Shared Load(ing): Efficient Bulk Loading into Optimized Storage

Stefan Noll[*§], Jens Teubner[§], Norman May[*], Alexander Böhm[*]

[*]SAP SE, firstname.lastname@sap.com

[§]TU Dortmund University, firstname.lastname@cs.tu-dortmund.de

ABSTRACT

Bulk loading into the optimized storage of a database system is a performance-critical task for data analysis, replication, and system integration. Depending on the storage layout, it may entail complex data transformations, making it also an expensive task that can disturb other workloads.

In this work, we demonstrate that for a commercial, in-memory columnar system with compression-optimized storage, data transformation dominates the cost of bulk loading. The transformations may cause resource contention on a stressed system, resulting in poor and unpredictable performance for both bulk loading and query processing. To mitigate this problem, we propose *Shared Loading*, a distributed bulk loading mechanism that enables dynamically offloading deserialization and data transformation to the machine where the input data resides. In our evaluation we demonstrate that, for different network bandwidths and data sets, *Shared Loading* accelerates bulk loading into compression-optimized storage and improves the performance and predictability of queries running concurrently.

1. INTRODUCTION

In today's heterogeneous system landscape, an ever-growing volume of data is available in plain text files. Plain text files are frequently used to transfer scientific data sets or business data, to facilitate replication and system integration, or to migrate to a new system. In the latter case, customers of SAP state that the bulk loading of text files can quickly become the bottleneck in mission-critical migration processes. Thus, fast and efficient bulk loading is imperative.

Related work [1, 2, 5] assumes that data resides on local storage. However, files are often stored close to the *client* machine, where an application produces data or data is preprocessed, and not close to the *server* machine running the DBMS. Thus, data needs to be transferred over the *network*. This can be a fast internal network, in case of a cloud-only or an on-premise scenario, or a slow Internet connection, e.g., when loading data from an on-premise setup into the cloud.

9th Biennial Conference on Innovative Data Systems Research (CIDR '19)
January 13-16, 2019 , Asilomar, California, USA.

In addition, related work [2,17] focuses on optimizing parsing and the creation of an index during bulk loading. They conclude that most time is lost on deserialization. However, modern systems [4,10,15] employ a (highly) compressed storage. In such a system, it is also challenging to *transform* data because compression is resource-intensive. In fact, we demonstrate in our analysis of a production system that most time is lost on data transformation—not on deserialization. In addition, such transformations may take away precious CPU cycles from queries running in parallel: we observed tail latencies degrade by a factor of **2.5** under load.

To address these issues, we study bulk loading in a distributed environment and its impact on query workloads running in parallel. Ultimately, we develop a new mechanism for efficient bulk loading into optimized storage. Our mechanism accelerates loading and improves the performance and predictability of concurrent query processing. Its architecture allows dynamically adapting data transformations and shifting work between client(s) and server *at loading time*—without the need for the user to partition the data or to manually parallelize bulk loading [16, 18]. To that end,

- (i) we analyze where time is lost in a complete bulk loading pipeline using SAP HANA as an example database;
- (ii) we present the architecture of the distributed bulk loading mechanism *Shared Loading*, which can dynamically offload work to the client machine; and
- (iii) we evaluate the performance characteristics of our approach and study whether it improves tail latency of concurrently running queries for different network bandwidths and data sets.

2. RELATED WORK & BACKGROUND

Bulk Loading. Commercial and open-source database systems offer a range of interfaces for bulk loading files. We group existing approaches into three categories: First, systems may offer a *command* for a terminal-based front-end. The user either manually transfers the file or it is transferred during the loading operation. Second, systems may support parameterized SQL queries with *array bindings*. In contrast to submitting a query for every row of a table, it allows batching multiple rows in a single query. In both approaches, the DBMS does the data processing. Third, some vendors provide *dedicated tools* for importing files such as `SQL*Loader` [18]. Their documentation does not detail, however, what is computed by the client and by the server.

