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

Stefan Noll
SAP SE & TU Dortmund University
stefan.noll@sap.com

Jens Teubner
TU Dortmund University
jens.teubner@cs.tu-dortmund.de

Norman May
SAP SE
norman.may@sap.com

Alexander Böhm
SAP SE
alexander.boehm@sap.com

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 running in parallel.

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. Popular formats include delimiter-separated values files, such as CSV, fixed-width values files, JSON, or XML. Plain text files are frequently used to transfer scientific data sets [44] 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, 10, 25, 35] assumes that data resides on local storage where the DBMS is installed on. 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.

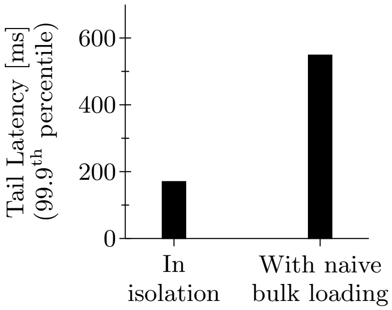


Figure 1: Tail latency of query workload when executed in isolation and in parallel to bulk loading without *Shared Loading*. Tail latency degrades by a factor of 3.2 under load.

In addition, related work [2, 32] primarily 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 [9, 16, 27, 29, 38] 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 commercial database 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. Figure 1 illustrates this problem. We execute a simple analytical workload both in isolation and in parallel to the bulk loading of the `lineitem` table of the TPC-H benchmark. We observe that tail latency degrades by a factor of 3.2 when bulk loading runs in parallel.

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 bulk loading and improves the performance and predictability of concurrent query processing. Its architecture allows dynamically adapting data transfor-

10th Annual Conference on Innovative Data Systems Research (CIDR '20)
January 12-15, 2020, Amsterdam, Netherlands.

mations and shifting work between client(s) and server *at loading time*—without the need for the user to partition the input data or to manually parallelize bulk loading [4,30,36]. 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 we study whether it improves tail latency of concurrently running queries for different network bandwidths and data sets.

2. RELATED WORK & BACKGROUND

Bulk Loading. Database systems offer a range of interfaces for bulk loading external files. We group existing approaches into three categories.

First, various systems offer a *command* for a terminal-based front-end. The user either manually transfers the file to the DBMS server or it is transferred during the loading operation. All data processing is done by the DBMS server. Examples include the commands `IMPORT FROM` from IBM Db2 [22], `BULK INSERT` from Microsoft SQL Server [31] or `COPY` from PostgreSQL [39].

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. The database system may support array bindings natively by processing a single SQL statement per batch or it may emulate array bindings by processing a prepared statement per row of a batch. Most systems with support for ODBC [19] or JDBC [5] support array bindings.

Third, some vendors provide *dedicated tools* for bulk loading external files. Some of these tools use array bindings, such as `bcp` from Microsoft [31], while other tools, such as `SQL*Loader` from Oracle [36] or the `LOAD` utility from Db2 [22], write data blocks "directly to the database". Their documentation does not detail, however, what is computed by the client and by the server.

Dziedzic et al. [15] analyze data loading with terminal-based commands 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, 8, 10, 18] focuses on analyzing large files with the goal to minimize the initial response times. Thus, first loading the entire file into a DBMS incurs a high initial cost and is usually avoided. In addition, it may suffice to read only a subset of the data for answering a query. Related work identifies the repeated parsing, tokenizing, and data type conversion as a performance bottleneck but does not consider network transfers or complex transformations.

Fast Parsing and Ingestion. Mühlbauer et al. [32] use SIMD instructions to accelerate deserialization and propose mergeable indexes for bulk loading. More recently, Langdale et al. [28] present techniques, e.g., for identifying escaped quote characters and converting digits to integer values using recent SIMD instructions. Xie et al. [46] propose a storage layer for analytics that combines parsing of selected fields and a dynamic, hash-based subset index for querying data during ingestion. They store data in an immutable, row-based log and do not consider data transformations such as compression or a columnar storage layout. We do not focus on optimizing parsing and deserialization because our analysis (see Section 3) shows that these operations consume less than 20 % CPU time in a commercial database system. Besides, optimizing parsing and deserialization by applying recent SIMD techniques and using just-in-time generated glue code is orthogonal to our work. We focus on a distributed environment where our approach can dynamically shift data transformations between client(s) and server to bring data into the optimized storage format of the database system.

