Data Transformation and Migration in Polystores

Adam Dziedzic and Aaron J. Elmore
Department of Computer Science
The University of Chicago
Chicago, Illinois
adam.dziedzic@cs.uchicago.edu
aelmore@cs.uchicago.edu

Michael Stonebraker
CSAIL
Massachusetts Institute of Technology
Cambridge, Massachusetts
stonebraker@csail.mit.edu

***Abstract*—Ever increasing data size and new requirements in data processing has fostered the development of many new database systems. The result is that many data-intensive applications are underpinned by different engines. To enable data mobility there is a need to transfer data between systems easily and efficiently. We analyze the state-of-the-art of data migration and outline research opportunities for a rapid data transfer. Our experiments explore data migration between a diverse set of databases, including PostgreSQL, SciDB, S-Store and Accumulo. Each of the systems excels at specific application requirements, such as transactional processing, numerical computation, streaming data, and large scale text processing. Providing an efficient data migration tool is essential to take advantage of superior processing from that specialized databases. Our goal is to build such a data migration framework that will take advantage of recent advancement in hardware and software.**

I. Introduction

The amount of available data for analysis is growing exponentially due to an abundance of data sources, such as smart cities deploying sensors, IoT and personal devices capturing daily activity, human curated datasets (e.g. OpenMaps or Wikipedia), large-scale collaborative data-driven science, satellite images, multi-agent computer systems, and open government initiatives. This abundance of data is diverse both in format (e.g. structured, images, graph-based, matrix, time-series, geo-spatial, and textual) and the types of analysis performed (e.g. linear algebra, classification, graph-algorithms, and relational-algebra). Such diversity results in related data items existing in a variety of database engines. For example, Figure 1 depicts data from the medical dataset MIMIC II [1], which contains structured patient data, text-based doctor notes, waveform data from medical devices, and alert data to monitor a patient's vital signs.

To support diverse types of data and analysis tasks, scientists and analysts often rely on ad-hoc procedures to integrate disparate data sources. This typically involves manually curating how data should be cleaned, transformed, and integrated, a process referred to as *extract, transform, and load (ETL)*. Such processes are brittle and time-consuming. Additionally, they involve moving all data and computation into a single system to perform the analysis, which usually is a system not ideal for all required computation. Examples include transforming graph oriented data into a relational database for analysis, or transforming sensor data into a distributed data flow system (e.g. Spark) to perform linear algebra. The target system is often chosen for familiarity reasons or that some perceived majority of data already resides on the system. Ideally, any sub-part of the analysis process would be executed on the optimal systems with the results collated based on minimizing data movement. However, most analysts and programmers are not well equipped to manage a plethora of systems, robustly handle the transformations between systems, nor identifying the right system for the task.

To address these limitations, *Polystores* simplify the use of disparate data management systems by seamlessly and transparently integrating underlying systems with a unified interface to users. Several recent systems explore integration of relational analytic databases with distributed data flow environments, such as Hadoop or Spark [2]–[4]. In these systems, the relational engines are often used as accelerators for certain tasks. As an alternative approach, researchers at the Intel Science and Technology Center (ISTC) for Big Data have proposed *BigDAWG* as a polystore to integrate a wide variety of data systems, such as relational databases, array databases, streaming databases, and graph databases. To balance the issues of system transparency (or mapping datasets to engines) and functionality, BigDAWG relies on Islands of Information that are each a collection of engines, query operators, and shims to transform data between underlying engines. Users express queries in terms of ISLANDS instead of learning underlying languages or APIs. For example, a user may issue queries such as `ARRAY( transpose(SENSOR_DATA14))` or `RELATIONAL(select names from patients where age = 24)`.

