Muses: Distributed Data Migration System for Polystores

Abdulrahman Kaitoua
DFKI
Berlin, Germany
abdulrahman.kaitoua@dfki.de

Tilmann Rabl
TU Berlin, DFKI
Berlin, Germany
rabl@tu-berlin.de

Asterios Katsifodimos
Delft University of Technology
Netherlands
a.katsifodimos@tudelft.nl

Volker Markl
TU Berlin, DFKI
Berlin, Germany
volker.markl@tu-berlin.de

***Abstract*—Large datasets can originate from various sources and are being stored in heterogeneous formats, schemas, and locations. Typical data science tasks need to combine those datasets in order to increase their value and extract knowledge. This is done in various data processing systems with diverse execution engines. In order to take advantage of each execution engine's characteristics and APIs data scientists need to migrate and transform their datasets at a very high computational cost and manual labor. Data migration is challenging for two main reasons: i) execution engines expect specific types/shapes of the data as input; ii) there are various physical representations of the data (e.g., partitions). Therefore, migrating data efficiently requires knowledge of systems internals and assumptions.**

In this paper we present Muses, a distributed, high-performance data migration engine that is able to forward, transform, repartition, and broadcast data between distributed engines' instances efficiently. Muses does not require any changes in the underlying execution engines. In an experimental evaluation, we show that migrating data from one execution engine to another (in order to take advantage of faster, native operations) can increase a pipeline's performance by 30%.

***Index Terms*—Distributed systems, data migration, data transformation, big data engine, data integration.**

I. Introduction

Polystores [1], [4], [10] combine a set of specialized data processing engines (e.g., graph engines, dataflow engines, array databases) in order to perform data analysis at scale, using each specialized engine according to its characteristics. Each engine uses its own format and storage location for the data it processes, often making data migration between those data processing engines the bottleneck in processing that data. Thus, the decision on whether two different execution engines are used for a single data pipeline, depends on whether the overhead of data migration is smaller than the speedup caused by a specialized execution engine.

Current Polystores consider execution engines and data storage as separated layers. Before performing any operation combining two different datasets, all datasets need to be loaded to the store performing that operation. In this paper, we show that data migration can be perform more than a simple extract-load process. During data migration, we can take into account the physical representation of data (partitioning, distribution, locality, etc.) and migrate data in the most efficient manner. For instance, a parallel equijoin operator can assume that the incoming data is partitioned by key. Then, partitioning while migrating data can yield significant performance benefits.

Efficient data migration between distributed engines is challenging. When migrating data one should take into account the different execution strategies and data representation assumptions of each of the individual participating execution engines. This requires deep knowledge of the participating execution engines, as well as the costs of each operation versus the costs of different migration strategies and their physical properties.

As a solution to these challenges, we present Muses, a distributed data migration system. Our contributions are:

- We present and evaluate Muses, a distributed data migration system that can connect to different distributed execution engines (currently supporting Apache Hadoop, Spark, Flink, Postgres, and SciDB) and migrate data respecting physical properties.
- *Muses is reactive to changes of engines instances topology*: Apache Spark [14] and Flink [2], [11], reserve application resources at run-time. Thus, instances topology differ based on the resources availability. Muses engine reacts to the executing engines topology changes.
- Muses is able to perform shuffling and data reorganization during data migration, improving performance up to 30%.
- We evaluate Muses on real data with a genomics use case.

The rest of this paper is organized as follows: Section II discusses the motivating example, a spatial join in genomics, which utilizes cross-engine execution. Section III shows the state of the art in data migration for Polystores. Section IV describes the architecture of Muses in detail. Section V, shows the performance evaluation. Finally, Section VI concludes the paper with ideas for future work.

II. Motivating example

Many genomics applications integrate several data types located in different storage technologies. The GenoMetric Query Language (GMQL) [9] was developed to perform queries on heterogeneous genomic data. GMQL is SQL-like language to query region-based genomic data. GMQL contains operations on the region data, RD (DNA intervals with a start and a stop position represent regions, such as genes regions), and the meta-data, MD, which describes the genomic data (clinical data). A single GMQL script might contain aggregation operations, meta operations, and domain-specific operations. In our use case, clinical data (metadata) is small in size relative to genomic data (regions data) and it is stored in PostgreSQL database. Previous work has shown that GMQL operations on metadata perform well on the PostgreSQL datbase management system. Cattani et. al. [3] analyzed performance differences between implementations of GMQL region commands in SciDB [12] (a multi-dimensional

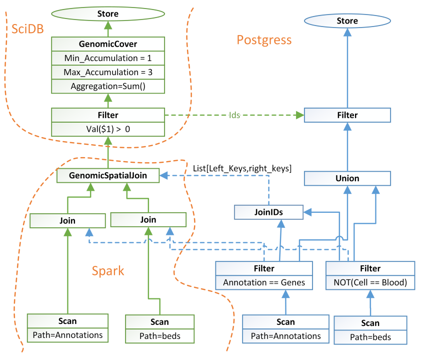

Fig. 1: Abstract Syntax Tree of GMQL script.

