for the difference in lines of code is the difference in language. SparkHook is written in Scala, Java, and C++, while all experiment guidance projects are installable Python-PIP [54]

Project Name	Time Investment (Hours)	Lines of Code	Lines of Comments	Lines Total
SparkHook (Arrow)	700	$\pm4,000$	±800	$\pm4,800$
SparkHook (Spark)	1000	$5,141$	840	$5,981$
Total	$1,700$	$9,141$	$\pm1,640$	$\pm10,781$

Table 1: Time investment and lines of code for SparkHook, sorted by name.

Project Name	Time Investment (Hours)	Lines of Code	Lines of Comments	Lines Total
Data-Deploy	72	$1,505$	454	$1,959$
metareserve	20	188	62	250
metareserve-GENI	250	$1,285$	376	$1,661$
Prometheus-Grafana-Deploy	50	$2,539$	604	$3,143$
RADOS-Deploy	250	$3,592$	$1,055$	$4,647$
Spark-Arrow-Experiments	300	$4,426$	$1,235$	$5,661$
Spark-Deploy	100	$2,616$	784	$3,400$
Total	$1,042$	$16,151$	$4,570$	$20,721$

Table 2: Time investment and lines of code for our Experimentation Guidance projects, sorted by name.

modules. Python is a quite verbose language, with enforced style requirements for placing newlines, tabs etc, while Scala, C++ allow for lengthy, complex one-line expressions. Another reason is the amount of work each system is in charge of: SparkHook has the complex task of collecting computations from Spark and offloading these to RADOS OSDs. The experimentation framework has to perform fairly large high-level tasks, such as reserving cluster hardware, installing software and data on nodes, preparing clusters for experimentation, guide experimentation, gathering and plotting results etc. The experimentation framework can be seen as an encompassing framework for SparkHook with many more high-level tasks, which logically requires more code than SparkHook itself, which has fewer high-level tasks. For the difference in time consumption is a simple explanation: for the experimentation framework, we were able to reuse several large files, functions and procedures across projects, making our progress a lot faster than with SparkHook.

The total, combined statics for this project can be found in Table 3. One might notice the total time spent for this project is approximately 2742 manhours. This is without including the time spent on writing this Thesis. An official MSc diploma requires a Thesis project of exactly $1,260$ hours, computed as $45\,\text{EC} \times 28\text{hours/EC} = 1260 hours$. If we define student payment for a Master Thesis to be their MSc diploma, I should get $2,742/1,260 \approx 2$ MSc diplomas, *rounded down*. Of course, this will not work in practice due to sensible constraints, such as the requirement that there must be a Thesis defence presentation for each diploma, the student in

Project Name	Time Investment (Hours)	Lines of Code	Lines of Comments	Lines Total
Experimentation framework	$1,042$	$16,151$	$4,570$	$20,721$
SparkHook	$1,700$	$9,141$	$\pm1,640$	$\pm10,781$
Total	$2,742$	$25,292$	$6,210$	$31,502$

Table 3: Total time investment and lines of code for both SparkHook and the Experimentation Guidance projects.

question must have followed all required courses for each MSc diploma, and it is not possible to get a diploma twice. Nonetheless, it is good to know that this MSc Thesis is worth (in time spent) over 2 MSc Thesis projects.

We spent approximately 2742 manhours in this project in a period of approximately 7.5 months. This equals approx 366 hours per month, approx 12.2 hours every day, assuming the average month has 30 days, including weekends. In practice, this is about right. A total of 4 free days were used. The longest workday was registered at 22 consecutive hours. Average workdays were from 10:00UTC+02:00 to 23:00UTC+02:00.

4.8 Availability

Among other things, the goal of this project was to build useful, practical, applicable code. We built this, and open-source our project to benefit the research, Spark, Arrow, and Ceph communities. The source code for SparkHook can be found here: https://github.com/Sebastiaan-Alvarez-Rodriguez/arrow-spark. It contains three Gradle modules: A core module with the connector, a module to run benchmarks, and a module containing sample tests.

The other frameworks we created to assist us with many experimentation-related tasks are available here:

- MetaReserve:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/metareserve.
- MetaReserve-GENI:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/metareserve-GENI.
- Spark-Deploy:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/Spark-Deploy.
- RADOS-Deploy:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/RADOS-Deploy.
- Data-Deploy:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/Data-Deploy.
- Prometheus-Grafana-Deploy:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/Prometheus-Grafana-Deploy.
- Spark-Arrow-Experiments:
  https://github.com/Sebastiaan-Alvarez-Rodriguez/Spark-Arrow-Experiments.

5 RQ3: Evaluation of SparkHook

To evaluate the performance and scalability of our prototype, SparkHook, we take the following steps. We compare performance between a standard (or vanilla) Spark cluster versus SparkHook, reading from our SkyhookDM-enabled [17] programmable storage backend RADOS [43], on a series of queries and datasets. As vanilla Spark is not capable of reading data directly, we use CephFS to present objects in RADOS as files to Spark. In the following subsections, we provide an overview of our experiments, we explain our environment and used hardware, and we display and discuss our results. This Section answers RQ3: what is the performance of our prototype end-to-end computation offloading framework?

5.1 Experimentation Overview

For our experiments, we compare the speed and efficiency for reading the columnar parquet- and row-wise CSV file formats, between our framework (using the Arrow Dataset API) and Spark's default reader, in a way that is fair and realistic. Here, we provide detailed information on both high- and low level experiment environment and parameters.

5.1.1 Experiment Description

To find answers to our Research Questions, we performed 6 separate experiments, in which we compare SparkHook with Spark's default reader, in several different situations.

1. We measured the influence of offloading computations with SparkHook on compute cluster CPU utilization.
2. We verified the network infrastructure alleviation by offloading data-reducing computations with SparkHook.
3. We experimented with offloading data-reducing computations with a range of row selectivities for SparkHook, to find out how the system behaves under different selectivities.
4. We measured the influence of different dataset objectsizes on overall performance.
5. We show the team's measured influence of selectivities and objectsizes on dataset reads in the backend, and compare it to overall performance.
6. We measured how the datasource size influences reading time to determine how both systems scale when processing more data.

The purpose for these experiments is to get comprehensive insight into the performance of SparkHook, and to show the benefits of intelligent cooperation between compute and storage clusters.

5.1.2 Hardware

We perform our experiments on `GENI` [55], an open infrastructure for at-scale networking and distributed systems research, from the U.S. National Science Foundation (NSF) [56]. The reason for this is simple: it is a well-known, open infrastructure. Many scientists around the world can access it, and reproduce our research on exactly equivalent hardware. Additionally, they can execute their own related frameworks there and make accurate comparisons with this work.

We allocated `m510` nodes from the `cl-utah` cluster site. `m510` nodes have an Intel Xeon D-1548 CPU running at 2.0 GHz, with 8 cores and 2 threads per core, for a total of 16 threads. It has 64 GB RAM, and a high-performance NVME drive for storage. When comparing experiments or reproducing results, we encourage the reader to reserve and run on `m510` nodes.

5.1.3 Execution, Parameters and Reproducibility

We explain the general structure of all experiments. We used every specification we give here as a default value for every experiment.

Experiment We ran every experiment 21 times for every configuration. We discard the first execution for every experiment, because the JVM- and memory caches are still 'cold' during this run, producing an outlier. Between runs of the same experiment we did not close the Spark session as to keep JVM and caches warm. We adhered to the guidelines provided by Uta et al. [60], to ensure our performance results are reproducible and significant. Due to the stability of our cluster, the difference between the $1st$ and $99th$ percentiles for each experiment were below 10%.

Data Our workload consists of a dataset with 20.48 billion rows and 17 columns. The origin of the data is the famous NYC yellow taxi dataset [61]. The default stored size of the dataset is ± 512 GiB, split in RADOS objects of 128 MB. We observed that data must be self-contained within a single RADOS object in Section 3.8, such that we can execute functions on single RADOS objects at a time. Following this requirement, each 128 MB RADOS object contains a single Parquet row group in all experiments. Note that we open-sourced our data generator, as well as all benchmarking code (see Section 4.8).

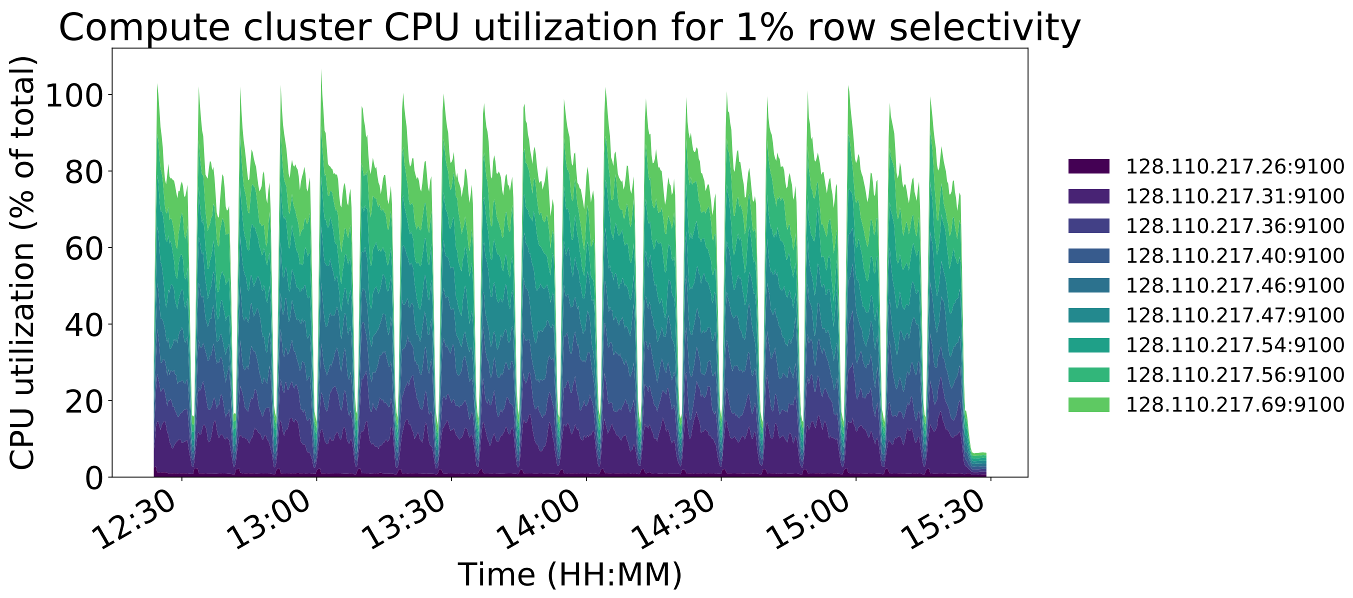
Execution In our experiments, we are interested in reading performance. To measure this we created queries which only read in all data, and count the number of rows as a way of triggering execution. This way, we can accurately measure the read performance. By default, we read this data into a Spark Dataframe (DF) to test with, reading 8192 rows per batch.

Hardware By default, we experiment on cluster size of 8 executor nodes, plus one node for the driver (running on the master node). In the experiments, we always provide the number of nodes in terms of executor nodes. In our systems, Spark is allowed to use up to 60GB of the available 64GB RAM in each node. Note that approximately 20GB of RAM is already in use by the data and symlinks which we placed on RAMDisk. When we need to read parquet files, we normally read uncompressed parquet files.

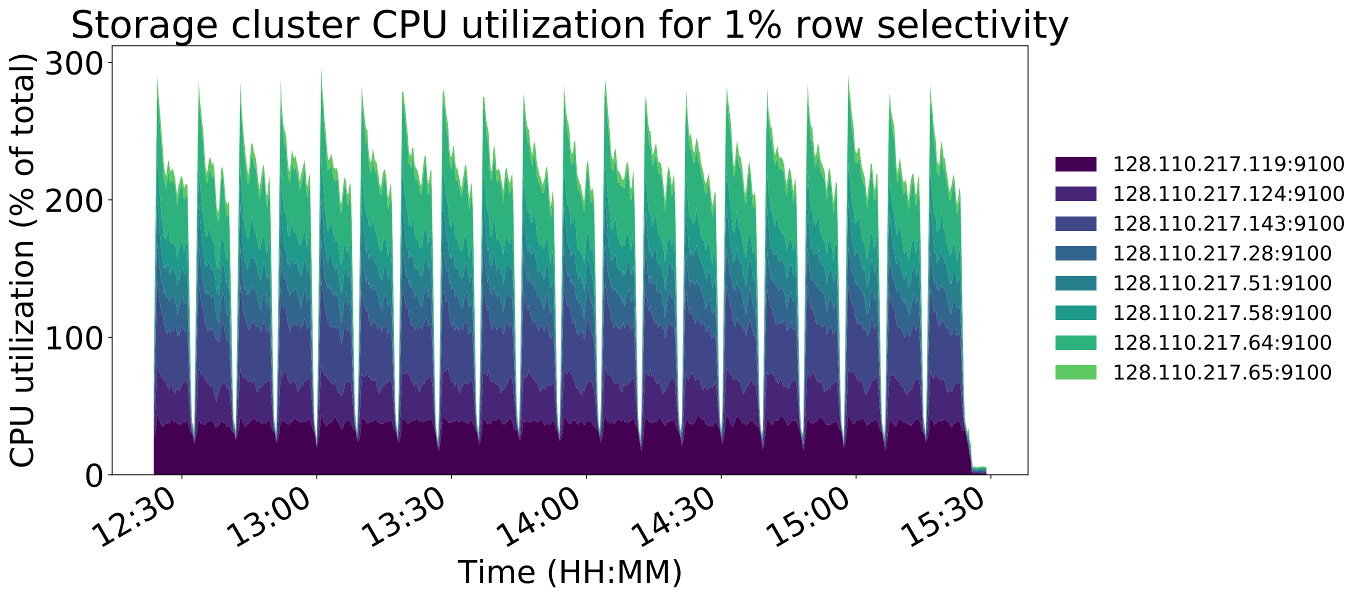
Final Words For every experimental result, we precisely describe all relevant experiment parameters. This includes both unchanged default parameters and those parameters which we varied. Finally, we made our benchmark and experimentation guidance software publicly available (see Section 4.8). Every experiment parameter configuration can be found there, exactly the same as we used them.

5.2 CPU Alleviation

CPUs are major bottlenecks for distributed data processing system performance [19, 18]. The goal of SparkHook is to offload computations to the storage backend, to alleviate CPU utilization on the compute cluster. The computation resources from the programmable storage backend take over some part of the total workload. In this experiment, we show that SparkHook indeed is able to alleviate compute cluster CPU utilization.



(a) Compute cluster CPU utilization. The total approximate area under curve is $774,793.2\%$.