Dziedzic et al. [9] analyze data loading for different DBMS. They assume that data already resides on the server. They corroborate our results by showing that bulk loading is slow and expensive, especially in compression-optimized systems. In fact, we close the gap discussed in their work: Our dynamic loading mechanism accelerates loading, takes the load off the system and improves query performance.

In-situ Query Processing. Query processing on files [1, 2, 5, 12] focuses on analyzing large files with the goal to minimize the (initial) response times for a sequence of queries. Related work identifies the repeated parsing, tokenizing, and data type conversion as a performance bottleneck but does not consider network transfers or complex transformations

Instant Loading. Mühlbauer et al. [17] use SIMD instructions to accelerate deserialization and create mergeable indexes during bulk loading. We do not focus on these operations because our analysis (see Section 3) shows these operations consume less than 20 % CPU time in a commercial system and are orthogonal to our work. We focus on a distributed environment where our approach can dynamically shift data transformations between client(s) and server.

Dictionary Compression. The in-memory database system SAP HANA [10] makes heavy use of *order-preserving* dictionary compression in its read-optimized storage—other systems employ similar ideas to varying degrees [4, 15]. An ordered dictionary maps domain values to a dense set of consecutive numbers. Instead of the actual value of the columns, the engine stores the index of the dictionary entry. A column or a dictionary may be further compressed. In our work, we focus on order-preserving dictionary compression. Applying additional compression such as bit packing or prefix encoding is orthogonal. We observed in experiments (not shown) that they can be combined efficiently by exploiting the sorting and the known distinct count of dictionary compression.

Buffered Updates. To facilitate data ingestion into optimized storage, SAP HANA transforms new data gradually, migrating records from write- to read-optimized storage. Our approach bypasses the write-optimized storage and merges new data directly into the read-optimized storage—similar to Lamb et al. [15]. However, our approach enables offloading data transformations at loading time to the client.

3. COST ANALYSIS OF BULK LOADING

We analyze where time is lost in a complete bulk loading pipeline using SAP HANA. While the results are specific to the data set and the implementation of the bulk loading pipeline in SAP HANA, we expect similar results for other systems with (complex) data transformations—especially for systems with compressed storage. We bulk load the `lineitem` table of the TPC-H benchmark from a local solid-state drive. The setup is described in Section 5.1.

The results shown in Figure 1 demonstrate that data transformations consume more than half of the CPU time. This includes the time it takes to insert new rows into the write-optimized storage, to compute a compressed representation using an unsorted dictionary, and to create a columnar in-memory representation. The deserialization including parsing, validating and creating an instance of the SQL data type in memory consumes around 15 % CPU time. Checking constraints, partitioning the table, or updating meta data such as indexes requires only a small amount of CPU time. Merging the write-optimized storage into the read-optimized storage consumes 10 % CPU time. Logging and persisting consume a negligible amount due to asynchronous I/O. The remaining 10 % are overhead from the transaction manager, lock handling, and memory management.

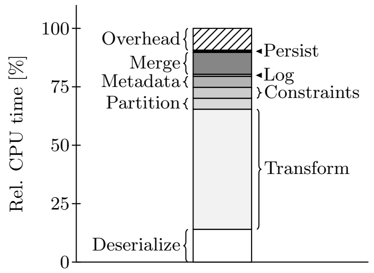


Figure 1: Cost analysis of bulk loading into SAP HANA. Most CPU time is spent on data transformation.

In summary, our results differ from previous results [2, 5, 17] that attribute the highest cost for bulk loading to deserialization. For compression-optimized systems, the cost of *transforming* the data dominates computing time and outweighs the cost of deserialization by a factor of **3.7**.

4. SHARED LOADING

Bulk loading and concurrently running queries compete for hardware resources. The result is poor loading throughput *and* poor query performance. To address the problem, we propose to *offload* part of the bulk loading. In particular, we can exploit that the input data of the bulk loading is often stored close to the *client* machine and not close to the *server* machine running the DBMS. Our cost analysis of the bulk loading pipeline identifies which steps are worth offloading: deserialization and data transformation.

