A Distributed Data Exchange Engine for Polystores

Abdulrahman Kaitoua, Tilmann Rabl, Volker Markl

Abstract:
There is an increasing interest in fusing data from heterogeneous sources. Combining data sources increases the utility of existing datasets, generating new information and creating services of higher quality. A central issue in working with heterogeneous sources is data migration: In order to share and process data in different engines, resource intensive and complex movements and transformations between computing engines, services, and stores are necessary.
Muses is a distributed, high-performance data migration engine that is able to interconnect distributed data stores by forwarding, transforming, repartitioning, or broadcasting data among distributed engines' instances in a resource-, cost-, and performance-adaptive manner. As such, it performs seamless information sharing across all participating resources in a standard, modular manner. We show an overall improvement of 30% for pipelining jobs across multiple engines, even when we count the overhead of Muses in the execution time. This performance gain implies that Muses can be used to optimise large pipelines that leverage multiple engines.

ACM CCS: Information systems → Data management systems → Middleware for databases

Keywords: Distributed systems, data migration, data transformation, big data engine, data integration.

1 Introduction

Polystores are designed to abstract applications from the underlying execution platform while harvesting the performance from cross-engines execution. Modern polystores [6, 22, 16] have four layers of abstraction: the language layer, the logical representation layer, the physical implementation layer, and the data storage layer. In this paper, we concentrate on the fourth layer and its connection with the third layer. Polystore consists of a set of heterogeneous data processing engines. An essential element of the storage layer management is the data migration among the various data processing engines. Thus, a crucial concern in cross-engine execution is the *data migration overhead*.

The key factors when deploying an application on polystore instead of a single processing engine are the performance gain from the application operations on polystore engines and the communication overhead introduced between the engines. Polystores are efficient when the performance gain by utilizing various engines overcome the cost of network communication.

An application in distributed systems consists of a set of logical operations organized in a graph. Each operation consists of one or more physical implementation tasks. The tasks are either computation-intensive or communication intensive. For example, a join query (an operation) in a distributed system consists of shuffle and merge tasks. The current polystore platforms consider execution engine and storage as separated layers [6], which makes sense for the simplicity of the design, but not for the performance. Executing the mentioned example would first migrate the data to the executing engines then performs the application first task, which is a shuffle task, resulting in new data movement. Polystore optimizers should be aware of the application tasks and have the flexibility to push shuffle logic to the data migration layer to optimize data migration.

Most of the current polystore engines consider single node stores in the architecture of the polystore, such as PostgreSQL and Neo4j. Efficient data migration between distributed engines, such as Apache Spark [14] or Apache Flink [13], is difficult due to numerous reasons. First, the topology of application deployment is determined at

run-time by the resource manager. Second, engines have different data production and consumption rates (varied based on the application tasks). Third, the execution engines are different in the number of input (m) and output (n) threads, which makes it hard to balance the data among engines, for example, moving data from n workers in a Flink cluster to m workers of a Spark cluster. Fourth, in cross-engine execution, many-to-many connections among engines are needed for efficient data migration. In a many-to-many connection, two distributed engines could send data to a single distributed engine, for example. Fifth, the polystore should not modify the source code of the underlying engines. Sixth, additional complexity comes from engine execution types; some engines are native streaming engines, and others are batch processing engines. These reasons make using a simple data migration approach based on direct socket connections or manual data migration not feasible.

Most of the polystore execution projects (e.g., [6, 22, 16]) have built prototypes with the data migration either hardcoded as a simple one-to-one connection [4], simplified as shared storage (HDFS [17]), or performed as manual data transformation, repartitioning, and migration [5]. We believe that these strategies for data migration are not suitable for distributed engines where the applications drive the deployment architecture.

In this paper, we discuss the architecture of Muses, a data migration engine for distributed systems. *Our contributions* can be summarized as follows:

1. *Muses is reactive to changes of the topology of the engine instances*: Distributed engines such as Apache Spark and Apache Flink, assign the application tasks to the cluster nodes at run-time based on the resource manager. Thus, the application deployment topology differs based on resource availability. Muses reacts to the application topology changes.
2. *Muses links heterogeneous data sources*: A data export/import in a single stream, multiple streams on a single machine (e.g., PostgreSQL with n threads for data export/import), or multiple streams on a cluster of machines (e.g., Apache Spark). To the best of our knowledge, the last case has not been solved by any previous system. Muses can connect all these types of data stores.
3. *Muses provides a rich platform for cross-engine execution layers*. To enable smart integration between the data migration layer (Muses) and a cross-engine execution layer, Muses provides the ability to perform data shuffling or broadcasting *during* the first data migration from a source engine to a destination engine. Muses also efficiently connects stream and batch processing engines.
4. *It is easy to connect engines to Muses*: Only a single Connector to Muses is needed to join Muses, as described in Section 3.2.2. No change to the source code of the engine is required.

This paper is an extended version of a short paper published in ICDE 2019 [21]. The rest of the paper is organised as follows: Section 2 discusses a motivating example, a spatial join in genomics, which utilises the cross-engine execution. Section 5 shows the state of art in data migration for polystores. Section 3 describes the architecture of Muses in detail. Section 4, shows the performance of the distributed data migration. Finally, Section 6 concludes the paper for future work.