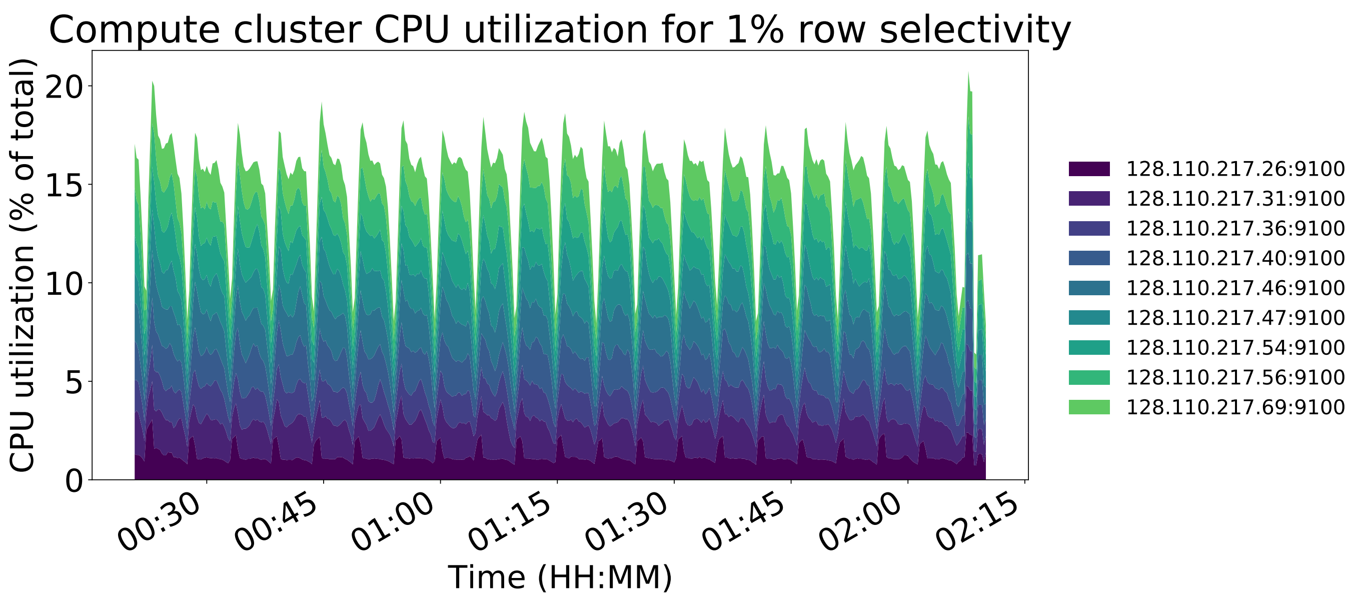


(b) Programmable storage backend CPU utilization. The total approximate area under curve is $2,160,871.7\%$.

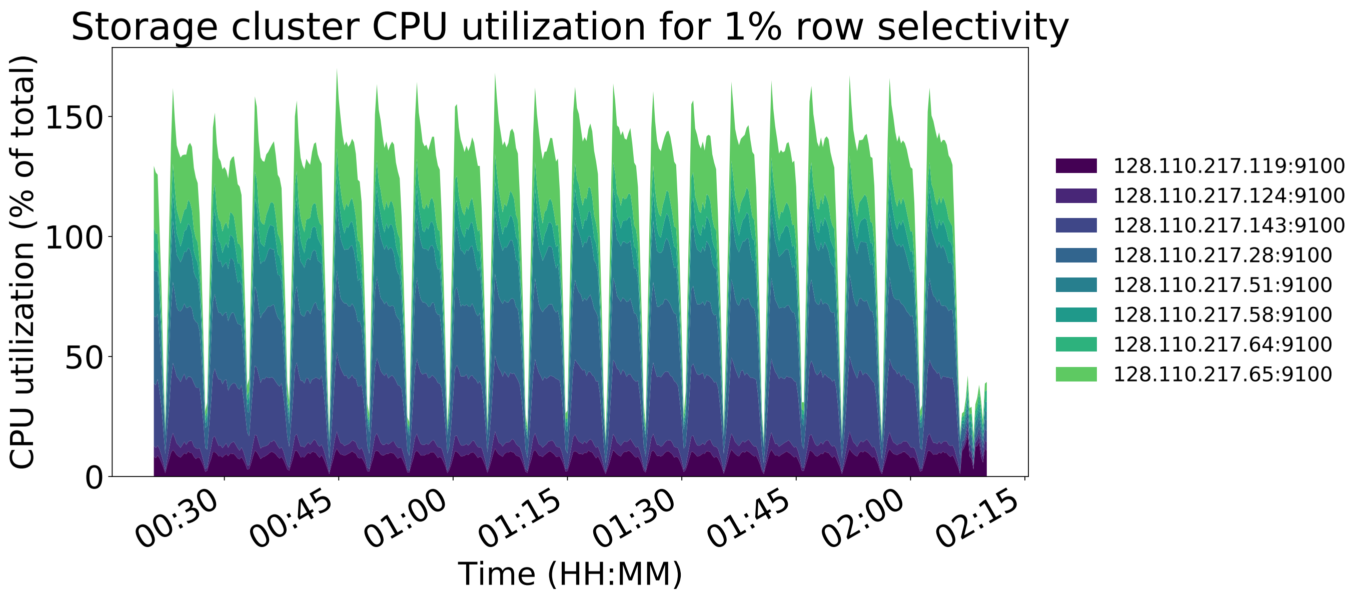
Figure 10: CPU utilization for compute cluster and programmable storage backend, reading 9.5 TiB uncompressed parquet data, using SparkHook, **without offloading** a 1% row selectivity query to the backend. Results are displayed as *stacked* lineplots for individual nodes.

To show that offloading with SparkHook indeed alleviates compute cluster CPU utilization, we compare CPU utilization when offloading versus not offloading computations. Hardware utilization was measured using Prometheus [58], a popular open-source monitoring solution.

In Figure 10, we read 9.5 TiB uncompressed parquet data, and apply a 1% selectivity



(a) Compute cluster CPU utilization. The total approximate area under curve is $\pm 97,214.8\%$.



(b) Programmable storage backend CPU utilization. The total approximate area under curve is $750,507.3\%$.

Figure 11: CPU utilization for compute cluster and programmable storage backend, reading 9.5 TiB uncompressed parquet data, using SparkHook, *offloading* a 1% row selectivity query to the backend. Results are displayed as *stacked* lineplots for individual nodes.

query, *without offloading* the query to the programmable storage backend. There, we find that compute cluster CPU utilization is moderate overall without offloading anything, with approximately 15% CPU utilization per cluster node for every execution. The programmable storage cluster is used more intensively during this time, with a CPU utilization of approximately 40% per programmable storage node. The reason for this high utilization is because the data is being read from parquet files, a CPU-intensive storage format, converted to Arrow IPC format internally, serialized, and compressed using the LZ4-compression algorithm, before being sent back. As we do not offload our 1% row selectivity query, we end up reading, converting, serializing and transmitting all $9.5\,\text{TiB}$ data.

There is a sharp contrast with Figure 11, which shows CPU utilization when executing the same query, but with offloading the data. The client CPU utilization is *strongly reduced*, to approximately 1.8% per cluster node. With SparkHook, using computation offloading, we are able to get a factor 8.0 (87.5%) compute cluster CPU utilization reduction!

Furthermore, the programmable storage CPU utilization decreased from approximately 40% per programmable storage node to approximately 18% per programmable storage node. Because we now offload computations to the storage nodes, the storage nodes can reduce the data size to convert, serialize, compress and transmit to only a fraction (namely 1%) of what it was before, causing the large reduction in CPU utilization. One might wonder why the storage cluster CPU utilization reduction is not exactly 99% when compared to not offloading. This is because we now have to do computations in the programmable storage backend, increasing utilization. Executing row filtering and then serializing, compressing and transmitting is much more efficient than not executing computations, serializing, compressing, and transmitting. In total, the CPU utilization in the programmable storage cluster is reduced by a factor 2.2.
Conclusion-1: SparkHook is capable of successfully offloading computations. SparkHook *greatly* alleviates CPU utilization for the compute cluster.

5.3 Network Alleviation

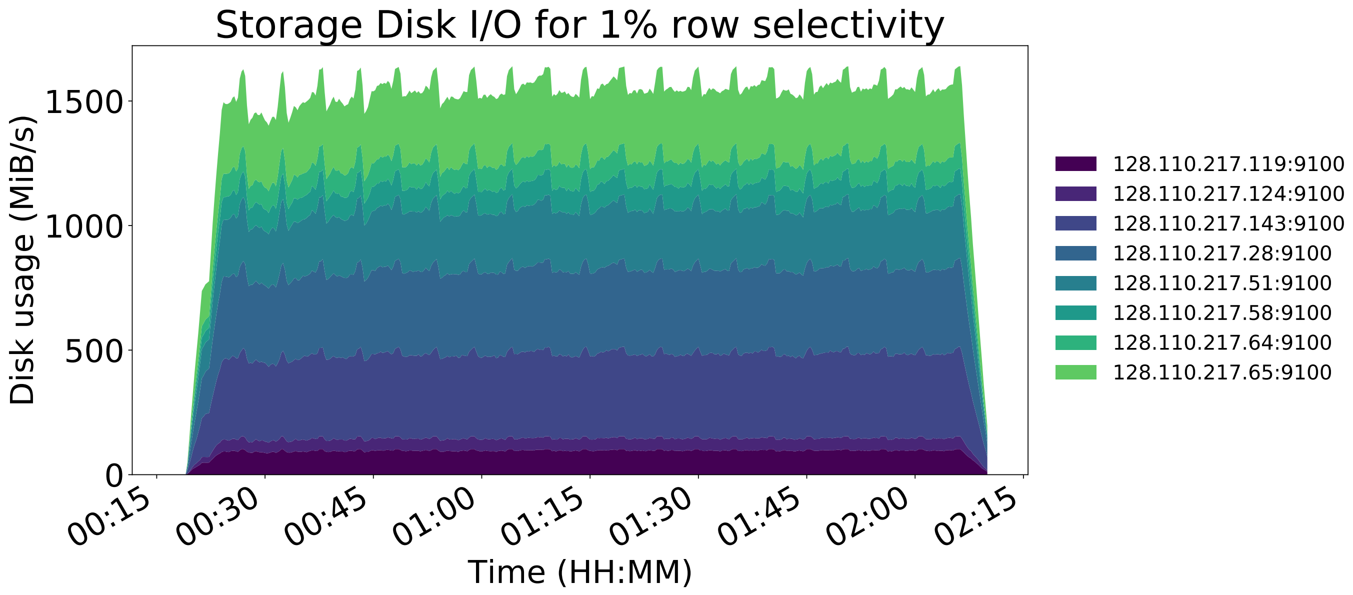
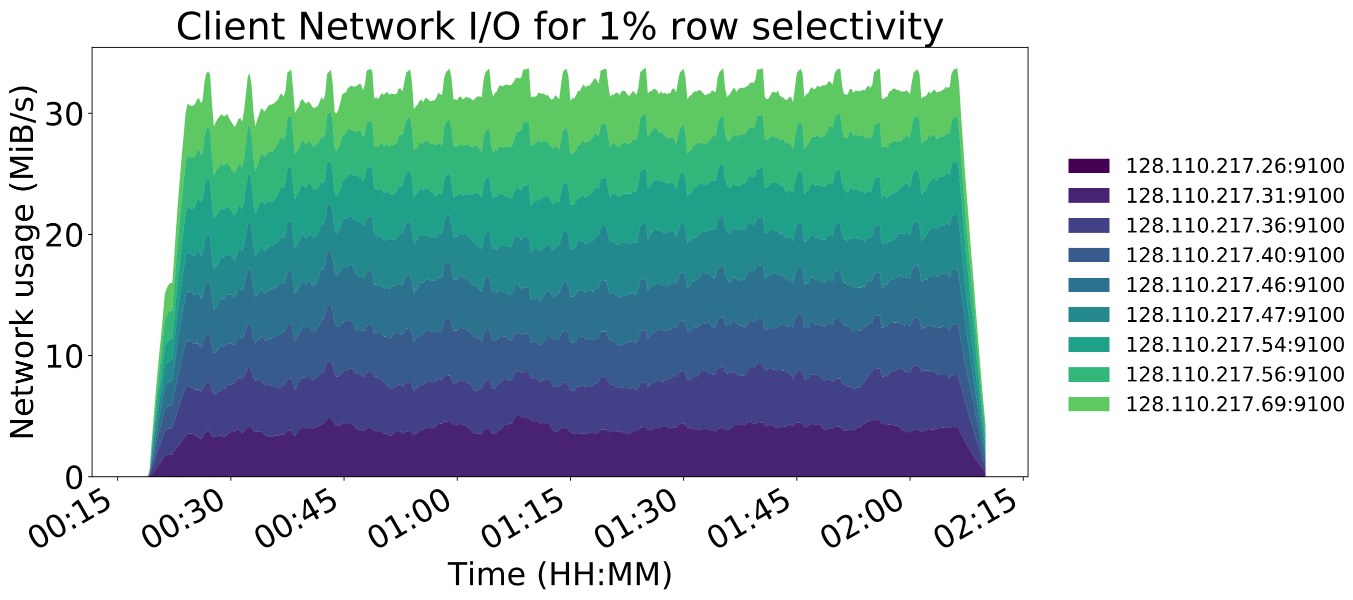
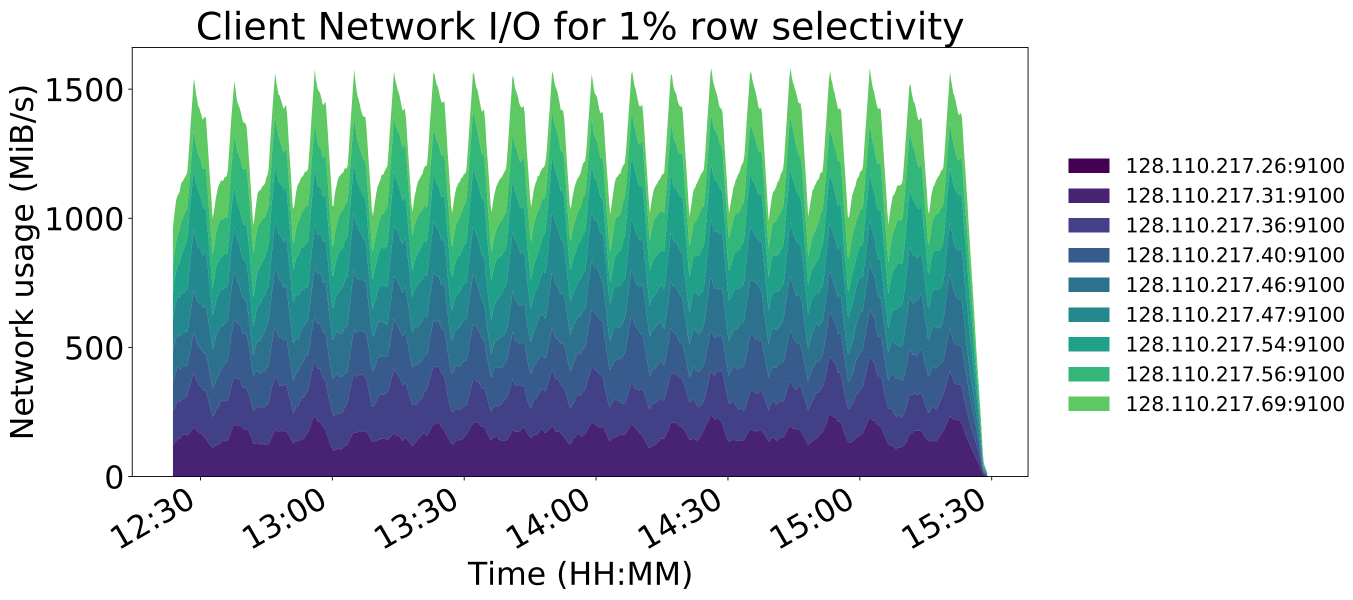


Figure 12: Storage Device utilization for RADOS cluster, reading $9.5\,\text{TiB}$ uncompressed parquet data, using SparkHook. Results are displayed as *stacked* lineplots for individual nodes. The total approximate area under curve is $9,685\,\text{GiB} \approx 9.5\,\text{TiB}(\approx 10.4\,\text{TB})$.