Regardless of the type of polystore utilized, all systems that unify disparate data processing frameworks face the issue of data transformation and migration in two common ways. First, to migrate data when a workload change results in poor performance due to the incorrect mapping of items to engines. Second, to transfer partial results of query executions, for example, a final output of a MapReduce job could be transferred to a relational database and joined with data selected from a table [2], [3]. In either case, when one system requires some data objects from another system, a logical and physical transformation must occur to *cast* the data between systems. Casting data between systems is an expensive procedure due

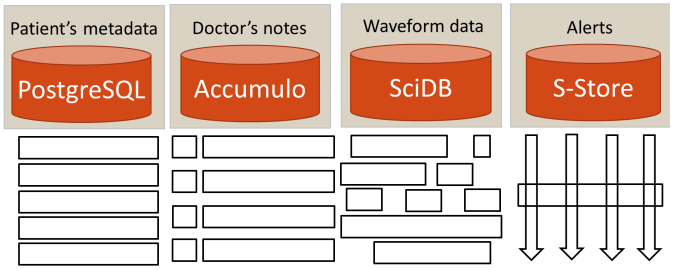

Fig. 1. The match of types of data with database models for: PostgreSQL (a row-oriented relational database), Accumulo (a key-value store), SciDB (an array database) and S-Store (a streaming database).

to parsing, serialization/deserialization, and extensive I/O to migrate data. To address this key issue, we propose **Portage** as a system to provide data transformation and migration between independent database instances. Currently, Portage supports a diverse set of database engines shown in Figure 1, including PostgreSQL (a row-oriented relational database) [5], Accumulo (a key-value store) [6], SciDB (an array database) [7] and S-Store (a streaming database) [8]. We focus on data migration between these systems.

In the remainder of the paper we discuss research challenges being explored in Portage. Section II outlines Portage's migration framework. Section III discusses the use of physical migration in the context of a relational and array-oriented database. Secion IV highlights how to enable parallel migration, presents migration in different data formats, and discusses adaptive migration for data stored in a distributed data-flow file system. Lastly, in Section V we discuss related work and in Section VI we describe future directions.

II. Data Migration Framework

A data migration and transformation typically involves four steps. First, the appropriate data items must be identified and extracted from the underlying engine. Depending on the system, this may involve directly reading a binary file, accessing objects in memory, or streaming data from a distributed storage manager. Second, the data must be transformed from the source database to the destination database. This involves both a logical transformation to change the data model and a physical transformation to change the data representation so the destination engine can operate on the data. Examples of the logical transformation include converting relational data to an array, or extracting a schema from a key-value store to build a relational format. To limit the scope of this work, we assume that the logical transformation is provided by the polystore. The physical transformation may be to a common shared format or directly to the destination's binary format. Third, the transformed data must be migrated between the machines. Depending on the instance locations and the infrastructure available, this step could involve use of pipes, sockets, files, shared memory, or remote direct memory access (RDMA). Lastly, the destination engine must load the data. Figure 2 highlights the various components in Portage to implement these four steps.

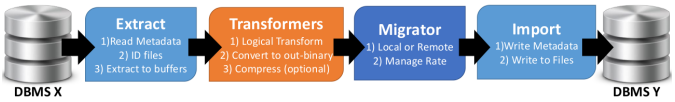

Fig. 2. Framework components for Portage.

III. Physical Migration

A general approach to physical data migration is to use a common format, such as CSV. We compare this approach with data migration in binary formats that includes intermediate transformation (between different binary formats) and direct migration in which both ends of the migration task process the same binary format. The three methods are presented in Fig. 3 and are detailed in this section.

Many databases support bulk loading and export of data in a CSV format. However, this process is compute bound with parsing (finding new line and field delimiters) and deserialization (transformation from text to binary representation) constituting the biggest consumers of CPU cycles. As shown in Fig. 3, CSV migration (top yellow line) is very slow with respect to other methods. Here, we migrate waveform data from the MIMIC-II dataset [1] from PostgreSQL to SciDB. This is a common case where CSV migration is easy to implement, but inherently slow.

Another approach is to migrate data using binary formats (middle blue line in Fig. 3). An intermediate binary transformation (external connector) is required, since there are different binary formats used by databases. Binary transformation requires conversion of data types. For example, SciDB's data types represent a subset of PostgreSQL's data types and the mapping between the data types has to be specified explicitly by the logical transformation. The binary migration with intermediate transformation is 3X faster than CSV migration when migrating data from PostgreSQL to SciDB. Nonetheless, this approach suffers from an extra conversion, so a natural question to explore is having only a single binary format for data transfer.