scientific database) and Apache Spark [14]. Apache Spark exhibited better performance in specific genomic operations because of the ease of use of UDFs in Spark for complex genomic operations, while SciDB processes aggregations in orders of magnitude faster. Based on these observations, the following GMQL operations can be deployed on the three engines as shown in Figure 1:

```
GENES = SELECT() ANNOTATIONS;
PEAKS = SELECT() BED_PEAKS;
MAPPED = GenometricJoin() PEAKS GENES;
SELECTED = SELECT(Count_PEAKS_GENES>0) MAPPED;
RELEVANT = COVER(1,2) SELECTED;
MATERIALIZE RELEVANT INTO OUTPUT;
```

The query consists of 5 operations and returns regions responding to a specific biological problem. The query performs both classic relational operations and domain-specific ones. The resulting operator graph consists of two directed acyclic graphs (DAG), one for regions operations and another for metadata operations. The regions graph (green in Figure 1) consists of three central operations; a selection, a genomics spatial join (finds interval intersections), and a genomic cover (histogram) operation. The operator graph has three types of connections between the data stores in this example: distributed to distributed (spark to SciBD), distributed to single node (SciDB to PostgreSQL), and single node to distributed (PostgreSQL to Spark). These can be optimized using Muses.

III. Related Work

Two proposals for data migration in PolyStores project are discussed in detail in Pipegen [7] and the work by Dziedzic et al. [5]. The former concentrates on achieving the best performance at the expense of generality of the approach, while the latter discusses more general approach at the expense of performance.

Lim et al. motivate running applications with varying parameters using cross-engine execution to optimize performance [10]. The authors propose ideas for data migration between engines and choose a shared HDFS as the medium between execution engines. Our experiments have shown that HDFS is a very slow migration medium. For this reason, we focus on on-the-fly migration between different engines by establishing connections among those engines. Agrawal et al. present a vision of a cross-engine execution engine architecture [1]. The authors concentrate on the engines and leave out data migration with a visionary discussion around data storage independency [8]. Gupta et al. present cross-engine join query execution in federated database systems [6]. The authors only discuss the reduction of data migration based on join operation parameters. Those ideas can be incorporated into Muses, as it can perform data partitioning for joins while connecting distributed data stores to hide the migration overhead.

IV. Architecture

The Muses engine is designed to connect distributed data engines for cross-engine execution. Figure 2, shows the general architecture of the Muses engine. Muses consists of a single **Muses Manager** node and one or more **Muses Nodes**. 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 Muses Nodes exchange only the application data streams.

Each data migration job in Muses contains a data *Producer* and a data *Consumer*. We refer to data engines as sources or sinks; one or more sources form a Producer and one or more sink form a Consumer. Sinks and sources might be distributed engines. Thus, we call a source's output threads as source instances and the sink input threads as sink instances. For simplicity of the description ofthe Muses architecture in Figure 2, Muses nodes are either marked as Producer or Consumer nodes. Since each machine can host several sinks/sources, any 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. Source/Sink 'A' is shown as a distributed source and has three instances distributed on two machines (Server 1, and Server 2). Source/Sink 'C' has one instance 'C1' on Server 1, and source/sink 'DB' is a database instance on Server 2.

A. *Muses Manager*

The Muses Manager manages registration of new sources/sinks, distributes job configurations on nodes, monitors job execution and collects profiling information.

The engine registration process requires the user to provide a list of machines that host the source/sink and the connectors of the registered source/sink. Migration jobs are submitted to the Muses Manager by the user or by a cross-engine execution dispatcher. Muses *jobs configurations* include: the producers and consumers for the migration job, the data distribution operation between the producer and the consumer, and the engine's topology, in case it has been dynamically set.

For every data migration job, the Muses Manager sends the *task configurations* to each Muses Node involved in the job execution, the task configurations includes: The number of Connector instances for each source/sink on the machine, which is equal to the number of source/sink instances on the 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.

