DBMS Data Loading: An Analysis on Modern Hardware

Adam Dziedzic[1⋆], Manos Karpathiotakis[2], Ioannis Alagiannis[2], Raja Appuswamy[2], and Anastasia Ailamaki[2,3]

[1] University of Chicago
ady@uchicago.edu
[2] Ecole Polytechnique Fédérale de Lausanne (EPFL)
firstname.lastname@epfl.ch
[3] RAW Labs SA

Abstract. Data loading has traditionally been considered a "one-time deal" – an offline process out of the critical path of query execution. The architecture of DBMS is aligned with this assumption. Nevertheless, the rate in which data is produced and gathered nowadays has nullified the "one-off" assumption, and has turned data loading into a major bottleneck of the data analysis pipeline.
This paper analyzes the behavior of modern DBMS 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 DBMS, 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 DBMS are unable to saturate the available hardware resources. We therefore identify opportunities to accelerate data loading.

1 Introduction

Applications both from the scientific and business worlds accumulate data at an increasingly rapid pace. Natural science experiments produce unprecedented amounts of data. Similarly, companies aggressively collect data to optimize business strategy. The recent advances in cost-effective storage hardware enable storing the produced data, but the ability to organize and process this data has been unable to keep pace with data growth.

Extracting value out of gathered data has traditionally required loading it into an operational database. For example, in data warehouse scenarios, ETL involves **E**xtracting data from outside sources, **T**ransforming it to fit operational needs, and then **L**oading it into the target database [16, 23]. The demand for reduced data-to-query time requires data loading to be a fast operation [27]. The demand for high availability requires minimizing, if not eliminating, batch loading windows during which Database Management Systems (DBMS) can be taken offline. Finally, the ever-increasing growth of data requires data loading to be a scalable operation that can exploit hardware parallelism to load massive amounts of data in very short amounts of time [16, 23]. Unfortunately, traditional DBMS are built around the assumption that data loading is a "one-time deal";

⋆ Work done while the author was at EPFL.

data loading is considered an offline process out of the critical path, with the user defining a schema and loading the majority of the data in one go before submitting any queries. When this architectural design assumption is combined with the explosive data growth, the result is the emergence of data loading as a major bottleneck in the data analysis pipeline of state-of-the-art DBMS.

While much research over the past few years has focused on innovative techniques to avoid or accelerate data loading [8, 13, 21, 24], there has been no systematic study till date that quantifies the ability of modern DBMS to exploit multicore processors and modern storage hardware in order to parallelize data loading. The importance of such quantification has been recognized by the Big Data community, and has led the "Big-Data Top100" benchmark to consider the time spent to load data into the system as part of the benchmark metric [11]. Still, although there is a wide variety of benchmarks [5–7] which use diverse queries to evaluate the query processing capabilities of DBMS, a similar analysis of the (bulk) data loading capabilities of DBMS is missing.

This paper presents a detailed data loading analysis with the following goals: 1) analyze how parallel data loading scales for various DBMS, 2) identify bottlenecks, and 3) provide development guidelines to enable the design of efficient data loading pipelines, and administration guidelines to accelerate the time-consuming loading process. The analysis considers three dimensions: software, hardware, and application workloads. Along the software dimension, we investigate architectural aspects (row stores vs column stores) of four state-of-the-art DBMS, implementation aspects (the threading model used for parallel loading), and runtime aspects (degree of parallelism, presence and absence of logging/constraints). Along the hardware dimension, we evaluate the impact of storage configurations with different I/O capabilities (HDD, SATA SSD, hardware RAID controller with SCSI disks, and DRAM). Finally, along the workload dimension, we consider data from popular benchmarks and real-world datasets 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: Each evaluated DBMS is 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 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 DBMS exhibit varying degrees of scalability. We also list cases in which the conventional drop indexes-load data-rebuild indexes rule-of-thumb which is applicable to single-threaded index building and constraint verification 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.

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 1: Storage devices and characteristics.

2 Setup and Methodology

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

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 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.

Analyzed Systems: The analysis studies four systems: a commercial row-store (DBMS-A), an open-source row-store (PostgreSQL [2]), a commercial column-store (DBMS-B), and an open-source column-store (MonetDB [1]). 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.