The goal of SparkHook is to offload computations to the storage backend, to alleviate the CPU load of the compute cluster. One of the major benefits of offloading computation to storage is that the computations can operate locally. We use this technique to offload filters to the backend, thereby saving a lot of data to be sent. This alleviates the network infrastructure. However, as discussed in Section 6.1, pushing down computations has a fundamental serialization overhead drawback. In this experiment, we verify that the network traffic is in-



(a) 1% row selectivity filter *with offloading*. The total approximate area under curve is 199 GiB.



(b) 1% row selectivity filter, *without offloading*. The total approximate area under curve is $13,701$ GiB ≈ 13.4 TiB.

Figure 13: Measured Network utilization for Spark compute clusters, reading 9.5 TB uncompressed parquet data, applying and offloading a 1% row selectivity filter, using SparkHook, with and without offloading. Results are displayed as *stacked* lineplots for individual nodes.

deed reduced, and quantify this alleviation by comparing it with equivalent situations without computation offloading.

In Figure 12, we see the disk utilization when reading 9.5 TiB worth of data in a series of 20 computations. The total approximate area under curve is $9,685$ GiB ≈ 9.5 TiB(≈ 10.4 TB), measured using Simpson's rule [62]. The reason we read slightly more data is simple: The RADOS storage backend we use, BlueStore, adds some extra metadata objects that are read during querying. Also, RADOS stores metadata alongside the objects containing the data. The compute cluster reads this metadata as well.

The compute cluster network I/O when pushing down a 1% selectivity query is depicted in Figure 13a. Note the extreme difference with Figure 12: even though the cluster reads 9.5 TiB worth of data at a speed of approximately 1550 MiB per second, the compute cluster only receives 199 GiB filtered data, at a speed of 30 MiB per second. Using SparkHook, we successfully alleviate the network infrastructure by a factor of **48.9**, an enormous amount! Because of offloading a 1% row selectivity query, we reduce network utilization by approximately 98.0%! One might wonder why we are not reducing network utilization by exactly 99%. The reason for this is simple: A small part of network utilization consists of metadata fetching from the programmable storage backend. These requests are counted toward network utilization, and always happen, regardless of row selectivity queries. Additionally, the Spark cluster performs some coordination over the network during computations. Overall, the performance we found is *excellent*.

We were interested in finding out what happens when we execute the same 1% row selectivity query, *without offloading*. These results can be found in Figure 13b. Our 20 compute requests are executed slower, leading to 20 clearly visible peaks. From this Figure, it becomes clear that not offloading computations is quite expensive. The compute cluster requests data, applies the not-pushed-down computations, and only requests the next batch of data after finishing, causing this jagged pattern visible in the Figure.

All data is transmitted to the compute cluster, with an area-under-curve of ± 13.4 TiB. This is approximately $6,895.3\%$ more data than when we do offload queries. Also, this transmission consisted of 41.1% more data than the ± 9.5 TiB we read from the storage backend disks. Approximately all data inflation is caused by serialization. Even when reading data from the backend without offloading any computation, data is still read, interpreted, and serialized before being transmitted back. This chain of events inflates the data by large margins. In the backend, every uncompressed parquet file is converted to the Arrow IPC format, which is much less compact. In most of our measurements, Arrow IPC data is commonly 9 times inflated, when compared to its equivalent in uncompressed parquet. After reading in the data to Arrow IPC, we compress the IPC data using the LZ4 compression algorithm. The LZ4-compressed IPC data is only ± 1.32 times its equivalent in parquet. So, approximately $32\%.$ of the total 41.1% data inflation can be explained directly by serialization. The other $\pm 9.1\%.$ is due to a combination of factors, such as the Spark cluster coordinating itself for 4 hours, metadata requests and replies, slight hardware utilization measuring overhead etc. Overall, network alleviation is excellent when offloading data reduction queries. However, the network infrastructure utilization increases when not offloading any data reduction queries.

Conclusion-2: SparkHook is capable of successfully offloading computations. Offloading greatly alleviates the pressure on network infrastructure, especially with data reduction queries.

5.4 Selectivity

In Section 5.2, we found that pushing down row selectivity queries alleviates compute cluster CPU utilization. In Section 5.3, it was interesting to see how row selectivity query pushdowns greatly alleviate the network infrastructure. We also want to know whether row selectivity offloading increases overall distributed data processing system performance. In this experiment, we offload row-filtering computations to the storage cluster, with a varying amount of row selectivities, and measure overall distributed data processing system processing time. The results are depicted in Figure 14. Not unexpectedly, offloading very strong data reduction queries speeds up performance significantly when compared to not reducing data at all, by

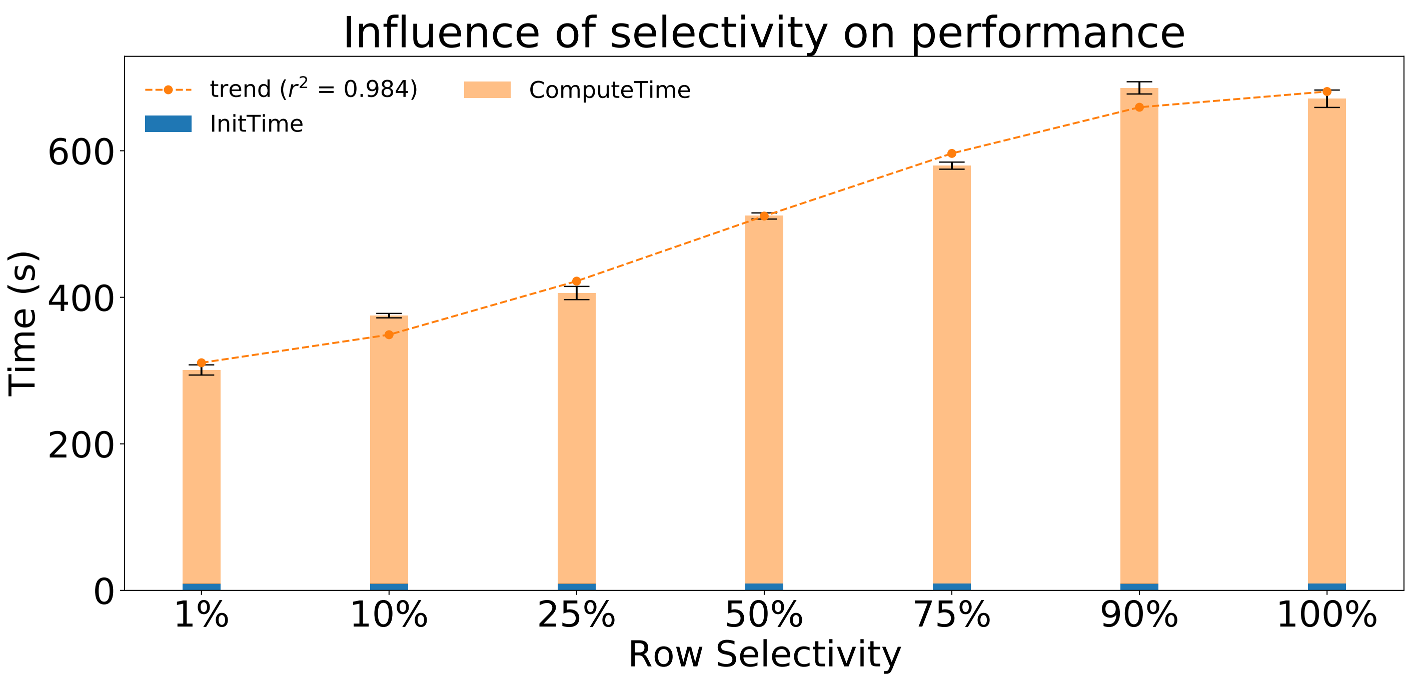


Figure 14: Overall SparkHook execution time for a range of row selectivities, on a 512 GiB NYC yellow taxi dataset.

approximately 45%.

In Figure 14, we see that there is a general trend between pushing down row selectivity computations and performance: when pushing computations with a lower row selectivity, Spark, the chosen distributed data processing system usually performs better. This makes sense, because we need to send back less data, as shown in Section 5.3, and we alleviate compute cluster utilization, as shown in Section 5.2. The saved hardware utilization reduces overall computation time.

There is one notable exception in this general trend: a 100% row selectivity filter is faster than a 90% row selectivity filter, which is counter-intuitive. This anomaly is caused by Spark, our distributed data processing system. A 100% row selectivity query simply means that we want to match all rows, i.e. we apply a filter resembling `SELECT * from Table`. When we execute such SQL-like queries in Spark, it simply translates this filter to no filter. SparkHook detects that there is no row filter to offload, and requests the data without offloading the computation. In the programmable storage backend, there is no computation to execute, so the data is simply serialized and sent back. The reason 100% row selectivity is faster is because SparkHook skips computation offloading and computing in the programmable storage backend.

Note that SparkHook is a prototype, still under development. As explained in Section 6.1, we work with Arrow's inefficient serialization libraries, and we plan to minimize this cost. This should further improve speed for all selectivities. Especially larger row selectivities would improve with the proposed serialization changes, since higher row selectivities have more data to serialize after computation.

Conclusion-3: Offloading data-reducing queries can improve overall distributed data processing speed by $\pm 45\%$ percent. Especially offloading greater data-reducing queries is beneficial for performance.

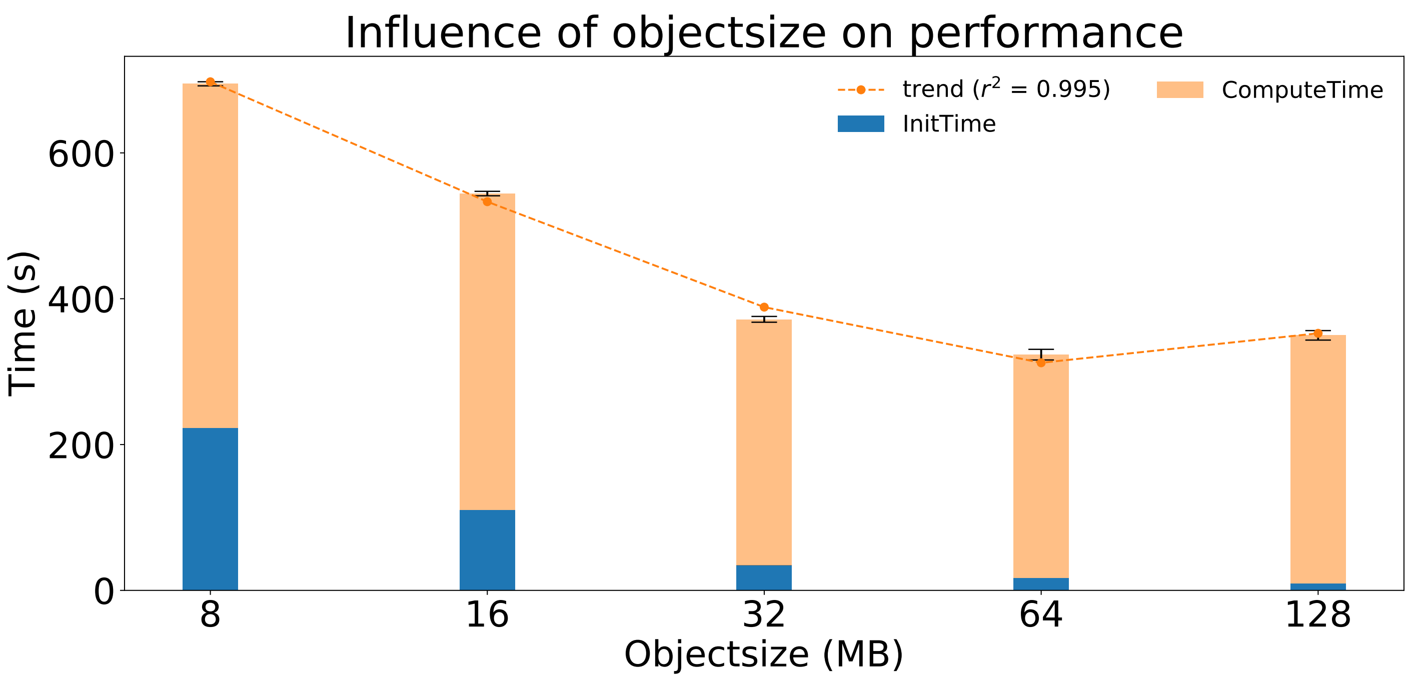


Figure 15: Total execution time for datasets of 512 GiB, stored in RADOS objects of 4, 8, 16, 32, 64, 128 MiB for a 10% row selectivity query.

5.5 Objectsize

We store the data in RADOS in self-contained objects, as explained in Section 3.8. Each RADOS has a minimum size of 4 MiB and a maximum size of 128 MiB. Different objects can have different sizes, and the sizes are not required to be a nice two-power, as RADOS supports storing objects with different sizes. Note that for SparkHook, we do require that all objects are of the same size, in order to easily calculate object offsets.

SparkHook always processes one object per Spark process at a time. The object size could greatly influence the overall system performance. For a dataset of a given size, fewer but larger objects would mean that a relatively larger part of the data could be stored inside each object, alongside a schema, magic bytes, headers, footers etc, leading to a relatively larger amount of information per bit.

On the other hand, fewer files also results in lesser potential parallelism, as Spark generates one task per one object. Additionally, larger objects means essentially that a larger single rowgroup is stored inside a file. Parquet metadata is available for rowgroups only, not for smaller units. Having larger rowgroups per object results in having coarser metadata information about the stored data. The coarser metadata will in practice mean that we can skip fewer row groups when filtering, for example. Using smaller object sizes results in greater potential parallelism, and finer metadata information. However, it also increases the amount of overhead associated with obtaining data: more offloading requests and data responses are needed for SparkHook to process all data. Additionally, we store relatively less information per bit inside each object, due to the schema, headers, footers, magic bytes that have to be present in each object, which results in more compute cycles going to reading redundant data.

We were interested in finding out the influence of objectsizes on processing speed for different queries, and created an experiment to understand this variable. The results are available in Figure 15. There, we find that object size indeed has a significant role in terms of processing speed: Some sizes lead to *double* performance when compared to the RADOS default object size of 4 MiB.

When reviewing SparkHook's initialization time (InitTime, bottom bars) and computation time (ComputeTime, top bars) for this experiment, we find two different patterns. For the initialization time, we see that there is an exponential relation: For each doubling of the objectsize, the initialization time SparkHook requires halves. The reason for this is simple: Every next objectsize in the Figure can store twice as much data, meaning we require half as many objects for our 512 GiB NYC taxi dataset. At initialization time, the Spark cluster requests metadata information from the RADOS cluster for each object. Having fewer but larger objects directly results in lesser time spent on requests. Especially for the lower object sizes, i.e. 4, 8, and 16 MiB, there is a lot of overhead in the initialization time due to the large amount of objects, greatly impacting overall performance.

For the computation time, we see another pattern. With each doubling of object size, we become slightly faster, until we reach top performance with 64 MiB objects. Apparently, using larger objectsizes is more efficient for computation with SparkHook. We store more information per bit for each object, and have to do fewer requests to the programmable storage from the distributed data processing system.

One interesting observation is that SparkHook can process datasets built from 64 MiB RADOS objects slightly faster than the 128 MiB object dataset variant, even though the initialization time is slightly lower. It turns out Ceph Storage cluster and RADOS work best when using smaller objects than the upper limit of 128 MiB objects. This limit was imposed because Ceph's and RADOS' internal functions were created with the idea that executing functions on objects would take a very short while. With very large objects, these functions take significantly longer to complete, producing a laggy, sometimes even crashing cluster. From personal experience, we found that manually increasing the limit and using datasets with e.g. 512 MiB objects indeed produced a laggy, unstable and often hanging RADOS cluster. Clusters hosting datasets with 128 MiB objects are stable, but still slightly slower than clusters hosting datasets with 64 MiB objects. We get our data slower with 128 MiB objects, slightly degrading the computation time when compared to the same dataset with 64 MiB objects.