We argue that offloading work needs to be done *dynamically* depending on, e.g., the input data, the compute power of client and server, or the available network bandwidth. To that end, we propose the architecture of a distributed bulk loading mechanism that enables offloading deserialization and data transformation to the client *at loading time.*

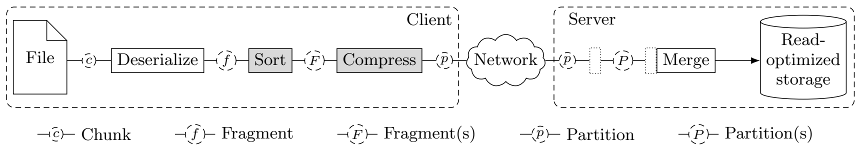
Figure 2 gives an overview of the processing steps on client and server. We assume that the input file is a delimiter-separated values file such as CSV. However, by adjusting the parsing step of the deserialization of the file, our approach may support other flat file formats. We use the example of order-preserving dictionary compression, but the concept of dynamically offloading deserialization and the transformation into a dictionary-compressed column store may be applicable to other transformations and compressions.

For the purpose of illustration, we first present client-centric and server-centric loading. Afterwards, we describe how we can combine both approaches dynamically.

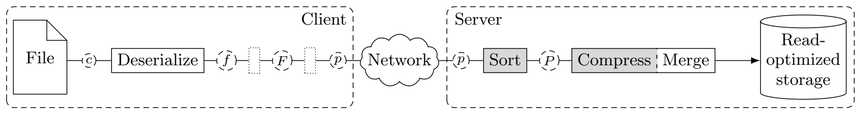
4.1 Client-Centric

Client Component. The client component transforms data by pushing file chunks through a processing pipeline. When we shift data transformations to the client, the client produces a dictionary-compressed, columnar partition and sends it to the server. This allows the DBMS to merge a partition directly into optimized storage. We briefly describe the individual steps, shown in Figure 2a, in the following.

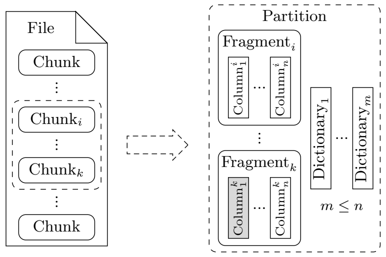
The *deserialization* step converts a file *chunk* to an in-memory instantiation of the data. It parses the chunk to identify delimiter symbols, validates fields, and instantiates



(a) Client-centric bulk loading into optimized storage.



(b) Server-centric bulk loading into optimized storage



(c) *Shared Loading* transforms a file chunk into a columnar in-memory fragment. Multiple fragments form a logical partition. A fragment's column (gray) may be dictionary-compressed.

Figure 2: Processing steps in *Shared Loading*. Computational work (gray) can dynamically shift between client (a) and server (b): at loading time, we can decide for a *fragment's column* (c) where to compute its data transformation.

data types in memory according to the schema of the table it gets from the server. Finally, the deserialization step assembles all rows of the chunk into a columnar in-memory representation, which we refer to as a *fragment*. The *sort* step adds a temporary dictionary to a fragment's column, which is a sorted copy of the column without duplicates. The *compression* step logically assembles multiple fragments into a horizontal *partition*, shown in Figure 2c, and (physically) merges all temporary dictionaries of a column into a single dictionary. Afterwards, it uses the dictionary to encode the columns of the fragments. The *transfer* step sends a partition to the server. Dictionary compression reduces the transfer size: we show in the evaluation in Section 5.2 that dictionary compression reduces the data size of the `warehouse` data set by 56 % compared to the original file size.

Server Component. The server component is designed as a part of the database system with internal access to the storage engine. When we shift the data transformation to the client, it receives a dictionary-compressed, columnar partition that allows the DBMS to merge the partition directly into its *read-optimized storage*.