2 Motivating Example

Recent advancements in genomic materials reading techniques, a.k.a., Next Generation Sequencing (NGS), make reading genomic materials faster and cheaper, which leads to a vast increase in the generation of genomic data. Several methods are used to extract signals from the genomic data that associate a region (interval) of the genome with some interesting information - such as a mutation or a peak of expression [15]. Each interval (record) in a sample (file) has a set of attribute/value associated with each interval, where the files are tab-delimited.

Genomic applications integrate several data types from different storage technologies. The GenoMetric Query Language (GMQL) [18] was developed to perform queries on heterogeneous genomic data. GMQL is a SQL-like language to query region-based genomic data. The complexity of a GMQL application comes from integrating heterogeneous data sources and performing operations with various complexities. GMQL contains operations on both the data of regions, DNA intervals with a start and a stop position representing regions, and the meta-data that describes the genomic data (clinical data). A single GMQL script can contain aggregation operations, meta operations, and domain-specific operations.

Our polystore deployment is shown in Figure 1 and contains two types of connections between engines. The connection between the SciDB cluster and the Spark cluster is **many-to-many**, and the connection between Spark/SciDB and PostgreSQL is **many-to-one**. In this paper, we will consider the GMQL execution engine as a running example for polystores. The reason for choosing this ploystore deployment is based on previous studies. The clinical data (metadata) is small in size relative to genomic data (region data), and it is stored in a PostgreSQL database. Previous work has shown that GMQL operations on metadata perform well on a PostgreSQL engine in comparison to other engines such as Spark and SciDB. Cattani et. al. [25] showed a performance divergence between implementations of GMQL commands in SciDB [2] (a multi-dimensional scientific database) and Apache Spark [14]. Apache Spark exhibited high performance for complex genomic operations using UDFs, while SciDB is an order of magnitude faster than Spark for aggregations. In some operations like region histograms, SciDB performed almost 12 times faster. The challenges in a polystore of both SciDB and Apache Spark for implementing GMQL engine are: 1. Independent storage units that make it hard to move the data

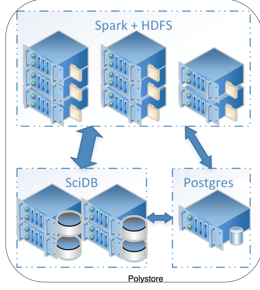

Figure 1: A polystore consisting of three engines, two of which are distributed engines.

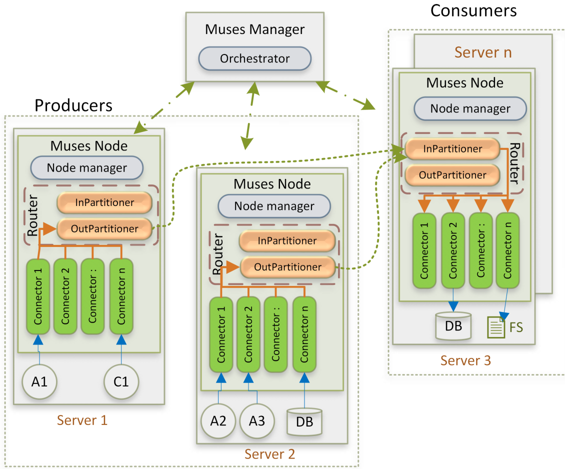

Figure 2: Muses general architecture.

between tasks running on the two engines. 2. Different physical and logical data representations. 3. A different number of engine instances, as shown in Figure 1.

3 Muses Architecture

Muses is designed to connect distributed data engines for cross-engine execution. Figure 2 shows the general architecture of Muses. Muses consists of a single **Muses Manager Node** and one or more **Muses Nodes**, which are described in detail in Sections 3.1 and 3.2, respectively. In Muses, two types of data are exchanged; *control data* and *application data streams*. The *Muses manager* sends configurations (control data) to node managers and receives *profiling information*. While the Muses nodes exchange only the data streams.

Each data migration job in Muses contains a logical data *Producer* and a data *Consumer*. We refer to engines as sources or sinks; one or more sources form a *Producer*, and one or more sinks form a *Consumer*. Sinks and sources might be distributed engines. We denote the output threads of a source as *source instances* and the sink input threads as *sink instances*. For simplicity of the description of Muses architecture in Figure 2, Muses nodes are logically either marked as producer or consumer nodes. Since each machine can host several sinks and sources, a Muses node can be a producer and a consumer node at the same time.

We represent sources and sinks in Figure 2 as either distributed sources/sinks or local sources/sinks. Engine 'A' is shown as a distributed source and has three instances ('A1', 'A2', and 'A3') distributed on two machines (Server 1 and Server 2). Engine 'C' has one instance 'C1' on Server 1 while engine 'DB' is a database instance on Server 2. The Muses manager node should be aware of the topology of the data engines. When allocating the instances of a data engine dynamically, the deployment topology must be passed to the Muses job configurations at runtime.

In the following, we discuss the details of the Muses architecture, its components, the resource management, and the distributed pipelines.

3.1 Muses Manager

The *Muses Manager* registers new sources/sinks, distributes job configuration on nodes, monitors job execution, and collects profiling information.

The manager registers sources/sinks before running any data migration operation. The registration process requires the user to provide a list of the machines hosting the source/sink, and the Connectors (described in Section 3.2.2) of the registered source/sink.

The manager controls the data migration and transformation between S sources and K sinks. The user, or the cross-engine execution dispatcher, submits the migration job to *Muses Manager* along with the *job configuration*, which includes:

- The producers and consumers for the migration job,
- the data distribution operation between the producer and the consumer,
- and the engines' topology, in case it has been dynamically set.

