THE UNIVERSITY OF CHICAGO

DATA LOADING, TRANSFORMATION AND MIGRATION FOR DATABASE MANAGEMENT SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ADAM DZIEDZIC

CHICAGO, ILLINOIS
OCTOBER 2017

To my parents & sisters

Dziękuję Wam za wsparcie, wiarę we mnie i dobre słowo. Tak wiele to dla mnie znaczy.

The data can be used because we are able to load, transform, and migrate them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Aaron Elmore for his extensive support, guidance, patience and constructive feedback.

I would like to thank all the people whom I met, worked and collaborated with while working on the topics described in this thesis starting from the data loading work, through the data migration and ending at the data movement accelerators: Anastasia Ailamaki, Raja Appuswamy, Ioannis Alagiannis, Satyanarayana R Valluri, Manos Karpathiotakis, Michael Stonebraker, Vijay Gadepally, Zuohao She, Tim Mattson, Jennie Duggan, John Meehan, Stan Zdonik, Nesime Tatbul, Shaobo Tian, Yulong Tian, Jeremy Kepner, Samuel Madden, Dixin Tang, Kyle O'Brien and Jeff Parkhurst.

ABSTRACT

The exponential rate in which data is produced and gathered nowadays has turned data loading and migration into major bottlenecks of the data analysis pipeline. Data loading is an initial step of data analysis where raw data, such as CSV files, are transformed to an internal representation of a Database Management System (DBMS). However, given that the analysis is complex and diverse, many users utilize more than a single system since specialized engines provide a superior performance for a given type of data analysis. Thus, the data have to be transformed and migrated between many DBMSs. To ease the burden of dealing with a diverse set of disparate DBMSs, polystores seamlessly and transparently integrate specialized engines to expose a single unified interface to users and applications. A polystore system has to migrate data when a workload change results in poor performance due to inadequate mapping of data objects to engines. Moreover, partial results of query executions need to be migrated, for example, a final output of a MapReduce job is transferred to a relational database and joined with data selected from a table. To alleviate the aforementioned problems, first, we identify the overheads of data loading by carrying out an empirical analysis in three dimensions: software, hardware, and application workloads. Our experimental results show that modern DBMSs are unable to saturate the available hardware resources of multi-core platforms. Second, we leverage our findings from the empirical analysis of data loading and demonstrate how to harness task and data level parallelism, concise binary formats, adaptive processing, compression, and modern hardware to achieve a performant data transformation and migration. We present the results of our accelerators for four DBMSs. Our data migration solution is part of BigDAWG, a reference implementation of a polystore system, and can be used as a general framework where fast data loading, transformation, and migration are required. We show how the data loading process should use all the resources efficiently to achieve 10 times speedup and that we can migrate data 4 times faster by applying concise binary formats.

CHAPTER 1

INTRODUCTION

Many organizations amass an enormous amount of information on a daily basis to optimize their business strategy. Large scale scientific experiments, such as LHC (Large Hadron Collider) at CERN, generate data at an unprecedented rate of tens of petabytes per year. Users can run queries to extract value from the accumulated information if the data is loaded, transformed, and can be easily migrated between different data management systems. DBMSs are created with the premise that the data loading is a "one-time deal" and data is rarely moved or extracted. When these design limitations are combined with the exponential data growth and the need to use the same data for different analyses, the net effect is that the data loading emerges as a major bottleneck in the data analysis pipeline of state-of-the-art DBMSs. Once the data is processed and loaded into a DBMS, users want to transform it and analyze further on or combine with data sources residing in separate DBMSs. We divide our work into two parts. First, we analyze the data loading process, in which the raw data in a text format, such as CSV, is inserted into a DBMS. Second, we examine and show how to accelerate the data migration and transformation between four DBMSs. For data migration, from performance point of view, we leverage the fact that the data is no longer a raw file on disk but stored in a structured and optimized way in a database.

ETL (Extraction, Transformation, Loading) [43] process is established to Extract data from many OLTP databases, Transform them to adhere to a common schema, and then to Load them to a target data warehouse. The need for a reduced data-to-query time requires optimization of the data loading process. The ever-increasing growth of data requires data loading to scale easily and exploit the hardware parallelism to load a massive amount of data in very short time. The mismatch between the architecture of DBMSs, based on the assumption that data loading is a one time and offline operation, and the explosive growth of data causes an emergence of the data loading process as the main bottleneck in the data analysis pipeline. We scrutinize data loading operations to quantify how parallel data loading scales for various DBMSs and identify bottlenecks of the process. The analysis considers three dimensions: software, hardware, and application

workloads. Along the software dimension, we investigate architectural aspects (e.g. row stores vs column stores) of four state-of-the-art DBMSs, implementation aspects (e.g. the threading model used for parallel loading), and runtime aspects (e.g. degree of parallelism and presence and absence of logging/constraints). Along the hardware dimension, we evaluate the impact of storage configurations with different I/O capabilities (e.g. HDD, SATA SSD, hardware RAID controller with SCSI disks, and DRAM). Finally, along the workload dimension, we consider data from our micro-benchmarks, popular benchmarks (e.g. TPC-H [14], TPC-C [13], and SDSS [10]), and real-world datasets (e.g. Symantec [12]) with diverse data types, field cardinality, and number of columns. The results of the analysis show that:

- Bulk loading performance is directly connected to the characteristics of the dataset to be loaded: evaluated DBMSs are stressed differently by the involved datatypes, the number of columns, the underlying storage, etc.

- Both single-threaded and parallel bulk loading leave CPU and/or storage underutilized. Improving CPU utilization requires optimizing the input I/O path to reduce random I/O and the output I/O path to reduce pauses caused by data flushes. Such optimizations bring a 2-10x loading time reduction for all tested DBMS.

- Despite data loading being 100% CPU bound in the absence of any I/O overhead, the speedup achieved by increasing degree-of-parallelism (DoP) is sub-linear. Parsing, tokenizing, datatype conversion, and tuple construction dominate CPU utilization and need to be optimized further to achieve further reduction in loading time.

- In the presence of constraints, different DBMSs exhibit varying degrees of scalability. We list cases in which the conventional approach: (1) drop indexes, (2) load data, (3) rebuild indexes, is applicable to single-threaded index building and constraint verification, however, it is inappropriate under parallel loading.

- Under high DoP, constraints can create unwarranted latch contention in the logging and locking subsystems of a DBMS. Such overheads are a side-effect of reusing the traditional query

execution code base for bulk data loading and can be eliminated by making data loading a first-class citizen in DBMS design.

Once the data is loaded into a DBMS, users may require to analyze it further in specialized systems, hence data has to be migrated between different DBMSs. Ideally, any sub-part of the analysis process would be executed on an optimal DBMS 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, *Polystore* systems simplify the use of disparate data management systems by seamlessly and transparently integrating underlying systems with a unified interface to users. All polystores have to migrate data when a workload changes and to transfer partial results of query executions. 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. To address this key issue, we propose **Portage** as a framework to provide fast data transformation and migration between independent database instances. Currently, Portage supports a diverse set of database engines including PostgreSQL (a row-oriented relational database) [7], Accumulo (a key-value store) [1], SciDB (an array database) [9] and S-Store (a streaming database) [8]. A 4 to 10 times speedup of data migration and loading can be achieved by applying the following optimizations:

- Parallel export from PostgreSQL (the source code of the DBMS was modified to export data on the physical level of database pages instead of the logical level of tuples). The new feature is integrated in the copy command where users specify the number of output partitions.

- Parallel export from SciDB (from each of the partitions of the data) in SciDB and PostgreSQL binary formats.

- Binary data migration between PostgreSQL and SciDB is supported in two ways: (1) data is exported from PostgreSQL in its binary format, transformed on the fly to SciDB binary format, and finally loaded to SciDB, (2) PostgreSQL source code was modified to enable export

in SciDB binary format and direct binary migration between the DBMSs in the common and concise binary format.

- SIMD-optimized parsing during data loading to PostgreSQL.
- Lightweight compression algorithms (e.g. snappy) for data transfer via network.
- Adaptive migration where the data loading or export method for Accumulo is selected based on the data size (Batch-Writer/Scanner for up to 2GB of migrated data and MapReduce for large-scale migration).

CHAPTER 2

BACKGROUND

In the following two sections we give an overview of the fundamental concepts that are used throughout the thesis: heterogeneous data and polystore systems.

2.1 Heterogeneous data

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 2.1 depicts data from the medical dataset MIMIC II [5], 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. 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

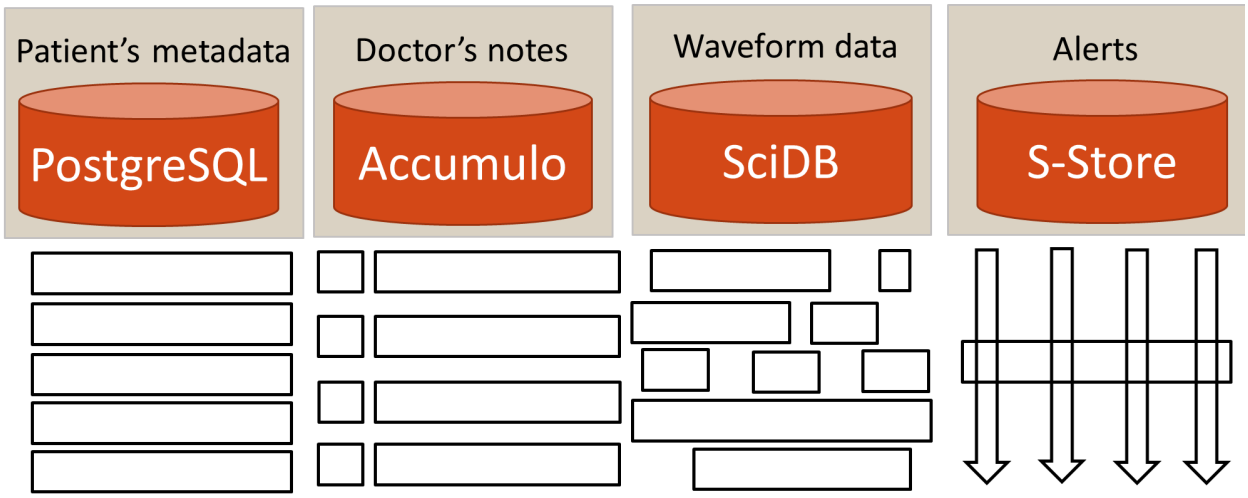


Figure 2.1: The match of types of data with database models for: PostgreSQL (a row-oriented relational database), Accumulo (a key-value store), SciDB (an array database) and S-Store (a streaming database).

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 limitation, polystore systems were introduced.

2.2 Polystore Systems

A polystore is a system supporting Big Data applications that exposes multiple storage engines through a single interface [37]. The difficulty with handling of the Big Data is caused by the variety, volume, and velocity with which the data is provided to the system. Several recent systems explore integration of relational analytic databases with distributed data flow environments, such as Hadoop or Spark [36, 25, 16]. 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 reference implementation of a polystore system that seamlessly and transparently integrates a wide variety of data systems, such as relational databases, array databases, streaming databases, and graph databases. It provides a unified interface to users. 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 changes and data objects has to be re-mapped to other DBMSs for better performance. Second, to transfer partial results of query execution. 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. We provide data transformation and migration between independent database instances. Currently, our data migration framework supports a diverse set of databases including PostgreSQL (a row-oriented relational database) [7], Accumulo (a key-value store) [1], SciDB (an array database) [9] and S-Store (a streaming database) [8]. We focus on data migration between these systems.

CHAPTER 3

DATA LOADING

3.1 Introduction

We analyze the behavior of modern DBMSs in order to quantify their ability to fully exploit multicore processors and modern storage hardware during data loading. We examine multiple state-of-the-art DBMSs, a variety of hardware configurations, and a combination of synthetic and real-world datasets to identify bottlenecks in the data loading process and to provide guidelines on how to accelerate data loading. Our findings show that modern DBMSs are unable to saturate the available hardware resources. We therefore identify opportunities to accelerate data loading.

3.2 Setup and Methodology

We now describe the experimental setup, the workloads employed to study and analyze the behavior of the different DBMSs during data loading, and the applied methodology.

3.2.1 Experimental Setup

Hardware: The experiments are conducted using a Dell PowerEdge R720 server equipped with a dual socket Intel(R) Xeon(R) CPU E5-2640 (8 cores, 2 threads per core resulting in 32 hardware contexts) clocked at 2.00 GHz, 64KB L1 cache per core, 256KB L2 cache per core, 20MB L3 cache shared, and 64GB RAM (1600 MHz DIMMs).

The server is equipped with different data storage devices, including i) individual SATA hard disk drives (HDD), ii) a hardware RAID-0 array with SAS HDD (DAS), and iii) a hardware RAID-0 array with SAS solid state drives (SSD). Table 3.1 summarizes the available storage devices and their characteristics.

OS: We run all the experiments using Red Hat Enterprise Linux 6.6 (Santiago - 64bit) with kernel version 2.6.32.

Name	Capacity	Configuration	Read Speed	Write Speed	RPM
HDD	1.8 TB	4 x HDD (RAID-0)	170 MB/sec	160 MB/sec	7.5k
DAS	13 TB	24 x HDD (RAID-0)	1100 MB/sec	330 MB/sec	7.5k
SSD	550 GB	3 x SSD (RAID-0)	565 MB/sec	268 MB/sec	n/a

Table 3.1: Storage devices and characteristics.