For our 512 GiB dataset, the cost of the RADOS clusters being slightly slower with 128 MiB objects was a bit higher than the reduction in initialization cost for SparkHook. With larger datasets, which have more objects, we expect that the benefit of using fewer but larger objects will outweigh the slight slowdown of using 128 MiB objects with the RADOS cluster. Overall, when blindly deciding which objectsize to use for a dataset, it is better to pick larger sizes than smaller sizes, due to the added overhead for smaller objects.

Conclusion-4: Practitioners should carefully select the dataset RADOS object size when building datasets for SparkHook. When estimating, it is better to use larger object sizes, up to 128 MiB, as this decreases overhead time considerably.

5.6 Objectsize and Selectivity Influence

When experimenting, we found that object size and query selectivity have stronger correlation on read performance when used together. Until now, we only experimented with selectivity and object size variations individually. Figure 16, measures programmable storage reading speed. We added our own experiment in Figure 17, where we measure overall performance of SparkHook with many variations in row selectivity and RADOS objectsizes.

From the measurements, we can clearly see an interesting pattern. There are the large row selectivities 100%, 99%, 90% and 75%, and the smaller row selectivities 50%, 25%, 10%

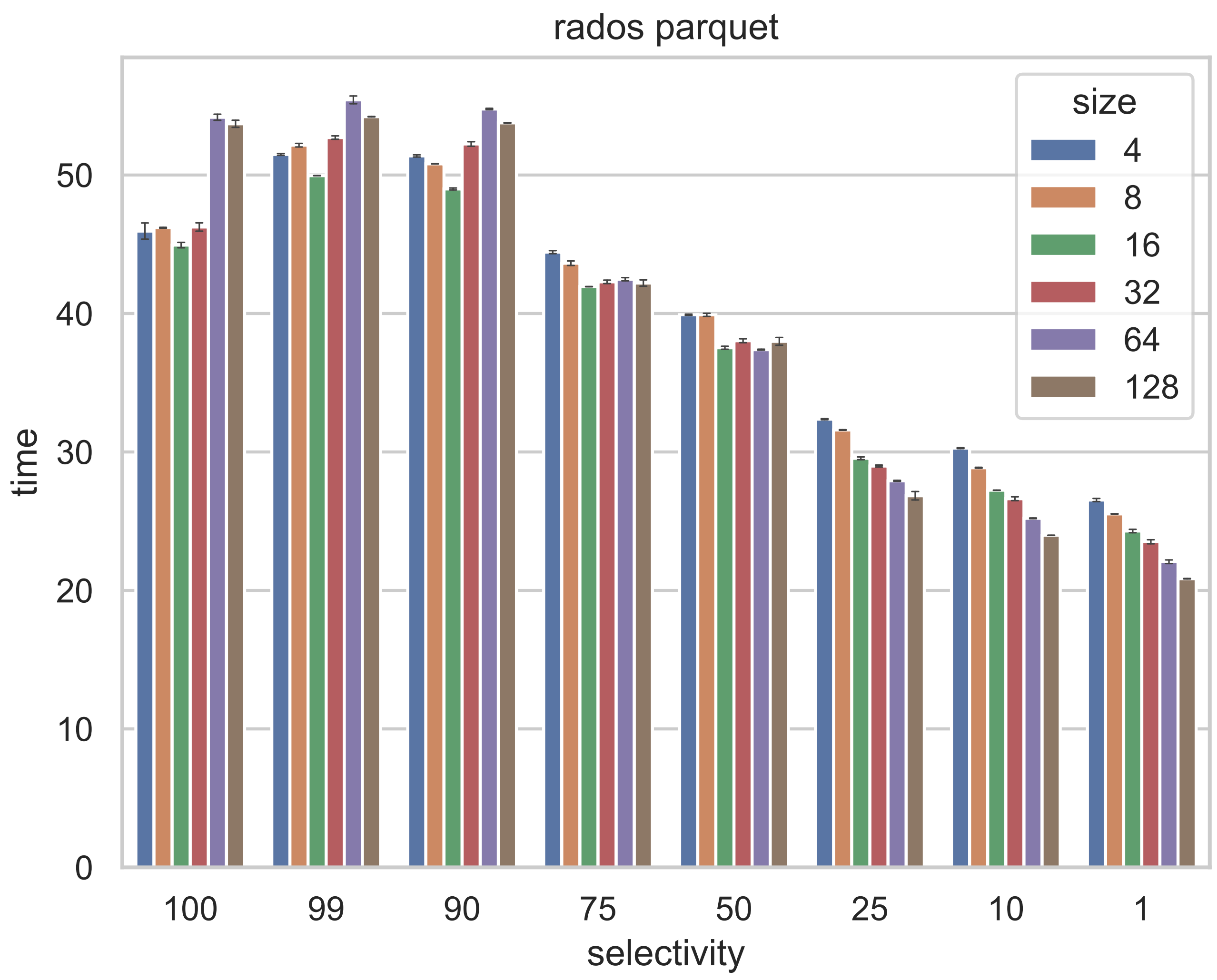


Figure 16: Programmable storage backend read times for a dataset of ±143 GiB, stored in RADOS objects of 4, 8, 16, 32, 64, 128 MiB, for a range of row selectivity queries. Due to the huge search space, the experiments are only repeated 5 times. Note that the performance variability is very low however, as with all other experiments. This Figure comes from other team members. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

and 1%. There are the large objectsizes 128, 64, 32 MiB, and the smaller objectsizes 16, 8, 4 MiB. For the larger row selectivities, the smaller objectsizes perform better than the larger objectsizes. For the lower row selectivities, this relation inverts: using lower row selectivities, larger objectsizes are more efficient.

It turned out that for sequential reading, every increase in objectsize is slightly faster. The maximum difference was between 4 MiB and 128 MiB, which was only approximately 23%. For parallel reading, we found that lower objectsizes perform best, with 16 MiB the fastest, and each next size becomes increasingly slower, up to 128 MiB. So, the parquet reader performs better with larger objectsizes, and parallel I/O performs better for smaller object sizes. 16 MiB objects are the sweet spot between parallel I/O and parquet reader optima. This explains why

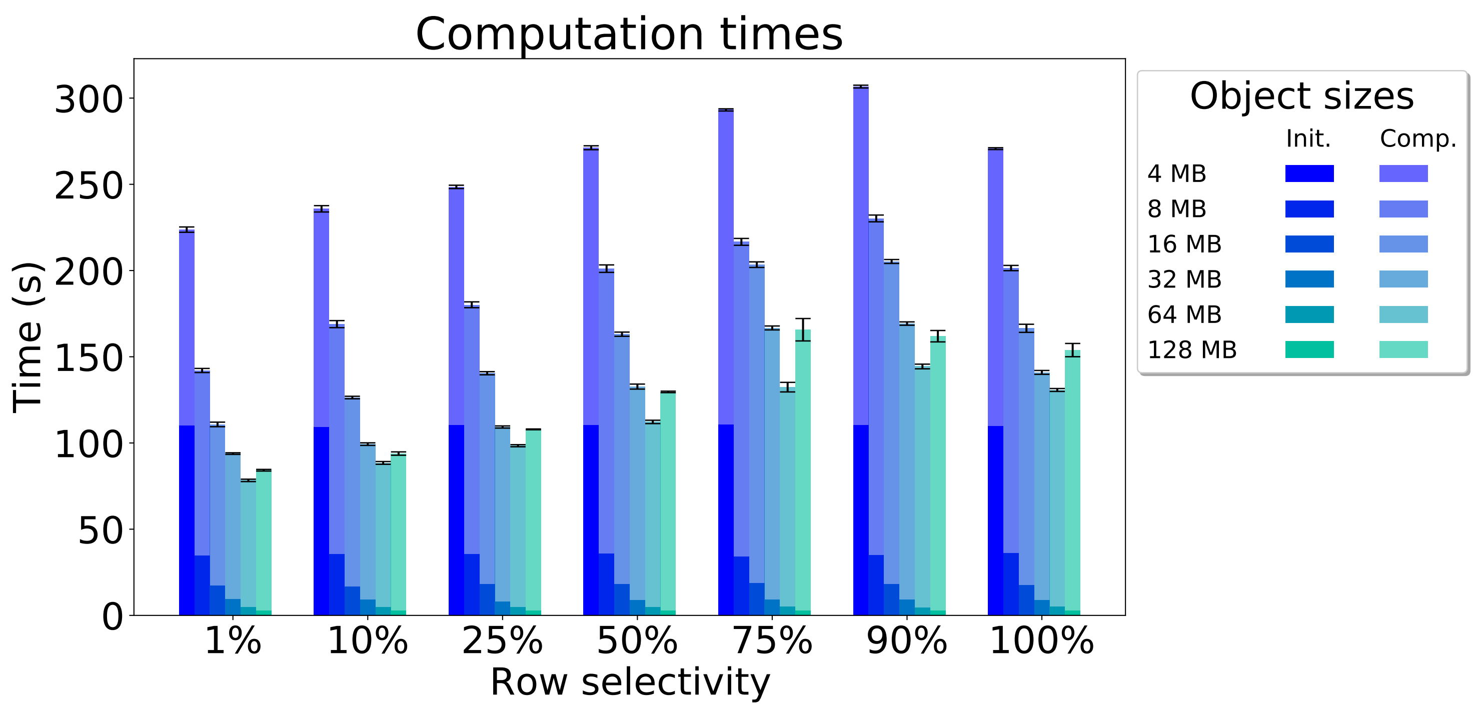


Figure 17: Total execution time for a dataset of 128 GiB, stored in RADOS objects of 4, 8, 16, 32, 64, 128 MiB for a range of row selectivity queries. Due to the huge search space, we repeated experiments only 6 times. Note that the performance variability is very low however, as with all other experiments.

16 MiB performs so well on the higher selectivities. The reason that the relation inverts is due to the 'blocking problem': transmitting large blobs of data causes blocking when executing parallel reads. This is what causes 128 MiB objects to be slower for the higher row selectivities. For the lower row selectivities, the blocking problem is less prominent, since we filter out a large portion of the data. This makes the higher objectsizes outperform the lower objectsizes when low row selectivities are applied.

We found it interesting to see whether these relations are visible in SparkHook as well. Figure 17 shows that this is not the case. Any positive effect on the computation time that the relations may have had for lower object sizes is immediately diminished by the extreme amount of initialization time. For all row selectivities, we see the same trend: Each doubling of objectsize increases performance, until 128 MiB objects are used. We explained this relation in detail in Section 5.5.

Conclusion-5: Objectsizes and selectivities have a strong influence on storage backend performance. Despite this influence, SparkHook performs best for larger objectsizes, in line with previous conclusions. Practitioners should generally pick higher objectsizes to increase overall performance more.

5.7 Data Scalability

An important property of a distributed data processing system is its *data scalability* behavior, which evaluates system performance with increasing dataset sizes. When performance scales (almost) linearly with dataset sizes, we can process extremely large quantities of data with minimal overheads.

In Figure 18, we display the execution time for increasing dataset sizes, for both SparkHook

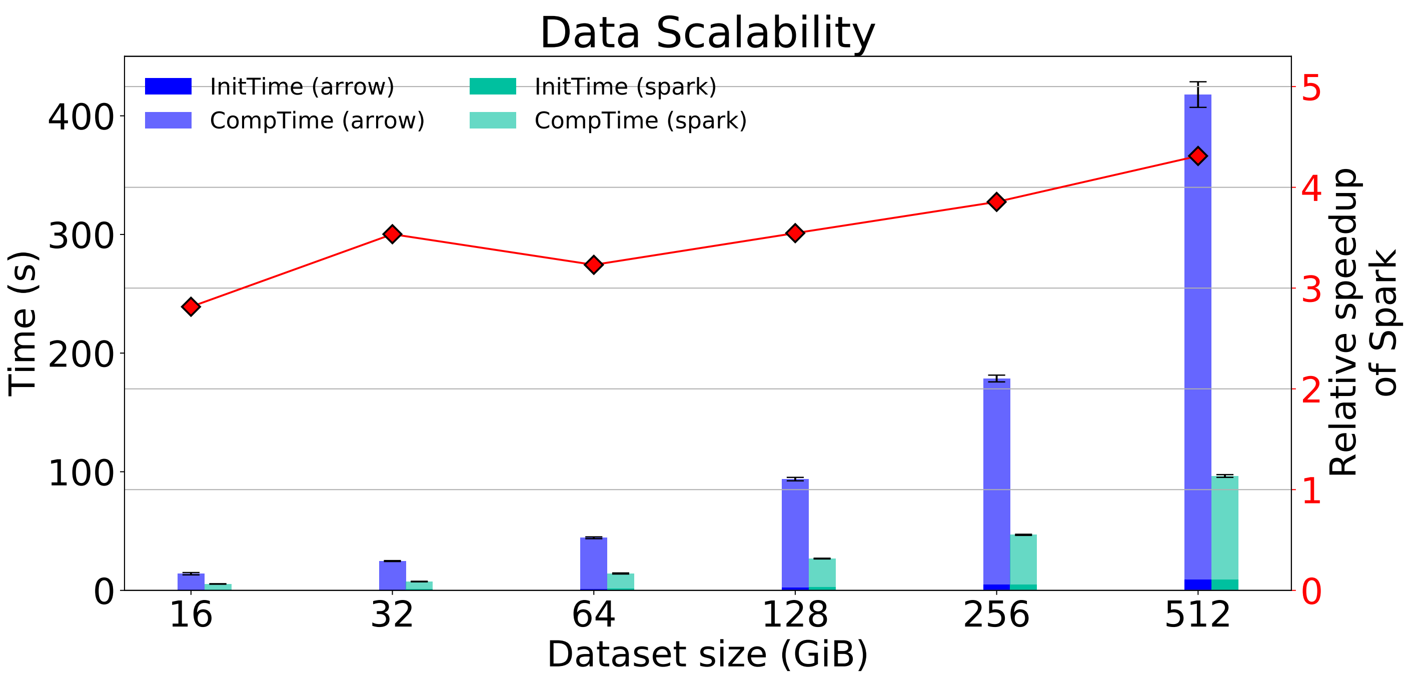


Figure 18: Measured execution time for SparkHook, with individual initialization time and computation time, and the relative speedup of vanilla Spark (line) on varying amounts of input rows, for a cluster of 8 Spark executor nodes and 8 Ceph OSD nodes, reading from uncompressed parquet files in batches of 1 KB, using Dataframes.

and vanilla Spark reading data from a mounted directory to Ceph (CephFS). Notice that, in general, the execution time approximately doubles as the dataset size doubles, following a clear exponential trend. Both systems scale well with increase in dataset size.

When compared to vanilla Spark, SparkHook is much slower than Spark, with a growing difference for larger datasets. Vanilla Spark fetches data from Ceph through CephFS. SparkHook fetches data through sending requests to a custom loaded Skyhook class interface.

One of the major reasons we are slower is because of serialization overhead. To send data back from our custom loaded Skyhook class interface, we need to serialize the computed-on data. Arrow is very inefficient in this. From the measurements described in Section 6.1, we believe we should be able to get a 20-25% speedup on overall performance when implementing a more efficient serializer for Arrow.