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 [7], which is designed for evaluating data warehouses, and the transaction processing benchmark TPC-C [5], 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 [3] dataset and a real-life dataset provided by Symantec [4].

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.

2.3 Experimental Methodology

The goal of the experiments is to provide insight on "where time goes" during loading in modern DBMS – 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 DBMS 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 DBMS. 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.

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 DBMS. Thus, the difference in loaded database size across DBMS is due to architectural differences, like the use of data compression.

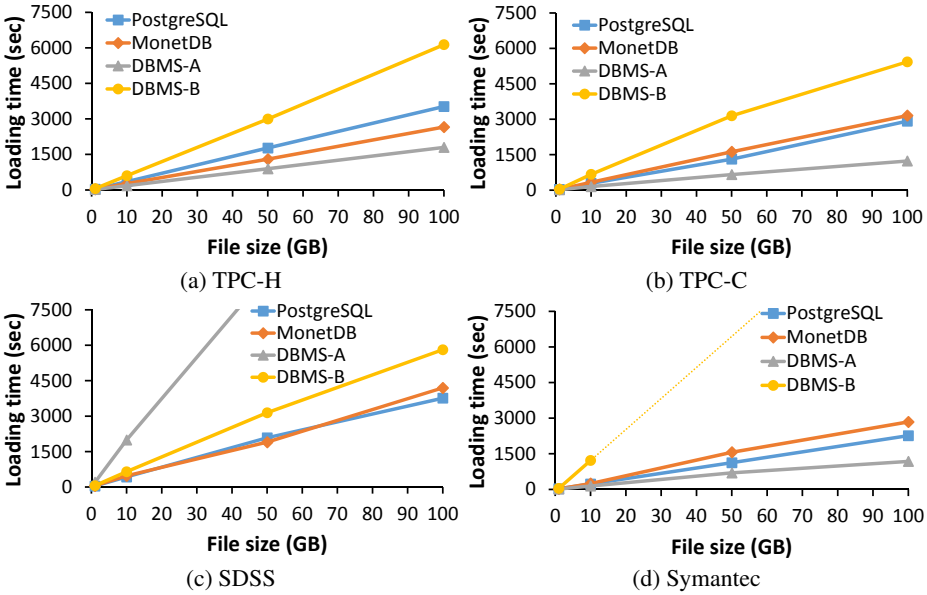
Profiling. We collect statistics about CPU, RAM, and disk I/O utilization of the OS and the DBMS. We use `sar` to measure the CPU and RAM utilization, and `iostat` to measure disk utilization statistics. In addition, we use `iosnoop` to record disk access patterns and the Intel VTune Amplifier to profile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

3 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.

3.1 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



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

(c) SDSS (d) Symantec

Fig. 1: Data loading time increases linearly with the dataset size (single-threaded loading, raw input from HDD, database on DAS).

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 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 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 DBMS. 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 up-

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

date 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, they apply compression to reduce the cost of data scans.

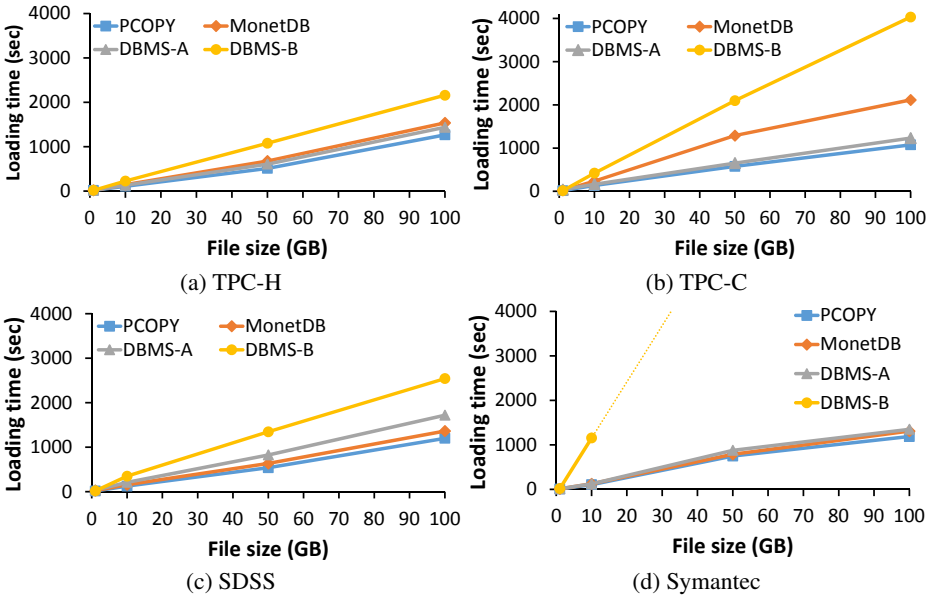
Table 2 shows the ratio between the input data file and the final database size for the experiments of Figure 1. Even though the tested systems read the same amount of data, they end up writing notably different amounts of data. Clearly, DBMS-B and MonetDB have smaller storage footprint than PostgreSQL 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.).