The manager configures Muses Nodes based on the topology, resource availability, and the shuffling method of the migration job. The manager sends the task configuration to each Node involved in the job execution, then executes the migration. The *task configuration* includes:

- The number of Connector instances for each source/sink on the machine, which is equal to the number of source/sink instances on that machine.
- The number of expected input streams for each sink instance, which depends on the number of sources instances and the data distribution strategy.
- The data distribution procedure for Muses Node routers with a list of destination (sinks) machines addresses.

3.2 Muses Node

Figure 2 shows *Muses Nodes*, which consists of a set of Connectors (P_n), consumer and producer routers, and a *Node manager*. The Node Manager is the only process

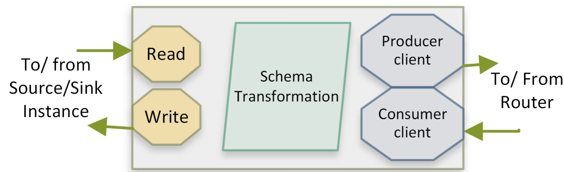

Figure 3: Muses Connector: Read, is the read operation. Write, is the write operation. Producer client, is the Connector to the producer router. Consumer client, is the Connector to the consumer router.

that keeps running on each node for Muses engine. At the same time, Muses routers and connectors are initialized on-demand at the job configuration phase by the Node Manager, which reduces the overall overhead of Muses.

For efficient data migration, Muses creates direct connections from the source machine to the destination machines without being mediated by the Muses Manager node. Muses establishes the connection as follows: First, the *Muses Manager* sends the job configurations to the *Node Managers*. Second, the Node Manager instantiates the Connectors and Muses routers on both the source and sink machines. Third, The router gets the data distribution procedure from the node manager. At this point the connection is established from the *source connectors* through the *producer routers* to the *consumers routers* and finally to the sink instances. In the end, Muses Manager triggers the sources and sinks executions.

3.2.1 Node Manager

The *Node Manager* is responsible for keeping a connection with Muses Manager for receiving job configurations and sending status and aggregated streams profiling information through heartbeats messages.

The *Node Manager* initialises the routers for the job by setting up consumer/producer stream. The *Node Manager* also manages resources reserved by the routers and connectors. The *Node Manager* is responsible for restoring the stream in case of nodes failure.

3.2.2 Connector

Intermediate data representation (Apache Arrow [11]) is used to facilitate connecting several engines with minimal coding requirements. The intermediate data representation connects a set of data engines with only two data transformations (from and to the intermediate data representation). More information about data transformation is mentioned in Section 3.4. We discuss the overhead of using an intermediate data format in Section 4.

Figure 3 shows the logical representation of the Muses Connector in details, the position in the complete architecture is illustrated in Figure 2. The Connector consists of a *Read* operator that reads from the source and translates the data into the Apache Arrow data structure, a *Write* operator that reads from an Arrow data structure and writes to the data sink, and a socket client to the routing module (described in Section 3.2.3).

3.2.3 Routers

The *Router* is an Akka [12] streaming graph that controls the flow of streams in and out of a Muses Node. The Router controls the connector instances. The Connectors share the same pool of threads on the machine. The number of threads is set by the node manager relative to the source/sink performance and the resources available on the hosting machine.

When the node has only one Consumer instance, we configure the Consumer Router as a *forward router* to optimise the link. When consumer engine instances vary in their consumption rate, we perform *stream balancing* as the default in case the job manager does not set a specific shuffling mechanism. Stream balancing uses an Akka *Balancing Router*. The *Balance Router* adjusts the stream when all the Connectors in the same node have the same buffer. Then the *Balancing Router* redistributes the load from busy consumer instances to idle consumer instances.

Muses can use either the same or different data distribution procedures (shuffle function) on the consumer and producer routers for the migration job. For example, when performing a data shuffling operation, the same shuffle function should be on both sides of the stream because the producer and consumer routers on the source node and the sink nodes should share the same shuffle mechanism. Different distribution procedures are used when the producer balances data to a set of sink nodes, and the consumer routers on the sink nodes use a forward routing to forward the data to the local instances without performing any distribution.

The *Consumer Routers* and *Producer Routers* start and stop the Connector instances based on the data streams tags and the job configurations. The end of a stream is determined by stream tags sent at the beginning and end of a stream.

In addition to data migration, the Router aggregates profiling information and sends it to the Node Manager. Profiling information includes amounts of data read or written from or to any data source or sink. This profiling information is useful for the cross-engine-execution optimiser.

3.3 Distributed Pipeline

Muses streams data between data stores to minimize the data spilled on hard disks. The most straightforward pipeline between two distributed data stores (multi-threaded), when the number of source's threads equals the number of sink's threads, is to connect each thread from the source to respective thread in the sink using a socket connection [4]. The above mentioned simple pipeline would perform the best under the following assumptions: The number of export and import threads in both data stores is identical. The data throughput of the source threads is unified on all the producer threads. The consumer's threads can keep up with the producers' throughput. There is no

need for physical schema change. Moreover, there is no need for shuffling or changing the logical format of the data between the source and the sink data stores.

In real-life examples, the above assumptions do not necessarily hold. Thus, Muses streams data from the source data store of N exporting threads to the sink data store of M importing threads, considering the following into account: The number of producer instances and consumer instances, the producer throughput, the speed of the consumer, the schema changes, and the shuffling process.