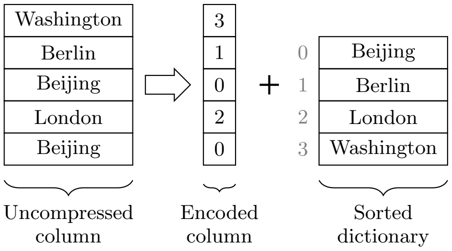


Figure 2: Example of applying order-preserving dictionary compression to a column.

Network Serialization. Raasveldt et al. [40] analyze the data *export* from various database systems. They conclude that protocols suffer from a per-row overhead and expensive (de)serialization. They propose to transfer data in column-major chunks with variable-sized strings as well as to employ columnar compression techniques. Our network protocol resembles their design. However, we transfer larger chunks of data, i.e., 500 MiB not 1 MB, and we choose to represent strings fixed-sized instead of variable-sized to simplify in-place sorting. We reduce transfer volume by employing dictionary compression.

Dictionary Compression. The in-memory database system SAP HANA [16] makes heavy use of *order-preserving* dictionary compression in its read-optimized storage—other systems employ similar ideas to varying degrees [9,27,29,38]. Figure 2 gives a short example. 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 typically much smaller index of the dictionary entry. The compression works especially well for large string columns with a low distinct count. Such data is very common in real-world business applications [7,33]. 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 as shown in Figure 3. New records are first appended to a write-optimized column store. Values are dictionary-

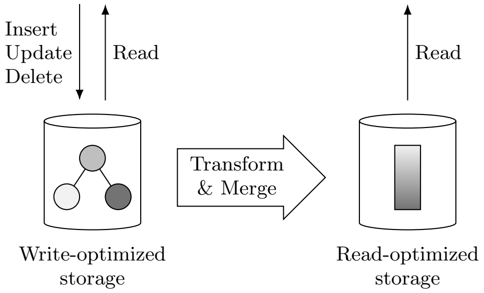


Figure 3: Buffered updates in SAP HANA. Records migrate from write- to read-optimized storage.

compressed without preserving the order in the encoding. A CSB$^+$-tree [41] provides a sorted view of the unsorted dictionary to accelerate accesses. Eventually, data is merged into the read-optimized column store, which employs more elaborate compression methods, such as order-preserving dictionary encoding.

Our approach bypasses the write-optimized storage and merges new data directly into the read-optimized storage—similar to Lamb et al. [27]. 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 in SAP HANA, we expect similar results for other systems with (complex) data transformations—especially for systems with a compressed storage. First, we describe the individual steps of a bulk loading operation. Then, we demonstrate where time is lost.

Loading Steps. Bulk loading starts with the *deserialization* of the file. For a delimiter-separated values file, it *parses* the file contents in search for symbols marking the end of a row (e.g., '`\n`') or the end of a column (e.g., '`|`') and splits the character stream into individual fields of the input table. It *validates* whether a value conforms to the specification of the SQL type given by the schema. If the validation succeeds, the operation creates an *instance* of the SQL data type in memory; otherwise it handles the error.

In addition, the system *transforms* new data into the physical storage layout, such as a row-based or a columnar representation. This might involve complex restructuring and different compression methods. In SAP HANA, new records migrate from a compressed, write-optimized storage to a compressed, read-optimized storage. If the target table is partitioned, each row needs to be assigned to its correct *partition*. If the system is distributed, it might be necessary to *route* data over the network to another node. The system also checks *constraints*, such as the absence of null values, and *updates metadata*, such as indexes or statistics. Ultimately, it writes a *log* to persistent storage to assure durability. Systems maintaining a (read-)optimized storage additionally *merge* new data (periodically) into their optimized storage. To speed up recovery, the optimized storage may be written to *persistent* storage.