Analyzed Systems: The analysis studies four systems: a commercial row-store (DBMS-A), an open-source row-store (PostgreSQL [7]), a commercial column-store (DBMS-B), and an open-source column-store (MonetDB [6]). To preserve anonymity due to legal restrictions, the names of the commercial database systems are not disclosed. PostgreSQL (version 9.3.2) and MonetDB (version 11.19.9) are built using gcc 4.4.7 with -O2 and -O3 optimizations enabled respectively.

3.2.2 Datasets

The experiments include datasets with different characteristics: both industry-standard and scientific datasets, as well as custom micro-benchmarks. All datasets are stored in textual, comma-separated values (CSV) files.

Industrial benchmarks. We use the TPC-H decision support benchmark [14], which is designed for evaluating data warehouses, and the transaction processing benchmark TPC-C [13], which models an online transaction processing database.

Scientific datasets. To examine more complex and diverse cases compared to the synthetic benchmarks, we also include in the experiments a subset of the SDSS [10] dataset and a real-life dataset provided by Symantec [12].

SDSS contains data collected by telescopes that scan parts of the sky; it includes detailed images and spectra of sky objects along with properties about stars and galaxies. SDSS is a challenging dataset because i) it includes many floating point numbers that require precision and ii) most of its tables contain more than 300 attributes.

The Symantec spam dataset consists of a collection of spam e-mails collected through the

worldwide-distributed spam traps of Symantec. Each tuple contains a set of features describing characteristics of the spam e-mails, such as the e-mail subject and body, the language, the sender's IP address, the country from which the spam e-mail was sent, and attachments sent with the e-mail. NULL values are common in the Symantec dataset because each e-mail entry may contain different types of features. In addition, the width of each tuple varies based on the collected features (from a few bytes to a few KB). The Symantec spam dataset also contains wide variable length attributes (e.g., e-mail subject) that considerably stress systems which use compression for strings.

3.2.3 Experimental Methodology

The goal of the experiments is to provide insight on "where time goes" during loading in modern DBMSs – not to declare a specific DBMS as the fastest option in terms of bulk loading performance. The experiments thus explore a number of different configurations (software and hardware) and datasets, and highlight how different parameters and setups affect loading performance.

All DBMSs we use in this analysis support loading data either by using a bulk loading `COPY` command or by using a series of `INSERT` statements. We found bulk loading using `COPY` to be much faster than using `INSERT` statements for all DBMSs. Therefore, all experimental results reported in this paper were obtained by using the `COPY` command.

In addition to bulk loading, DBMS-A, MonetDB, and DBMS-B also offer built-in support for parallel loading. PostgreSQL, in constrast, does not support parallel loading. We work around this limitation by building an external parallel loader which we describe in Section 3.3.

Tuning. All tested systems are tuned following guidelines proposed by the DBMS vendors to speed up loading. For MonetDB, we also provide the number of tuples in the dataset as a hint to the parallel loader as we found that loading does not scale without providing this hint. To ensure fair comparison between the systems, we map datatypes used in benchmarks to DBMS-specific datatypes such that the resulting datatype size remains the same across all DBMSs. Thus, the difference in loaded database size across DBMSs is due to architectural differences, like the use of data compression.

Profiling. We collect statistics about CPU, RAM, and disk I/O utilization of the OS and the DBMS. We use `sar` to measure the CPU and RAM utilization, and `iostat` to measure disk utilization statistics. In addition, we use `iosnoop` to record disk access patterns and the Intel VTune Amplifier to profile the different systems and derive performance breakdown graphs.

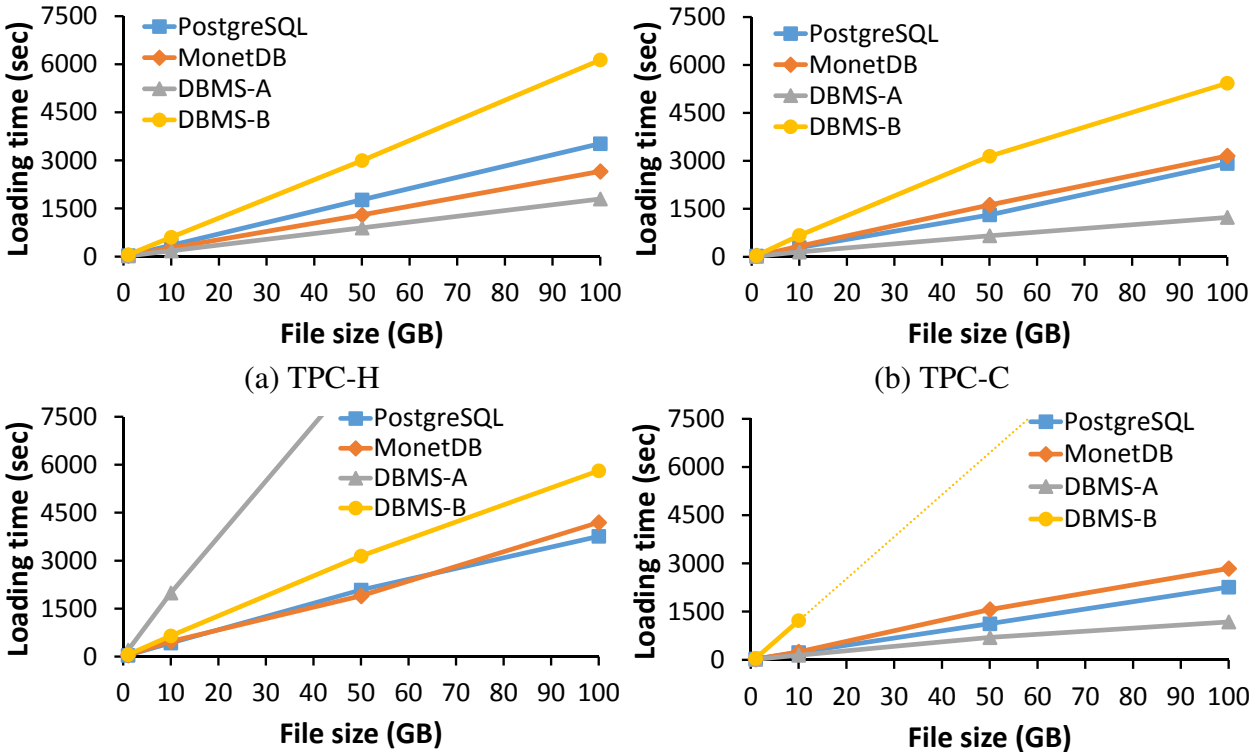
3.3 Experimental Evaluation

We conduct several experiments to evaluate the bulk loading performance of the tested systems. We start with a baseline comparison of single-threaded data loading using a variety of datasets. We then consider how data loading scales as we increase the degree of parallelism. Following this, we analyze I/O and CPU utilization characteristics of each DBMS to identify where time is spent during data loading and investigate the effect of scaling the storage subsystem. Finally, we examine how each system handles the challenge of enforcing constraints during data loading.

3.3.1 Baseline: Single-threaded data loading

This experiment investigates the behavior of PostgreSQL, MonetDB, DBMS-A and DBMS-B as their inputs increase progressively from 1GB to 100GB. Each variation of the experiment uses as input (a) TPC-H, (b) TPC-C, (c) SDSS, or (iv) Symantec dataset. The experiment emulates a typical enterprise scenario where the database is stored on a high-performance RAID array and the input data to be loaded into the database is accessed over a slow medium. We thus read the input from HDD, a slow data source, and store the database on DAS, a high-performance RAID array.

Figure 3.1(a-d) plots the data loading time for each system under the four benchmarks. As can be seen, the data loading time increases linearly with the dataset size (except when we load the SDSS dataset in DBMS-A and the Symantec dataset in DBMS-B). DBMS-A outperforms the rest of the systems in the majority of the cases; when considering 100GB database instances, DBMS-A is 1.5× faster for TPC-H, 2.3× faster for TPC-C, and 1.91× faster for Symantec compared to the second fastest system in each case. DBMS-A, however, shows the worst performance for



(a) TPC-H (b) TPC-C

(c) SDSS (d) Symantec

Figure 3.1: Data loading time increases linearly with the dataset size (single-threaded loading, raw input from HDD, database on DAS).

the SDSS dataset (5× slower than the fastest system). The reason is that SDSS contains numerous floating-point fields, which are meant to be used in scientific processing. DBMS-A offers a compact datatype for such use cases, which facilitates computations at query time but is expensive to instantiate at loading time, thus stressing the storage engine of DBMS-A. Among the other systems, PostgreSQL exhibits robust performance, having the second fastest loading time in most experiments.

PostgreSQL and DBMS-A outperform DBMS-B and MonetDB under the Symantec dataset because of the architectural differences between the two types of DBMSs. PostgreSQL and DBMS-A are row stores that follow the N-ary storage model (NSM) in which data is organized as tuples (rows) and is stored sequentially in slotted pages. OLTP applications benefit from NSM storage because it is more straightforward to update multiple fields of a tuple when they are stored sequentially. Likewise, compression is used less frequently because it makes data updates more

expensive. On the other hand, DBMS-B and MonetDB are column stores that follow the decomposition storage model (DSM) and organize data in standalone columns; since they typically serve scan-intensive OLAP workloads and apply compression to reduce the cost of data scans. Moreover, the compression in column stores is efficient because values from the same domain are stored consecutively.

Table 3.2 shows the ratio between the input data file and the final database size for the experiments of Figure 3.1. Even though the tested systems read the same amount of data, they end up writing notably different amounts of data. Clearly, DBMS-B and MonetDB have smaller storage footprint than PostgreSQL and DBMS-A. The row-stores require more space because they store auxiliary information in each tuple (e.g., a header) and do not use compression. This directly translates to improved performance during query execution for column stores due to fewer I/O requests.

The downside of compression, however, is the increase in data loading time due to the added processing required for compressing data. DBMS-B compresses all input data during loading. Thus, it has the worst overall loading time in almost all cases. MonetDB only compresses string values. Therefore, the compression cost is less noticeable for MonetDB than it is for DBMS-B. The string-heavy Symantec dataset stresses MonetDB, which compresses strings using dictionary encoding. This is why MonetDB exhibits the second worst loading time under Symantec. Despite this, its loading time is much lower than DBMS-B. The reason is that MonetDB creates a local dictionary for each data block it initializes, and flushes it along with the data block. Therefore, the local dictionaries have manageable sizes and reasonable maintenance cost. We believe that DBMS-B, in contrast, chooses an expensive, global compression scheme that incurs a significant penalty for compressing the high-cardinality, wide attributes in the Symantec dataset (e.g., e-mail body, domain name, etc.).

Name	TPC-H	TPC-C	SDSS	Symantec
DBMS-A	1.5	1.3	1.5	1.5
PostgreSQL	1.4	1.4	1.4	1.1
DBMS-B	0.27	0.82	0.18	0.25
MonetDB	1.1	1.4	1.0	0.92

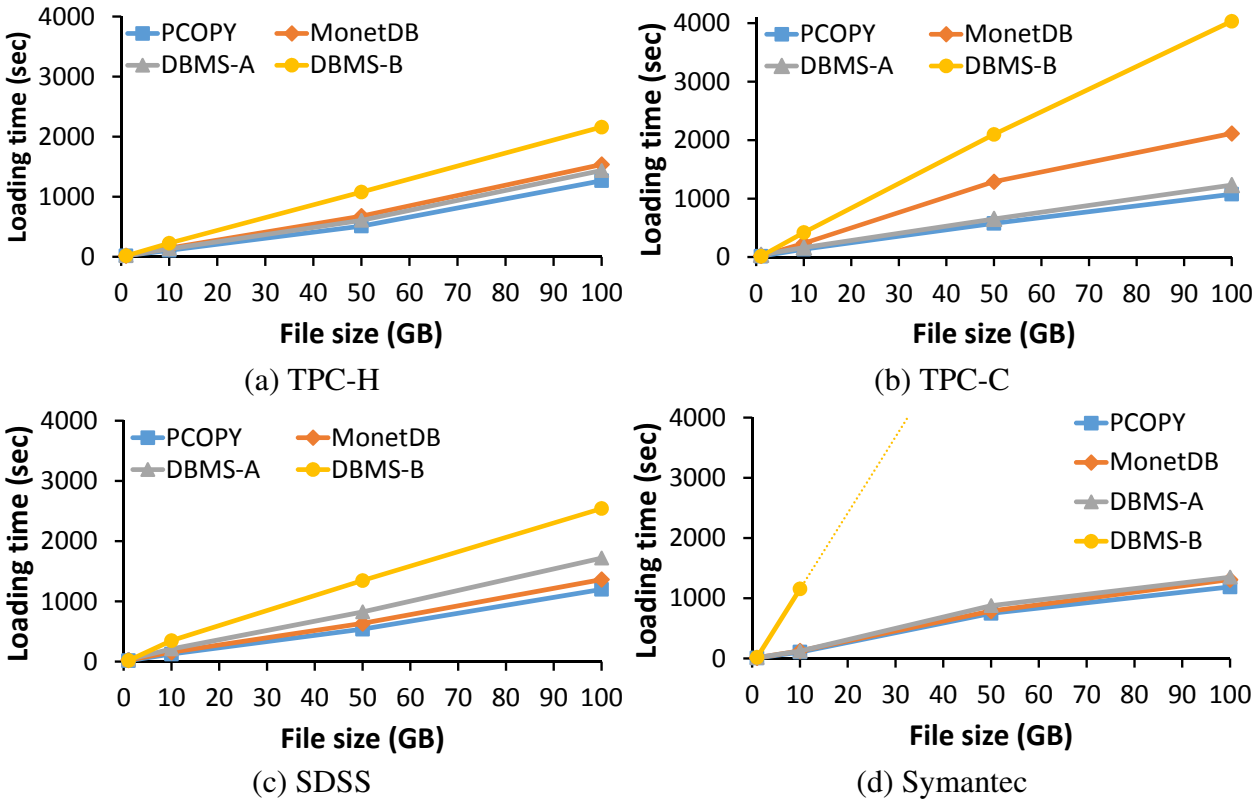
Table 3.2: Input data file/Database size ratio for each dataset (10GB instance). Column stores achieve a better ratio (less is better).