To solve the general data migration problem in Muses, we define a stream by a start and end tags. In case the number of threads of the consumer does not match the number of threads in the producer, we perform stream fan-out at the producer routers and streams fan-in at the consumer routers to balance the data on the consumers and producers. The data received at the consumer router does not necessarily come from a single data store.

As a fan-out operator creates several streams with start and end tags, the number of output streams is dependent on the distributing operation (balance routing, key-based shuffle, or broadcasting). A fan-in operator (consumer router) will receive several streams and will not close the sink stream until it gets all the end tags of all the streams it is configured to receive. The Manager configures the consumer routers with the number of expected streams for each sink instance. For example, the number of streams generated from a balance producer router will be the same as the number of nodes running the sink instances. In contrast, the number of streams the consumer router would receive is equal to the number of nodes hosting the source data store instances.

3.4 Data Transformation

Data transformation is the main challenge in connecting engines for polystores. There are two types of schema transformations between distributed systems, *the logical and physical schema* transformations. The *logical schema*, for example, represents a row-based, column-based, key-value, or array-based data layout. Therefore, a transformation example would be to transform a row data representation into a key-value schema. We select from the row-based data format the attributes that represent the key and the attributes that represent the value; this step represents the logical transformation. *The physical schema* represents the data types; for example, an unsigned 64-bit integer. Data types might differ from the source to the sink engine. The unsigned-integer at the source might need to be transformed by Muses into a Long as sink data type.

For Muses to be a general approach for connecting different distributed engines, we decided to translate all engine data format to an intermediate data format. The Muses Connector, Section 3.2.2, is the main block in the data transformation mechanism. All newly added engines need to define the Connector functions. Thus, describe the physical transformation between the engine types and the standard data types.

Muses streams support Muses Connectors for solving the data transformation between distributed engines. A Muses stream consists of several Arrow streams and the Muses schema, as described in Section 3.3. Muses schema contains the stream ID and extra information about the data set in the original logical format, such as the source logical schema, primary keys, secondary keys, the dimensions for array databases, and the keys and values for key-value databases.

The receiving Connector reads the stream schemata from the first batch, maps the Arrow data types to the consumer engine data types using Arrow schema, then constructs the receiver's logical schema from Arrow data using Muses Schema.

3.5 Resource Management of Engines Under Shared Resources

The Muses data streaming strategy between engines takes into account the shared resources between the sources and sinks, which is essential for performance measures. Some systems, such as databases, keep daemon processes always running as background processes, while other systems, such as Flink and Spark, reserve resources only on demand. Running two applications (source and sink application) on the same machine (or sharing resources) can result in a resource-draining problem and lead to the CPU context switching problem, which will degrade the overall performance.

For example, Apache Hadoop Yarn [1] is considered as a distributed operating system that allows distributed applications to reserve resources for execution. Flink, Spark, and other data flow engines can run on top of Yarn at the same time (if free resources are available). When we pipeline data from Spark to Flink (or any other Yarn application) in a Yarn setup, they have to share resources. Let's consider the Yarn cluster provides 100 slots (resource units) for processing; we have three ways to schedule the data migration along with managing the resources running. As a first option, we can run the Spark application with all the resources (100 slots), which will allow higher parallelism until Spark finishes execution, spilling the data on HDFS. Then run Flink with input as the distributed data on HDFS from the previous Spark run. Finally, spill the data to HDFS. HDFS will distribute, replicate the data on cluster nodes, which is efficient for the next engine run. The second option is to split the resources into half and run Spark on 50 slots and Flink on 50 Slots, then stream the data between the two engines. The third option is to profile the two applications, the producer application running on Spark and the consumer application running on Flink, then assign a percentage of the resource to each of the applications based on the computational needs for each stage. In this option, we stream the data between the engines for better performance.

The Muses Manager decides to choose one of the three options above based on profiling information received in the job parameters from higher-level optimizer. We call this *physical deployment optimization*. Muses can orchestrate all the three options since it considers HDFS as a distributed source/sink for other data engines (when a Connector for HDFS is available).

3.6 Lightweight Implementation - Performance Efficiency

To make Muses lightweight, we built Muses as a reactive application, which is resilient, interactive, scalable, and event-driven. Muses is built on top of Akka [12], which provides high-level and straightforward abstractions for distribution, concurrency, and parallelism. Akka is an asynchronous, non-blocking, and highly performant message-driven programming model.

Since Akka is event-based, all actors only get activated if data are in the sending queue of the actor, it does not use blocking threads for communication and data transfer. For efficient implementation, in cases where the source and the sink are on the same machine, Muses routes the data internally by message passing between actors referencing the same in-memory Arrow structure. Muses also uses Akka streaming between nodes to optimize the data transfer and provides back-pressure to optimize IO communication.

4 Evaluation

In this section, we perform three sets of experiments: an end-to-end experiment based on the motivating example, a simple profiling for data migration techniques, and a specific experiment to show Muses performance in connecting distributed systems.

The data set is collected from the Encode genomics repository [15]. The Encode data is a tab-delimited text with ten columns. The schema of the data has two Long fields, two String fields, one Char field, and four Double fields. We built 9 data sets of different sizes to show the performance with increasing data sizes. Our experimentation platform is a cluster comprised of machines containing Intel $E5620$ processors with eight hyper-threads and $32GB$ memory.