Another major performance bottleneck derives from the way we currently pushdown row selectivity filters. We obtain them from the Spark internals in a location that allows *optional* filtering, meaning that we are free to apply or not apply provided filters, and Spark will re-apply the filter during computation. In order to reduce overall time consumption further, we need to obtain computations from another place in Spark, such that Spark no longer executes the filtering operation again during computation. This could reduce SparkHook execution time by another 20%.

Another way to improve performance beyond the already mentioned points is to improve the cooperation between the compute and storage layers. Currently, the compute layer sends requests for data to the storage, with computations attached. The storage layer reads all data at once, applies computations, serializes the data, compresses the data and sends it in one bulk to the compute layer. In the meanwhile, the compute layer remains idle, since it requires data to perform all non-pushed down computations. Vice-versa, if the compute layer is processing bulk data, it has no outgoing requests for more data until it needs more data, at which point it has to wait again for the storage layer to process the request. It would be better for performance if there is very little latency between request and response, as this reduces waiting. Solving the aforementioned serialization problem will reduce this latency partially. We could solve the majority of the latency problem by transforming our bulk processing to a stream instead. By using a stream, we send back data more quickly, meaning that the compute layer can already start applying non-pushed down computations. Another solution would be to use more Spark executor instances than there are cores on the cluster. This would increase the number of outgoing requests for the compute cluster. When all cores are busy with computing, there would still be outgoing requests, and the storage layer would remain busy, alongside the compute cluster. Special care has to be taken in choosing the amount of extra Spark executors. Too few extra executors could result in the storage clusters still being partially idle at some intervals, while too many executors could result in the storage cluster becoming overburdened, creating queues that are too large to fit in memory.

We ask for patience. Spark is a framework of over 7 years of development now (at the time of writing), having seen 3 major versions. Our system is produced in 7 months, without a large software foundation. By implementing optimizations and analyzing performance, we are certain we can and will greatly improve performance, to a point where it is comparable to, or even faster than Spark.

Conclusion-6: SparkHook scales well with dataset sizes, but has several disadvantages when compared to vanilla Spark. Its disadvantage to Spark increases with scale. There are several improvement points for SparkHook, which should provide on-par or even better performance than vanilla Spark.

6 RQ4: Prototype to Production-Ready & Lessons learned

During this project, there were many implementation choices, many optimizations, and many more ideas for optimizations, but most of all: there are many lessons learned. In this Section, we provide insights into the issues we faced, we discuss the main current performance bottlenecks and explain how to solve them. As this project is ongoing, we expect to implement proposed solutions in the near future. This Section answers RQ4: what are the lessons learned and how can we make this production-ready, achieving good performance and efficiency?

6.1 The Double Interpretation Problem

The first and most fundamental problem in our design and system implementation was the 'Double Interpretation Problem': In regular distributed data processing systems, we fetch the data from 'standard' storage clusters, which just return the requested data as-is to the client, in large byte sequences. With our system, we have to interpret the data in the storage nodes to apply computations on it *before* sending it back to the client in large byte sequences. If we would not interpret the data, we would not be able to apply computations in storage nodes. On the client side, we have to re-interpret the received data again. With our pushdown design approach, we are forced to interpret the data *twice*, whereas regular distributed data processing systems only have to do so *once*. This is an inherent drawback to distributed data processing system design with computations transferable to decentralized devices. We can reduce the impact of this problem by minimizing the cost for interpreting the data, and by focusing on pushing down operations that significantly reduce the data to send back.

During experimentation, our team analyzed the time composition that SparkHook requires. [4] In Figure 19, we depict the cost of moving data from the storage backend to the compute cluster. To better understand what happens, we first review the different time segments visible in this Figure, in chronological order. When pushing down a computation for an object, we first need to stat a fragment to obtain an object ID (1). Then, we serialize (2), transfer, and deserialize (3) the data scan request to the OSD owning the correct object ID. The OSD reads the object (4), scans (reads, interprets) the raw data from the object as a parquet file (5). During this stage, we also apply our filtering and projection computations. Our output data is a *Result Table*. Once all data has been read and processed, we serialize the resulting *Result Table* (6), and send it back to the client. The client deserializes the Table (7), and our cycle is complete.

Intuitively, the Double Interpretation Problem spans stages 5, 6, and 7: stage 1, 2 and 3 are required to make a request to the storage backend. Together, these stages form less than 0.1% of the total request time. We assume that a regular storage backend (as opposed to our programmable storage backend) requires a similarly small amount of time to receive and interpret a request. Stage 4 is a necessary step, present in regular storage backends as well. It reads the data (without interpreting anything). Stage 5 is part of the problem, since this step is only required because we need to interpret and compute on the data in the storage backend. Stage 6 and 7 are part of the problem as well, since we need to somehow transfer the interpreted and computed-on data back to the client. However, we only consider stages 6 and 7 to be part of the problem, and not stage 5: after all, if a regular storage cluster would be used, we would execute stage 5 (interpreting data, executing computations) on the compute

[4]One feature described in this Subsection (analyzing the time composition that SparkHook requires, backend) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

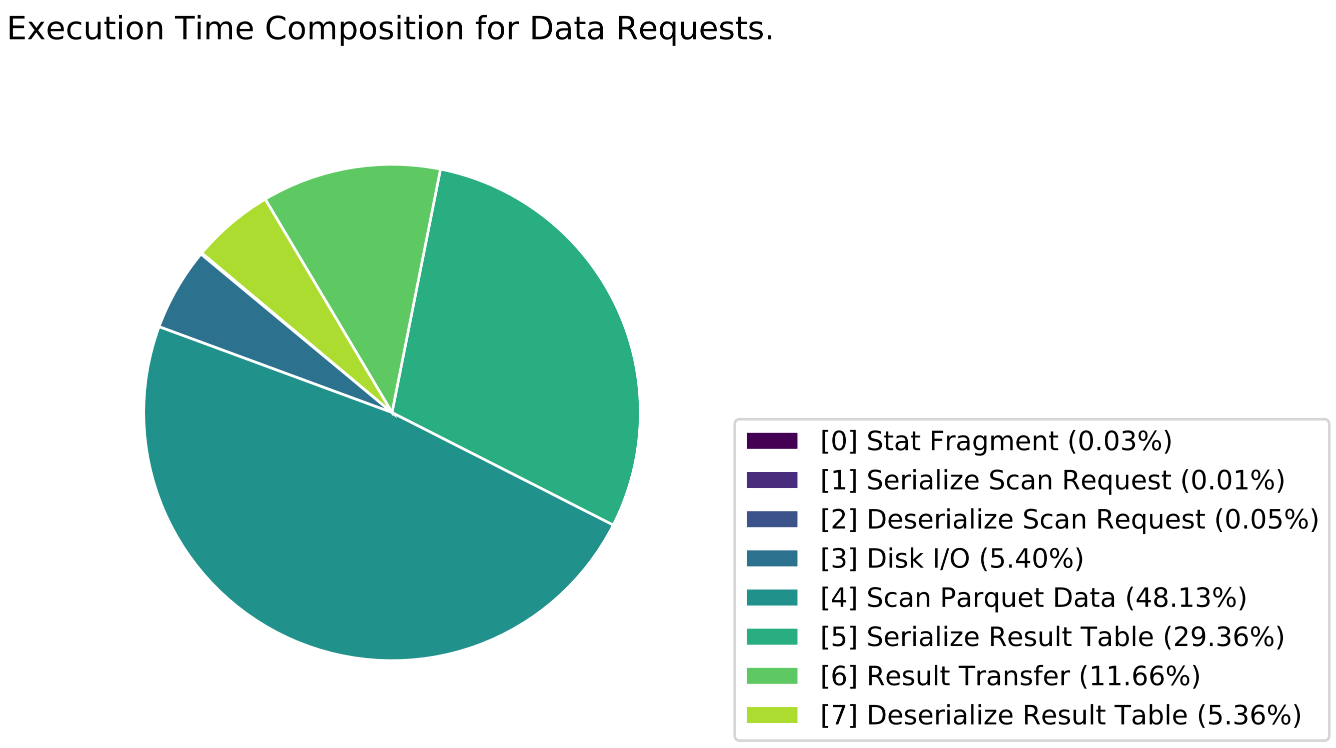


Figure 19: Time composition for obtaining data from the storage backend, requesting a compressed IPC file. The data from this Figure comes from other team members. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

cluster in an equivalent manner as we do now in the storage backend. Just as with stage 1, 2 and 3, stage 5 is an essential stage, executed in both regular and programmable storage backends.

Stage 6 and 7 are the stages where we send data from the programmable storage backend and receive it on the compute cluster. With a traditional storage backend, we would require some amount of time to send data back to the client as well. However, this data is raw, uninterpreted data. Sending data back with traditional storage backends would not require any serialization or deserialization. With our programmable storage backend, serialization consumes 42.8% of the total time to fetch data. This is a quite large amount of time, significantly slowing SparkHook. The Arrow developers confirmed the measurements, and confirmed that the Arrow serialization modules are indeed inefficient and not well-maintained.

The Arrow developers and ourselves did additional experiments to find out why the serialization time is (relatively) so extreme, the results of which can be found in Figure 20. From the measurements, our team and the Arrow developers noticed that $\pm 50\%$ of the time is spent in `__memmove_sse2_unaligned_erms`, which is the major function in libc's well known *memcpy* command, which copies raw memory bytes between two buffers. This makes sense, as the current *Arrow Table* serialization protocol enforces that all fragments making up the table are copied to one, contiguous buffer. As all data is widely spread in small buffers across memory with no detectable patterns, this constraint triggers many *memcpy* calls. In Figure 21, we display the cached request time consumption. When repeating requests, we found that the relative time consumption of all functions dropped significantly, except the internal *memcpy* function. Approximately all waiting time is caused by copying data in these situations.

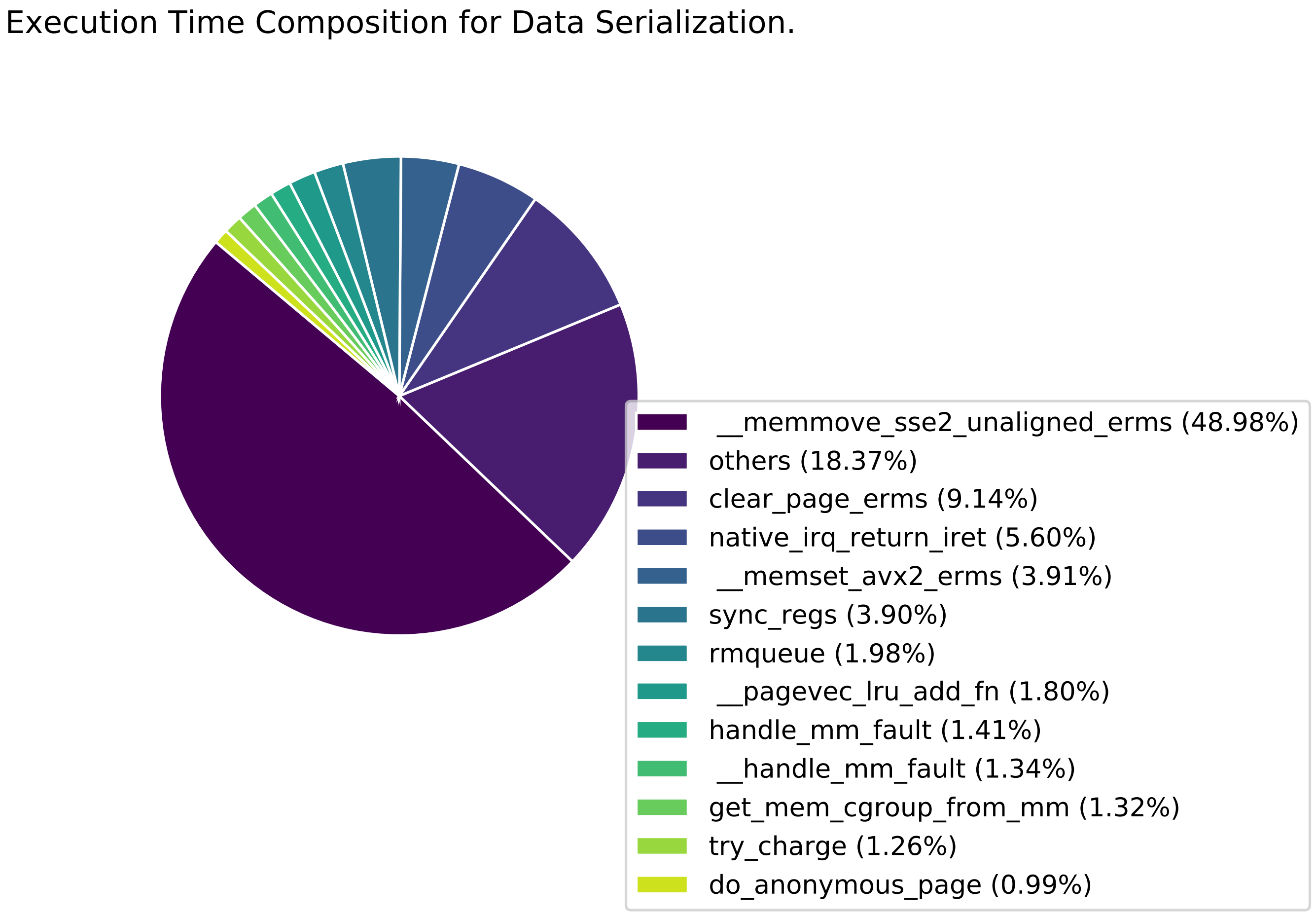


Figure 20: Serialization time composition when requesting a compressed IPC file, executed with `perf`. The resulting function categories are sorted based on relative size. The data from this Figure comes from Arrow developers. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

To solve this issue, and in turn alleviate the Double Interpretation Problem, we are currently building a system without contiguous-buffer-constraints, such that no memcopies need to occur, improving the serialization speed. Initial results are promising, showing significantly reduced execution time spent in serialization and deserialization. It is still a far way from being applied in practice, however.