The *merge* step merges one or more *partitions* into the *read-optimized storage*. It merges all partitions available since the last merge operation. For each column of the partitions, it merges the dictionaries with the corresponding dictionary of the optimized storage. First, the merge step creates mappings from the dictionaries of the partitions to the new dictionary. Afterwards, it uses the mappings to update the optimized storage as well as to update the data of the partitions, which is then appended to the optimized storage. Note that we could merge the data into a specific partition of the target table or create a new partition to avoid updating the dictionary compression.

4.2 Server-Centric

Client Component. When we shift only deserialization to the client, but compute data transformations on the server, the client component of *Shared Loading* produces an uncompressed, columnar partition (see Figure 2b). The *deserialization* step produces a *fragment* just as in case of the client-centric loading. Afterwards, the client groups *fragments* logically into a horizontal *partition* without applying dictionary compression. Subsequently, it transfers a partition to the server by sending fragment after fragment over the network.

Server Component. The server component receives an uncompressed, columnar partition from the client. It needs to transform the data before merging it into optimized storage. The *sort* step receives a *partition* consisting of *fragments* from the client. This allows the server to process each fragment independently and as soon as a fragment arrives. For each column of a fragment, it creates a temporary dictionary—similar to the sort step of the client component. The *merge* step merges all *partitions* available since the last merge operation into the *read-optimized storage*. For each column of the partitions, it first merges the temporary dictionaries with the corresponding dictionary of the optimized storage. Afterwards, the merge step maps the dictionaries of the optimized storage to the merged dictionaries. It uses the mappings to update the optimized storage and the merged dictionaries to compresses the partitions, which are then appended to the optimized storage.

4.3 Dynamic Offloading

The architecture of *Shared Loading* combines client- and server-centric loading. It allows deciding whether to transform a *fragment's column*, shown in Figure 2c, on the client or on the server. The decision can be made *at loading time*. To that end, *Shared Loading* can use heuristics during the sort step at the client. It either creates a temporary dictionary and performs client-centric loading for a fragment's column, or it omits the creation of the temporary dictionary and performs server-centric loading.

The remaining steps adapt to the decision (dotted boxes in Figures 2a and 2b). The compression step at the client only compresses a fragment's column if it has a temporary dictionary. Otherwise the fragment's column remains uncompressed in the partition. The sort step at the server checks if a fragment's column is not already dictionary-compressed. Only if that is the case, it creates a temporary dictionary. Thus, the sort step produces a partition where a fragment's column either is dictionary-compressed or has a temporary dictionary. The merge step either updates the dictionary compression of a fragment's compressed column or it encodes a fragment's uncompressed column when writing to the optimized storage.

Heuristics. The architecture of *Shared Loading* enables the use of different heuristics. For instance, we use a heuristic for *minimizing* the amount of data sent over the network. In particular, we estimate the number of unique values in a column using the HyperLogLog algorithm [11] with HIP estimator [6]. The algorithm allows us to estimate the total memory size of the dictionary-compressed column and the corresponding dictionary. If we estimate the memory size to be smaller than the uncompressed column, we transform the column at the client; otherwise we delegate the transformation to the server.

Note that other heuristics could decide to shift data transformations based on the server's utilization, the client's and the server's compute capabilities, or the network bandwidth. In particular, heuristics can use information that is only available at runtime.

4.4 Implementation

Our C++ implementation of the data processing pipeline shown in Figure 2 exploits independent work whenever possible to achieve a high degree of parallelism. The implementation is independent of the codebase of SAP HANA but simulates major characteristics.

To facilitate in-place sorting, we store variable-sized strings of type `VARCHAR(N)` as *fixed-sized* strings of length `N`. Note that we avoid the increased memory and transfer size of fixed-sized strings by employing dictionary compression. We reduce the memory footprint of the merge operation by merging at most two columns in parallel. The same configuration is the default in SAP HANA. This also limits the impact of the merge operation on other workloads.