4.1 Single Engine Solution Versus Polystores for Genomic Application

In this section, we discuss the comparison between two deployments of the motivation example, discussed in Section 2, on a cluster of eight machines: the first deployment is the single system deployment, and the second deployment is a polystore deployment. We have an application of three parts to run on three engines; Apache Spark, SciDB, and PostgreSQL. In the single system deployment, Apache Spark uses all the eight machines resources; once spark applicaiton finishes, PostgreSQL and SciDB share the eight machines. For the polystore deployment, the three engines share the eight machines at the same time. Apache Spark is installed on the eight machines while 4 out of those eight machines host a SciDB installation, and one machine hosts Postgres, all running.

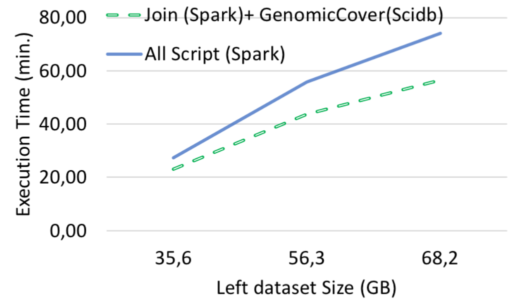

Figure 4: GMQL Script execution.

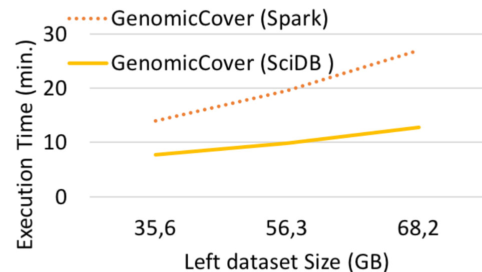

Figure 5: GenomicCover performance over different platforms.

Figure 4 shows the performance of both the single system and polystore solution. In the polystore solution, we run only a part of the execution plan on SciDB, which is mainly the GenomicCover operator. We show the performance gain from running the GenomicCover operator over SciDB in Figure 5.

The automated distributed data migration using Muses allows overlapping the engines' executions and hides the overhead of the data migration between the engines. As soon as Apache Spark generates the first output sample, it is streamed to SciDB nodes and imported into SciDB so SciDB can start execution even though Spark did not finish the whole job execution. For a more in-depth look inside the GenometricJoin operator, we use GenometricJoin as an optimal example to link distributed systems in Section 4.3.

4.2 Data migration profiling

We designed our experiments to show the overhead of using intermediate data representation. Our goal is not to replicate the work of Haynes et al. [4] but to extend it, so we did not cover all the data representations experiments discussed in this work. The authors in [4] shows that Apache Arrow is the best performing intermediate

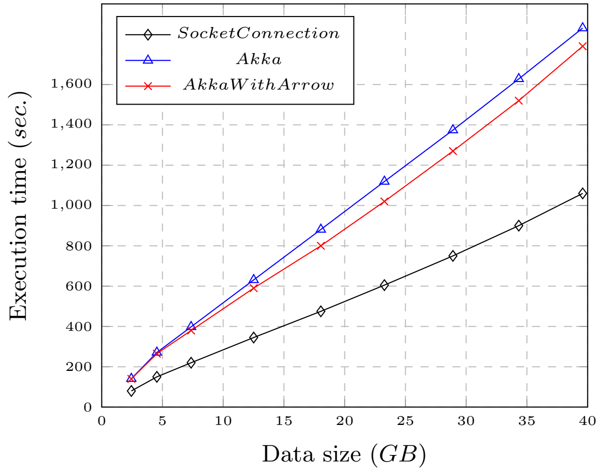

Figure 6: Three data migration scenarios: A socket connection is a direct connection between two engines. An Akka pipeline with data transformation during migration. An Akka pipeline with data transformation to intermediate Arrow stream representation.

representation, thus we concentrated on using Apache Arrow in our experiments.

Exporting/importing data throughput is relative to the architecture of the engine. Some engines have a single thread for importing and another thread for exporting; others have multiple threads for importing and exporting. In addition to that, different engines have different processing throughput that affects the importing and exporting stream. Therefore, we exclude the data export and load execution time in our experiments. We concentrate on the data migration process, but we discuss the effect of the export and import parallelism.

The socket connection in Figure 6 exports the data from the source, transfers the data to the sink engine machine, and finally load the data into the sink engine. In this scenario, *no data is parsed* but only a stream of bytes. Though this scenario is the simplest with the highest performance, it needs both engines to be almost identical.

Migrating data between two engines with different data types requires data translation process. In the Akka scenario, Figure 6, the data translation operation is used to translate the data types from the source schema types to the destination schema types. The drawback of this translation is that it is connection specific. If one of the two engines changes its data schema, the whole connection becomes invalid. The performance of the data migration with data transformation drops because of the data transformation of data records to the destination engine types. Increasing process parallelism can avoid data transformation overhead.

By using Arrow, we abstract the engines internal schema from the migrated data schema. For our experiments, the tab-delimited files are translated to an Arrow structure and then back from Arrow to text. Even though converting data to/from the intermediate data representation produces overhead, Muses performances similar to the second scenario because Arrow columnar structure

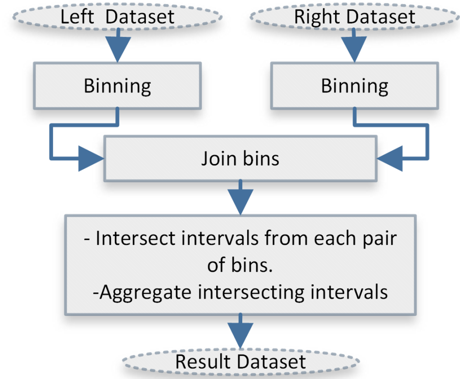

Figure 7: Genomic Spatial Join sub-tasks.

reduced the data transferred on the network.