6.2 The Pushing Problem

Another problem to discuss is the 'Pushing Problem', which deals with the question whether we should or should not pushdown a computation. We define a computation C should be pushed down to the storage layer if and only if the total execution time of C decreases when we do so, i.e. if the performance benefits of pushing down C are greater than the costs, when comparing to not pushing down C. We could formalize the cost of pushing down a computation C as:

$$\text{Cost}(C) = C_{\text{client}} + I_{\text{client}} + T(D_{\text{pushdown}}) + S_{\text{OSD}} + C_{\text{OSD}} + I_{\text{OSD}}$$

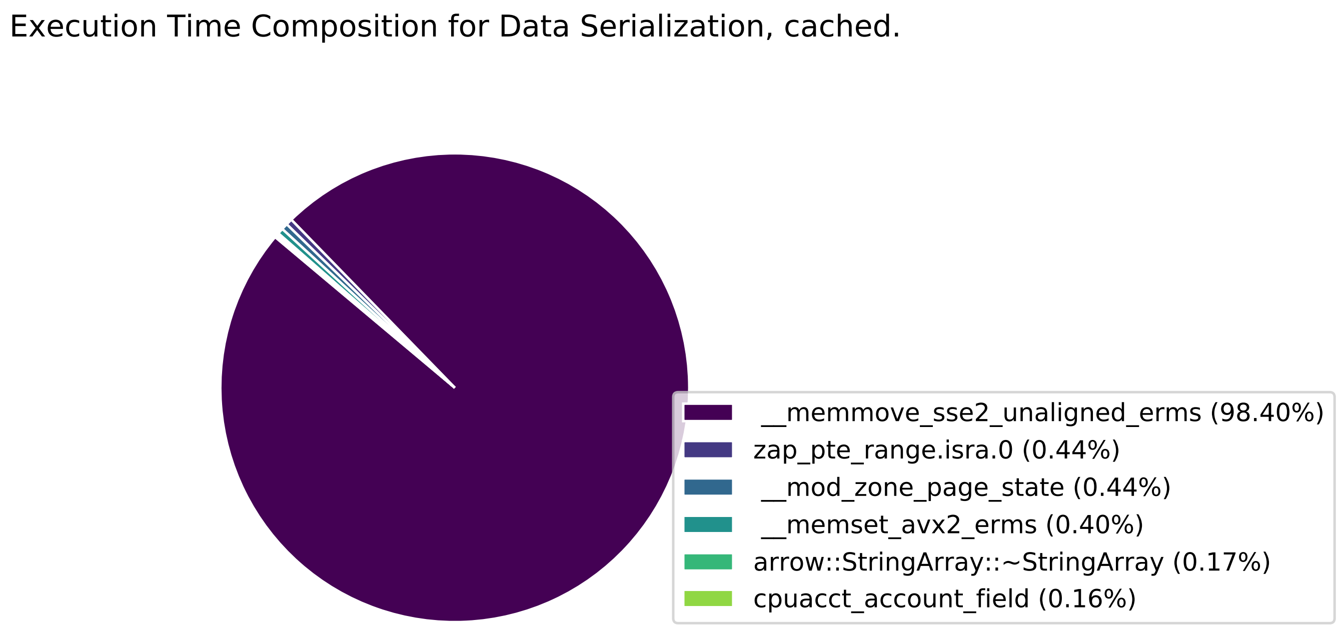


Figure 21: Serialization time composition when requesting a *cached* compressed IPC file, executed with `perf`. The resulting function categories are sorted based on relative size. The data from this Figure comes from Arrow developers. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

Where

$$\begin{aligned}
C_{\text{client}} &= \text{computation cost on the client machine} \\
I_{\text{client}} &= \text{interpretation cost on the client machine} \\
T(X) &= \text{transmission cost to get } X \text{ data units from the remote to the client machine} \\
D_{\text{pushdown}} &= \text{amount of data units after computations have occurred} \\
S_{\text{OSD}} &= \text{serialization cost to get data on the OSD in a shape that allows it to be sent} \\
C_{\text{OSD}} &= \text{computation cost of the pushed down operations on the OSD} \\
I_{\text{OSD}} &= \text{interpretation cost on the OSD}
\end{aligned}$$

The formalized cost formula for a regular, non-pushed down computation C_{regular} with remote data could be:

$$\text{Cost}(C_{\text{regular}}) = C_{\text{regular}} + I_{\text{regular}} + T(D_{\text{regular}}$$

Where

$$\begin{aligned}
C_{\text{regular}} &= \text{computation cost on the client machine} \\
I_{\text{regular}} &= \text{interpretation cost on the client machine} \\
T(X) &= \text{transmission cost to get } X \text{ data units from the remote to the client machine} \\
D_{\text{regular}} &= \text{amount of data units to transfer}
\end{aligned}$$

Using these two formulas, we can derive a formula indicating the Pushdown Cost (PC) of pushing down vs not pushing down some computation:

$$
\begin{aligned}
\mathrm{PC}_{\mathrm{Computation}}(C) &= C_{\mathrm{regular}} - C_{\mathrm{client}} - C_{\mathrm{OSD}} \\
\mathrm{PC}_{\mathrm{Interpretation}}(C) &= I_{\mathrm{regular}} - I_{\mathrm{client}} - I_{\mathrm{OSD}} \\
\mathrm{PC}_{\mathrm{Transmission}}(C) &= T(D_{\mathrm{regular}} - D_{\mathrm{pushdown}}) \\
\mathrm{PC}(C) &= \mathrm{PC}_{\mathrm{Computation}}(C) + \mathrm{PC}_{\mathrm{Interpretation}}(C) + \mathrm{PC}_{\mathrm{Transmission}}(C) + S_{\mathsf{OSD}}
\end{aligned}
$$

Note that not all, but all *major*, datasize-sensitive variables are present in the formulas we use. Also: for the sake of simplicity, we review only a single computation at a time, ignore the many side-effects that can occur when pushing down multiple computations.

As we can see, there are *three* main factors to take into account when deciding to offload a computation C: computation, Interpretation, and Transmission cost. Additionally, there is the constant drawback of serialization cost to take into account for offloading a computation. Note that any of the three factors could be negative, however, indicating that pushing down C is beneficial for given negative factors. For example, pushing down a filter computation that reduces the data size by 99% will likely show a strong negative cost for the Transmission factor, as we can now skip sending all filtered data, while we had to send all data before.

The question of whether to push or not to pushdown computations is now reduced to a single cost formula. However, before we can apply this formula in practice, there are several obstacles: What if the pushdown cost formula shows a big favor toward pushing down a computation C, but the storage cluster is overburdened already? We would make the wrong decision, and pushdown C unjustly. We solve this problem by specifying that the storage system is allowed to skip any computation C_0 and all following non-executed computations $C_1, C_2, ...$, pushing the data back in the current state to the client. $C_0, C_1, C_2 ...$ will be executed on the client machine, in the Arrow layer, before the data is returned to Spark.

A more persistent practical problem, one found in many formal cost functions: we do not know the cost of Computation, Interpretation and Transmission for a computation C before we actually executed C locally and in the storage cluster through pushdowns. In many papers, this is solved by requiring users to specify an estimated cost. Here, we cannot ask users to do so, as it would greatly reduce the practical applicability of this work. Instead, we let the user decide whether to push down any kind of computation of not. For example, users can specify whether SparkHook should pushdown filters or not, predicates or not etc.

6.3 Runtime-based Load Balancing and Coordination

In our experiments, we were the sole user for both compute cluster and storage backend. In practice, however, we expect that many users will offload computations to the same backend. If we just always offload queries, there will be situations where the storage cluster is overburdened by computations already when we want to offload. This would cause the storage backend to take longer than if we would not offload. [5]

Most distributed data processing systems have functionality to balance loads across nodes. They can do so, because of the great amount of information available to the chosen distributed data processing system at runtime. The utilization of every node can be reviewed, as well as task queues, performance for similar jobs in the past, etcetera. This makes one wonder: how

[5]One feature described in this Subsection (computation pushback) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

do we balance load across the programmable storage backend? Just as with distributed data processing systems, the load could be highly unbalanced, with storage nodes receiving many requests for data and computation offloading, while others are completely idle. This is a more difficult problem to tackle than with distributed data processing systems. For the programmable storage backend, we do not have the rich information that distributed data processing systems have, because we do not impose information-tracking requirements on the storage backend, so we cannot guarantee that a backend keeps track of node utilization information, or that it provides any runtime information at all.

We could create a projection system, which maps each request in all compute nodes to storage nodes using some coordination heuristic, in such a way that requests are balanced. For example, a mapping using client node ID and requested object ID could be used to pick a storage node from a list of nodes containing the data we want to request. The mapping would ensure that all clients together would request data in a evenly distributed fashion, ensuring each storage node has as much chance of being chosen as all others. This system would still have several problems, however. One problem is data load. If data is not spread approximately evenly between nodes, it is not possible to make a mapping that ensures requests are balanced between storage nodes, since many requests need to be directed to one storage node only. Another problem is data and request skew. If some requests take much longer to process than others, using a scheme to evenly balance requests across storage nodes will not help to balance the programmable storage cluster. Some nodes would get many requests that take much longer to process than the others, causing them to get overburdened and unbalanced.

To perfectly and soundly balance loads in the programmable storage cluster, we require fast access to the runtime status of candidate nodes for data and offloading requests, much like distributed data processing systems. In other terms: we need strong global runtime information about the storage cluster, which is something we cannot have.

To alleviate overburdened nodes, we propose to enhance SparkHook with storage-level decision making capabilities: A storage node can measure whether it is overburdened internally, by checking its request buffer size and limit. The request buffer simply contains the requests to process on that storage node. If a storage node is overburdened, it could skip the computation part, and just send all requested data back, alongside the not-executed computations. Alternatively, the storage node could execute part of the computations, to attempt to reduce dataset sizes while only slightly increasing the workload. The parts of computations that were not executed are sent back to the client in this case, alongside the data. This is what we would call '*a pushback*'.

An intelligent pushback service could even add priorities to pushed down computations. Operations which are expected to greatly reduce the dataset size at a low cost get higher priority than computations which do not decrease or even increase dataset sizes. Our intelligent scheduler would schedule computations with higher priorities first, and favour executing high priority computations in most situations, rather than pushing them back with all other computations. All computations that are pushed back to the client along requested data are executed on the client machine, ensuring data will be correct when delivered to Spark.

Since our system pushes down computations to the individual storage nodes, it may occur that some programmable storage nodes are overburdened, while others are not. In those situations, the nodes not overburdened will execute the pushed down computations, while others will skip computation and only return data. This leads to variable performance between requests and unbalanced compute nodes on the client. Luckily, most, if not all modern distributed data processing systems dynamically balance loads, which means that faster client nodes get new workloads, after finishing part of the total workload much quicker than other nodes.

Note that the notion of *pushing back* computations does not require any global runtime information about the storage cluster, making this a viable, efficient solution. However, we distribute load between the storage and compute layers with this solution, while our original goal is to balance load between storage nodes in the storage layer.

In order to use *pushbacks* to distribute load between storage node, only a minor adaption is needed. When pushing back a request in a storage node, we no longer send the data along. Also, we no longer execute part of the computations. Instead, we only send a flag back to the compute cluster, indicating rejection. When a compute node receives a reject flag for a pushdown, it accesses the list of nodes containing the required data, and repeats the request to a different node, until it finds a storage node that accepts. If a threshold is reached, a normal read request is issued without offloading to any node containing the data. The data is returned, and the client node executes the computations itself.

With these modifications, client nodes search for available storage nodes to handle offloading, balancing load without global information. This solution works, but is not optimal. Compute nodes could stop searching for available storage nodes too soon, while there exists a storage node that is available. Additionally, searching consumes some amount of time. The solution can be implemented to be near-optimal, however. For example, by making each compute node search for available storage nodes in an other pattern than the other nodes, the chance of finding an available storage node early increases considerably. This could be achieved by search heuristics depending on compute node ID, object ID etcetera.

We have not yet implemented load balancing, and aim to implement this using *pushback* strategies with decentralized coordination optimizations.

6.4 Data Set Partitioning

With this project, we require data in a very specific shape on the OSDs. We require that datasets of potentially many terabytes are stored in objects, in sizes of 10s of megabytes. Inside each object, we have to store a full, uncompressed parquet file with one rowgroup, with headers and footers, as explained in Section 3.8. This is impeding practical applicability of this work, since many existing datasets do not satisfy all current requirements. These datasets would require to be repartitioned to fit inside Ceph, following a one-to-one mapping from files to objects, with a single rowgroup in each file.

In practice, however, individual Parquet files could be arbitrary large. From personal experience, datasets regularly contain parquet files which are tens of gigabytes in size. [6]

We require small parquet files with constant sizes and single rowgroups that fit inside a RADOS object due to our file design, as explained in Section 3.8. We use this strategy because it makes each object independent. When the programmable storage backend receives a request for a specific object, it can satisfy it without touching any object other than the requested one. This makes the enables the reading implementation to scale out: the backend only has to fetch the object, and use a regular parquet reader implementation on the data inside it.

If we were to support existing datasets, with parquet files of unconstrained size, we would have to implement a complex reader and writer interface. To get the data stored in RADOS, we would need to cut the files in chunks of a two-power in size, e.g. 128 MB. We would need to write some kind of index file, containing the object ID's of the objects that form a parquet file, such that we can reconstruct the file later. When receiving a request for a file, we would need to request the index file. The OSD with the index file would service the request. That

[6]One feature described in this Subsection (the data shape) is a collaborative effort, originating from the research team we are part of rather than only us. We are not the main contributor for this feature, but rather a member of the team. See more about originality, contributions and acknowledgements in Section Declaration of Originality.

OSD would query all other OSDs for the objects listed in the index file to reconstruct the file. Then, it would be able to read the data as a parquet file by processing the contained data from all objects iteratively. This design has several problems, the most fundamental one being: to read a file, an OSD would need to perform much intercommunication, to fetch the right objects. This severely impedes scalability of the storage backend design. Additionally, using such large files at once could fill up available RAM, as there is no file size constraint and all data has to be read in to be able to compute on it. In principle, we could alleviate these problems if we only read the data at a given offset, up to a given length. It is not possible to accurately calculate the correct object to fetch for a given offset, however. Moreover, row groups could span multiple objects, which would require us to load more objects at read-time when we find ourselves at the end of an object buffer. this would in turn require us to write a highly specialized reader. Finally, if we would change the data contents at given offset and write back, we have a severe synchronisation problem with other objects. For example, the parquet metadata would need to be updated to point to new byte offsets for the changed rowgroup, and the schema might need to be adjusted. Because we split the arbitrarily large file in multiple objects in this example, we would need to load more objects, and make metadata and schema adjustments. if we edit the schema, however, all objects belonging to the file must be updated as well, because new fields might be present or some fields could be removed. The complexity of this problem becomes much larger when considering multiple parallel writes could occur. This would require locking constraints for all objects forming the file, to ensure that only write process operates on all objects composing a file at a time, and to ensure no reading processes use any of these objects. RADOS was not meant for that, and would greatly slowdown overall storage backend performance. With singular, independent RADOS objects, our system is simple and efficient.

6.5 The Placement Group Autoscaling Problem

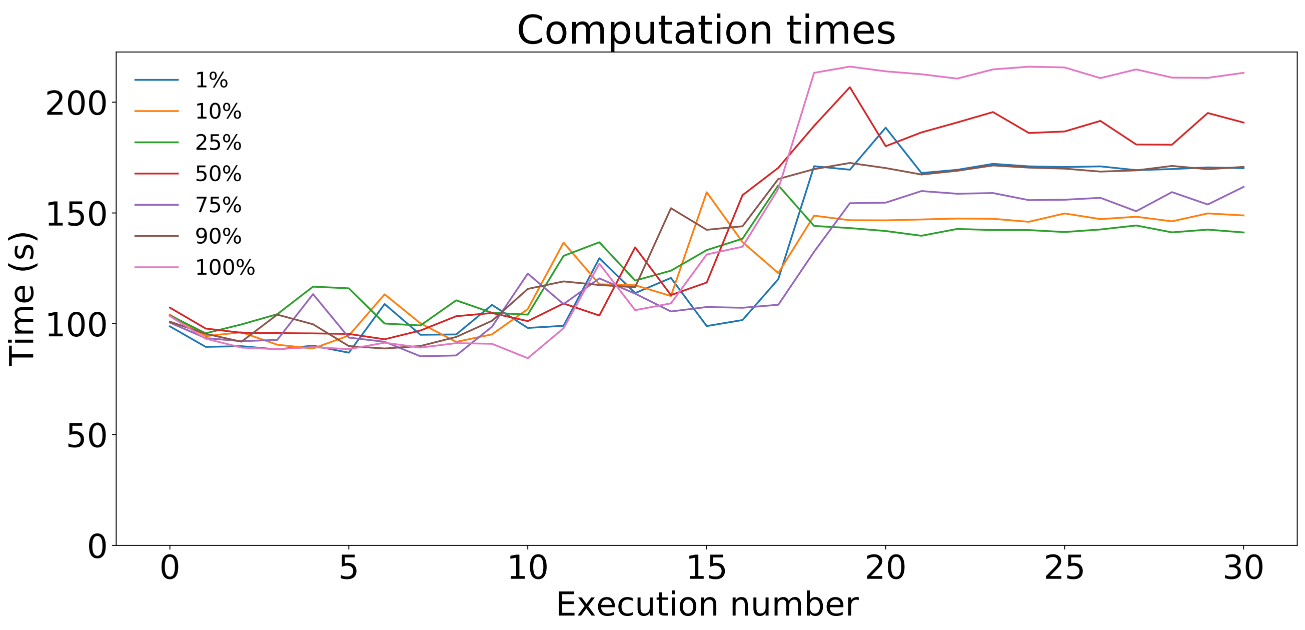


Figure 22: Gradually deteriorating performance when repeating the same experiment for the same data on the same hardware, for given *row selectivities*.