We set the size of a file chunk to 10 MiB and we group 50 fragments into a partition. This means that a partition corresponds to 500 MiB of the input file. We experimentally confirmed that both parameters are robust. The chunk size needs to be big enough to contain multiple rows and to amortize the parallelization overhead (e.g., 10 KiB) and small enough to allow a high degree of parallelism (e.g., 100 MiB). Similar arguments apply to the partition size: a partition should contain between 10 and 100 chunks.

5. EVALUATION

We evaluate the performance of *Shared Loading* and state-of-the-art architectures without and with concurrent query processing, we analyze how much work *Shared Loading* can offload, and, more importantly, whether it improves throughput or query performance and predictability.

5.1 Setup

Data Set. We evaluate two data sets: the `lineitem` table of the TPC-H benchmark and the `warehouse` table, which was extracted from the data warehouse of a customer. Both data sets are available in the file format of the TPC-H benchmark. We use the `lineitem` table with a scale factor of 10. The file has a size of 7.24 GiB. The file of the `warehouse` table has a size of 17.57 GiB, $12 \cdot 10^6$ rows and 155 columns.
Hardware. Our system has 128 GiB of DRAM and two Intel Xeon E5-2660 v3 processors with 10 physical cores each. The client process and the server process run on different sockets of the same machine. We allocate 10 physical cores to the server and vary the number of cores for the client from 2 to 8. The files reside on an local SSD with a read bandwidth of up to 530 MB/s. Note that, for a single HDD, a weak client would never be compute-bound. In addition, the SSD is only used for reading. We do not evaluate persisting and logging because we assume the server's storage to be more powerful than the client's.
Network. We use the `tc` utility to emulate different network bandwidths—similar to [19]. TCP/IP messages are sent to the localhost address. We evaluate a network bandwidth of 1 Gbit/s and 10 Gbit/s because these represent 69 % of the market share [14]. 1 Gbit/s represents the maximum Internet bandwidth when loading data from an on-premise solution into the cloud. 10 Gbit/s, on the other hand, represents a typical sizing option/quota within the cloud [3,13] when performing a cloud-internal bulk loading operation.
Configurations. We evaluate *Shared Loading* in three different configurations: SL_A corresponds to Figure 2a, i.e., the client *compresses all*—it transforms all data into dictionary-compressed partitions. SL_S uses the heuristic to *minimize* data *size*—the client dynamically decides for each column of a fragment to compress it on the client or on the server. SL_N corresponds to Figure 2b, i.e., the client *compresses none*—it transforms all data into uncompressed partitions.

In addition, we compare *Shared Loading* against two state-of-the-art approaches: PIPE and SEQ. They represent the bulk loading mechanism of current systems using terminal-based commands. PIPE means that the client sends file chunks to the server, while the server ingests the data. SEQ means that the client first transfers the file. Then, the server performs the bulk loading.
Query Workload. We use two analytical queries inspired by the TPC-H benchmark to evaluate the impact of concurrent query processing. Queries run against a second instance of the `lineitem` table with a scale factor of 10. Thus, query processing is independent from bulk loading. We execute each query ten times in a batch, wait for all queries to finish and then execute the next batch. In each experiment, we base the duration of the query workload on the maximum loading time of all configurations.

```
Q1: select sum(l_extendedprice) from L;
Q2: select count(*) from L where l_shipdate
        between '1994-1-1' and '1995-1-1';
```

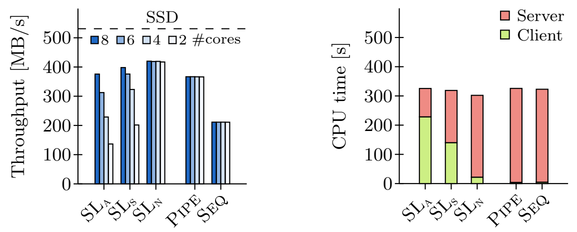
Measurement Method. We measure *throughput*, i.e., the size of the file divided by the elapsed time. In addition, we measure *CPU time*, the total time which processor cores spent on executing instructions, to quantify computational work. To evaluate the impact of query processing, we measure *tail latency*—a performance metric for mission-critical systems with stringent SLAs [7,8].