Loading Costs. To analyze where times is lost, we bulk load the `lineitem` table of the TPC-H benchmark from a local solid-state drive. The experimental setup is described in Section 5.1. The results shown in Figure 4 demonstrate that data transformations consume 55 % CPU time. This includes the time it takes to insert new rows into the write-optimized storage and to compute a compressed, columnar in-memory representation.

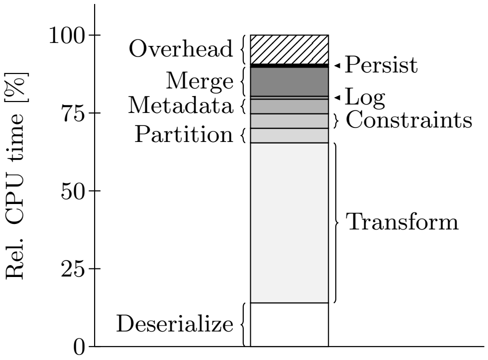


Figure 4: Cost analysis of bulk loading into SAP HANA. Most CPU time is spent on data transformation.

Deserialization consumes around 15 % CPU time. Checking constraints, partitioning the table, or updating meta data 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 of CPU time due to asynchronous I/O. The remaining 10 % are overhead from the transaction manager, lock handling, and memory management.

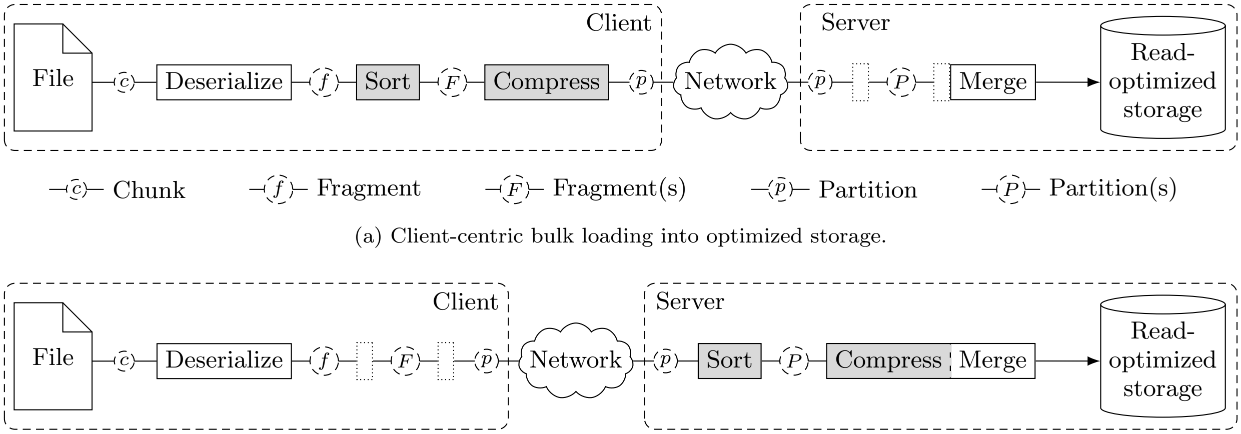
In summary, our results differ from previous results [2, 10, 32] that attribute the highest cost for bulk loading to deserialization. For compression-optimized systems, the cost of *transforming* the data dominates computing time and outweighs the cost of deserialization by a factor of **3.7**.

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*.

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 in a column store, but the concept of dynamically offloading deserialization and data transformation may be applicable to other storage formats and compression techniques. Note that we propose that the client transforms data *towards* the storage format. This allows the server to ingest data with little effort. The server still needs to validate all client data.



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

(b) Server-centric bulk loading into optimized storage

Figure 5: 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* (cf. Figure 6) where to compute its data transformation.