The third approach is to use a binary format which can be processed by both source and destination databases. Here, we change the source code of PostgreSQL to export binary data in SciDB's format and load the data directly to SciDB without intermediate transformation. This method (presented as the bottom red line in Fig. 3) enables us to achieve about 4X faster data migration than the CSV one. To understand the results using the three approaches, we analyze each step of the data migration process as depicted in Fig. 2.

The CSV migration from the Fig. 3 is split into CSV export and CSV load (two first yellow columns in Fig. 4). The CSV loading is slow because of parsing and deserialization [9].

In the second case, we use our data migrator as an external connector for intermediate binary transformation. The three blue columns in Fig. 4 correspond to the blue line in Fig. 3.

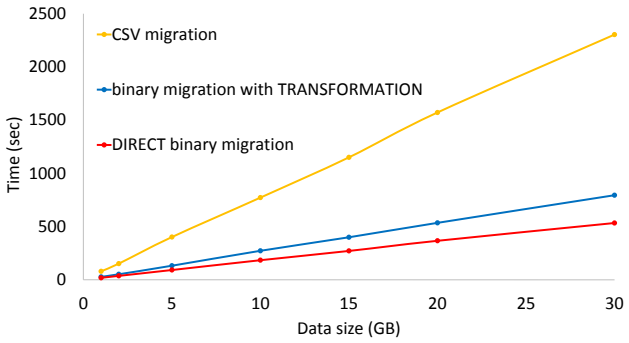

Fig. 3. Data migration from PostgreSQL to SciDB (3 approaches).

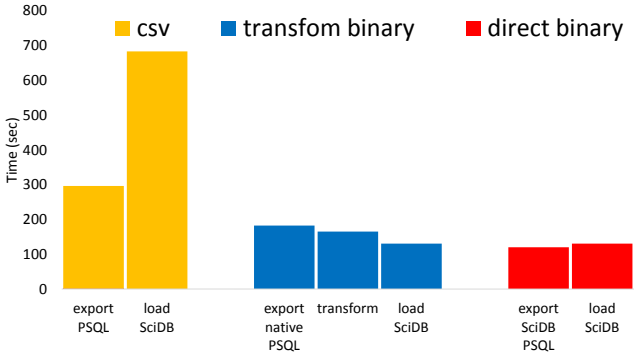

Fig. 4. Breakdown of data migration from PostgreSQL to SciDB (flat array) for waveform data of size 10 GB (PSQL denotes PostgreSQL).

In this case, the export is slower than loading. We argue that this unexpected situation is caused by bloated binary format in PostgreSQL (with data in PostgreSQL binary format often larger than in CSV format) and a concise SciDB binary format.

Finally, we present the direct binary migration with only two phases: export and load (presented as two last red columns in Fig. 4). When we modify PostgreSQL to export data directly in SciDB binary format, the export time is faster than binary loading to SciDB. The direct binary migration is the fastest approach due to the concise binary data format and no need for any intermediate transformation. However, there is a trade-off between performance and being user-friendly. The more metadata we put in the binary format, the slower the loading/export is because of bigger data size and additional processing. On the other hand, we can verify the data during migration process and if there is any error then we can determine the exact row and column where it occurred.

It is worth noting that we also explore use of archival formats, but due to space and comparable performance we omit the results from this paper.

IV. The Optimization of Data Migration

We can optimize the migration and transformation process using many techniques, such as dynamic compression, parallelism, rate-limiting, and adaptive extraction. In this section, we focus on parallelism and adaptive data migration.

A. Parallel migration

Many databases can load data in parallel from separate clients. For instance, the input data that should be loaded to a single table in a database could be provided as a collection of files (instead of one single file for the whole table). However, many databases do not support parallel data export. This section presents how we enable parallel export in PostgreSQL.

The default implementation of data export from PostgreSQL executes a full table scan, processes data row by row, and outputs a single file. This process could be parallelized in two ways. The logical approach is to distribute the rows from a table in a round-robin fashion to separate files. This approach guarantees that the sizes of the output files will be almost the same (with possible small differences of a single row size). However, this requires each of the clients to fully scan the same table, even though a part of the processed data is eventually exported. The physical approach distributes data on the level of pages, which every database table is organized into. The pages can be distributed in round-robin fashion to separate files. The sizes of the output files are also evenly distributed with possible differences of a single page size. Each client processes and exports (to a single file) part of a table (subset of its pages), instead of the whole table. Additionally, batches of pages are processed by each client to improve sequential I/O. The sequence (batch) size is a parameter that has to be found experimentally or provided by a database optimizer, which can use statistics on number of rows in a table and number of rows per page, so that output data files are similarly sized.