4.3 Linking Distributed systems

To present Muses performance in migrating data between two distributed engines, we use the Genomic Spatial join operation from our genomic application. We run our experiments with the Genomic Spatial Join application under three deployments, illustrated in Figure 8. The first deployment is a **single cluster** of six machines. The second deployment consists of **two clusters with Muses** as the connecting medium between the cluster. The third deployment consists of **two clusters with distcp** to connect the clusters. Distcp is an Apache Hadoop MapReduce tool for *distributed data copy* between HDFS storages [3]. Clusters A and B, Figure 8, are independent clusters in storage and processing.

We use two datasets of Encode data [15] in the join experiment: *376 million* intervals in the left dataset and around *47 Billion* intervals in the right dataset. The total number of intervals is 70GB (7500 files).

4.3.1 Genomic Spatial Join

We perform the join operation in a polystore that utilizes Muses. In order to focus on the performance of Muses, we set the execution engines to Apache Spark in our polystore. We created a custom input format (extends *CustomInputFormat* class) and a custom output format (extends *CustomOutputFormat* class) as our Muses Connector (Section 3.2.2).

Genomic Spatial Join is considered a critical operation in genomic applications. It combines the genomic data from independent sources (stored on different engines) and joins thousands of samples of genomic data (hundreds of billions of records). It is, therefore, essential to perform this operation efficiently.

Genomic spatial joins differ from relational joins in the fact that joining an interval (record) from the left data set with another interval (record) from the right data set is not based on equality but on intervals (start and stop) intersection [20].

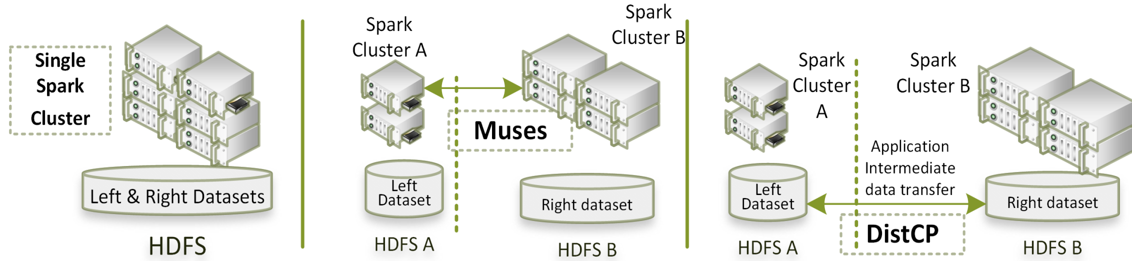

Figure 8: Deployments of the Genomic Spatial Join application.

Figure 9: Execution time of the Genomic Spatial Join application.

The genomic spatial join consists of three sub-tasks; data binning, joining (find the intersecting intervals), and finally aggregating the values of the intersected intervals, as shown in Figure 7. We bin the data to reduce the complexity of the join operation, by partitioning the data into buckets (bins); where the intervals crossing a bucket border are replicated to both adjacent bins [18, 19]. Then pairs of bins with the same bin number from the two datasets are selected. These pairs of bins are searched for intersecting intervals. The intersection between two intervals is defined as follows: the left end of the left dataset interval is less than or equal to the right end of the right dataset interval, and the left end of the right dataset interval is less than or equal the right end of the left dataset interval. Finally, an aggregation function is applied on the values of the right dataset intervals that intersect with the same interval from the left dataset.

4.3.2 Results

In Figure 9, we show the performance of the three deployments described in Figure 8: The first scenario is a deployment of the Genomic Spatial Join application on a *Single Spark Cluster* of six machines where both the left and the right data sets are located on the cluster HDFS. The first scenario, a single cluster, is the ideal case and is only an *ideal baseline* to show Muses overhead.

For the second and the third deployments, we use two clusters in the deployment. Cluster A contains two machines of Spark with its own HDFS installation and cluster B contains four machines of Spark with its own HDFS installation. We chose to have the same execution engine on the two clusters to concentrate only on the data migration performance regardless of the variation of systems performances running different operations. Proving that distributing the execution of two engines yields to better performance is the topic to be covered by polystore optimizer and is not the scope of this paper. In this paper, we concentrate on the best practice for efficient data migration between distributed systems. The left and right data sets are located on different data centers (the left data set is on Cluster A and the right data set is on B).

The second deployment scenario uses Muses as the distributed data migration engine between Spark clusters A and B. The deployment plan for the application's subtasks in Figure 7 is split into three parts: the first part is the binning of the left data set, to be executed on Spark Cluster A. The second part is the shuffling of the binned data, to be performed by Muses while migrating. The third part is the rest of the operations (bin right data set, shuffle right data set and join with data form A) and they are executed on Spark cluster B.

The binning operation of the left data set involves replicating the left data set for the number of files in the right data set. Thus, the increase in the number of files in the right data set will affect the intermediate data sizeIn our experiments, even-though, we fixed the size of the left data set, the intermediate data increases with the increase of a number of files in the right data set (because of the left data set binning replication).

Muses creates a pipeline between the two Spark installations. The pipeline consists of a set of data Akka streams from spark distributed instances in A to Spark distributed instances in B, as discussed in the previous sections. Therefore, the connection has no intermediate staging on HDFS. The performance using Muses was gained from both pipe-lining the distributed systems and the shuffle performed in Muses while migrating the intermediate data. The execution time overhead that results from distributing this application on two engines while using Muses as the distributed data migration is 15% of the execution time for 70GB of data.