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.

3.2 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



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

(c) SDSS (d) Symantec

Fig. 2: Data loading time increases linearly with the dataset size (parallel loading, input on HDD, database on DAS).

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.

Figure 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 1 and 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.

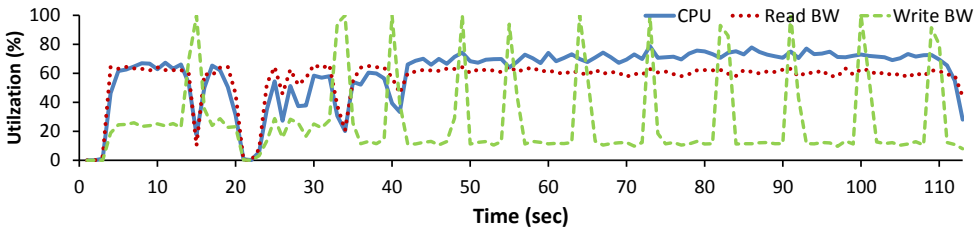
Summary. Figure 2 again shows that there is no system that outperforms the others across all the tested datasets. Generally, parallel loading improves data loading performance in comparison to single-threaded loading in many cases. However, scaling is far from ideal, as loading time does not reduce commensurately 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 Data Loading: Where does time go?

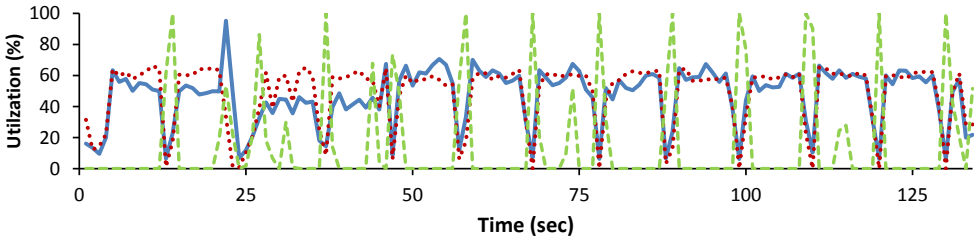
The next experiment looks into CPU and I/O resource usage patterns to identify where time goes during the data loading process so that we understand the reason behind lack of scalability under parallel data loading. This experiment presents an alternative view of Figure 2(a): It monitors the usage of system resources (CPU, RAM, I/O reads and writes) when a 10GB version of TPC-H is loaded using the parallel loaders for various DBMS. 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:

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 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.

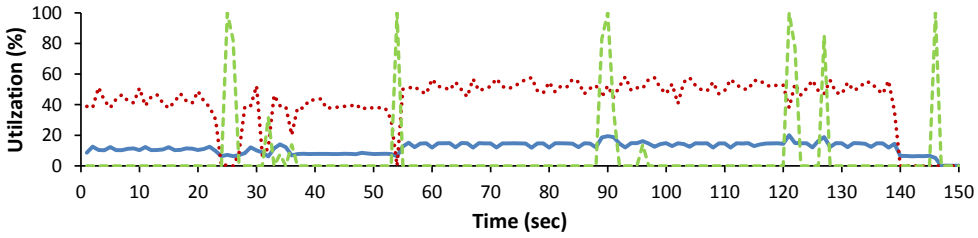
The second pattern that can be observed across all systems is that CPU and I/O resources remain under-utilized. MonetDB exhibits the lowest CPU utilization among all systems. This is due to the fact that it uses one "producer" thread that reads, parses, and tokenizes raw data values, and then N "consumer" 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