To illustrate our method, we compare the read patterns from disk for traditional extraction from PostgreSQL and our modified page-wise version. To capture the disk traces we use iosnoop[1]. For this experiment, we export a single *lineitem* table from the TPC-H benchmark (scale factor 1) [10]. Fig. 5 presents the original version of the COPY command with export from PostgreSQL. There is a single thread which reads the table block by block. Our modified version of the COPY command shown in Fig. 6 uses four concurrent clients (each client is represented as a different color). The blocks read by separate clients do not overlap. Our modified version of COPY completes the export task about 2X faster than the original version because it more effectively utilizes resources (CPU and storage read bandwidth). A key question we are exploring in Portage is how much parallelism should be applied to a migration and transformation process?

B. Parallel Migration from S-Store

The streaming main-memory database S-Store combines the best of two worlds: fast binary export and parallel processing. Here, the data is partitioned which enables us to extract each partition in a separate process. We enhance the S-Store export utility to directly transform its internal data representation to a required external binary format. The transformations occur internally in S-Store, so there is no need for the external transformation.

[1]http://www.brendangregg.com/blog/2014-07-16/iosnoop-for-linux.html

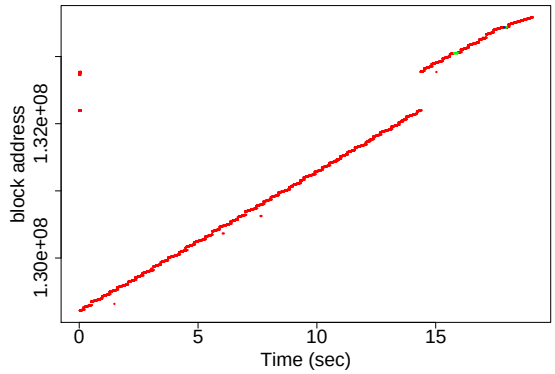

Fig. 5. Single-threaded export from PostgreSQL (current version).

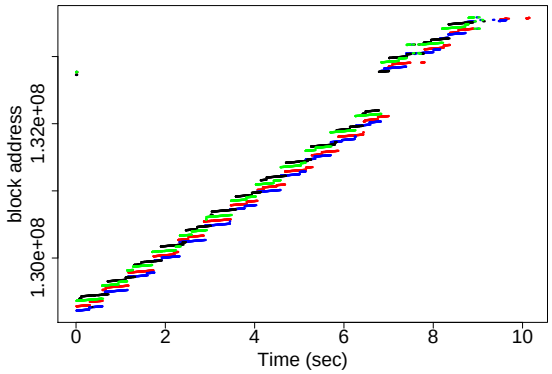

Fig. 6. Parallel export from PostgreSQL (modified version).

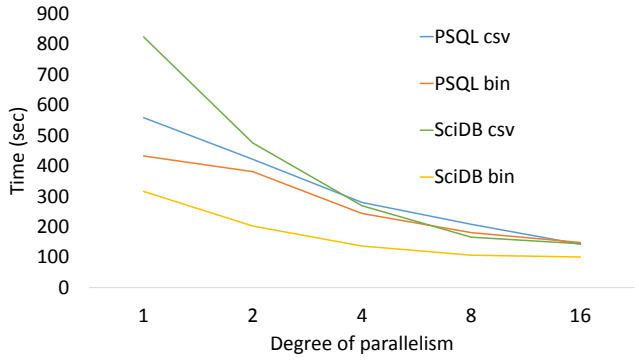

Fig. 7. Data migration from S-Store to PostgreSQL and SciDB (data from TPC-C [11] benchmark of size about 10 GB).