The third scenario is the intuitive way of connecting two distributed clusters with independent distributed storage. In this scenario, HDFS plays the role of intermediate

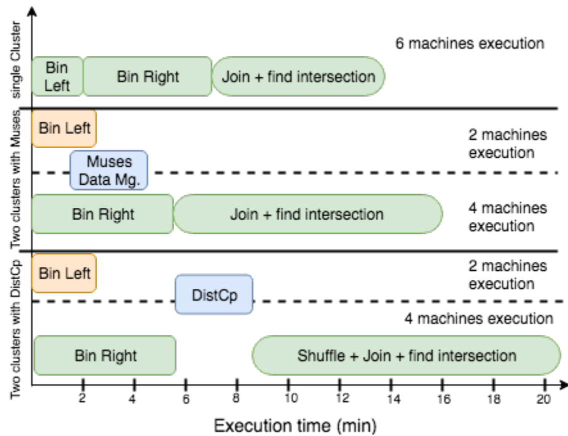

Figure 10: Genomic Spatial Join performance breakdown.

storage for the data migration between the two Spark engines. After the left data set binning on Cluster A, the result is stored on the HDFS of A. Then the binned left data set is migrated to the HDFS on Cluster B. Distcp creates a MapReduce application that reads from the HDFS on A and writes into the HDFS on B. The rest of the operations are performed on Spark cluster B.

The *DistCp* setup is sensitive to the intermediate data size as shown in Figure 9. Muses transfers the intermediate data in parallel to the execution of the two servers, see Figure 10. With Muses, the intermediate data between the engines are streamed and shuffled therefore decreases the overhead of the linking the two distributed systems.

Overall, the deployment with Muses outperforms the staging on HDFS. Figure 10 show a profile of one execution of the application on the three setups. Figure 10 shows Muses improves performance by 30% over the DistCp while it is 15% slower in comparison to a single engine execution. In the second scenario, both clusters run in parallel to the intermediate data transfer. The join execution in the single engine execution is faster than the other cases because six machines are involved in the join. The slowest join is performed in the third scenario because only four machines are involved. Even though only four machines are executing the join in the second scenario, the performance is higher than the last scenario. The improvement in the join performance is because Muses shuffles the left data set while migrating the data. As a result, the **join operation does not require additional data movement**.

5 Related Work

There are several proposals for polystore implementations. Here, we focus on the state of the art data migration methods in polystores.

The PolyStores project [6] is most related to our work. Two separated proposals for data migration in the PolyStores project are discussed in details in Pipegen [4] and the work by Dziedzic et al. [5]. The former concentrates on achieving the best possible performance at the expense of generality of the approach, while the latter discusses a more general approach at the expense of performance. We discuss these approaches in detail below.

In order to connect engines, Bradon et al. [4] recompile the source code of the data stores after modifying the import and export methods code; adding a socket connection and serialization operations. These source code modifications are intended to enable only one pair of data stores connection. For parallel data migration between two data stores, Pipegen creates a global directory for the connections between execution engines using sockets. Pipegen connects an only a number of threads. In case of difference in parallelism degree between the source and destination engines, the reminder threads are neglected. In contrast to these works, Muses does not require the source-code of the execution engine to be available. Muses can connect one source data store to many destination engines. Muses makes it possible to establish many-to-many connections using specific data shuffle or balance operators.

Dziedzic et al. [5] discuss the overhead of intermediate data representations for a direct link between database. Data migration is performed manually in three independent steps: export, translate, and import. Binary files are staged in between those steps. In case of a multithreaded export/import operation, the data balancing between data stores is performed by manually splitting the exported files equally to the number of threads in the importing data store. Muses streams the data between data stores and with no manual intervention. In case of different number of parallelism between source and destination engines, Muses balances/shuffles the data between the engines threads.

Both approaches, Pipegen and Dziedzic, are manually configured to migrate data between two data engines. They do not have an automatic configuration mechanism that can connect engines such as Flink [13] and Spark [14], which have dynamic topologies setup at runtime when allocating tasks on distributed machines based on resource availability. We propose Muses as a solution for connecting dynamically distributed execution engines.

To optimize cross-engine platform, Lim et al. [16], executed applications with diverse sets of parameters. In this work, the authors propose ideas for data migration between engines and choose a shared HDFS as the medium between execution engines for testing. Finally, they left the data migration and transformation as an open problem to be solved. Agrawal et al. [22] present the architecture of a cross-engine polystore. However, they do not address the issue of the data migration beyond discussing data storage independence [23].

Gupta et al. present the cross-engine join query execution in federated database systems [24]. The authors only discuss the reduction of data movement based on join operation parameters. Muses performs a sub-task (shuffle) of the join query while connecting distributed data stores to hide the migration overhead. Therefore,

the work is orthogonal and can be incorporated in Muses.

In general, most of the solutions that perform data migrations in related work are either require hardcoding in the source and sink engines, contain manual steps, or dependent on shared storage. In contrast Muses does not require any special code for engines and thus is engine independent. It represents a general approach, and is reactive to the engines dynamic topology.

6 Conclusions

We present the Muses architecture for efficient data migration between distributed data analysis systems. Muses is a flexible, reconfigurable, distributed data migration engine, and scalable in connecting new distributed engines. Scalability is achieved by the design of the connectors to add new engines and the usage of the intermediate data structure. Muses can connect engines with or without intermediate storage. Our experiments show that Muses outperforms baseline connections for cross engine execution. For future work, our main future focus is to optimize the Muses overhead and data packing.