5.2 Loading in Isolation

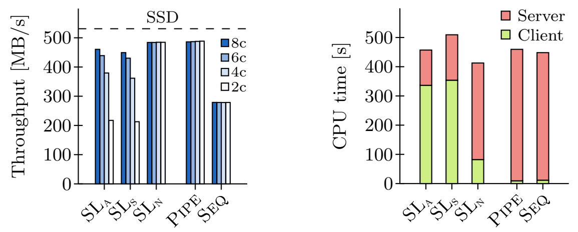
10-Gbit Network. Figure 3 shows the results for a network bandwidth of 10 Gbit/s: *Shared Loading* can offload a large amount of work to the client (independent of the number of cores allocated to the client). For the `lineitem` table, we can shift 71 % (SL_A), 44 % (SL_S), and 7 % (SL_N) of the total CPU time to the client. For the `warehouse` table, we are able shift 74 %, 69 %, and 20 %, respectively.

We observe that for SL_S the amount of CPU time shifted to the client varies due to the heuristic: the client compresses 10 out of 16 columns for the `lineitem` table and 140 out of 155 columns for the `warehouse` table. In addition, the results demonstrate that the server component of *Shared Loading* always consumes less CPU time than the state-of-the-art bulk loading architectures PIPE and SEQ.

The throughput of *Shared Loading* is comparable to the state of the art and in some cases even up to 15 % higher. We observe that without concurrent query processing shifting only deserialization to the client (SL_N) results in the highest

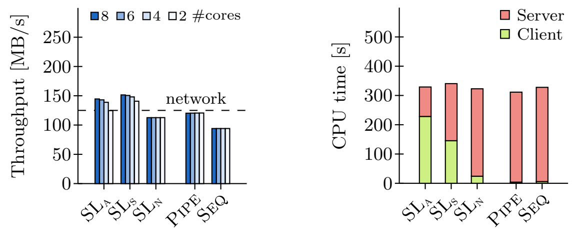


(a) `lineitem`

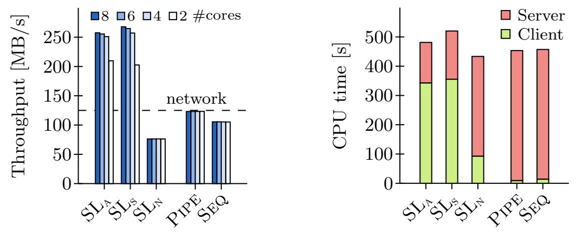


(b) `warehouse`

Figure 3: Bulk loading over a 10-Gbit network without query processing.



(a) `lineitem`



(b) `warehouse`

Figure 4: Bulk loading over a 1-Gbit network without query processing.

throughput. The results come as no surprise because the bulk loading is not network-bound but limited by the SSD's read-bandwidth and the client's compute capability.

1-Gbit Network. Figure 4 shows the results for a network bandwidth of 1 Gbit/s: The measured CPU times resemble the results for a network bandwidth of 10 Gbit/s due to asynchronous network transfer. In addition, the results demonstrate that throughput is network-bound. It differs significantly between configurations. We attribute this to the amount of data that the client transfers. Note that SL$_\text{N}$ increases the transfer size compared to the original file because we store strings with a fixed size (cf. Section 4.4).

For the `lineitem` table, SL$_\text{S}$ reduces the transfer size to 77 % of the size of the input file and it increases throughput by 26 % compared to PIPE. For the `warehouse` table, SL$_\text{S}$ transfers data with 41 % of the file size and it increases throughput by 117 % compared to PIPE. Thus, using the heuristic results indeed in the smallest transfer size.

5.3 Loading With Concurrent Queries

10-Gbit Network. Figure 5 shows the results for a network bandwidth of 10 Gbit/s. We notice that the results differ from previous results without query processing. Compared to the baseline PIPE, SL$_\text{A}$ increases throughput by 90 % for the `lineitem` table and by 29 % for the `warehouse` table. This demonstrates that, by shifting all transformations including compression to the client, *Shared Loading* can maintain a high loading rate even when the server is stressed—unlike state-of-the-art bulk loading methods.