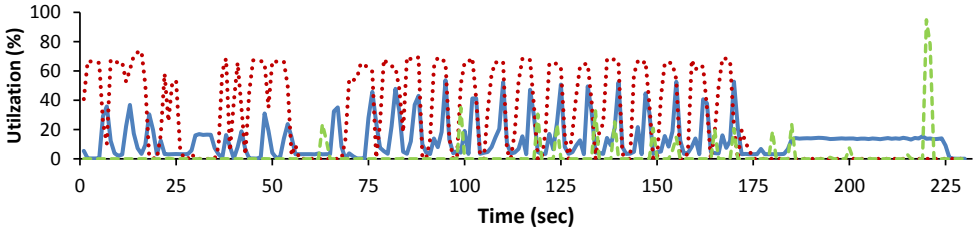
(a) PCOPY



(b) DBMS-A



(c) MonetDB



(d) DBMS-B

Fig. 3: Different CPU, Read/Write bandwidth utilization per system (Input TPC-H 10GB, source on HDD, database on DAS).

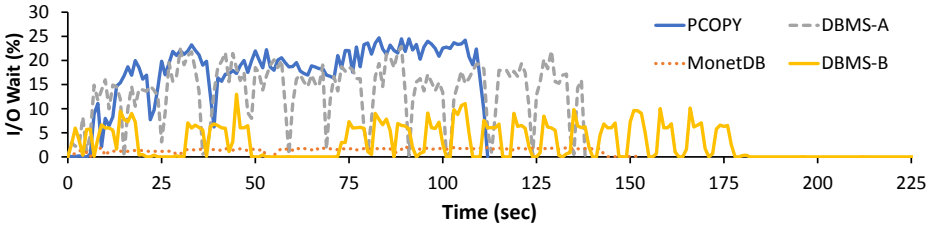


Fig. 4: I/O Wait per system (Input TPC-H 10GB, source on HDD, database on DAS).

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.

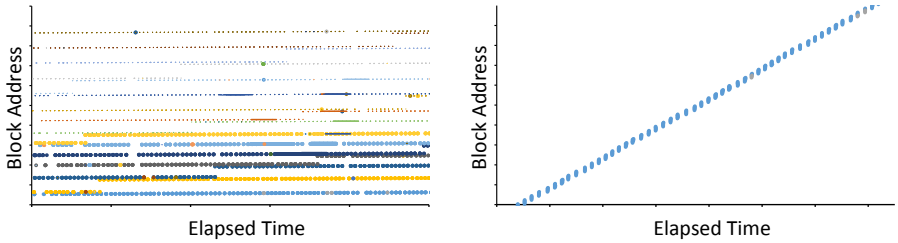
Figure 4 illustrates the percentage of time that each DBMS spends waiting for I/O requests to be resolved during each second of execution. Except MonetDB, all other systems spend a non-trivial portion of time waiting for I/O which explains the low CPU utilization. Still, read throughput utilization of various systems in Figure 3 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.

3.4 Impact of underlying storage

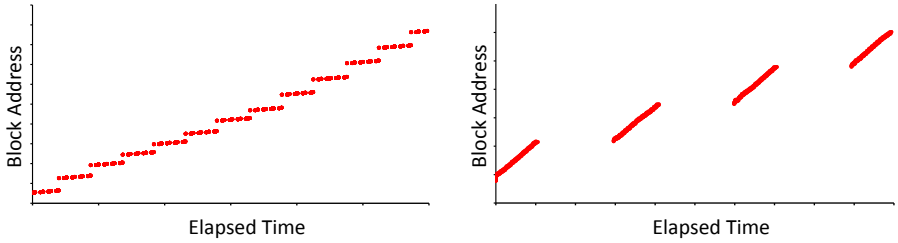
The previous experiments showed that a typical DBMS setup under-utilizes both I/O bandwidth as well as the available CPUs because of the time it spends waiting for random I/O completion. This section studies the underutilization issue from both a software and a hardware perspective; it investigates i) 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 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



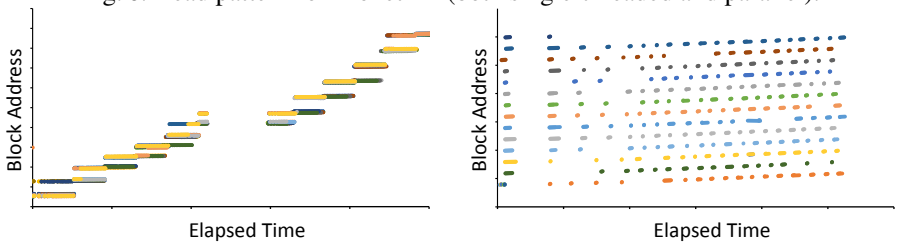
(a) Reading the whole file (b) A chunk (zoom-in)

Fig. 5: Read pattern for PCOPY (parallel).