This experiment compares performance of data extraction from S-Store and direct data loading to PostgreSQL and SciDB. The goal of the experiment is twofold. First, we compare extraction from S-Store in three formats: CSV, PostgreSQL binary, and SciDB binary. Second, we test the performance of parallel data migration as a function of number of partitions in S-Store. The results presented in Fig. 7. show that binary migration is much faster than a CSV-based migration for low degrees of parallelism, however, the speed of two methods converges as parallelism increases.

As CSV migration is CPU bound, it scales better with more cores than binary migration. This is due to increased CPU cycles for parsing and deserialization. The main benefit of using the binary format is the elimination of the CPU intensive tasks. The single-threaded migration from S-Store to SciDB in CSV format is about 3X slower than binary migration (for TPC-C data when loading to SciDB is executed from a single instance). On the other hand, the binary migration to PostgreSQL is only about 20% faster than CSV migration because of PostgreSQL's bloated binary format. When enough CPU cycles are available for CSV migration, performance difference between CSV and binary migration is negligible. However, the requirement of even data distribution and available cores for CSV migration cannot be fulfilled in many cases, for example, because of data skew and limited CPU resources.

C. Adaptive Migration

Accumulo is a distributed, persistent, multidimensional sorted map, based on the BigTable system [12], with fine-grained access control. Accumulo adopts diametrically different architecture than other systems which we discussed. Data must be sorted and this can be done externally using MapReduce job, which incurs long start-up time, or directly via a programming interface. We explore two data loading/extraction methods in Accumulo and decide when each should be used.

Batch-Writer/Scanner is an internal part of the API provided by Accumulo. For batch-writer, one client first reads data from PostgreSQL and connects to an Accumulo master server to request information about tablet (e.g. chunk or shard) servers. The client then writes data to one tablet server. If the size of one tablet increases above a specified threshold, this tablet is split into two tablets and one of them is migrated to another tablet server for load balancing.

Accumulo requires all records to be sorted by row ID. One method is to directly load unsorted data (using Batch-Writer) and let Accumulo itself complete the ordering task. However, when faced with many records this method is costly since the whole dataset is reshaped, which incurs many network and disk I/O operations. Therefore, direct data loading of unsorted data is only effective for small amount of data. An alternative method is to load pre-sorted data to Accumulo. This requires the data-migration framework to take the responsibility for sorting data during its execution. Considering the distributed nature of Accumulo, we can order data in a MapReduce job during data migration. However, this method is only effective when loading large amount of data since overheads of distributed computing frameworks themselves are relatively high when used to process small amount of data. Therefore, we have to find a cutoff point (in terms of data size) in which we should switch from Batch-Writer to MapReduce job.

We compare performance between Batch-Writer/Scanner and MapReduce job in Accumulo for data migration by varying size of TPC-H data [10]. Fig. 8 and Fig. 9 show

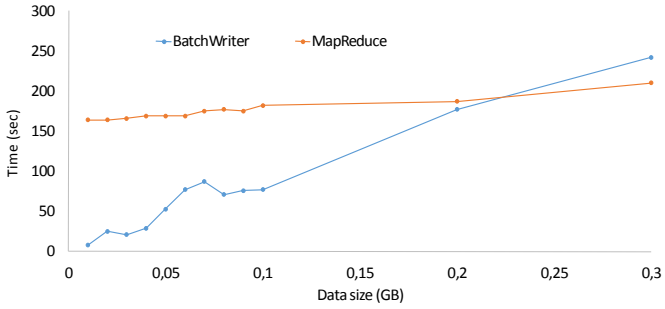

Fig. 8. From PostgreSQL to Accumulo (Small data size)

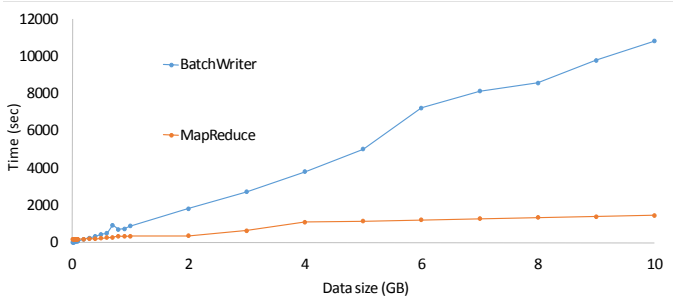

Fig. 9. From PostgreSQL to Accumulo (Big data size)