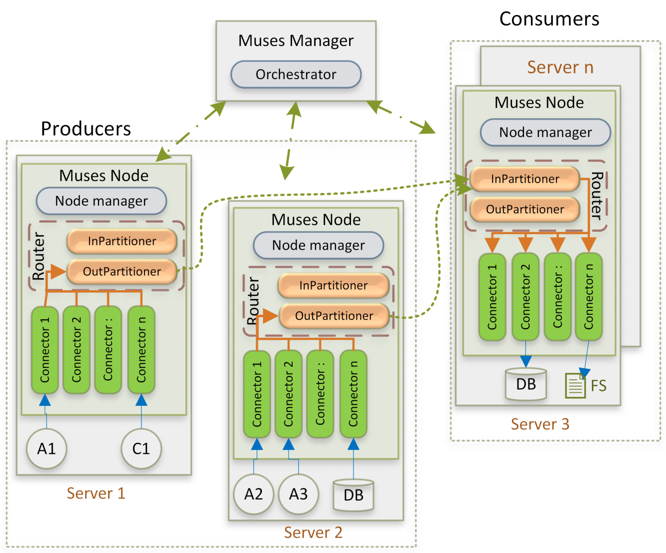

Fig. 2: Muses Architecture

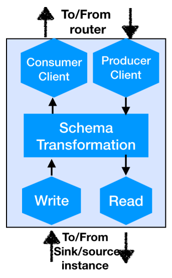

Fig. 3: Muses Connector

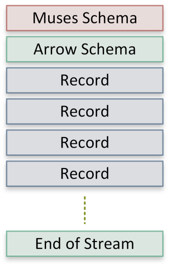

Fig. 4: Muses stream format

B. *Muses Node*

Figure 2 shows *Muses Nodes*, which consist of a set of Connectors (P_n), consumer and producer routers, and a Node manager. The Node Manager is the only daemon process that keeps running on each node for the Muses engine, while Muses routers and connectors are initialized on demand at the job configuration phase which reduces the overall overhead of Muses.

1) Node Manager: The Node Manager is responsible for keeping a connection with the Muses Manager for receiving job configurations and sending status and aggregated streams profiling information through heartbeats messages. The Node Manager initializes the routers for the job by setting up consumer/producer streams, manages resources reserved by the routers and Connectors, and restores the stream in case of nodes failure.

2) Connector: Apache Arrow is used as an intermediate data representation to facilitate connecting several engines with minimal coding requirements.

Figure 3 shows the logical representation of a Muses Connector in detail. The Connector consists of a Read operator that reads from the source and translates the data into a Muses Stream which encapsulates an Apache Arrow [1] data structure, a Write operator that reads from the Muses Stream and writes to the data sink, and a socket client to the routing module The engines' topologies and the Connectors are the two main parameters needed to connect engines.

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

Two types of schema transformations have to be considered in connecting distributed systems; The *logical schema*, for example, represents a row-based, column-based, or array-based data layout. *The physical schema* is represented by data types, for example an unsigned 64 bit integer. Muses streams are used for the data transformation between distributed engines.

A Muses stream consists of several Arrow streams with an associated schema. The schema contains the stream ID and information about the data set in the original logical format, such as: source logical schema, primary keys, secondary keys, values used as dimensions for Array databases, and values used to as keys and values for key/value databases. Arrow streams have a structure that is shown in Figure 4, they consist of the *Arrow Schema* and one or more *Record Batches*.

3) Routers: The Muses Router is an Akka streaming graph that controls the flow of streams in and out of a Muses Node. The Routers control the Connector instances. When the node has only one Consumer instance, we configure the Consumer Router as a *forward router* to optimize the link. When consumer engine instances vary in their throughput, we perform *stream balancing* in case there is no specific shuffling mechanism already set using Akka *Balancing Router*.

Once the streams end, the Connector instances are no longer needed. The Consumer Routers and Producer Routers start and stop the Connectors instances based on the data streams tags and the job configurations. 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.

V. Performance evaluation

Data for the experiments was collected from the Encode genomics repository [13]. Encode data is a tab-delimited text with ten columns. The schema has two Long fields, two String fields, one Char field, and four Double fields. We use 9 data sets of different size to show the performance with increasing data size. Our experimentation platform is a cluster comprised of 8 machines containing Intel® $E5620$ processors with 8 hyper-threads, and $32GB$ memory.

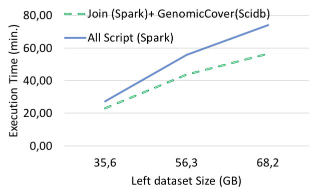

Fig. 5: Script execution

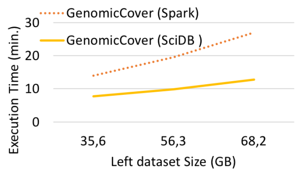

Fig. 6: Cover performance.

A. *Single engine solution versus Polystores*

In this section, we discuss the comparison between two deployments of the motivation example, shown in Section II; a single system deployment and Poystores deployment. In the single system deployment, Apache Spark uses all the eight machines resources. While for the Polystores deployment, the eight machines are shared between three engines: Apache Spark is installed on all eight machines while 4 out of those eight machines host a SciDB installation and one machine hosts PostgreSQL.