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

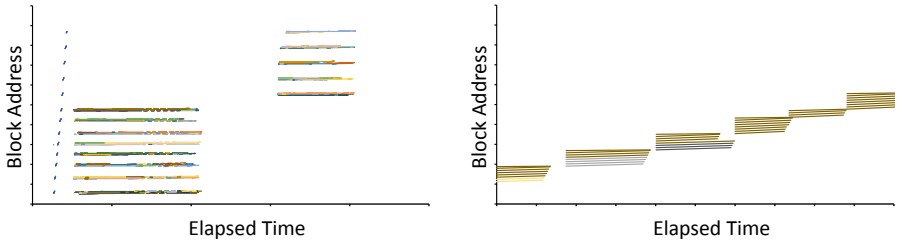
Fig. 6: Read pattern for MonetDB (both single-threaded and parallel).



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

Fig. 7: Read pattern for DBMS-A (parallel).

LBAs (Logical Block Addresses), further on called chunk. Different colors in the graphs represent distinct processes/threads.



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

Fig. 8: Read pattern for DBMS-B (parallel).

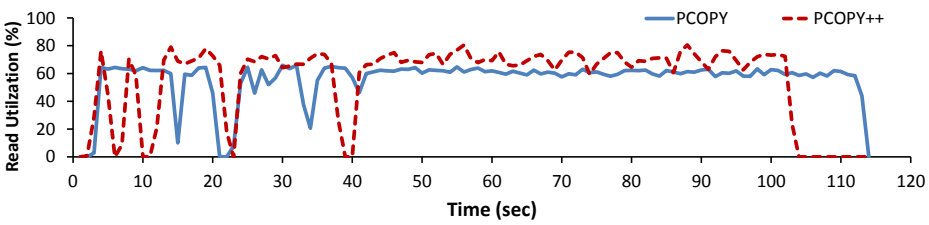


Fig. 9: Using a serial reader improves the read throughput of PCOPY.

Figures 5, 6, 7, and 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 sequentially using one thread, while the other systems use multiple concurrent readers (plotted with different colors in the graphs). Figure 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 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 5(b)).
- MonetDB uses a single "producer" thread to read data and provide each data block to a "consumer" (Figure 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 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 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 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 10 plots the loading time when the slow HDD is the data source storage media, as in previous experiments, while varying the destination

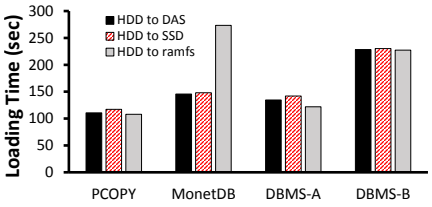


Fig. 10: Loading 10GB TPC-H: Varying data destination storage with slow data source storage (HDD).

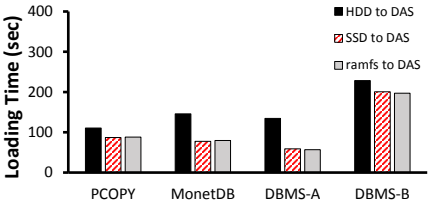


Fig. 11: Loading 10GB TPC-H: Varying data source storage with DAS data destination storage.

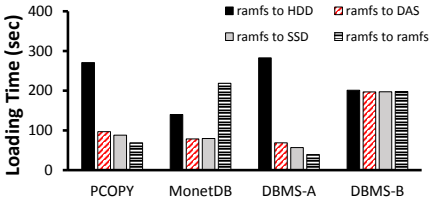


Fig. 12: Varying data destination storage with fast data source storage (ramfs).