This work has been supported through grants by the German Ministry for Education and Research as s BIFOLD (01IS18025A and 01IS18037).

Literature

[1] Hadoop Yarn, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[2] *SciDB array DataBase*, http://www.paradigm4.com/

[3] *DistCp tool*, Apache Hadoop, https://hadoop.apache.org/docs/current3/hadoop-distcp/DistCp.html

[4] Haynes, Brandon and Cheung, Alvin and Balazinska, Magdalena, *PipeGen: Data Pipe Generator for Hybrid Analytics*, Proceedings of the Seventh ACM Symposium on Cloud Computing (SOCC), 2016 Oct 5 (pp. 470-483). ACM.

[5] Dziedzic, Adam and Elmore, Aaron J and Stonebraker, Michael *Data transformation and migration in polystores*, IEEE High Performance Extreme Computing Conference (HPEC) 2016 Sep 13 (pp. 1-6). IEEE.

[6] J. Duggan, A. Elmore J, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdonik, *The bigdawg polystore system*, ACM Sigmod Record. 2015 Aug 12;44(2):11-6.

[7] Y. Papakonstantinou *Polystore Query Rewriting: The Challenges of Variety*

[8] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, *A demonstration of the BigDAWG polystore system* Proceedings of the VLDB Endowment. 2015 Aug 1;8(12):1908-11.

[9] P. Chen, V. Gadepally, M. Stonebraker, *The bigdawg monitoring framework*, IEEE High Performance Extreme Computing Conference (HPEC) 2016 Sep 13 (pp. 1-6). IEEE.

[10] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson M. Stonebraker, *The BigDAWG polystore system and architecture*, IEEE High Performance Extreme Computing Conference (HPEC) 2016 Sep 13 (pp. 1-6). IEEE.

[11] *Apache Arrow*, https://arrow.apache.org/

[12] *AKKA*, http://akka.io/

[13] *Apache Flink*, https://flink.apache.org/

[14] *Apache Spark*, https://spark.apache.org/

[15] *Encode: Encyclopedia of DNA Elements*, https://www.encodeproject.org/

[16] H. Lim, Y. Han, S. Babu, *How to Fit when No One Size Fits.*, CIDR 2013 (Vol. 4, p. 35).

[17] K. Shvachko, H. Kuang, S. Radia, R. Chansler, *The hadoop distributed file system*, MSST 2010 May 3 (Vol. 10, pp. 1-10).

[18] A. Kaitoua, P. Pinoli, M. Bertoni, S. Ceri, *Framework for supporting genomic operations*, IEEE Transactions on Computers. 2016 Aug 29;66(3):443-57.

[19] Lo, Ming-Ling and Ravishankar, Chinya V, *Spatial hash-joins*, ACM SIGMOD Record 1996 Jun 1 (Vol. 25, No. 2, pp. 247-258). ACM.

[20] M. Bertoni,S. Ceri, A. Kaitoua, P. Pinoli, *Evaluating cloud frameworks on genomic applications*, IEEE International Conference on Big Data (Big Data) 2015 Oct 29 (pp. 193-202). IEEE.

[21] A. Kaitoua, T. Rabl, A. Katsifodimos, V. Markl, *Muses: Distributed Data Migration System for Polystores*, IEEE 35th International Conference on Data Engineering (ICDE) 2019 Apr 8 (pp. 1602-1605).

[22] D. Agrawal, S. Chawla, A. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, M. Zaki, *Road to Freedom in Big Data Analytics*, EDBT 2016 Jan 1 (pp. 479-484).

[23] A. Jindal, J. Quiané-Ruiz, S. Madden, *CARTILAGE: adding flexibility to the Hadoop skeleton*, Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data 2013 Jun 22 (pp. 1057-1060). ACM.

[24] A. Gupta, V. Gadepally, M. Stonebraker, *Cross-engine query execution in federated database systems*, IEEE High Performance Extreme Computing Conference (HPEC) 2016 Sep 13 (pp. 1-6). IEEE.

[25] S. Cattani, S. Ceri, A. Kaitoua, P. Pinoli, *Evaluating Genomic Big Data Operations on SciDB and Spark*, International Conference on Web Engineering 2017 Jun 5 (pp. 482-493). Springer, Cham.

Dr. Abdulrahman Kaitoua Abdulrahman Kaitoua is a Senior big data architect and a team lead in the innovation and research team of GK-Software SE company in Berlin, Germany. He received his Ph.D. with honor in Information Technology from Politecnico di Milano in 2017.

Address: Technical University of Berlin,Germany, E-Mail: abdulrahman.kaitoua@polimi.it

Prof. Dr. Tilmann Rabl Tilmann Rabl is a full professor and Chair of the Data Engineering Systems Group at Hasso Plattner Institute and the University of Potsdam. He is also cofounder of the startup bankmark.

Address: Hasso Plattner Institute, University of Potsdam, Germany, E-Mail: tilmann.rabl@hpi.de

Prof. Dr. Volker Markl Volker Markl is a Full Professor and Chair of the DIMA Group at TU Berlin and an Adjunct Full Professor at the University of Toronto. He is Director of the Intelligent Analytics for Massive Data Research Group at DFKI and Director of the Berlin Big Data Center.

Address: Technical University of Berlin, Germany, E-Mail: volker.markl@tu-berlin.de