the experimental results of migrating data from PostgreSQL to Accumulo. When the dataset is small, Batch-Writer is slightly better than MapReduce (shown in Fig. 8) because of the overheads of launching MapReduce job. However, when the data size increases beyond 0.2 GB, MapReduce runs faster. Similar experiment was carried out for migration from Accumulo to PostgreSQL. For small data sizes, Batch-Scanner performs better than MapReduce but MapReduce indicates better performance when the data size is greater than 2 GB. Ongoing research is exploring active learning to identify the cut-offs for selecting the optimal loading method when more than one is available.

V. Related work

Here we present related research on data loading, extraction and migration in the context of traditional relational databases, main-memory databases, high performance shared-nothing databases, and data warehouses.

The live datastore transformation proposed in [13] is a general framework for data migration, but presents migration only in one direction: from MySQL to Cassandra. The authors devise a canonical model which is a general intermediate form for data migration between any two databases. The canonical model resembles relational model. This allows new databases via implementing the transformation from the database to the canonical model. However, such an approach adds an additional overhead.

Instant loading [14] is based on a scalable bulk loader for main-memory systems. It is designed to parallelize the loading phase of the HyPer in-memory database [9] by utilizing vectorization primitives (SIMD instructions) to eagerly load CSV datasets as they arrive from the network. It applies task- and data- parallelization on every stage of data loading phase to fully leverage the performance of modern multi-core CPUs and reduce the time required for parsing and deserialization. Additionally, instant loading also considers the problem of bulk creation of index structures and proposes the use of merge-able index structures. Disk-based database systems can also benefit from vectorized algorithms proposed by instant loading to reduce the CPU processing cost.

Sridhar, et al. [15] present the load/extract implementation of dbX, a high performance shared-nothing database system that can be deployed on commodity hardware systems and the cloud. To optimize loading performance the authors apply a number of techniques: (i) asynchronous parallel I/O for read and write operations, (ii) force every new load to begin at a page boundary and use private buffers to create database pages to eliminate lock costs, (iii) use a minimal WAL log, and (iv) force worker threads to check constraints on column values. In our work, we also decreased the loading time for PostgreSQL and Accumulo by setting a minimal level of write-ahead logging (WAL).

While Portage is focusing on building efficient custom end-points for a set of specific database engines, PipeGen [16] explores the use of a general-synthesized approach to automatically build data pipes based on existing unit tests for import/export. PipeGen enhances Java DBMSs, optimizes CSV and JSON formats, and uses Arrow as an intermediate format for network transfer. Portage uses named pipes to avoid data materialization via the disk and exploits native binary formats.

MISO [2] is a polystore system that combines Hadoop with a data warehouse. The main motivation is to run big data exploratory queries in a Hadoop environment and use the spare capacity of a data warehouse to speed up the queries. One of the crucial parts of MISO is to decide what data and when should be transferred to the data warehouse. The MISO tuner is responsible for computing a new multi-store design which would minimize future workload cost. It uses a heuristic that takes as input the observed workload, budgets for storages (in data warehouse and Hadoop), and transfer as well as previous multistore design with new materialized views. The MISO Tuner Algorithm consists of 4 steps: (i) calculate benefit of each separate view, (ii) compute interacting sets of views, (iii) sparsify the sets, and (iv) pack the MISO knapsacks. The MISO approach was able to give insight into data about 3 times faster than the sole use of a data warehouse or Hadoop. The MISO system can selectively migrate the data from Hadoop to data warehouse and argues that: *current approaches to multistore query processing fail to achieve the full potential benefits of utilizing both systems due to the high cost of data movement and loading between the stores*. We address exactly this problem in our data migrator.