Summary

The time taken to load data into a DBMS depends on both the dataset being loaded and the architecture used by the DBMS. No single system is a clear winner in all scenarios. A common pattern across all experiments in the single-threaded case is that the evaluated systems are unable to saturate the 170 MB/sec I/O bandwidth of the HDD – the slowest input device used in this study. The next section examines whether data parallelism accelerates data loading.

3.3.2 Parallel data loading

The following experiments examine how much benefit a DBMS achieves by performing data loading in a parallel fashion. As mentioned earlier, PostgreSQL lacks support for parallel bulk loading out-of-the-box. We thus develop an external loader that invokes multiple PostgreSQL COPY commands in parallel. To differentiate our external loader from native PostgreSQL, we will refer to it as PCOPY. PCOPY differs from other systems that support parallel loading as native feature in that it uses PostgreSQL as a testbed to show how parallelism can be introduced to an existing RDBMS without tweaking its internal components. PCOPY is a multithreaded application that takes as input the file to be loaded into the database, memory maps it, computes aligned logical partitions, and assigns each partition to a different thread. Each thread sets up a pipe and forks off a PostgreSQL client process that runs the COPY command configured to read from a redirected standard input. Then the thread loads the data belonging to its partition by writing out the memory mapped input file to the client process via the pipe.



(a) TPC-H (b) TPC-C

(c) SDSS (d) Symantec

Figure 3.2: Data loading time increases linearly with the dataset size (parallel loading, input on HDD, database on DAS).

Figure 3.2(a-d) plots the results for each dataset. We configure all systems to use 16 threads – the number of physical cores of the server. Comparing Figures 3.1 and 3.2, we can see that parallel loading improves performance compared to single-threaded loading in the majority of cases. Similar to the single-threaded case, loading time increases almost linearly as the dataset size increases. DBMS-B shows the same behavior as in the single-threaded case for the Symantec dataset. On the other hand, parallel loading significantly improves the loading time of DBMS-A for SDSS; abundant parallelism masks the high conversion cost of floating-point values intended to be used in scientific computations.

The data loading code path of PostgreSQL proves to be more parallelizable for this experiment as PCOPY achieves the lowest loading time across the different datasets. Compared to single-threaded PostgreSQL, PCOPY is 2.77× faster for TPC-H, 2.71× faster for TPC-C, 3.13× faster for SDSS, and 1.9× faster for Symantec (considering the 100GB instances of the datasets).

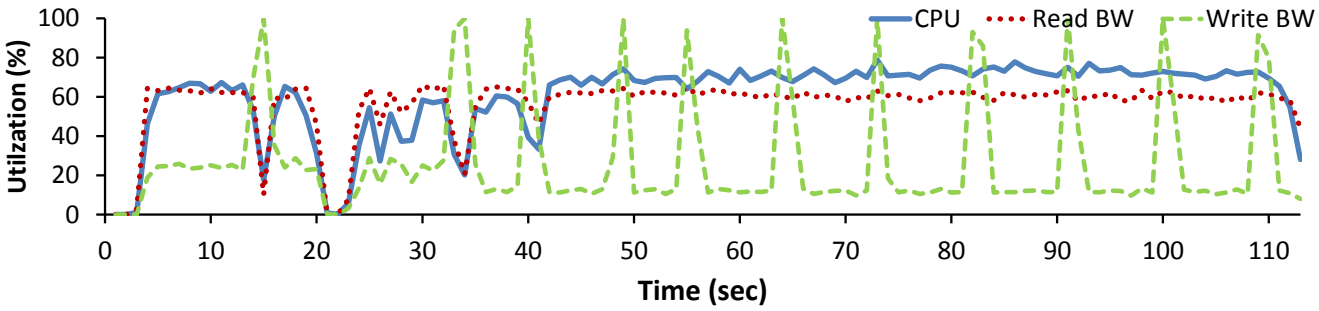
MonetDB benefits from parallel loading as well, being 1.72× faster for TPC-H, 1.49× faster for TPC-C, 3.07× faster for SDSS, and 2.16× faster for Symantec (100GB instances). The parallel version of DBMS-A is 1.25× faster for TPC-H and 10.78× faster for SDSS compared to the single-threaded version (100GB instances). On the other hand, DBMS-A fails to achieve a speed-up for TPC-C and Symantec. Finally, DBMS-B is 2.84× faster for TPC-H, 1.34× faster for TPC-C, and 2.28× faster for SDSS (100GB instances) compared to its single-threaded variation. Similar to the single-threaded case, DBMS-B requires significantly more time to load the long string values of the Symantec dataset. As a result, DBMS-B still processes the 10GB dataset when the other systems have already finished loading 100GB.

Summary. Figure 3.2 again shows that there is no system that outperforms the others across all the tested datasets. Generally, parallel loading improves data loading performance in comparison to single-threaded loading in many cases. However, scaling is far from ideal, as loading time does not reduce commensurately with the number of cores used. In fact, there are cases where a 16× increase in the degree of parallelism fails to bring any improvement at all (e.g., DBMS-A for TPC-C and Symantec).

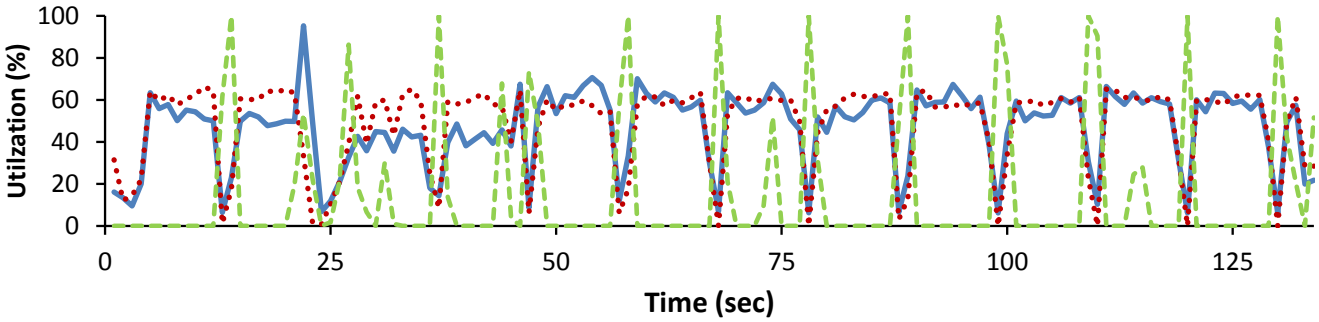
3.3.3 Data Loading: Where does time go?

The next experiment looks into CPU and I/O resource usage patterns to identify where time goes during the data loading process so that we understand the reason behind lack of scalability under parallel data loading. This experiment presents an alternative view of Figure 3.2(a): It monitors the usage of system resources (CPU, RAM, I/O reads and writes) when a 10GB version of TPC-H is loaded using the parallel loaders for various DBMSs. As before, raw data is initially stored on HDD and the database is stored on DAS. There are two patterns that can be observed across all systems in Figure 3.3:

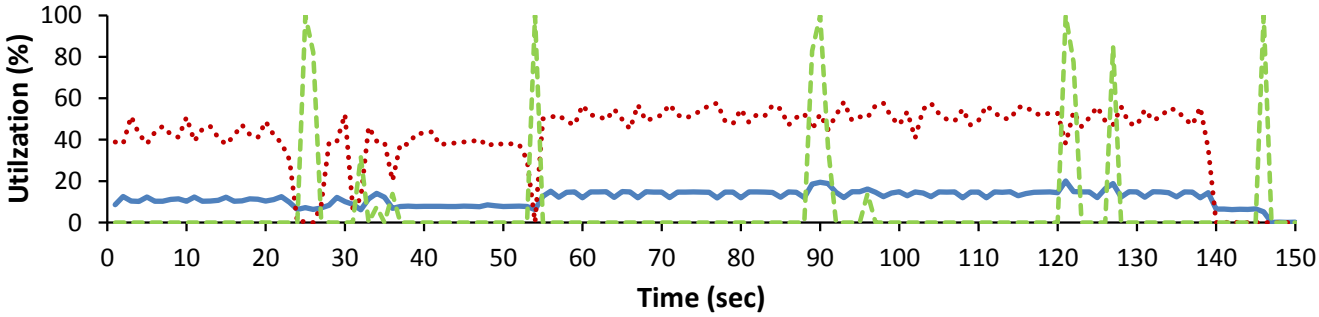
First, both CPU utilization and write I/O bandwidth utilization exhibit alternating peak and plateau cycles. This can be explained by breaking down the data loading process into a sequence of steps which all systems follow. During data loading, blocks of raw data are read sequentially



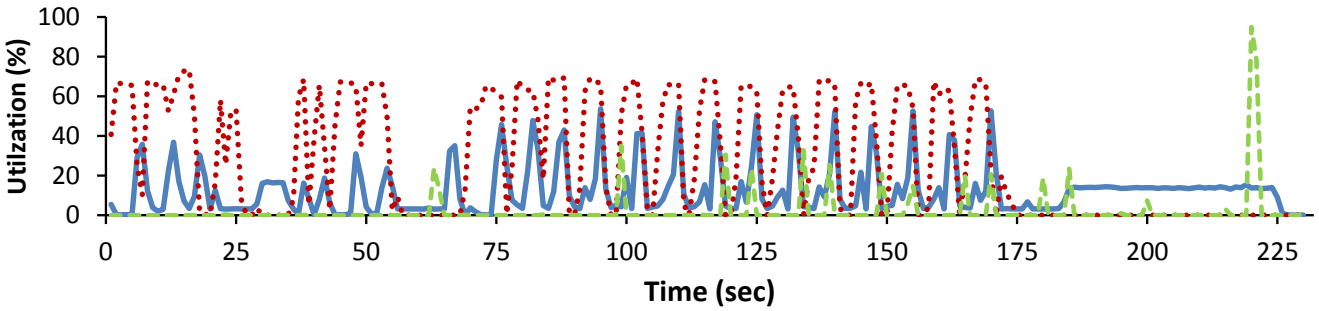
(a) PCOPY



(b) DBMS-A



(c) MonetDB



(d) DBMS-B

Figure 3.3: Different CPU, Read/Write bandwidth utilization per system (Input TPC-H 10GB, source on HDD, database on DAS).

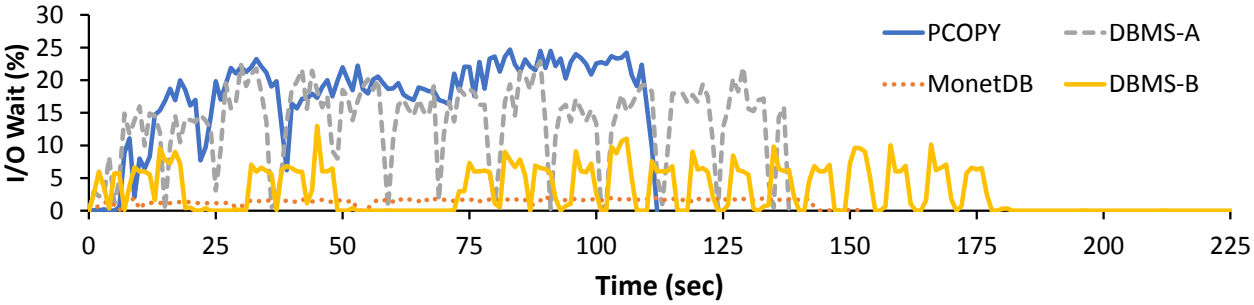


Figure 3.4: I/O Wait per system (Input TPC-H 10GB, source on HDD, database on DAS).

from the input files until all data has been read. Each block is parsed to identify the tuples that it contains. Each tuple is tokenized to extract its attributes. Then, every attribute is converted to a binary representation. This process of parsing, tokenization, and deserialization causes peaks in the CPU utilization. Once the database-internal representation of a tuple is created, the tuple becomes part of a batch of tuples that are written by the DBMS and buffered by the OS. Periodically, these writes are flushed out to the disk. This caching mechanism is responsible for the peaks in write I/O utilization. During these peaks, the CPU utilization in all systems except PCOPY drops dramatically. This is due to the buffer cache in DBMS blocking on a write operation that triggers a flush, thereby stalling the loading process until the flush, and hence the issued I/O operation, complete.