Figure 5 shows the performance of both the single system and the Polystores setup. In the Polystore setup, we run only a part of the execution plan on SciDB, which is the GenomicCover operator. The performance gain from running the GenomicCover operator over SciDB is shown in Figure 6. For input data sizes of 35GB, 56GB, and 70GB, the performance speed up for the polystore deployment are 1.2, 1.25, and 1.3 respectively. The performance speed up trend is positive with respect to the input data size.

The automated distributed data migration, thanks to Muses engine, allows the overlapping of the engines executions and the data migration; thus minimizes the overhead of the data migration. As soon as the first output sample is being processed by Apache Spark, it is streamed to SciDB nodes and imported into SciDB.

B. Data migration profiling

We designed our experiments in this section to show the intermediate data representation overhead and the performance from connecting distributed engines using a hard-coded pipeline. Al. Haynes et al. show that Apache Arrow as an intermediate representation performs best out of a variety of data representations [7], thus we concentrated on using Apache Arrow in our tests. We exclude the data export and load execution time in our profiling. We focus on the data migration process, but also evaluate the effect of export and import parallelism.

The socket connection scenario, in Figure 7, transfers the data from the source to the sink engine machines as binary stream. In this scenario, *no data is parsed*. Though, this scenario is the simplest with the highest performance, it needs both engines to be almost identical in both physical and logical data formats.

Migrating data between two engines with different data types requires a data translation process. In the Akka setup, Figure 7, the data translation operation translates the data types from the source types to the destination engine's types. The drawback of this setup is that it is connection specific. The data transformation introduces additional overhead on the migration process while increasing process parallelism can decrease this overhead.

In Akka with Arrow setup, the data is translated into an intermediate data representation then translated back to the sink engine's data types. The connection setup is independent of the engines' data types. Changes of one of the engines' schemas only changes the translation at its part of the connection, and the connection remains valid. Even though this scenario has the overhead of the intermediate data representation, the performance is comparable to transferring the data without intermediate data format because Arrow structure reduces the data transferred on the network, and thus the performance is comparable with the second scenario. Of course, this is not always the case, because some engines produce data in binary format and thus Arrow format might not be more compact than the engines binary.

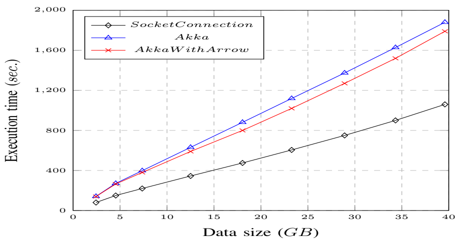

Fig. 7: Three data migration setups.

VI. Conclusions

In this paper we presented Muses, an efficient data migration engine that enables migrating data among different distributed execution. Our experiments show very promising results: using a migration engines such as Muses, one can speed up the migration of data making the case for distributed Polystores systems stronger. Our focus for the future is to automatically generate connectors from high-level descriptors in order to make the integration of new connectors easier and less labor intensive. Moreover, plan to look at data compression techniques and integrate more execution engines in our framework.

Acknowledgment. This work has been supported through a grant by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] D. Agrawal, S. Chawla, A. K. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and M. J. Zaki. Road to freedom in big data analytics. In *EDBT*, pages 479–484, 2016.
[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB JournalThe International Journal on Very Large Data Bases*, 23(6):939–964, 2014.
[3] S. Cattani, S. Ceri, A. Kaitoua, and P. Pinoli. Evaluating genomic big data operations on scidb and spark. In *International Conference on Web Engineering*, pages 482–493. Springer, 2017.
[4] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.
[5] A. Dziedzic, A. J. Elmore, and M. Stonebraker. Data transformation and migration in polystores. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.
[6] A. M. Gupta, V. Gadepally, and M. Stonebraker. Cross-engine query execution in federated database systems. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.
[7] B. Haynes, A. Cheung, and M. Balazinska. Pipegen: Data pipe generator for hybrid analytics. In *Proceedings of the ACM Symposium on Cloud Computing, ser. SOCC 16, New York, NY, USA*. ACM, 2016.
[8] A. Jindal, J. Quiané-Ruiz, and S. Madden. Cartilage: adding flexibility to the hadoop skeleton. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1057–1060. ACM, 2013.
[9] A. Kaitoua, P. Pinoli, M. Bertoni, and S. Ceri. Framework for supporting genomic operations. *IEEE Transactions on Computers*, 66(3):443–457, 2017.
[10] H. Lim, Y. Han, and S. Babu. How to fit when no one size fits. In *CIDR*, 2013.
[11] T. Rabl, J. Traub, A. Katsifodimos, and V. Markl. Apache flink in current research. *it-Information Technology*, 58(4), 2017.
[12] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. volume 15, pages 54–62. IEEE, 2013.
[13] S. University. Encode: Encyclopedia of dna elements, oct 2017.
[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.