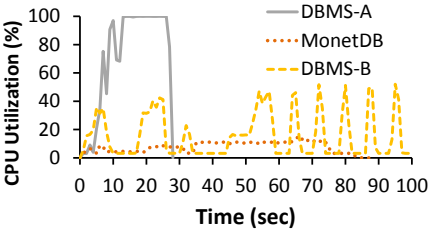


Fig. 13: CPU Utilization for ramfs-based source and destination.

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 DBMS 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[4].

Figure 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 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 its thread overprovisioning; the numerous low-priority threads it spawns get pre-empted often. For PCOPY and DBMS-A, using ramfs as the

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

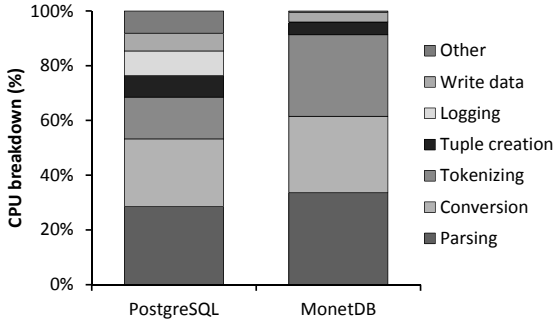


Fig. 14: CPU breakdowns for PostgreSQL and MonetDB during data loading of integers. Most of the time is spent on parsing, tokenizing, conversion and tuple creation.

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 13 shows the CPU utilization for the DBMS 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 DBMS 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.

3.5 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 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 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 [21] (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.

3.6 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 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[5]. We consider PK constraints as specified in the TPC-H schema.

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

Figure 16 shows the total time taken to load the database using 16 physical cores. Comparing Figures 15 and 16, the following observations can be made:

[5] `https://www.monetdb.org/Documentation/SQLreference/TableIdentityColumn`

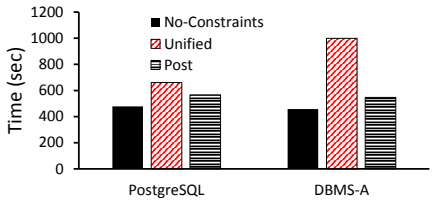


Fig. 15: Single-threaded loading in the presence of PK constraints.

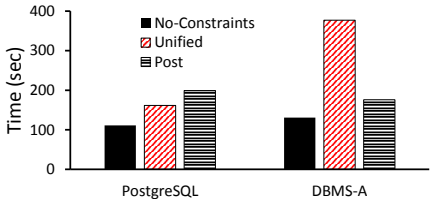


Fig. 16: Parallel loading in the presence of PK constraints.

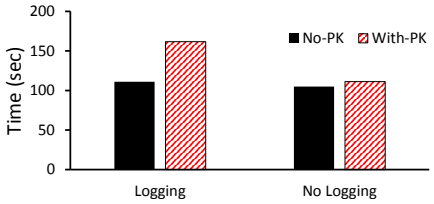


Fig. 17: Effect of logging in PostgreSQL.

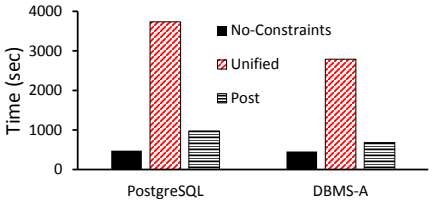


Fig. 18: Single-threaded loading in the presence of FK constraints.

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

Foreign key constraints. Figure 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 15 and 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.

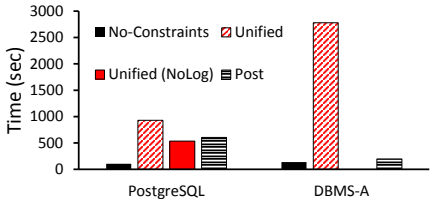


Fig. 19: Parallel loading in the presence of FK constraints.

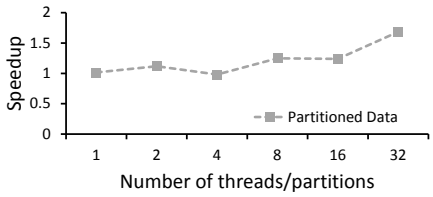


Fig. 20: Re-organizing input to facilitate FK validation.

Figure 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 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 The traditional rules of thumb for loading data with constraints have to be updated. 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 which 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.

4 Related Work

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. This section briefly reviews this body of related work.