The second pattern that can be observed across all systems is that CPU and I/O resources remain under-utilized. MonetDB exhibits the lowest CPU utilization among all systems. This is due to the fact that it uses one "producer" thread that reads, parses, and tokenizes raw data values, and then N "consumer" threads convert the raw values to a binary format. The parsing and tokenization steps, however, are CPU-intensive and cause a bottleneck on the single producer; CPU utilization is therefore low for MonetDB. The CPU usage for DBMS-B has bursts that are seemingly connected with the system's effort to compress input values, but is otherwise very low. DBMS-B spawns a very high number of threads with low scheduling priority; they get easily pre-empted due to their low priority and they fail to saturate the CPU. PCOPY and DBMS-A have higher CPU usage (61% and 47% on average, respectively) compared to MonetDB and DBMS-B, yet they also fail to fully exploit the CPU resources.

Figure 3.4 illustrates the percentage of time that each DBMS spends waiting for I/O requests to be resolved during each second of execution. Except MonetDB, all other systems spend a non-trivial portion of time waiting for I/O which explains the low CPU utilization. Still, read throughput utilization of various systems in Figure 3.3 barely exceeds 60% even in the best case. This clearly indicates that all systems except MonetDB issue random read I/O requests during parallel data loading which causes high I/O delays and an underutilization of CPU resources.

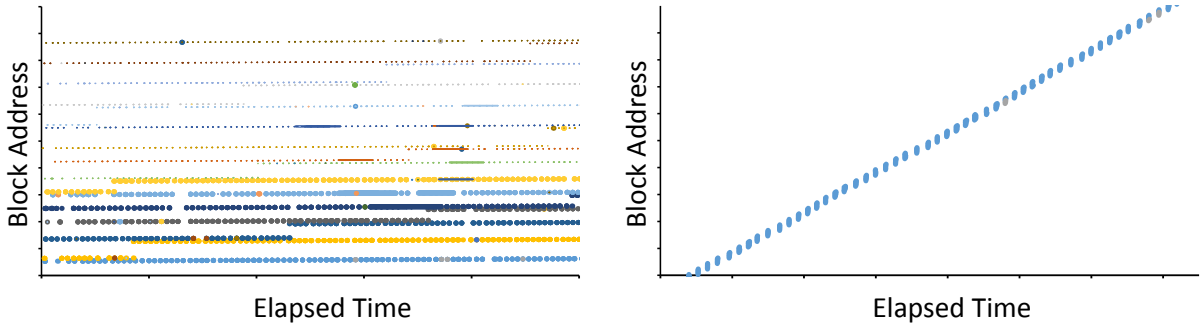
Summary. Contrary to single-threaded loading, which is CPU bound, parallel data loading is I/O bound. Except MonetDB, parallel data loaders used by all systems suffer from poor CPU utilization due to being bottlenecked on random I/O in the input data path. MonetDB, in contrast, suffers from poor CPU utilization due to being bottlenecked on the single producer thread that parses and tokenizes data.

3.3.4 Impact of underlying storage

The previous experiments showed that a typical DBMS setup under-utilizes both I/O bandwidth as well as the available CPUs because of the time it spends waiting for random I/O completion. This section studies the underutilization issue from both a software and a hardware perspective; it investigates i) how the different read patterns of each tested DBMS affect read throughput, and i) how different storage sub-systems affect data loading speed.

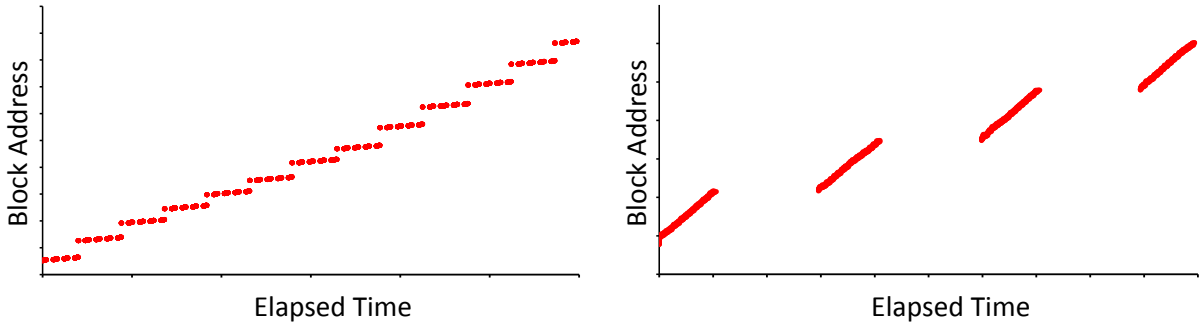
I/O read patterns.

This set of experiments uses as input an instance of the orders table from the TPC-H benchmark with size 1.7GB and records the input I/O pattern of different systems during data loading. We extract the block addresses of the input file and the database file using *hdparm*, and we use *iosnoop* to identify threads/processes that read from/write to a disk. The input data file is logically divided on disk into 14 pieces (13 with size 128 MB and a smaller one with size 8MB). To generate each graph, we take i) the start and end times of the disk requests, ii) the address of the disk from where the reading for the request starts and iii) the size of the operation in bytes. Then, we draw a line



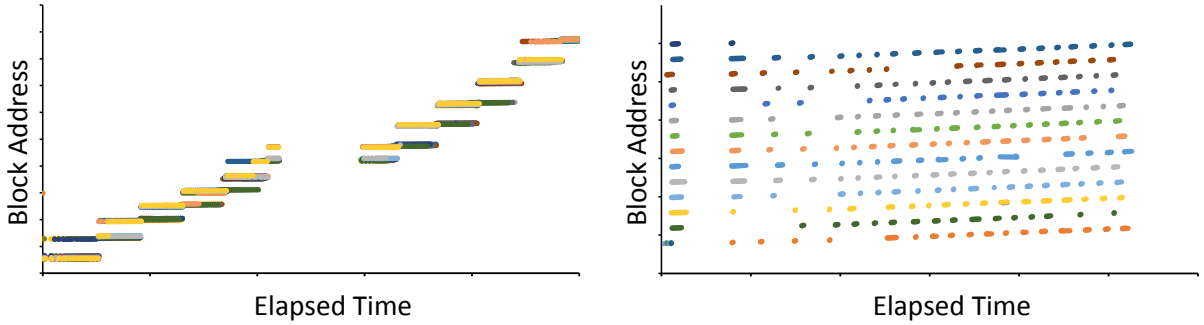
(a) Reading the whole file (b) A chunk (zoom-in)

Figure 3.5: Read pattern for PCOPY (parallel).



(a) Reading the whole file (b) A chunk (zoom-in)

Figure 3.6: Read pattern for MonetDB (both single-threaded and parallel).

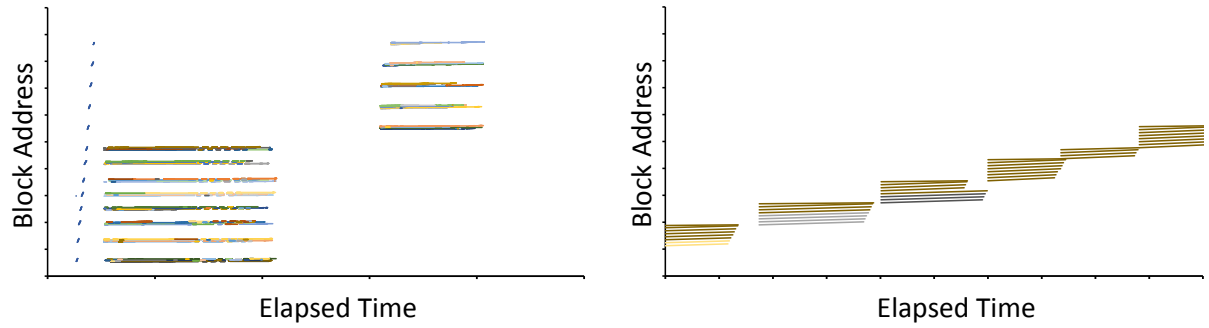


(a) Reading the whole file (b) A chunk (zoom-in)

Figure 3.7: Read pattern for DBMS-A (parallel).

from the point specified by (start time, start address) to the (end time, start address + # of read bytes). There are two kinds of plots: The first one depicts the whole file address space, while the other zooms in the first contiguous LBAs (Logical Block Addresses), further on called chunk. Different colors in the graphs represent distinct processes/threads.

Figures 3.5, 3.6, 3.7, and 3.8 plot the read patterns for PCOPY, MonetDB, DBMS-A, and DBMS-B respectively. All systems operate in parallel mode. MonetDB reads data from disk



(a) Reading the whole file (b) A chunk (zoom-in)

Figure 3.8: Read pattern for DBMS-B (parallel).

sequentially using one thread, while the other systems use multiple concurrent readers (plotted with different colors in the graphs). Figure 3.6(a) depicts the 13 pieces read serially which mirror the 13 main contiguous address spaces of the input file. By looking closer into Figure 3.6(b) we observe that MonetDB reads big contiguous chunks. In total, the plots depict four different read patterns:

- PCOPY exploits 16 threads to read different parts of the file simultaneously. Then, each thread reads consecutive blocks of the assigned part of the file (Figure 3.5(b)).
- MonetDB uses a single "producer" thread to read data and provide each data block to a "consumer" (Figure 3.6).
- DBMS-A accesses a part of the file and processes it using multiple threads. Each thread is assigned its own contiguous area within the accessed chunk (Figure 3.7(b)).
- DBMS-B first samples the whole file with one process, and then it accesses a big chunk of the file (roughly 1GB) in one go. Similar to DBMS-A, each thread is assigned a contiguous portion of a chunk. Contrary to DBMS-A, which reads one part of the file at a time (1 out of the 13 chunks), DBMS-B accesses a wider address space in the same period of time (8 out of the 13 chunks). However, they behave alike on the lower level where each thread reads a contiguous sequence of blocks.

Analyzing each system further reveals that MonetDB uses a sequential read pattern, whereas

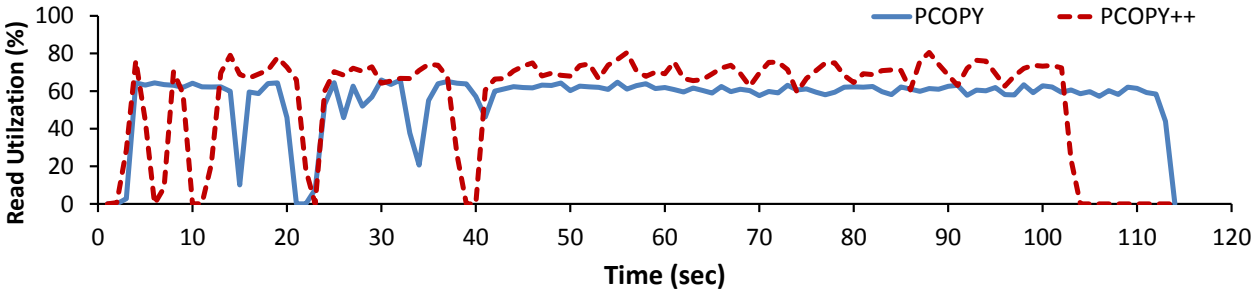


Figure 3.9: Using a serial reader improves the read throughput of PCOPY.

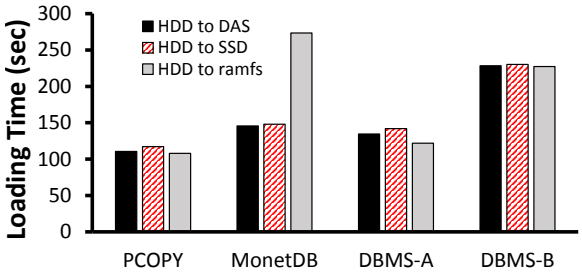


Figure 3.10: Loading 10GB TPC-H: Varying data destination storage with slow data source storage (HDD).

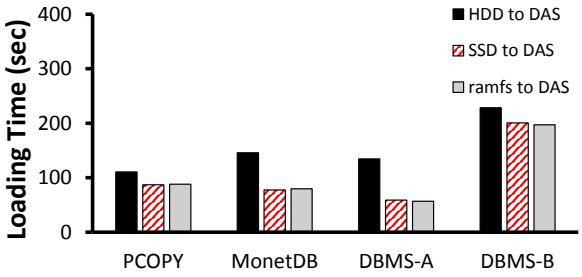


Figure 3.11: Loading 10GB TPC-H: Varying data source storage with DAS data destination storage.

the rest of the systems use parallel readers that read small data chunks from different seeks to the disk and cause random I/O. To gauge which of the two approaches is more beneficial for a system, we implement PCOPY++, which is a variation of PCOPY that uses a single serial reader. As depicted in Figure 3.9, PCOPY++ achieves higher read throughput because it uses a serial data access pattern, which minimizes the costly disk seeks and is also disk-prefetcher-friendly. As a result, PCOPY++ reduced loading time by an additional 5% in our experiments.

Effect of different storage devices.

While sequential accesses are certainly useful for slow HDD-based data sources, it might be beneficial to use multiple readers on data sources that can sustain large random IOPS, like SSD. Thus, another way to eliminate the random I/O bottleneck is to use faster input and output storage media.

To examine the impact of the underlying data storage on parallel loading, we run an experiment where we use as input a 10GB instance of TPC-H and vary the data source and destination storage devices. Figure 3.10 plots the loading time when the slow HDD is the data source storage media,

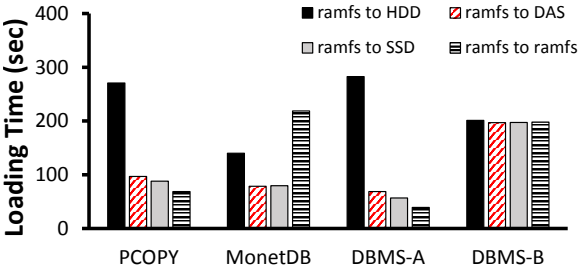


Figure 3.12: Varying data destination storage with fast data source storage (ramfs).