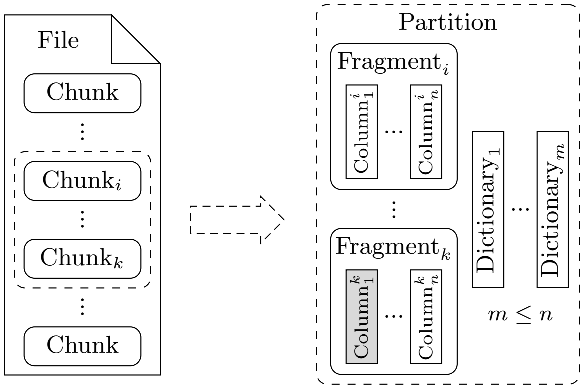


Figure 6: *Shared Loading* transforms a file chunk into a columnar in-memory fragment. Multiple fragments form a logical partition. A fragment's column (gray) may be dictionary-compressed.

Figure 5 gives an overview of the data flow and the processing steps on client and server. First, we introduce client-centric loading shown in Figure 5a. Afterwards, we present server-centric loading shown in Figure 5b. Finally, 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 enabling a high degree of parallelism. It produces horizontal *partitions* of the input table as shown in Figure 6. When we shift data transformations to the client machine, the client component of *Shared Loading* produces a dictionary-compressed, columnar partition and sends it to the server machine. This allows the DBMS to merge a partition efficiently into its optimized storage. We describe the individual steps 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 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. For each column of the fragment, it creates a copy, sorts the copy, and removes duplicates. The temporary dictionary facilitates dictionary encoding in the next step.

The *compression* step logically assembles multiple fragments into a horizontal partition shown in Figure 6, and (physically) merges all temporary dictionaries of a column into a single dictionary. This means that a partition's dictionary is locally ordered—not globally with respect to the entire table, previous partitions or preexisting data on the server. We assure that the order of rows matches their original occurrence in the input file. This preserves the ordering if the file contents are already sorted. Afterwards, it uses the dictionary to encode the columns of the fragments.

The *transfer* step sends a partition to the server. The client component transfers fragment after fragment. This enables the server component to process a fragment before the partition is transferred completely. 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.

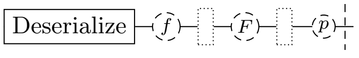
Server Component. The server component is designed to be part of the database system with internal access to the storage engine. When we shift the data transformation to the client, it receives a dictionary-compressed, columnar partition. This allows the DBMS to merge the partition efficiently into its read-optimized storage by updating the dictionary encoding (instead of creating the dictionary encoding from scratch) and speeds up network transfers.

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 incoming data into a specific partition of the target table or we could create a new partition to avoid updating the dictionary compression.

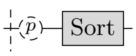
4.2 Server-Centric

Client Component. When we shift only deserialization to the client, but transform data on the server, the client component of *Shared Loading* produces an uncompressed, columnar partition (see Figure 5b).



The *deserialization* step produces a fragment just as in case of client-centric loading, described in Section 4.1. 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. It needs to transform the data before merging it into optimized storage.



The input for the *sort* step is a partition consisting of multiple fragments. 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 6, 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 of the bulk loading pipeline adapt to the decision (dotted boxes in Figure 5). 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 (see Figure 6). 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 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 [17] with HIP estimator [11, 34] *at loading time*. 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. We demonstrate that the heuristic produces indeed the smallest size in our evaluation.

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, e.g., to implement a feedback loop.

4.4 Implementation

Our C++ implementation of the data processing pipeline exploits independent work whenever possible to achieve a high degree of parallelism. The implementation is independent of the code base of SAP HANA, but simulates major characteristics of SAP HANA.

We execute each pipeline step in at least one thread. In addition, we use the thread pool provided by Intel Thread Building Blocks [24] to implement fine-grained task parallelism: we sort, compress and merge columns in parallel and we implement parallel algorithms for compressing and merging, e.g., a parallel merge algorithm similar to [12, p. 800].