By offloading transformations to the client, *Shared Loading* relieves the server as well: When we compare PIPE with SL$_\text{A}$, tail latency improves by 32 % for Q1 and 53 % for Q2 for the `lineitem` table. For the `warehouse` table, tail latency improves by 47 % and 60 %, respectively. Thus, *Shared Loading* can reduce stress in peak load situations, which in return improves query performance and predictability.

Note that the client can be relatively weak: 4 cores suffice to transform and compress all data on the client while achieving a higher throughput than the state of the art. Average response times (not shown) improve by 20 % and 50 % for the `lineitem` table and by 15 % and 37 % for the `warehouse` table when comparing SL$_\text{A}$ with PIPE.

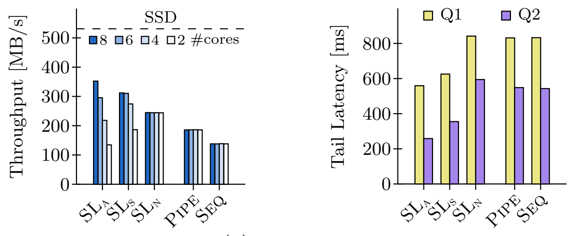
1-Gbit Network. Figure 6 shows the results for of a network bandwidth of 1 Gbit/s. The results for throughput are similar to the ones without query processing: bulk loading is again network-bound. SL$_\text{S}$ improves throughput by up to 27 % and 116 % compared to PIPE. In particular, we observe that throughput does not degrade even though the server is stressed: the low loading rate induces smaller load spikes.

SL$_\text{A}$ improves tail latency by 12 % for Q1 and by 33 % for Q2 compared to PIPE for the `lineitem` table. For the `warehouse` table, tail latency only improves by 7–17 %, which demonstrates that the transformations of the `warehouse` table cause fewer load spikes than the `lineitem` table. *Shared Loading* primarily improves throughput due to the reduced transfer size, while its efficient offloading never degrades query processing on the server.

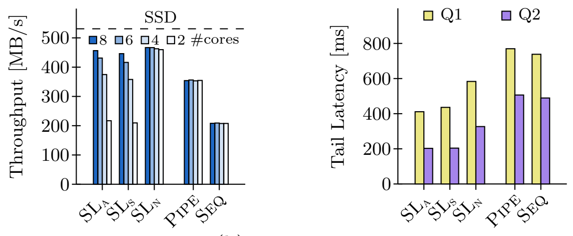
5.4 Discussion

Our evaluation demonstrates that *Shared Loading* performs up to **2**× better than state-of-the-art architectures in 1-Gbit environments due to the compressed network transfer. The performance advantage in 10-Gbit environments becomes clear once the server is stressed, e.g., by running a query workload concurrently: offloading data transformations to the client reduces CPU contention on the server which benefits query processing *and* bulk loading. The results also demonstrate why work needs to be shifted *dynamically*: different configurations of *Shared Loading* perform best or show trade-offs depending on network bandwidth, server load, and compute capability of the client.

Ultimately, we envision the client component of *Shared Loading* to be part of a lightweight SQL client. Its complexity stays low because it performs only deserialization and data transformation. In addition, *Shared Loading* consumes a low amount of memory—it buffers only one partition in-between processing steps. Thus, the total memory consumption will not exceed the equivalent of $7 \cdot 50 \cdot 10\,\text{MiB} \approx 3.5\,\text{GiB}$