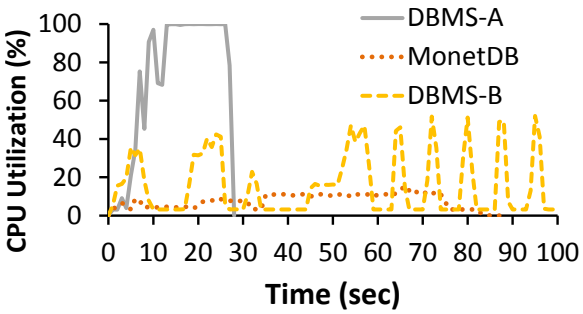


Figure 3.13: CPU Utilization for ramfs-based source and destination.

as in previous experiments, while varying the destination storage used for storing the database. Varying the database storage has little to no impact on most systems despite the fact that ramfs is an order of magnitude faster than DAS. This again shows that all systems are bottlenecked on the source media. The random I/O requests that the DBMSs trigger when loading data in parallel force the HDD to perform many seeks, and therefore the HDD is unable to serve data fast enough.

For MonetDB, the loading time increases when the database resides on ramfs. To clarify this trend, we analyze the performance of MonetDB using VTune. We notice that most of the CPU time is spent in the $internal\_fallocate$ function. MonetDB by default uses the $posix\_fallocate$ function, which instructs the kernel to reserve a space on disk for writes. ramfs, however, lacks support for the $posix\_fallocate$ function and as a result the *glibc* library has to re-create its semantics – a factor that slows down the loading process[1].

Figure 3.11 plots the loading time when we vary the data source storage while using DAS as the data destination storage. Using a faster data source storage accelerates loading for all the systems. Nevertheless, the difference between the configurations that use SSD- and ramfs-based source storage is marginal, which implies that the write performance of DAS eventually becomes a bottleneck for very fast input devices.

To further look into the write bottleneck, Figure 3.12 plots the loading time when we vary the data destination storage while using ramfs – the fastest option – as the data source storage. The observed behavior varies across systems: DBMS-B has little benefit from ramfs because of

1. We reported this behavior to the MonetDB developers, and it is fixed in the current release.

its thread overprovisioning; the numerous low-priority threads it spawns get pre-empted often. For PCOPY and DBMS-A, using ramfs as the data destination storage achieves the best overall performance. Loading time reduces by 1.75× for DBMS-A and 1.4× for PCOPY when ramfs is used as the destination storage compared to DAS. This clearly shows that DAS, despite being equipped with a battery-backed cache for buffering writes, is still a bottleneck to data loading due to the negative impact that dirty data flushing has on the data loading pipeline.

Figure 3.13 shows the CPU utilization for the DBMSs that support parallel loading when source and destination are ramfs. Only DBMS-A reaches 100% CPU utilization, with its performance eventually becoming bound by the CPU-intensive data parsing and conversion tasks.

Summary. The experiments demonstrate the effect of the interaction between the DBMS and the underlying storage subsystem, both from a software and a hardware perspective. Our analysis showed that the way in which DBMS issue read requests and the degree of parallelism they employ has an effect on the read throughput achieved.

Writes are also challenging for parallel loading because multiple writers might increase I/O contention due to concurrent writes. In fact, slow writes can have a bigger impact on the data loading performance than slow reads, as slow flushing of dirty data stall the data loading pipeline. Thus, it is important to use storage media that perform bulk writes quickly (using write caches or otherwise) to limit the impact of this problem.

Finally, for the fastest combinations of data source and destination storage, which also allow a high degree of IOPS, there are DBMSs that become CPU-bound. However, on measuring the storage bandwidth they use in that case (250MB/s), we found that they will still be unable to fully utilize modern storage devices, like PCIe SSD, indicating that the data loading code path needs to be optimized further to reduce loading time.

3.3.5 Hitting the CPU wall

Data loading is a CPU-intensive task – a fact that becomes apparent after using the fastest data source and destination storage combination. This section presents a CPU breakdown analysis

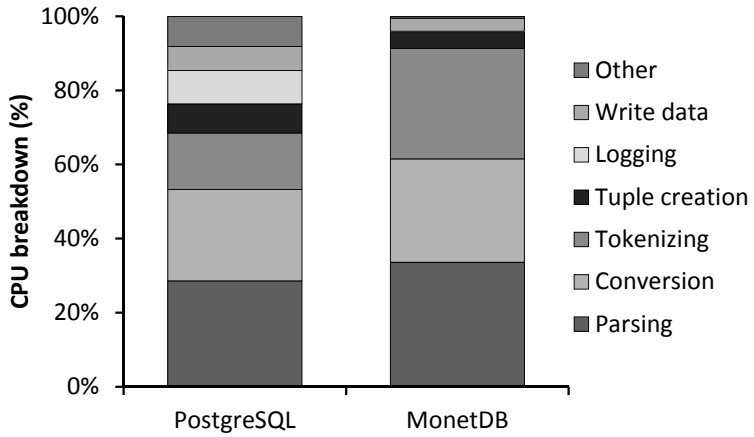


Figure 3.14: CPU breakdowns for PostgreSQL and MonetDB during data loading of integers. Most of the time is spent on parsing, tokenizing, conversion and tuple creation.

using VTune for the two open-source systems (PostgreSQL and MonetDB). For this experiment, we use as input a custom dataset of 10GB which contains 10 columns with integer values and we examine the CPU overhead of loading this data file. Figure 3.14 shows the results; we group together the tasks corresponding to the same functionality for both systems based on the high-level steps described above.

Even though PostgreSQL is a row-store and MonetDB is a column-store, both databases go through similar steps to perform data loading. Both systems spend the majority of their processing time to perform the parsing, tokenizing, and data type conversion steps (69% for PostgreSQL and 91% for MonetDB). Overall, data loading is CPU-intensive; however, parsing and tokenizing data from a file and generating tuples can be decomposed into tasks of smaller size. These tasks do not require any communication (i.e., there are no dependencies between them), thus they are ideal candidates for parallel execution. Modern DBMS are based on this property to provide parallel loading. The CPU cost for parsing and tokenizing can also be further reduced if the general-purpose file readers used by the DBMS for bulk loading are replaced by custom, file-specific readers that exploit information regarding the database schema [33] (e.g., number of attributes per tuple, datatype of each attribute). Finally, PostgreSQL spends 8% of the time creating tuples and 9% of the time for logging related tasks. On the other hand, MonetDB spends 5% of the loading time on the same steps.

3.3.6 *Data loading in the presence of constraints*

Enforcing integrity constraints adds overheads to the data loading process. An established rule of thumb claims that populating the database and verifying constraints should be separated and run as two independent phases. Following this adage, database administrators typically drop all preexisting primary and foreign key constraints, load data, and add constraints back again to minimize the total data loading time. This section investigates the performance and scalability implications of primary-key (PK) and foreign-key (FK) constraint verification, and tests conventional knowledge.

Primary key constraints.

Figure 3.15 shows the total time taken to load the TPC-H SF-10 dataset in the single-threaded case when a) no constraints are enabled, b) primary key constraints are added before loading the data (*"Unified"* loading and verification), c) primary key constraints are added after loading the data (*"Post"*-verification). All the experiments use an HDD as the input source and DAS to store the database. We omit results for DBMS-B because it lacks support for constraint verification, and for MonetDB because its *Unified* variation enforces a subset of the constraints that this section benchmarks[2]. We consider PK constraints as specified in the TPC-H schema.

Figure 3.15 shows that for both DBMS-A and PostgreSQL, enabling constraints before loading data is 1.16× to 1.82× slower than adding constraints after loading. The traditional rule of thumb therefore holds for single-threaded data loading. A natural question that arises is whether parallel loading techniques challenge this rule of thumb.

DBMS-A supports explicit parallelization of both data loading and constraint verification phases. Thus, DBMS-A can parallelize the *Unified* approach by loading data in parallel into a database which has PK constraints enabled, and parallelize the *Post* approach by performing parallel data loading without enabling constraints and then triggering parallel PK constraint verification. PostgreSQL is unable to independently parallelize constraint verification. Thus, the *Post* approach for

2. `https://www.monetdb.org/Documentation/SQLreference/TableIdentityColumn`

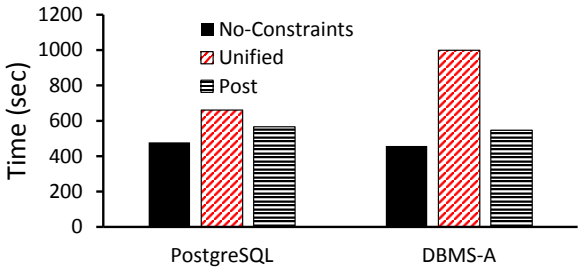


Figure 3.15: Single-threaded loading in the presence of PK constraints.

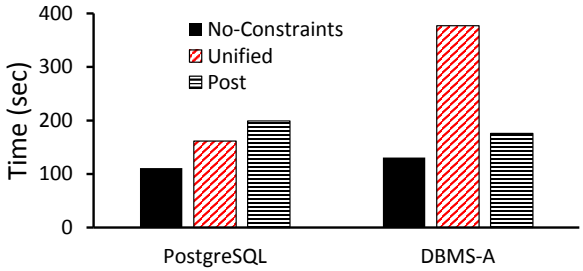


Figure 3.16: Parallel loading in the presence of PK constraints.

PostgreSQL performs parallel data loading (using PCOPY) and single-threaded constraint verification.

Figure 3.16 shows the total time taken to load the database using 16 physical cores. Comparing Figures 3.15 and 3.16, the following observations can be made:

- Parallel loading reduces the execution time of both *Unified* and *Post* approaches for both DBMS (4.08× under PCOPY, 2.64× under DBMS-A).

- The conventional rule of thumb is no longer applicable to both DBMS shown in Figure 3.16: While *Post* provides a 2.14× reduction in loading time over *Unified* under DBMS-A, the trend reverses under PostgreSQL as *Unified* outperforms *Post* by 19%. The reason is that the *Post* configuration for PostgreSQL performs single-threaded constraint verification while *Unified* parallelizes loading and constraint verification together as a unit.

- Despite outperforming *Post*, *Unified* is still 1.45× slower than the *No-Constraints* case for PostgreSQL. Similarly, *Post* is 1.34× slower than *No-Constraints* for DBMS-A. The PostgreSQL slowdown is due to a cross interaction between write-ahead logging and parallel index creation; Figure 3.17 shows the execution time of *No-Constraints* and *Unified* over PostgreSQL when logging is enabled/disabled. DBMS-A lacks support for an explicit logging deactivation, therefore it is not presented. Logging has minimal impact in the absence of constraints, but it plays a major role in increasing execution time for *Unified*. In the presence of a PK constraint, PostgreSQL builds an index on the corresponding attribute. As multiple threads load data into the database, the index is updated in parallel, and these updates are logged. Profiling

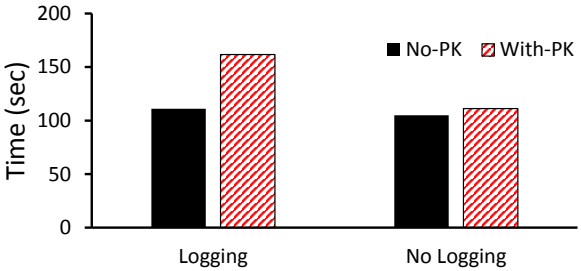


Figure 3.17: Effect of logging in PostgreSQL.

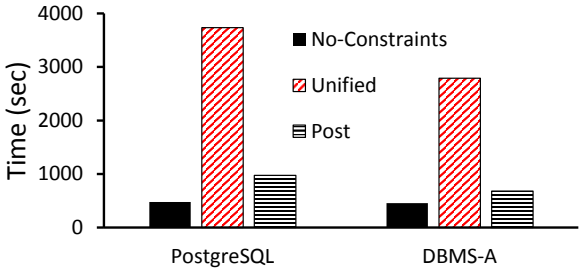


Figure 3.18: Single-threaded loading in the presence of FK constraints.

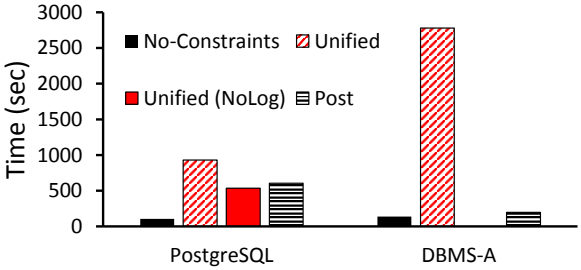


Figure 3.19: Parallel loading in the presence of FK constraints.

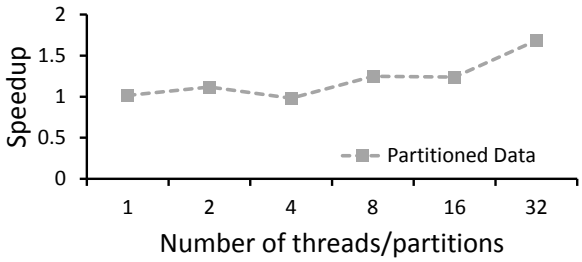


Figure 3.20: Re-organizing input to facilitate FK validation.

revealed that this causes severe contention in the log manager as multiple threads compete on latches.