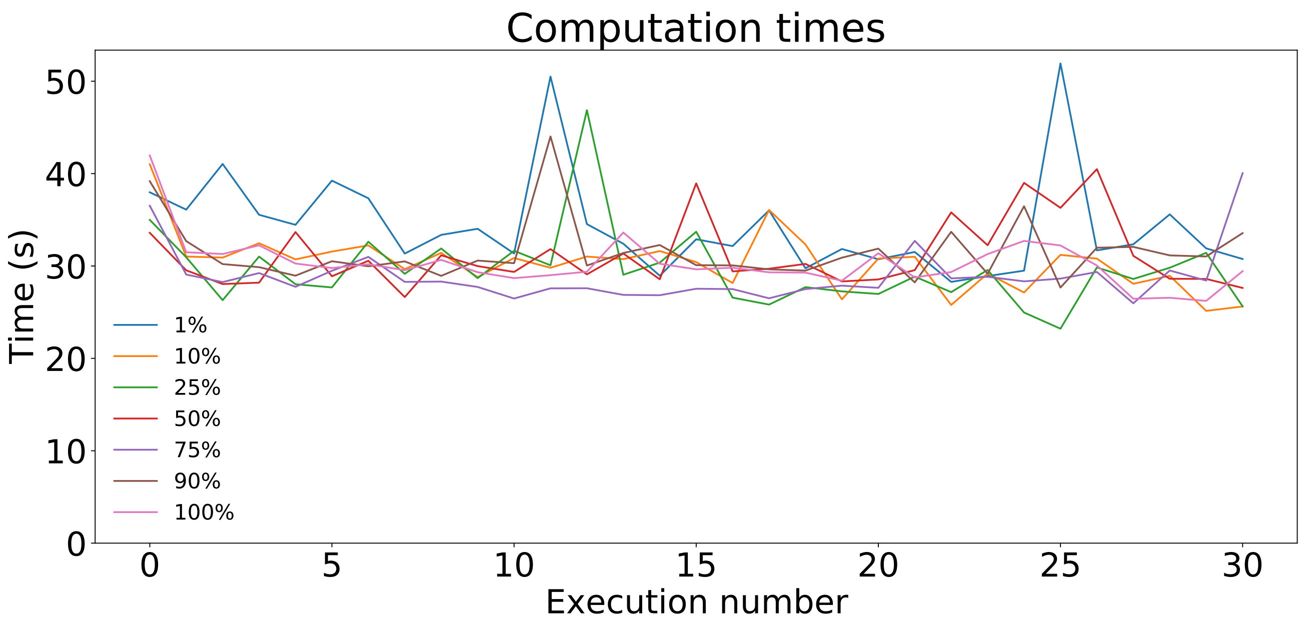


Figure 23: Resolved deteriorating performance by turning off the Ceph Placement Group Autoscaler. Verified on smaller dataset than in Figure 22, otherwise unchanged. Showing computation time for given *row selectivities*.

A more practical problem we faced during this project was the 'Placement Group Autoscaling problem'. When repeating experiments for e.g. 30 times, we noticed our performance gradually deteriorated. An example of this issue is depicted in Figure 22. It turned out that the Ceph Placement Group Autoscaler was the cause.

A Ceph Placement Group is an internal implementation detail of data distribution. When Ceph requires to fetch an object, it first needs to find the Object Placement for given object to be able to find it. The Object Placement contains the data needed to locate given object. Systems could have millions, billions or even trillions of objects. Tracking the Object Placement for each object individually would be highly inefficient or even infeasible. Instead, Placement Groups are used to maintain Object Placement information for collections of objects within a data pool. The objects are distributed over Placement Groups by hashing their object ID value and applying an operation based on the number of available Placement Groups in the given data pool and the pool ID.

When running Ceph, it automatically optimizes the Placement Groups, by resizing or scaling Placement Groups. Ceph reviews the total available storage, target number of Placement Groups for the system, and data quantity stored in each pool. Using this information, the Autoscaler tries to redistribute the Placement Groups. For example, when existing Placement Groups are only partially filled, it might be better for overall lookup performance to merge them.

During experimentation, the Placement Group Autoscaler would greatly reduce the number of placement groups we precalculated to be a good amount, even leading to some objects having no objects while others had many. Since SparkHook offloads computations to the nodes owning the requested objects, the nodes containing the objects after autoscaling were overburdened by work, while the nodes without any objects went idle. This caused the significant performance degradation from Figure 22. Luckily, Ceph has an option to turn of autoscaling. After applying this option, performance returned to normal. We verified this solution by executing a small query, as displayed in Figure 23.

6.6 GENI

Although the GENI infrastructure is a cluster with excellent hardware quality and quantity, we experienced quite many problems during this project.

Many problems came from the fact that there was no good documentation available for the programming interface. We needed the programming interface, as we wanted to automate resource acquisition. After consulting the source code implementation of the available GENI APIs, after asking fellow team members who had more experience with GENI, and after much experimenting, we finally got a fully working program to handle allocations and deallocations without any user interaction. Still, many days were lost when struggling with GENI that could have been spent otherwise, if better documentation had been available.

Other problems we experienced where with the nodes themselves. Some of the problems we did encounter:

- Reserved nodes never start, or start only after days of waiting.
- Nodes have no connection with the public internet, making direct access impossible.
- Nodes lose their local interconnect to other nodes seemingly randomly.
- After a reboot, nodes never return online.

The unpredictable hardware state often led to failing experiments.

7 Related Work

In this Section, we discuss work related to the topics we cover. Additionally, we discuss their contrast to this work, and provide links to other work for curious readers. In particular, we review related work in three categories: (1) Obtaining Data and Performance, (2) Computation with Generic Data Interfaces, (3) Computation Offloading with Specialized Hardware Support and (4) Computation Offloading with Generic Hardware.

Obtaining Data and Performance There is much research available on the subject of providing data to distributed data processing systems. For example, in Albis [19], the authors propose an entirely new columnar format to be used with modern hardware, following the idea that the main performance bottleneck has shifted from networking and storage to processing hardware in the past decade. In more recent times, research focus seems to have shifted away from specialized file types. Instead, research focuses on using universal in-memory formats, like Apache Arrow [41]. Li et al. [63] make the case for using in-memory datasets in Database Management Systems to greatly reduce the time needed to export data to Distributed Data Processing Systems. Much like us, they aimed to share data using a universal in-memory dataset format. Our generic computation offloading system could benefit from this work, since SparkHook requires storage layers capable of presenting data in the generic Arrow format. A major difference between this work and our own is that we are interested in designing a generic computation offloading framework for distributed data processing systems, while their work is about providing generic data interfaces.

Computation with Generic Data Interfaces The first part of our connector consists of Arrow-Spark, a framework that provides a generic Arrow Dataset API interface to Spark. The concept of providing a generic data interface for a computation engine is still recent, but already quite popular. Many computation engines are implementing generic data interfaces, and datasource developers are implementing reading strategies in generic data interfaces as well.

When the data is only available from remote datasources, we already have some sort of a separation. Databricks [64], a company providing cloud resources involving Spark, Delta Lake, etc, provides several interesting connectors, such as a JDBC and ODBC connector [65] for Spark. These connectors allows users to query many MySQL-capable backends. Databricks also has a connector to connect to Amazon RedShift resources [66].

Decoupling execution and data ingestion on one cluster is something else, however. While the goal of the aforementioned connectors is to allow users to obtain data from remote services, our goal is to improve performance on the way. Actually, we were unable to find scientific work about decoupling execution and data and experimenting with it. The only things to note here are the projects we used code from and developed upon, from H. Zhang [49] and the bigdata project [50] from Intel corporation. Finally, in Spark+AI Summit 2020, V. Ganesh [67] speaks very briefly about using Arrow for reading data. Of course, they refer to using PySpark (Spark in Python) with Arrow. There is no official support for using Core (Scala) Spark together with Core Arrow (C++).

Computation Offloading with Specialized Hardware Support In this work, the main point is to generalize computation pushdowns, to alleviate the bottleneck present on CPUs inside clusters. Although we are the first to implement a system to pushdown computations in a *general* way, several implementations exist to pushdown some form of computations using specialized hardware support.

In their work, Nonnenmacher et al. [68] explore pushing down operations from Spark into FPGAs, using a JNI bridge to Apache Arrow to cross the bounds of the JVM into native code. Their work is about hardware-specific pushdowns, whereas our work focuses on generic, general-purpose hardware in cluster environments.

Recently, Amazon introduced the Amazon Redshift Advanced Query Accelerator (AQUA) [69] project, which pushes specific reduction and aggregation Redshift computations closer to the data, using specialized proprietary hardware devices. Additionally, S3 introduced S3 Select [21], which allows users to scan files inside S3 for several formats, to improve query performance. Many systems already support reading from S3 Select, such as the compute engines Redshift [70], Spark [3], and Snowflake [4], and DBMSs such as PushdownDB [71]. However, as is common with Infrastructures as a Service (IaaSs) [72], there are no exposed APIs to tune the performance of S3 Select. Also, there is no way to implement support for reading other file formats than the few currently supported. Our work is significantly different, in the fact that it allows general query offloading, without conceding customizability and extensibility of the datasource reading implementation. Additionally, our work is open source, and can be executed without the need of any public cloud infrastructure.

Systems like IBM Netezza [73], PolarDB [22], Ibex [74] and YourSQL [75] depend on sophisticated and costly hardware like FPGAs and Smart SSDs [15, 16], to perform table scanning inside the storage layer. These systems use hardware-software co-design for specific use-cases. Our work is significantly different, as we allow arbitrary computations to be offloaded, as opposed to allowing only table scans by these systems. Additionally, we only require general-purpose hardware instead of sophisticated hardware, making our work usable in a much broader sense than aforementioned related research.

Computation Offloading with Generic Hardware Our system, SparkHook, is able to offload computations using generic hardware, not to be confused with FPGA's, SmartSSDs, specialized proprietary offloading controllers etcetera. Several other systems exist that handle computation offloading without depending on specialized hardware, for remotely comparable contexts.

One of the earliest concepts of computation offloading was by Riedel et al. [76], who introduced the Active Disk system. With the Active Disk system, storage I/O intensive parts of a host application can be offloaded to microcontrollers in harddrives, in order to alleviate the bus interconnect and host processor. Lim et al. [77] expanded on this concept by implementing a distributed filesystem that offloads filesystem functionality, such as file lookups. Another early concept is the Intelligent Disk system [78], which enables disk-to-disk communication without utilizing the host processor and bus interconnect.

An example of a more recent, distributed data processing system-related concept is Rhea [79], which brings filter offloading to MapReduce applications for unstructured and semi-structured data. It uses static analysis to generate executables that apply filters when executed in storage clusters. Our work differs in that we support any operation to be offloaded, as long as the operation is supported by the computation offloading interface and the chosen storage backend. Additionally, we offload computations at runtime, without requiring static code analysis.

8 Conclusion

For the conclusion, we revisit each Research Question one more time, show how we answered them, and summarize the interesting details.

RQ1: How can we design and prototype generic end-to-end computation offloading between Distributed Processing Systems and Programmable Storage? Designing a generic computation offloading system is a complex task. Many design challenges arise when touching concepts such as inter-program, inter-machine, movable computations, offloading decisions, inherent drawback alleviation etcetera. We visited these design challenges, went over all possible solutions, and explained our decisions accurately. We have shown all stages a request flows through in order to offload computations, and get data back to the original requesting compute node.

RQ2: How to assess the performance of SparkHook, our prototype, in a end-to-end fashion? In order to maximize reproducibility, we executed all experiments on open hardware, and implemented our own declarative experimentation framework, data generators, result plotters etcetera. SparkHook requires an environment that would take quite long to setup manually, which is why we implemented and provided many support tools. All of these tools are publicly available. Additionally, to help performance assessment, we provide hardware monitoring support.

RQ3: What is the performance of our prototype end-to-end computation offloading framework? In this work, we have shown the performance of our prototype, SparkHook, extensively. We have shown that generic computation offloading is a viable, efficient solution to alleviate compute cluster CPU utilization. We discussed the influence of data reduction queries and objectsizes on overall performance, and have shown our system scales well with dataset sizes. Finally, we show that our system has a long way to go when compared to vanilla Spark. We discussed several techniques to greatly improve performance to mitigate this issue.

RQ4: What are the lessons learned and how can we make this production-ready, achieving good performance and efficiency? We discussed the potential and inherent drawbacks of computation offloading at length, going deeper in the concept of movable computations and serialization issues that arise. Additionally, we have shown the main performance bottlenecks for SparkHook, and provided our very best ideas to solve the underlying problems and causes. Finally, we discussed our most persistent problems and how we solved them.

In this project, we created an unprecedented prototype system for generic computation offloading between compute and storage clusters. We have shown that our prototype successfully alleviates the CPU utilization of the compute cluster, reducing it by 87.5%. Additionally, our system greatly alleviates network infrastructure, up to 98.0%. This is a manifest to the great potential benefits of intelligent cooperation between compute clusters and storage hardware. Our hope is that this research opens the eyes of the scientific community, to study the exciting concept of intelligent cluster cooperation. Finally, we hope that this research will eventually lead to new datacenter designs, focusing on intelligent cooperation between compute and storage clusters.

Declaration of Originality

During this thesis, I (Sebastiaan) and professor Uta extensively worked together with a team of researchers. This was a most valuable and enjoyable experience. During this thesis, I often refer to team work, since all of my contributions fit into a bigger picture, but many would not make sense if the team work is left unexplained. This team collaboration requires a clear separation between our contributions, and contributions of other members of the team. In this Section, I give additional insights into the team collaboration and contributions, and attribute proper credit.

The Team

This research is a cooperation between Leiden University and University of California Santa Cruz. The composition of the research team is depicted in the table below.

Name	Role	Institution	Country
Sebastiaan Alvarez Rodriguez	MSc student	Leiden University	Netherlands
Alexandru Uta	Thesis supervisor, Assistant Professor	Leiden University	Netherlands
Carlos Maltzahn	Thesis supervisor, Adjunct Professor, Founder & Director of CROSS	University of California Santa Cruz	USA
Ivo Jimenez	Postdoc	University of California Santa Cruz	USA
Jayjeet Chakabroty	IRIS-HEP fellow, student	National Institute of Technology, Dagapur	India
Jeff LeFevre	Adjunct Professor	University of California Santa Cruz	USA
Aaron Chu	Student	University of California Santa Cruz	USA
Jianshen Liu	Student	University of California Santa Cruz	USA

Team Work

As a team, we have done several publications. Here, I list and acknowledge them, and provide attribution details.

The Leiden-based team members (Sebastiaan, Alexandru Uta) are co-authors of *Towards an Arrow-native Storage System* [80]. The main authors of this paper are the UC Santa Cruz team members, led by Jayjeet Chakraborty and Carlos Maltzahn. I refer to parts of this paper in our work. At every location where I do so, a footnote with attribution notices is present.