We use different sorting algorithms depending on the data type: a radix sort implementation for integers and dates, `boost::string_sort` [42, 43] for strings, and `pdqsort` [37] for the remaining types. To facilitate in-place sorting, we represent variable-sized strings of type `VARCHAR(N)` as *fixed-sized* strings of length `N` in partitions sent by the client, while the server stores variable-sized strings. Note that we avoid the increased memory and transfer size of fixed-sized strings by employing dictionary compression. To implement asynchronous network communications, we use the library `boost::asio` [26].

We currently do not support delimiter symbols marking the end of a column or the end of a row when they are enclosed in quotes without being escaped. This is orthogonal to our work and only affects the deserialization step of the bulk loading. To identify unescaped delimiter symbols, others propose, e.g., to perform an additional scan over the input file [32] or to employ speculative parsing [18].

We additionally annotate each chunk/fragment with an identifier and assure that the order of the rows in a partition corresponds to the order of the rows in the file. This allows the database system to exploit the fact that rows in a file might be sorted, e.g., by the primary key to accelerate index creation. 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 and query performance and predictability.

5.1 Setup

Data Set. We evaluate two data sets: the `lineitem` table of the TPC-H benchmark [45] 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, close to $6 \cdot 10^7$ rows, and 16 columns. The file of the `warehouse` table has a size of 17.57 GiB, around $12 \cdot 10^6$ rows, and 155 columns.

Hardware. Our system has 128 GiB of DRAM and two Intel Xeon E5-2660 v3 processors with 10 physical cores. We enable simultaneous multithreading. 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 text files reside on a local SSD. We measured a sequential read bandwidth of up to 530 MB/s using `fio` [6]. Note that in case of a significantly slower storage devices, such as a single HDD, a weak client could compute all data transformations without ever being 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. We clear the page cache of the Linux kernel (LTS version 4.4) before every run.

Network. We use the `tc` utility to emulate different network bandwidths—similar to [40]. TCP/IP messages are sent to the localhost address. Note that we profiled the transfer of a file both between two machines in a local network and between two processes via the localhost address on the same machine, where we emulated the same bandwidth: we did not observe a difference in execution or CPU time.

We evaluate a network bandwidth of 1 and 10 Gbit/s because these represent 69 % of the market share [23]. 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, 20] when performing a cloud-internal bulk loading operation.

Configurations. We evaluate *Shared Loading* in three different configurations: SL_A corresponds to Figure 5a, 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 5b, 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. Note that in all configurations we bypass the write-optimized storage and ingest data directly into the read-optimized storage.

Query Workload. We use two analytical queries inspired by the TPC-H benchmark to evaluate the impact of concurrent query processing:

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

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. This way, we induce a constant query load from the time the client starts reading the file until the server has merged all data into its read-optimized storage. For a given data set and network bandwidth, we base the duration of the query workload on the maximum loading time of all configurations.

Measurement Method. We measure *throughput*, i.e., the size of the file divided by the elapsed time of bulk loading. In addition, we measure *CPU time* of the bulk loading to quantify computational work. CPU time is the total time which processor cores spent on executing instructions. This does not include idle time, i.e., the time cores spend waiting for asynchronous I/O operations.