Foreign key constraints.

Figure 3.18 shows the time taken to load the TPC-H dataset in the single-threaded case when both PK and FK constraints are enabled. Comparing Figures 3.15 and 3.18, it is clear that FK constraints have a substantially larger impact on loading time compared to PK constraints. *Unified* is 7.8× and 6.1× slower under PostgreSQL and DBMS-A when the systems perform FK checks as well, compared to 1.38× and 2.1× when they only perform PK checks.

Figure 3.19 shows the time it takes to enforce FK constraints in parallel. Unlike the PK case, each approach tested benefits differently from additional parallelism. While the *Unified* approach benefits from a 4× reduction in loading compared to the single-threaded case for PostgreSQL, it fails to benefit at all for DBMS-A. However, the *Post* approach benefits from parallelism under both systems; DBMS-A and PostgreSQL achieve a 3.45× and 1.82× reduction in loading time

respectively. In addition, similarly to the PK case, disabling logging has significant impact on loading time: *Unified (No Log)* is 1.73× faster than the logging approach.

The conventional rule of enforcing constraints after data has been loaded is again only applicable under specific scenarios: Under DBMS-A, *Post* is indeed the only approach to scale and thus, the rule of thumb holds. Under PostgreSQL, *Post* lags behind *Unified (NoLog)* by 1.12×.

In total, adding constraint verification to the loading process increases time by 1.43× under DBMS-A and 5.1× under PostgreSQL for the parallel case. We profiled PostgreSQL with logging disabled to identify the root cause of performance drop; latching was the reason. PostgreSQL implements foreign keys as triggers, therefore each record insertion into a table fires a trigger which performs a select query on the foreign table to verify that the insertion does not violate the FK constraint. Such selections acquire a Key-Share lock on the target record to preserve consistency. As multiple threads load data into the database, they compete over the latch (spin lock) that must be acquired as a part of Key-Share locking. This contention causes performance deterioration.

Reducing contention. One way to reduce contention is to modify the DBMS by implementing more scalable locks. An alternative that this study adopts is to avoid contention by re-organizing the input. Specifically, we partition the raw data of the "child" table so that any records having an FK relationship with the same "parent" record are grouped together within a partition. Therefore, two threads loading two different partitions will never contend over latches while acquiring Key-Share locks.

Figure 3.20 shows the speedup achieved when loading the TPC-H lineitem table using the *Unified* approach over partitioned input data, compared to the case when the input is not partitioned. In the partitioning case, we split lineitem in N chunks, one per thread, such that two records in different partitions will never refer to the same parent record in the supplier table. In cases of low contention (1-4 threads), speedup is marginal. When multiple threads are used to load the input data, the input partitioning strategy yields up to a 1.68× reduction in loading time.

Summary

This section showed that enforcing constraints during loading (i.e., the *Unified* approach) offers performance which is competitive to applying constraints after loading the data (the *Post* approach), and even outperforms it in almost all cases. Besides the performance benefits, the *Unified* approach enables a DBMS to be kept online while loading data, compared to the *Post* approach that requires the DBMS to be taken offline. Thus, administrators should be wary of these trade-offs to optimize bulk loading.

In addition, it is time to refactor the loading pipeline in traditional DBMS. The loading pipeline is typically implemented over the same code base that handles single-record insertions and updates, therefore parallelizing loading externally using several client threads results in latch contention in several DBMS subsystems like the lock and log managers. Instead, DBMS should make bulk loading a first-class citizen and develop a code path customized for loading. With such changes, the loading time can be substantially reduced further even in the presence of constraints, all while the DBMS remains online during data loading.

CHAPTER 4

DATA TRANSFORMATION AND MIGRATION

4.1 Introduction

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. Once the data is loaded into one of the DBMSs, we have to enable data mobility. There is a need to transfer data between systems easily and efficiently. We present how a rapid data transfer can be achieved by taking advantage of recent advancement in hardware and software. We 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.

4.2 Data Migration Framework

A data migration and transformation typically involves four steps [3]. 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 4.1 highlights the various components in Portage to implement these four steps.

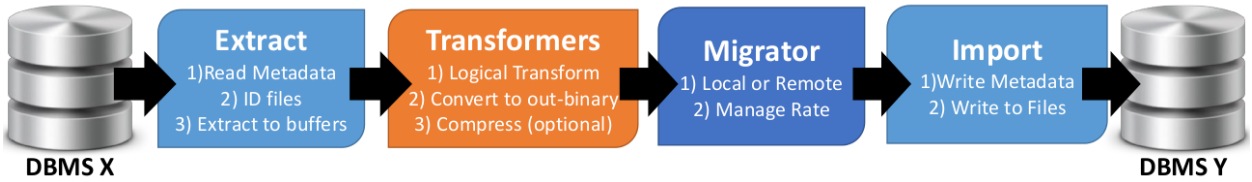


Figure 4.1: Framework components for Portage.

4.3 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 [26]. The three methods are presented in Fig. 4.2 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. 4.2, CSV migration (top yellow line) is very slow with respect to other methods. Here, we migrate waveform data from the MIMIC-II dataset [5] 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. 4.2). 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 in-

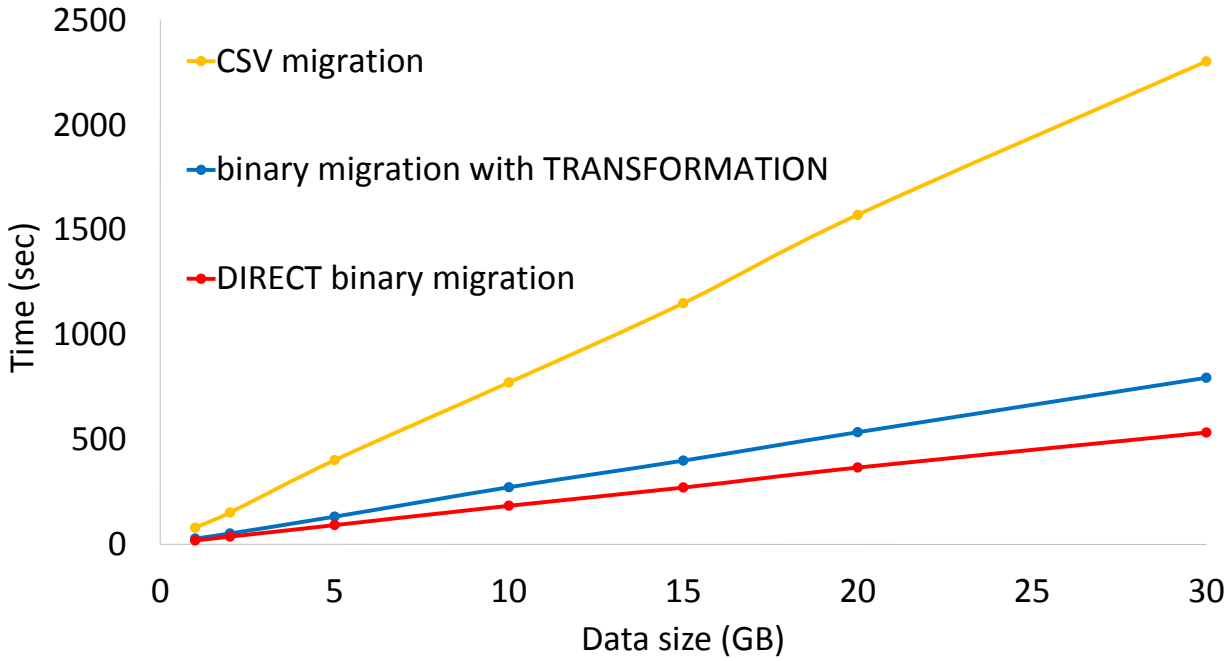


Figure 4.2: Data migration from PostgreSQL to SciDB (3 approaches).

termediate 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. 4.2) 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. 4.1.

The CSV migration from the Fig. 4.2 is split into CSV export and CSV load (two first yellow columns in Fig. 4.3). The CSV loading is slow because of parsing and deserialization [34].

In the second case, we use our data migrator as an external connector for intermediate binary transformation. The three blue columns in Fig. 4.3 correspond to the blue line in Fig. 4.2. In this case, the export is slower than loading. We argue that this unexpected situation is caused by

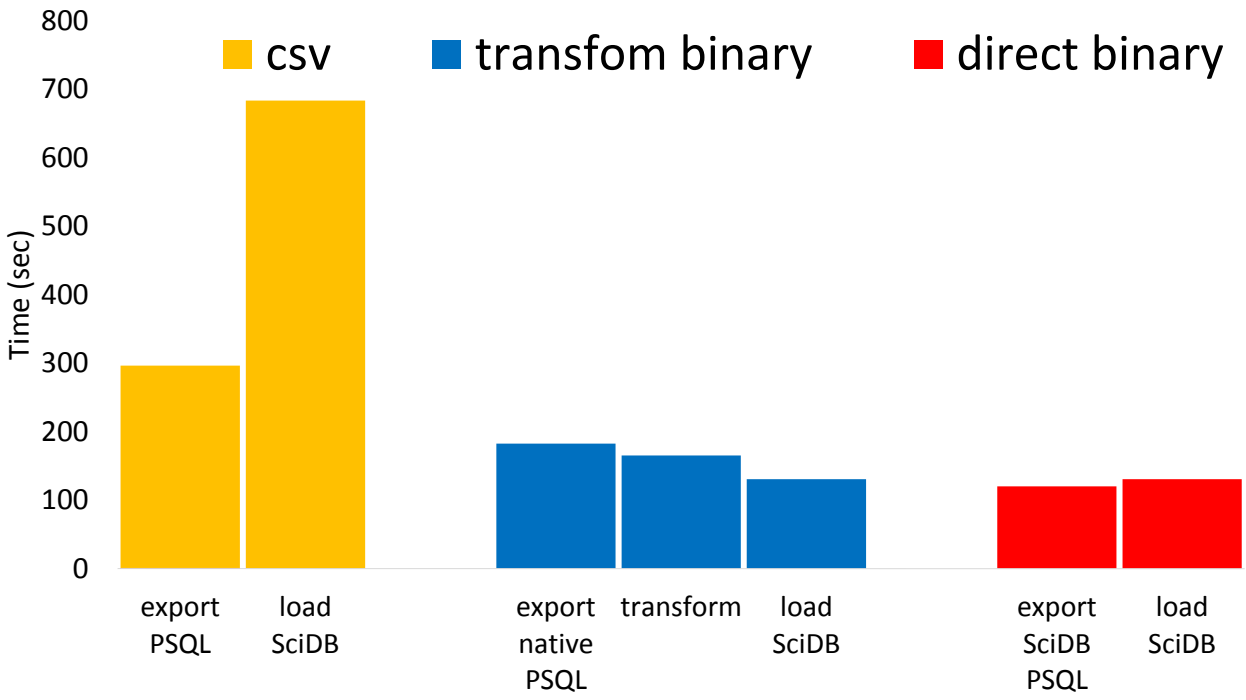


Figure 4.3: Breakdown of data migration from PostgreSQL to SciDB (flat array) for waveform data of size 10 GB (PSQL denotes PostgreSQL).

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. 4.3). 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 comparable performance we omit the results.

4.4 The Optimization of Data Migration

We can optimize the migration and transformation process using many techniques, such as (aforementioned) binary formats, task-level parallelism with many exporters and loaders, SIMD operations (data-level parallelism), dynamic compression, and adaptive execution method. In this section, we provide more details about these techniques.

4.4.1 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

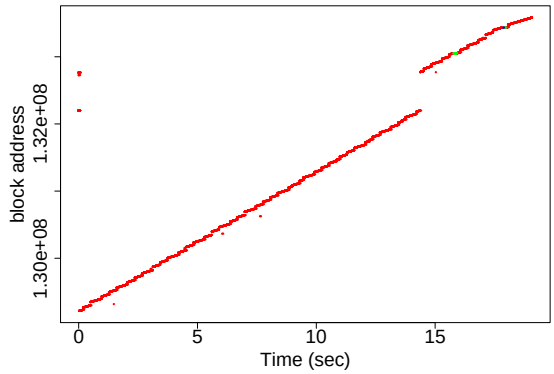


Figure 4.4: Single-threaded export from PostgreSQL (current version).

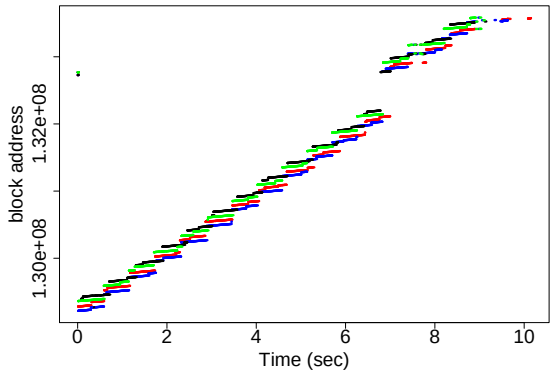


Figure 4.5: Parallel export from PostgreSQL (modified version).

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) [14]. Fig. 4.4 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. 4.5 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).

Parallel Migration from S-Store

The streaming main-memory database S-Store combines the best of two worlds: fast binary export and parallel processing [27, 38]. 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.