The Leiden-based team members (me, Alexandru Uta) are the main authors of *Zero-Cost, Arrow-Enabled Data Interface for Apache Spark* [44]. Jayjeet Chakraborty, Carlos Maltzahn, Ivo Jimenez, Jeff Lefevre and Aaron Chu are co-authors of this paper. Since this is my own work, I did not add a footnote with attribution notices wherever I refer to this paper. Instead, I explicitly mention that this is our previous, related work.

Contributions

The end goal of this collaboration, to design and evaluate a system to offload computations from the compute to storage layer, requires a great many steps. Many of the required designs and implementations for such a system have been created by the team, and not by us. Therefore, for improving readability of this manuscript, I have written about designs and implementations that were team contributions. I provide an overview of each chapter of this work, and note which team members contributed (parts of) it.

Abstract, Introduction, Declaration of Originality, Background These Sections do not contain design and implementation details for this work. They were written by Sebastiaan.

RQ1: Design and SparkHook Prototype Subsections 3.1,3.2 contain the high-level design plans for creating a system with the ability to offload computations to storage. Subsection 3.3 contains a concrete design plan for creating such a system, offloading computations from Apache Spark to Ceph Storage Clusters. These design plans were discussed and validated with the entire team and led by Sebastiaan.

Subsections 3.4,3.5,3.6 contain the 'frontend' of our system, SparkHook, which is the part running on the compute cluster. Here, SparkHook obtains computations in Apache Spark, and pushes them down to native code using a Java Native Interface (JNI) bridge. Additionally, these Subsections explain the concept of a computation aggregator, which aggregates (collects) computations in Spark before the actual execution stage starts, and caches them on all nodes transparently. All work found in these Subsections originates from Sebastiaan.

Subsection 3.7 explains the required communication the 'backend' of our system. The backend of the system is the part running on the programmable storage cluster. Jayjeet is the main contributor of work found in this Subsection. Sebastiaan only contributed to this work by building a connection caching system.

Subsections 3.8,3.9 contain the 'backend' of our system, which is the part running on the programmable storage cluster. Jayjeet is the main contributor of work found in these Subsections.

Subsection 3.10 contains the final stage of the frontend, which returns obtained data from the backend to Apache Spark. This work originates from Sebastiaan.

RQ2: Performance, Reproducibility, Measuring for SparkHook The entirety of this Section contains designs and implementations related to performing end-to-end experiments with our system, SparkHook. All described work there originates from Sebastiaan.

RQ3: Evaluation of SparkHook The entirety of this Section contains experiments. All experiments were designed and performed by Sebastiaan. Note that there is one Figure showing results obtained by Jayjeet, namely Figure 16. This Figure shows the read performance of the backend. Sebastiaan performed a similar experiment, displaying overall performance of the distributed data processing application instead, which includes read performance of the backend among other things.

RQ4: Prototype to Production-Ready & Lessons learned Subsection 6.1 explains the double interpretation problem, which is about having to interpret data on multiple machines when moving computations between nodes. The problem description and discussion originate from Sebastiaan. The data for Figure 19 was measured by Jayjeet, and plotted by Sebastiaan. The data for Figure 20 was measured by Arrow developers, and plotted by Sebastiaan. The data for Figure 21 was measured by Arrow developers, and plotted by Sebastiaan.

Subsections 6.2 explains the abstract principle of whether a computation should be pushed down or not, and translates this to reality. All concepts explained here originate from Sebastiaan.

Subsection 6.3 explains possible methods to achieve better load balancing on the programmable storage backend in a decentralized, non-global manner. The majority of ideas explained here originate from Sebastiaan. The concept of 'pushing back' computations when the storage cluster is overburdened originates from the team as a whole.

Subsection 6.4 explains the restrictions of the data shape SparkHook requires (at the time of writing) to properly function. It also explains the pros and cons of this approach. The data shape originates from Jayjeet. The pros and cons were described by Sebastiaan after having a short session about it with Jayjeet, and therefore is a shared contribution between them.

Subsections 6.5,6.6 describe more practical problems experienced by Sebastiaan. The content originates from Sebastiaan.

Related Work, Conclusion These Sections do not contain design and implementation details. They were written by Sebastiaan. Note that for the related work, we reused parts of the related work sections from team publications, making the Related Work a team contribution.

Credits & Thanks

This work would not have been possible at all without our team, whose members have all significantly contributed to this very challenging project. I greatly appreciated and enjoyed working with our team, during debugging, brainstorming, sharing experiences and all other interesting things we did together! I look forward to working together again in the future. Many thanks to the other team members for inviting me to the team!

References

[1] Statista. *Amount of data created worldwide.* https://www.statista.com/statistics/871513/worldwide-data-created/. Accessed:2020-10-08. Statista.

[2] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th python in science conference.* Vol. 126. Citeseer. 2015.

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[4] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. "The snowflake elastic data warehouse". In: *Proceedings of the 2016 International Conference on Management of Data.* 2016, pp. 215–226.

[5] Ezz El-Din Hemdan and DH Manjaiah. "Spark-based log data analysis for reconstruction of cybercrime events in cloud environment". In: *2017 International Conference on Circuit, Power and Computing Technologies (ICCPCT).* IEEE. 2017, pp. 1–8.

[6] Naman Shah. "Determining disease outbreak influence from voluminous epidemiology data on enhanced distributed graph-parallel system". In: *2000-2019-CSU Theses and Dissertations* (2017).

[7] Manar A Elmeiligy, Ali I El Desouky, and Sally M Elghamrawy. "A Multi-Dimensional Big Data Storing System for Generated COVID-19 Large-Scale Data using Apache Spark". In: *arXiv preprint arXiv:2005.05036* (2020).

[8] CERN. *CERN Computing.* https://home.cern/science/computing. Accessed:2021-07-03. CERN.

[9] Laura-Diana Radu. "Green cloud computing: A literature survey". In: *Symmetry* 9.12 (2017), p. 295.

[10] Apache Software Foundation. *Hadoop.* Version 0.20.2. Feb. 19, 2010. URL: https://hadoop.apache.org.

[11] 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).* Ieee. 2010, pp. 1–10.

[12] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. "Network support for resource disaggregation in next-generation datacenters". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* 2013, pp. 1–7.

[13] Amazon. *Amazon S3.* https://aws.amazon.com/s3/. Accessed: 2020-11-16. Amazon.

[14] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The datacenter as a computer: Designing warehouse-scale machines". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189.

[15] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. "Query processing on smart ssds: Opportunities and challenges". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 1221–1230.

[16] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. "SmartSSD: FPGA accelerated near-storage data analytics on SSD". In: *IEEE Computer architecture letters* 19.2 (2020), pp. 110–113.

[17] Jeff LeFevre. *Skyhook Data Management*. https://skyhookdm.com/. Aug. 2020.

[18] 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)*. 2015, pp. 293–307.

[19] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. "Albis: High-performance file format for big data systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 615–630.

[20] Mark T Bohr and Ian A Young. "CMOS scaling trends and beyond". In: *IEEE Micro* 37.6 (2017), pp. 20–29.

[21] Amazon Randall Hunt. *S3 Select and Glacier Select – Retrieving Subsets of Objects*. https://aws.amazon.com/blogs/aws/s3-glacier-select/. Nov. 2017.

[22] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. "{POLARDB} meets computational storage: Efficiently support analytical workloads in cloud-native relational database". In: *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 2020, pp. 29–41.

[23] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. ISBN: 1931971471.

[24] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[25] Muhammad Hussain Iqbal and Tariq Rahim Soomro. "Big data analysis: Apache storm perspective". In: *International journal of computer trends and technology* 19.1 (2015), pp. 9–14.

[26] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar. "KAFKA: The modern platform for data management and analysis in big data domain". In: *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*. 2017, pp. 1–5. DOI: 10.1109/TEL-NET.2017.8343593.

[27] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. "IBM streams processing language: Analyzing big data in motion". In: *IBM Journal of Research and Development* 57.3/4 (2013), pp. 7–1.

[28] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. “Ray: A Distributed Framework for Emerging AI Applications”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 9781931971478.

[29] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. “Graphx: A resilient distributed graph system on spark”. In: *First international workshop on graph data management experiences and systems*. 2013, pp. 1–6.

[30] 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*. 2015, pp. 1383–1394.

[31] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. “Mllib: Machine learning in apache spark”. In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[32] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. “Benchmarking streaming computation engines: Storm, flink and spark streaming”. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, pp. 1789–1792.

[33] Deepak Vohra. “Apache avro”. In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 303–323.

[34] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. “Column-stores vs. row-stores: how different are they really?” In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 967–980.

[35] Peter A Boncz, Martin L Kersten, and Stefan Manegold. “Breaking the memory wall in MonetDB”. In: *Communications of the ACM* 51.12 (2008), pp. 77–85.

[36] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. “Monetdb: Two decades of research in column-oriented database”. In: *IEEE Data Engineering Bulletin* (2012).

[37] Deepak Vohra. “Apache parquet”. In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.

[38] Apache. *Apache ORC - High-Performance Columnar Storage for Hadoop*. https://orc.apache.org/. Accessed: 2020-11-06. Apache Software Foundation.

[39] Martin Odersky. *The Scala Programming Language*. https://www.scala-lang.org/. Accessed: 2020-11-08. Programming Methods Laboratory of École polytechnique fédérale de Lausanne.

[40] Matei Zaharia, Reynold Xin, Xiao Li, Fan Wenchen, and Yin Huai. *Introducing Apache Spark 3.0*. https://databricks.com/blog/2020/06/18/introducing-apache-spark-3-0-now-available-in-databricks-runtime-7-0.html. Accessed:2020-10-16. Databricks.

[41] Arrow Development Team. *Apache Arrow*. https://arrow.apache.org. Oct. 2018.

[42] Arrow Development Team. https://arrow.apache.org/docs/python/dataset.html#reading-from-cloud-storage. Accessed: 2020-11-08. Apache Software Foundation.

[43] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. "RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters". In: *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*. PDSW '07. Reno, Nevada: Association for Computing Machinery, 2007, pp. 35–44. ISBN: 9781595938992. DOI: 10.1145/1374596.1374606. URL: https://doi.org/10.1145/1374596.1374606.

[44] Sebastiaan Alvarez Rodriguez, Jayjeet Chakraborty, Aaron Chu, Ivo Jimenez, Jeff LeFevre, Carlos Maltzahn, and Alexandru Uta. *Zero-Cost, Arrow-Enabled Data Interface for Apache Spark*. 2021. arXiv: 2106.13020 [cs.DC].

[45] Arrow Development Team. *Serialization and Interprocess Communication (IPC)*. https://arrow.apache.org/docs/format/Columnar.html#serialization-and-interprocess-communication-ipc. June 2021.

[46] Google LLC. *Protocol Buffers - Google's data interchange format*. https://github.com/protocolbuffers/protobuf. Accessed: 2021-06-27. Google LLC.

[47] Arrow Development Team. *Apache Arrow Dataset API Java implementation*. https://arrow.apache.org/docs/java/. Oct. 2018.

[48] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for High Performance and Scientific Computing* 14.9 (2011), pp. 1–9.

[49] Hongze Zhang. *ARROW-7808: [Java][Dataset] Implement Dataset Java API by JNI to C++*. https://github.com/zhztheplayer/arrow-1/tree/ARROW-7808. Accessed: 2020-09-14. Github, Apache Arrow community.

[50] Intel Corporation. *Optimized Analytics Package for Spark Platform (OAP)*. https://github.com/Intel-bigdata/OAP. Accessed: 2021-01-15. Github, Intel Corporation.

[51] Dhruba Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[52] HDFS Block Size. http://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Apache Software Foundation.

[53] Guido Van Rossum et al. *Python*. 1991.

[54] The pip developers. *PIP Installs Packages (Python3-PIP)*. https://pypi.org/project/pip/. Accessed:2020-10-16. PSF.

[55] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. "GENI: A federated testbed for innovative network experiments". In: *Computer Networks* 61 (2014), pp. 5–23.

[56] Randy L Phelps. "National Science Foundation". In: (2015).

[57] alfredodeza et al. *remoto*. https://github.com/alfredodeza/remoto. Accessed: 2021-06-27. Github.

[58] Prometheus contributors. *Prometheus - Monitoring system & time series database*. https://prometheus.io/. [Online; accessed 11-June-2021]. 2021.

[59] Grafana Labs. *Grafana — The open observability platform*. https://grafana.com. [Online; accessed 11-June-2021]. 2021.

[60] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. “Is big data performance reproducible in modern cloud networks?” In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 2020, pp. 513–527.

[61] *NYC Yellow Taxi Trip Record Data*. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[62] Eric W Weisstein. “Simpson’s rule”. In: *https://mathworld. wolfram. com/* (2003).

[63] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. “Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats”. In: *arXiv preprint arXiv:2004.14471* (2020).

[64] Databricks. *Databricks*. https://databricks.com/. Accessed: 2021-02-01. Databricks.

[65] Databricks. *Databricks*. https://docs.databricks.com/data/data-sources/sql-databases.html. Accessed: 2021-02-01. Databricks.

[66] Databricks. *Databricks*. https://github.com/databricks/spark-redshift. Accessed: 2021-02-01. Databricks, Github.

[67] Vinoo Ganesh. *The Apache Spark File Format Ecosystem*. https://databricks.com/session_na20/the-apache-spark-file-format-ecosystem. Accessed: 2021-02-01. Apache, VeraSet.

[68] FM Nonnenmacher. “Transparently Accelerating Spark SQL Code on Computing Hardware”. In: (2020).

[69] Amazon. *Amazon AQUA*. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/. Accessed: 2021-07-09. Amazon.

[70] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. “Amazon redshift and the case for simpler data warehouses”. In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1917–1923.

[71] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. “PushdownDB: Accelerating a DBMS using S3 computation”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1802–1805.

[72] Rafael Moreno-Vozmediano, Rubén S. Montero, and Ignacio M. Llorente. “IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures”. In: *Computer* 45.12 (2012), pp. 65–72. DOI: 10.1109/MC.2012.76.

[73] Malcolm Singh and Ben Leonhardi. “Introduction to the IBM Netezza warehouse appliance”. In: *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. 2011, pp. 385–386.

[74] Louis Woods, Zsolt István, and Gustavo Alonso. “Ibex: An intelligent storage engine with support for advanced sql offloading”. In: *Proceedings of the VLDB Endowment* 7.11 (2014), pp. 963–974.

[75] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. “YourSQL: a high-performance database system leveraging in-storage computing”. In: *Proceedings of the VLDB Endowment* 9.12 (2016), pp. 924–935.

[76] Erik Riedel, Garth Gibson, and Christos Faloutsos. “Active storage for large-scale data mining and multimedia applications”. In: *Proceedings of 24th Conference on Very Large Databases*. Citeseer. 1998, pp. 62–73.

[77] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David H-C Du. “Active disk file system: A distributed, scalable file system”. In: *2001 Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*. IEEE. 2001, pp. 101–101.

[78] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. “A case for intelligent disks (IDISKs)”. In: *Acm Sigmod Record* 27.3 (1998), pp. 42–52.

[79] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony Rowstron. “Rhea: automatic filtering for unstructured cloud storage”. In: *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 343–355.

[80] Jayjeet Chakraborty, Ivo Jimenez, Sebastiaan Alvarez Rodriguez, Alexandru Uta, Jeff LeFevre, and Carlos Maltzahn. “Towards an Arrow-native Storage System”. In: *arXiv preprint arXiv:2105.09894* (2021).