VI. Future Work

We plan to incorporate other databases to extend the range of data formats. Once we have a full set of binary connectors

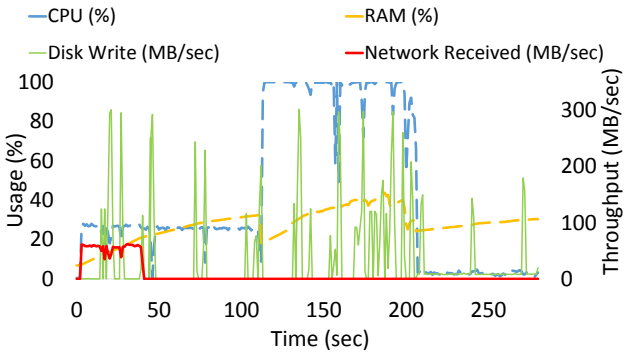

Fig. 10. Machine resources utilization while loading data to SciDB (0-50 sec - network transfer and transformation from csv to SciDB dense load format, 50 - 100 sec - loading, 100 - 250 - redimension).

we plan to build adaptive controller to optimize the data migration process and build a cost model for the time to migrate data between database engines. One of the goals of the project is to build a comprehensive and practical data migrator framework. This requires taking into account resource usage during the process.

In order to enable fast transformation and migration, Portage will need to address several open challenges. First, various and shifting resource bottlenecks can emerge. CPU cycles are needed for transformation, parsing, validation and serialization. Memory is used for internal buffers. Disk I/O is needed for extracting and importing the physical data. Network I/O is required for data transfer. A key requirement of Portage is to adapt its process based on constrained and available resources. For example, if network I/O is limited due to an over-utilized network and CPU cycles are abundant, then Portage should aggressively apply compression to reduce the data transferring over the network. On the other hand, when a given process is CPU-bound, for example when re-dimensioning from a flat array to a target array in SciDB (presented in the Fig. 10), and the network bandwidth is underutilized (below the maximum of about 125 MB/sec) then data compression should not be applied. We explore adaptive compression.

Second, given the increase in the number of cores available on machines and newer storage devices, Portage will need to control the amount of parallelization to apply for the migration. Too little and the process will be slow, and too much will result in resource thrashing. Similarly, the data transfer rate may need to be throttled to allow for slower loading or an over-loaded transport medium. To address these issues, we are exploring the use of adaptive controllers to manage the migration process.

Lastly, we are investigating the use of recent hardware and software trends, such as SIMD for data conversion, a unified automata processor (UAP) for fast data compression, JIT compilation for efficient data transformations code, and RDMA to achieve high-throughput and low-latency networking.

VII. Conclusions

Data migration is an inherent part of polystore systems, however, the data movement is costly and slow. In this paper, we introduce Portage - a performant data migration framework. While we present initial optimizations, we are exploring several research questions. We show that the binary migration can be much more performant than CSV migration. The parallelism is the key for faster CSV migration and can be enabled when the data is exported and imported in chunks by both ends of the migration process. The adaptivity is another interesting branch of our work which we want to research further on the level of resource usage.

Acknowledgment

This work is supported by the ISTC for Big Data.

References

[1] "MIMIC II: Waveform database overview." http://www.physionet.org/mimic2/mimic2_waveform_overview.shtml.

[2] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "Miso: Souping up big data query processing with a multistore system," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1591–1602. [Online]. Available: http://doi.acm.org/10.1145/2588555.2588568

[3] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in polybase," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1255–1266. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463709

[4] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687731

[5] "PostgreSQL," http://www.postgresql.org/.

[6] "Accumulo," http://accumulo.apache.org/.

[7] "SciDB," http://www.paradigm4.com/HTMLmanual/14.12/scidb_ug/.

[8] "S-Store," http://sstore.cs.brown.edu/.

[9] A. Kemper and T. Neumann, "Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.

[10] "TPC-H Benchmark: Standard Specification." http://www.tpc.org/tpch/.

[11] "TPC-C Benchmark: Standard Specification." http://www.tpc.org/tpcc/.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[13] T. Vanhove, G. van Seghbroeck, T. Wauters, and F. D. Turck, "Live datastore transformation for optimizing big data applications in cloud environments," in *IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, Ottawa, ON, Canada, 11-15 May, 2015*, 2015, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/INM.2015.7140270

[14] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *Proc. VLDB Endow.*, vol. 6, no. 14, 2013.

[15] K. T. Sridhar and M. A. Sakkeer, "Optimizing database load and extract for big data era," in *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part II*, 2014, pp. 503–512. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05813-9_34

[16] 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.