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

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

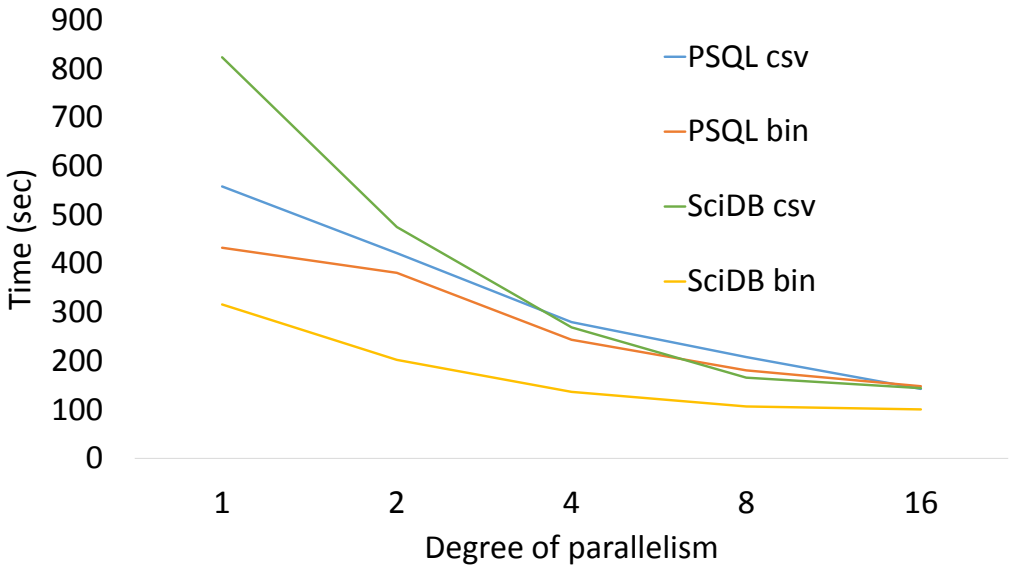


Figure 4.6: Data migration from S-Store to PostgreSQL and SciDB (data from TPC-C [13] benchmark of size about 10 GB).

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. 4.6. 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.

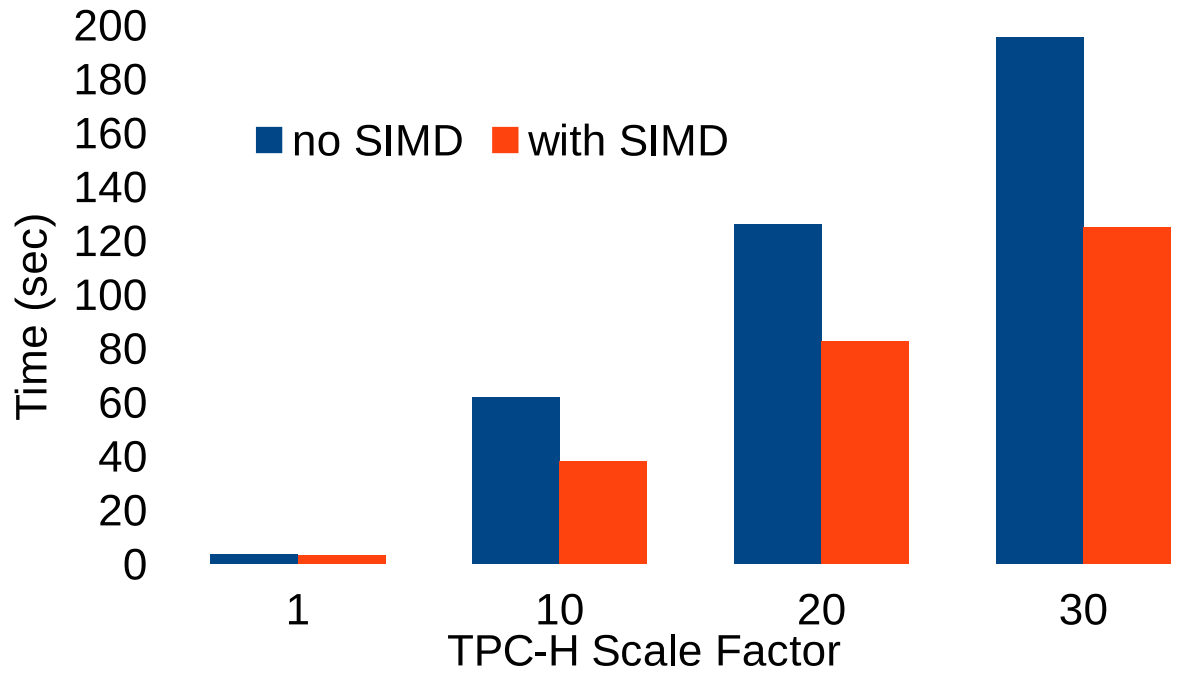


Figure 4.7: The execution time for parsing (finding new lines) with and without SIMD. The achieved speed-up is about 1.6X. (during loading of the *lineitem* table from the TPC-H [14] benchmark).

4.4.2 SIMD Operations

We investigated how SIMD (SSE 4.2) can be used for faster CSV parsing during data loading to PostgreSQL [39]. Parsing in PostgreSQL is split in two phases: (1) extract lines, (2) identify fields. First, it scans through the input file and searches for new lines. Once a new line is found, it is copied to a separate line buffer and sent to the next stage which finds fields (within the same line buffer). It occurs that the most time consuming part of the parsing process is extracting new lines. The process of finding lines for a plain text and CSV format is fused into a single block of code whereas there is a separate processing of fields for both plain text and CSV format. A new buffer is prepared for each line during parsing. The size of the line is not known beforehand (as there can be fields of varying size), thus we have to check if the new characters to be added do not exceed the size of the buffer and re-size it from time to time.

Finding lines requires many comparisons for each character. Thanks to SIMD, we skip many if-else comparisons for each character, as they are executed much faster using the *equal any* pro-

gramming model from SSE 4.2. Moreover, skipping many characters means avoiding many checks for the line buffer and coping more characters to the buffer at a time. For line parsing, we keep the SIMD registers full - we always load and process 16 characters in a SIMD register until the end of the file. In our micro-benchmark, we changed the parsing of lines in PostgreSQL and measured directly its performance while loading the *lineitem* table from the TPC-H benchmark. The achieved speed-up for finding new lines with SIMD is about 1.6X (Fig. 4.7)[2].

A similar approach of using SIMD for finding fields (especially numeric or short strings) within a line does not exhibit the same performance boost as the line parsing. The main reason is that the field delimiters are much more frequent than new line delimiters. For each character, during field parsing, we only check if it is a field separator or a quote so there are not as many branches as in the case of parsing the lines. SIMD registers cannot be kept full for finding the fields. The line in which we search for fields is already in its own buffer and we do not execute any memory operations, in contrast to line parsing.

4.4.3 Adaptive migration

Accumulo is a distributed, persistent, multidimensional sorted map, based on the BigTable system [22], with fine-grained access control. Accumulo adopts diametrically different architecture than other systems which we discussed, and is an example of a NoSQL system [28]. Data must be sorted and this can be done externally using MapReduce job [24], 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

2. The experiment was conducted on Intel Core i7-5557U CPU @ 3.1GHz (4 physical cores), 32 KB L1, 256 KB L2, 4 MB L3 cache shared, and 16 GB RAM, running Ubuntu Linux 14.04 LTS (64bit) with kernel version 3.16.0-38. 250GB SATA SSD: bandwidth about 250 MB/sec for writes and 500 MB/sec for reads.

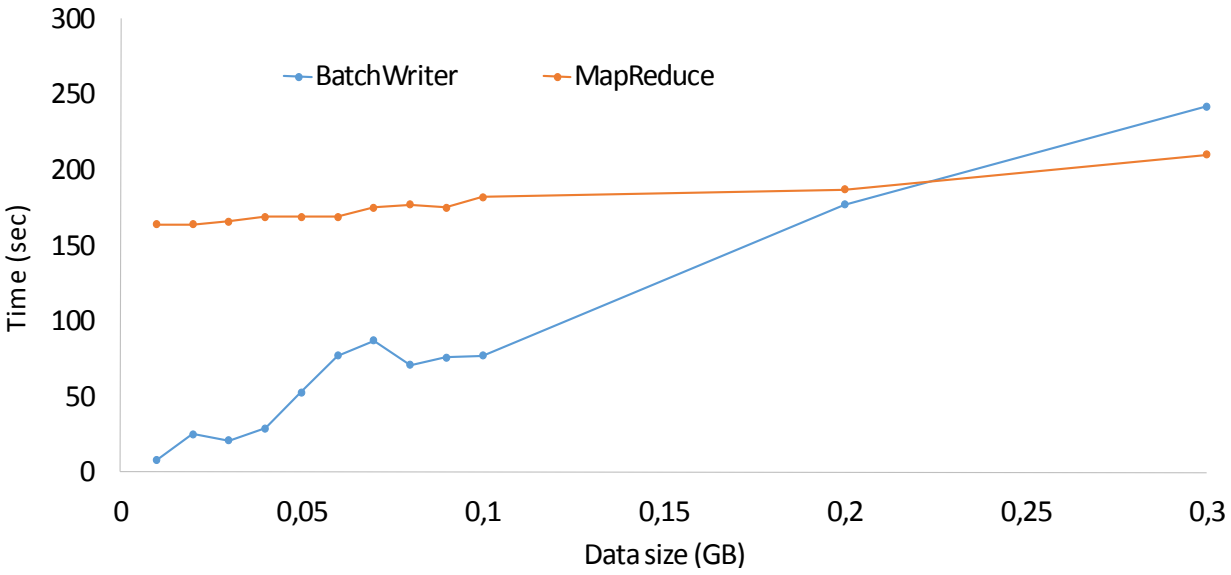


Figure 4.8: From PostgreSQL to Accumulo (Small data size).

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 [14]. Fig. 4.8 and Fig. 4.9 show 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. 4.8) because of the overheads of

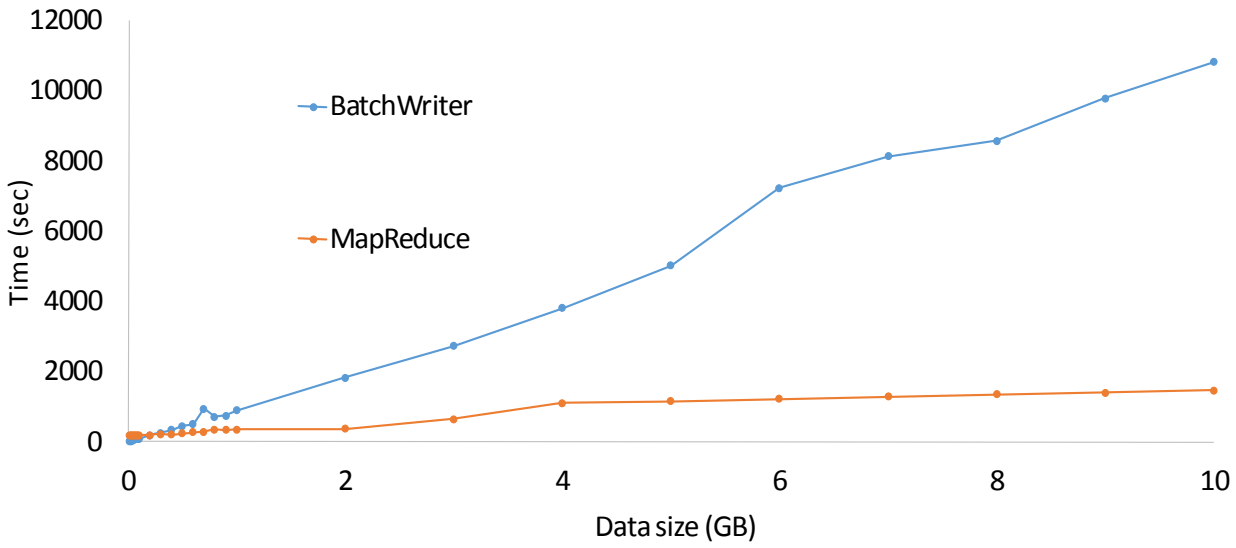


Figure 4.9: From PostgreSQL to Accumulo (Big data size).

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.