Bulk loading. Numerous approaches examine ways to accelerate data loading. Starting from general-purpose approaches, the authors of [10] introduce the idea of partitioning the input data and exploiting parallelism to load the data partitions faster. Instant loading [24] presents a scalable bulk loader for main-memory systems, designed to parallelize the loading phase of HyPer [22] 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. [26] 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 (*aio*) 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 works offer specialized, domain-specific solutions: The authors of [9, 28] 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 [14, 25]. Finally, the authors of [12] put together a parallel bulk loading pipeline for a specific repository of astronomical data.

Querying external data. Motivated by the blocking nature of data loading, vendor lock-in concerns, and the proliferation of different data formats, numerous works 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 from the query engine. Still, external tables lack support for advanced database features such as DML operations, indexes or statistics.

Speculative loading [13] 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 [15] 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 [8] 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 [19–21] further reduce raw data access costs by generating custom data access paths at runtime via code generation.

Data vaults [17] 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 [18] 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).

5 Conclusion

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.

This study evaluates the data loading performance of four popular DBMS 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 under some circumstances.

Despite such improvement, we still 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 improving the data loading pipeline. Our analysis reveals that moving forward, DBMS designers should refactor the data loading pipeline by using a dedicated code base for bulk loading to avoid latch contention in various subsystems. While some systems (like DBMS-A) do use such techniques when the database can be taken offline, we believe that it is necessary to apply the same principles to load data while keeping the database online in order to eliminate DBMS down time.

References

1. MonetDB. http://www.monetdb.org/.
2. PostgreSQL. https://www.postgresql.org/.
3. SkyServer project. http://skyserver.sdss.org.
4. Symantec Enterprise. http://http://www.symantec.com.
5. TPC-C Benchmark: Standard Specification. http://www.tpc.org/tpcc/.
6. TPC-DS Benchmark: Standard Specification. http://www.tpc.org/tpcds/.
7. TPC-H Benchmark: Standard Specification. http://www.tpc.org/tpch/.
8. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.
9. S. Amer-Yahia and S. Cluet. A declarative approach to optimize bulk loading into databases. *ACM Trans. Database Syst.*, 29(2):233–281, 2004.
10. T. Barclay, R. Barnes, J. Gray, and P. Sundaresan. Loading Databases Using Dataflow Parallelism. *SIGMOD Record*, 23(4):72–83, 1994.
11. C. Baru, M. Bhandarkar, R. Nambiar, M. Poess, and T. Rabl. Benchmarking Big Data systems and the Bigdata Top100 list. *Big Data*, 1(1):60–64, March 2013.
12. Y. D. Cai, R. A. Aydt, and R. Brunner. Optimized data loading for a multi-terabyte sky survey repository. In *SC2005*, page 42, 2005.
13. Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *SIGMOD*, 2014.
14. J. V. den Bercken and B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *VLDB*, pages 461–470, 2001.
15. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. Here are my queries. Where are my results? In *CIDR*, 2011.
16. C. Imhoff, N. Galemmo, and J. Geiger. *Mastering Data Warehouse Design*. Wiley Publishing, Inc, Indianapolis, Indiana, 2nd edition, 2003.
17. M. Ivanova, M. L. Kersten, and S. Manegold. Data Vaults: A Symbiosis Between Database Technology And Scientific File Repositories. In *Proceedings of International Conference on Scientific and Statistical Database Management 2012*, June 2012.
18. Y. Kargin, M. L. Kersten, S. Manegold, and H. Pirk. The DBMS - your big data sommelier. In *ICDE*, 2015.
19. M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12):972–983, 2016.
20. M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*, 2015.
21. M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.
22. A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.
23. R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.
24. T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14), 2013.
25. A. Papadopoulos and Y. Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Computing*, 29(10):1419–1444, 2003.
26. K. T. Sridhar and M. A. Sakkeer. Optimizing Database Load and Extract for Big Data Era. In *DASFAA*, pages 503–512, 2014.
27. P. Vassiliadis and A. Simitsis. Near real time ETL. In *New Trends in Data Warehousing and Data Analysis*, volume 3 of *Annals of Information Systems*, pages 1–31. Springer, 2009.
28. J. L. Wiener and J. F. Naughton. OODB bulk loading revisited: The partitioned-list approach. In *VLDB*, pages 30–41, 1995.