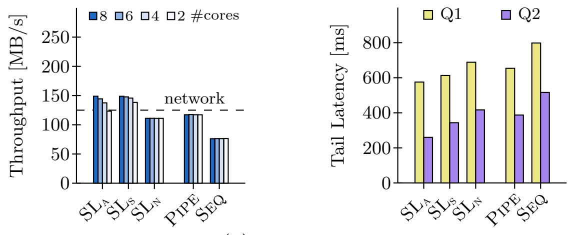


(a) `lineitem`

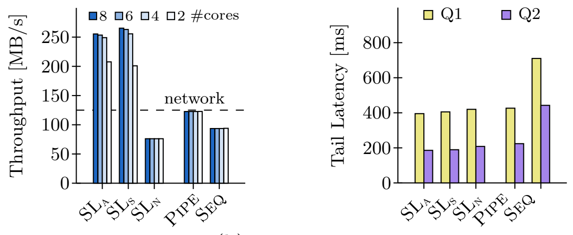


(b) `warehouse`

Figure 5: Bulk loading over a 10-Gbit network with concurrent query processing.



(a) `lineitem`



(b) `warehouse`

Figure 6: Bulk loading over a 1-Gbit network with concurrent query processing.

of file data for any table. The low resource consumption makes *Shared Loading* also a good candidate for implementing bulk loading in a cloud-native database. When loading a large volume of data, the system could start a (small) instance running the client component of *Shared Loading* to ensure elasticity and reduce costs.

6. CONCLUSION

In today's heterogeneous system landscape, bulk loading plain text files is a performance-critical task for data analysis, replication, system integration, and migration. However, for systems that employ a (highly) compressed storage, bulk loading can stress the system significantly. In particular, the data *transformation* during bulk loading can be very expensive and negatively impact workloads running in parallel.

In this work, we analyze the costs of bulk loading into a commercial in-memory database system with a compression-optimized storage. Our analysis shows that most processing time is spent on transforming the data into a compressed format—not on deserializing the file. Moreover, we confirm that state-of-the-art bulk loading significantly degrades tail latency of a query workload running in parallel, while the performance of the bulk loading suffers as well.

To mitigate this problem, we propose *Shared Loading*, a distributed bulk loading mechanism that enables *dynamically offloading* deserialization and data transformation to the client machine holding the file. Our evaluation using the lineitem table of the TPC-H benchmark and a real-world data set determines that *Shared Loading* increases bulk loading throughput especially in slower network environments or when the DBMS is stressed. At the same time, it can significantly improve tail latency of a query workload to enable efficient bulk loading into compression-optimized storage without sacrificing query performance and predictability.

7. REFERENCES

[1] Abouzied et al. Invisible loading: Access-driven data transfer from raw files into database systems. In *Proc. EDBT*, 2013.

[2] Alagiannis et al. NoDB: Efficient query execution on raw data files. In *Proc. SIGMOD*, 2012.

[3] Amazon. EC2 instance types. [Online].

[4] Boncz et al. Breaking the memory wall in MonetDB. *Commun. ACM*, 2008.

[5] Cheng et al. Parallel in-situ data processing with speculative loading. In *Proc. SIGMOD*, 2014.

[6] Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *Proc. PODS*, 2014.

[7] Dean et al. The tail at scale. *Commun. ACM*, 2013.

[8] DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, 2007.

[9] Dziedzic et al. DBMS data loading: An analysis on modern hardware. In *In Proc. ADMS/IMDM*, 2016.

[10] Färber et al. The SAP HANA database – an architecture overview. *Data Eng. Bull.*, 2012.

[11] Flajolet et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. AOFA*, 2007.

[12] Ge et al. Speculative distributed csv data parsing for big data analytics. In *Proc. SIGMOD*, 2019.

[13] Google. Virtual private cloud resource quotas. [Online].

[14] IDC. Worldwide Ethernet switch and router trackers, 2019. [Online].

[15] Lamb et al. The Vertica analytic database: C-store 7 years later. *PVLDB*, 2012.

[16] Microsoft. The Data Loading Performance Guide, 2009. [Online].

[17] Mühlbauer et al. Instant loading for main memory databases. *PVLDB*, 2013.

[18] Oracle Database 18c Documentation. [Online].

[19] Raasveldt et al. Don't hold my data hostage: A case for client protocol redesign. *PVLDB*, 2017.