4.4.4 Data compression

We investigate if the migration process can benefit from compression when data is transferred via network. The first requirement for this process, to bring any benefit, is to ensure that the speed of the migration is greater than the network throughput. We use the waveform data (with two integer columns and a single column of type double) from the MIMIC II dataset [5] of size about 10 GB. The migration is pipelined with 5 stages: data export, compression, transfer via network, decompression and data loading. We are able to achieve about 204 MB/sec throughput for parallel binary export from PostgreSQL for the level of parallelism set to four, which is executed in about 50 sec for the considered dataset. On the other end of the pipeline, we manage to obtain the throughput of about 136 MB/sec for the binary parallel data loading to SciDB for the level of

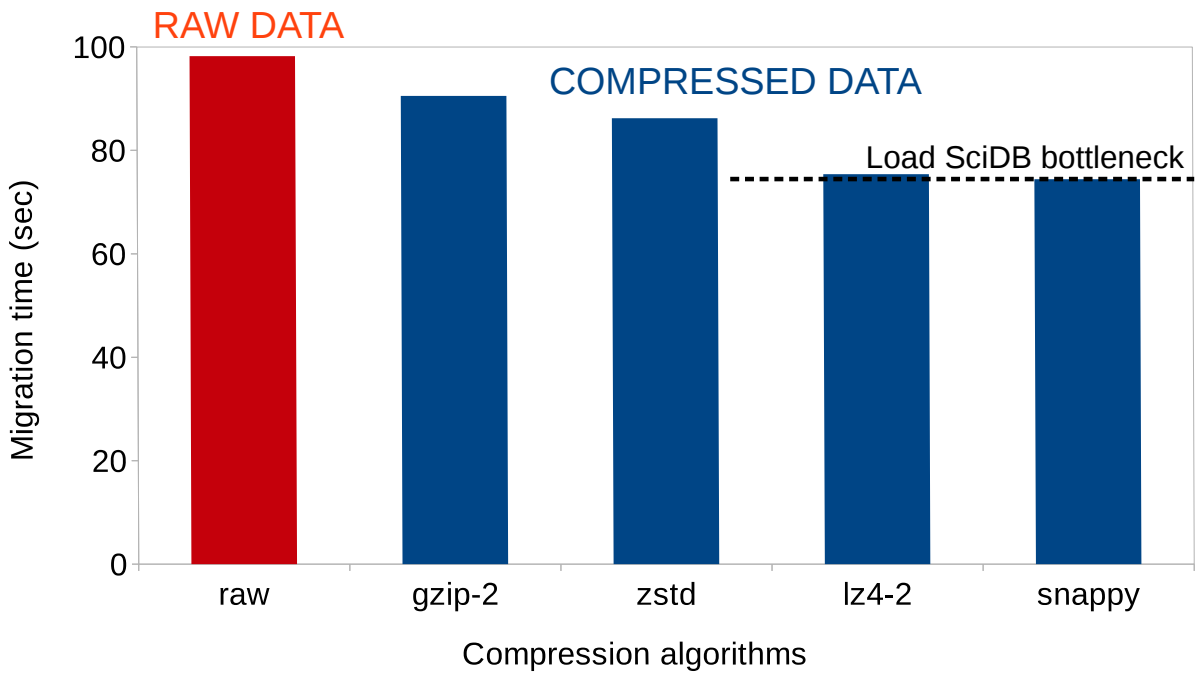


Figure 4.10: Migration time from PostgreSQL to SciDB via network for different compression algorithms.

parallelism set also to four and the whole dataset is loaded in about 75 sec. The data is moved between two machines belonging to a small cluster of 5 nodes, each of which has 4 cores and is connected via 1 Gbit/sec network. The migration time for raw data (without any compression) is of speed about 104 MB/sec (and takes about 98 sec), which reaches the practical network throughput. The application of gzip algorithm [2] for the compression and decompression stages (with the compression level set to two from range 1-9) gives us a slight improvement and reaches about 113 MB/sec (95 sec) and is equal to the speed of gzip compression (an exclusive execution of the compression stage in the pipeline). Next, zstd (z-standard), the new compression algorithm from Facebook [15], is a bit faster than gzip but is bottlenecked on the decompression stage of the pipeline and allows us to achieve the throughput of about 119 MB/sec (86 sec). Finally, lz4 (set to the level of two) [4] and snappy [11] compression algorithms are fast and efficient enough to decrease the migration time to the maximum ingest speed measured for SciDB: about 136 MB/sec (the migration takes about 75 seconds).

Overall, for the data transfer via network, we are able to accelerate the process by about 25%

(to reach its limit on the loading stage of the pipeline) as a result of application of the lightweight compression algorithms.

CHAPTER 5

RELATED RESEARCH

As the growth of collected information has turned data loading into a bottleneck for data analysis tasks, researchers from industry and academia have proposed ideas to improve the data loading performance and in some cases to enable data processing without any requirement for data loading. There were also attempts to improve data migration and new polystore systems emerged. We briefly review the body of related work in this section.

Numerous approaches examine ways to accelerate data loading. Starting from general-purpose approaches, the authors of [19] introduce the idea of partitioning the input data and exploiting parallelism to load the data partitions faster. Instant loading [39] presents a scalable bulk loader for main-memory systems, designed to parallelize the loading phase of HyPer [34] by utilizing vectorization primitives to load CSV datasets as they arrive from the network. It applies task- and data- parallelization on every stage of the data loading phase to fully leverage the performance of modern multi-core CPUs and reduce the time required for parsing and conversion. Disk-based database systems can also benefit from such vectorized algorithms to reduce the CPU processing cost. Sridhar et al. [41] 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 the loading performance the authors apply a number of techniques: (i) asynchronous parallel I/O for read and write operations, (ii) forcing every new load to begin at a page boundary and using private buffers to create database pages to eliminate lock costs, (iii) using a minimal WAL log, and (iv) forcing worker threads to check constraints on column values. Other related work offer specialized, domain-specific solutions: The authors of [18, 44] consider the problem of bulk loading an object-oriented DBMS, and focus on issues such as inter-object dependencies. Bulk loading for specialized indexing structures is also an active research area [20, 40]. Finally, the authors of [21] put together a parallel bulk loading pipeline for a specific repository of astronomical data.

Motivated by the blocking nature of data loading, vendor lock-in concerns, and the proliferation

of different data formats, many researchers advocate launching queries directly over raw data. Multiple DBMS allow SQL queries over data files without loading them a priori. Such approaches, such as the External tables of Oracle and the CSV Engine of MySQL, tightly integrate data file accesses with query execution. The integration happens by *linking* a data file with a given schema and by utilizing a scan operator with the ability to access data files and create the internal structures (e.g., tuples) required by the query engine. Still, external tables lack support for advanced database features such as DML operations, indexes or statistics.

Speculative loading [23] proposes an adaptive loading mechanism to load data into the database when there are available system resources (e.g., disk I/O throughput). Speculative loading proposes a new database physical operator (SCANRAW) that piggybacks on external tables. Adaptive loading [30] was presented as an alternative to full a priori loading. The main idea is that any data loading operations happen adaptively and incrementally during query processing and driven by the actual query needs. NoDB [17] adopts this idea and extends it by introducing novel data structures to index data files, hence making raw files first-class citizens in the DBMS and tightly integrating adaptive loads, caching, and indexing. RAW and Proteus [33, 32] further reduce raw data access costs by generating custom data access paths at runtime via code generation.

Data vaults [31] aim at a symbiosis between loaded data and data stored in external repositories. Data vaults are developed in the context of MonetDB and focus on providing DBMS functionality over scientific file formats, emphasizing on array-based data. The concept of just-in-time access to data of interest is further extended in [35] to efficiently handle semantic chunks: large collections of data files that share common characteristics and are co-located by exploiting metadata that describe the actual data (e.g., timestamps in the file names).

The live datastore transformation proposed in [42] 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.

While our focus is on building efficient custom end-points for a set of specific database engines, PipeGen [29] explores the use of a general-synthesized approach to automatically build data pipes based on existing unit tests for import/export. PipeGen enhances DBMSs written in Java, optimizes CSV and JSON formats, and uses Arrow as an intermediate format for network transfer. In our body of work on Portage, we use named pipes to avoid data materialization via the disk and exploit native binary formats.

MISO [36] 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 the thesis.

CHAPTER 6

CONCLUSIONS

Our experimental study and the accelerators for data loading, transformation and migration provide a possibility of faster insight to the ever-increasing data volume.

Data loading is an upfront investment that DBMS have to undertake in order to be able to support efficient query execution. Given the amount of data gathered by applications today, it is important to minimize the overhead of data loading to prevent it from becoming a bottleneck in the data analytics pipeline.

Our study evaluates the data loading performance of four popular DBMSs along several dimensions with the goal of understanding the role that various software and hardware dimensions play in reducing the data loading time of several application workloads. Our analysis shows that data loading can be parallelized effecively, even in the presence of constraints, to achieve a $10\times$ reduction in loading time without changing the DBMS source code. However, in order to achieve such improvement, administrators need to be cognizant of the fact that conventional wisdom that applies to single-threaded data loading might no longer hold in for parallel loading.

We find that most of the systems are not able to fully utilize the available CPU resources or saturate available storage bandwidth. This suggests there is still room for improvement of the data analysis pipeline. This led us to investigating accelerators for the data movement.

Once data is loaded into a DBMS, there is a need to provide a performant data mobility. Data migration is an inherent part of polystore systems, however, the data movement is costly and slow. We show that 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 binary migration can be much faster than the CSV migration, especially when not all of the CPU cycles can be used exclusively for the loading process or when there is a skew in the data and it is not evenly divided into partitions. The SIMD processing is beneficial for parsing of the text data (e.g. CSV) and shows improvement primarily for finding new line delimiters. Further, the data compression speeds up the migration process if the compression and decompression stages are not the slowest parts of

the data migration pipeline and there is some redundancy in the data. The adaptive approach to migration is based on selecting the fastest data export or loading methods for a given amount of data, which speeds up the migration process, however, it requires us to identify the cut-off point in which we change the migration method. This selection methods has to be tuned so that the system is robust and the migration time does not change dramatically for a relatively small change in the size of the processed data.

We believe that our findings and proposed acceleration methods can benefit database:

- **users** - as they will be able to speed-up their data analysis pipeline;
- **administrators** - who will be able to harness all the hardware resources for faster data delivery to the users;
- **designers and engineers** - who could refactor the database bulk loading/export and provide a dedicated code base for the operations; support a concise, compressed and common binary format for loading/export; and finally to build indexes efficiently during loading so that the data movement does not require a DBMS down time.

REFERENCES

[1] Accumulo. http://accumulo.apache.org/.

[2] gzip compression algorithm. https://gzip.org.

[3] Just-in-time Data Transformation and Migration in Polystores. http://istc-bigdata.org/index.php/just-in-time-data-transformation-and-migration-in-polystores/.

[4] lz4 compression algorithm. https://github.com/lz4/lz4.

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

[6] MonetDB. http://www.monetdb.org/.

[7] PostgreSQL. http://www.postgresql.org/.

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

[9] SciDB. http://www.paradigm4.com/HTMLmanual/14.12/scidb_ug/.

[10] SkyServer project. http://skyserver.sdss.org.

[11] Snappy compression algorithm. https://github.com/google/snappy.

[12] Symantec Enterprise. http://www.symantec.com.

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

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

[15] Zstandard (zstd) compression algorithm. https://github.com/facebook/zstd.

[16] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009.

[17] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.

[18] Sihem Amer-Yahia and Sophie Cluet. A declarative approach to optimize bulk loading into databases. *ACM Trans. Database Syst.*, 29(2):233–281, June 2004.

[19] Tom Barclay, Robert Barnes, Jim Gray, and Prakash Sundaresan. Loading databases using dataflow parallelism. 23:72–83, 12 1994.

[20] Jochen Van den Bercken and Bernhard Seeger. An evaluation of generic bulk loading techniques. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 461–470, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[21] Y. Dora Cai, Ruth Aydt, and Robert J. Brunner. *Optimized data loading for a multi-terabyte sky survey repository*, volume 2005. 2005.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[23] Yu Cheng and Florin Rusu. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1287–1298, New York, NY, USA, 2014. ACM.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[25] David J. DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1255–1266, New York, NY, USA, 2013. ACM.

[26] Adam Dziedzic, Jennie Duggan, Aaron J. Elmore, Vijay Gadepally, and Michael Stonebraker. Bigdawg: a polystore for diverse interactive applications. In *IEEE Viz Data Systems for Interactive Analysis (DSIA)*, October 2015.

[27] Adam Dziedzic, Aaron J. Elmore, and Michael Stonebraker. Data transformation and migration in polystores. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–6, 2016.

[28] Adam Dziedzic and Jan Mulawka. Analysis and comparison of NoSQL databases with an introduction to consistent references in big data storage systems. In *Proc.SPIE*, Warsaw, Poland, March 2014.

[29] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. Pipegen: Data pipe generator for hybrid analytics. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '16, New York, NY, USA, 2016. ACM.

[30] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. Here are my queries. Where are my results? In *CIDR*, 2011.

[31] Milena Ivanova, Yağiz Kargin, Martin Kersten, Stefan Manegold, Ying Zhang, Mihai Datcu, and Daniela Espinoza Molina. Data vaults: A database welcome to scientific file repositories. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 48:1–48:4, New York, NY, USA, 2013. ACM.

[32] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, August 2016.

[33] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[34] Alfons Kemper and Thomas Neumann. Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[35] Yagiz Kargin Martin L. Kersten, Stefan Manegold, and Holger Pirk. The DBMS – your Big Data Sommelier. In *ICDE*, 2015.

[36] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael 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*, SIGMOD '14, pages 1591–1602, New York, NY, USA, 2014. ACM.

[37] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. Demonstrating the bigdawg polystore system for ocean metagenomics analysis. *CIDR*, 2017.

[38] John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, and Aaron J. Elmore. Integrating real-time and batch processing in a polystore. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–7, 2016.

[39] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14), 2013.

[40] Apostolos Papadopoulos and Yannis Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Comput.*, 29(10):1419–1444, October 2003.

[41] 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*, pages 503–512, 2014.

[42] Thomas Vanhove, Gregory van Seghbroeck, Tim Wauters, and Filip De 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*, pages 1–8, 2015.

[43] Panos Vassiliadis. A survey of extract-transform-load technology. 5:1–27, 07 2009.

[44] Janet L. Wiener and Jeffrey F. Naughton. Oodb bulk loading revisited: The partitioned-list approach. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 30–41, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.