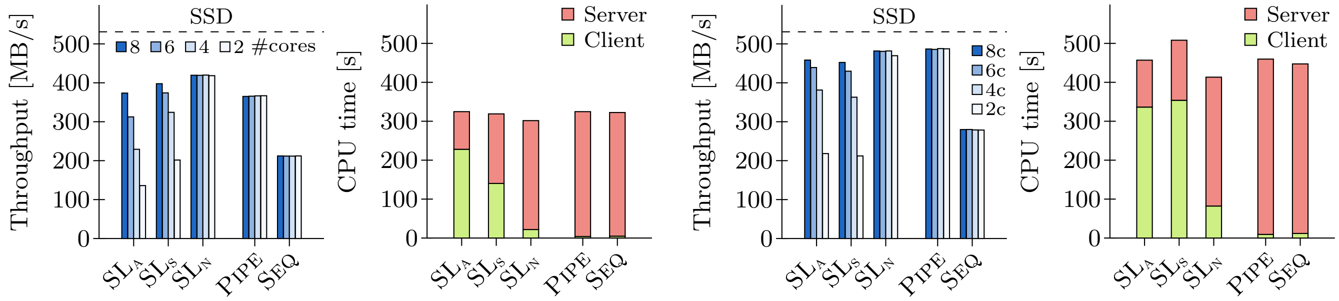
To evaluate the impact of query processing, we measure *tail latency*—a performance metric for mission-critical systems with stringent service level agreements [13, 14, 21]. In particular, we focus on the maximum response time of 99.9 % of the requests, i.e., the 99.9^{th} latency percentile. According to our experience, customers often prefer predictable response times over peak performance.

5.2 Loading in Isolation

10-Gbit Network. Figure 7 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.

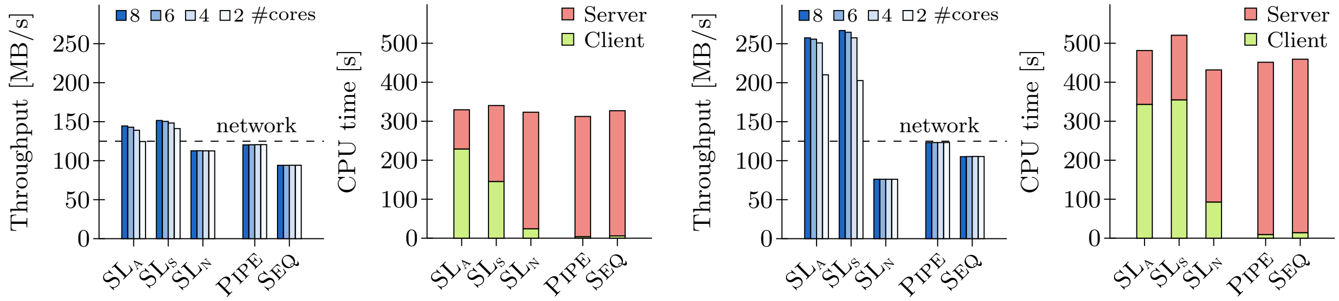
The throughput of *Shared Loading* is comparable to the state of the art and in some cases even up to 15 % higher. We observe that without concurrent query processing shifting only deserialization to the client (SL_N) results in the highest throughput. The results come as no surprise because the bulk loading is not network-bound but limited by SSD's read-bandwidth and the client's compute capability.



(a) `lineitem`

(b) `warehouse`

Figure 7: Bulk loading over a 10-Gbit network without query processing.



(a) `lineitem`

(b) `warehouse`

Figure 8: Bulk loading over a 1-Gbit network without query processing.

	SL$_A$	SL$_S$	SL$_N$	Pipe	Seq
`lineitem`	5.80	5.58	7.65	7.24	7.24
`warehouse`	8.06	7.76	28.33	17.57	17.57

Table 1: The amount of data (in GiB) per data set and configuration that is transferred over the network during bulk loading.

1-Gbit Network. Figure 8 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 and that throughput differs significantly between configurations. We attribute this to the amount of data, shown in Table 1, that the client transfers to the server. This is especially notable for the `warehouse` table, where dictionary compression achieves a compression ratio of 2.3. Note that SL$_N$ increases the transfer size compared to the original file because we store strings with a fixed size (cf. Section 4.4).

We observe that in slower network environments *Shared Loading* performs best when the client selectively transforms the data in order to minimize transfer size (SL$_S$). For the `lineitem` table, SL$_S$ 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$_S$ transfers data with 44 % of the file size and it increases throughput by 117 % compared to Pipe. Thus, using the heuristic results indeed in the smallest transfer size.

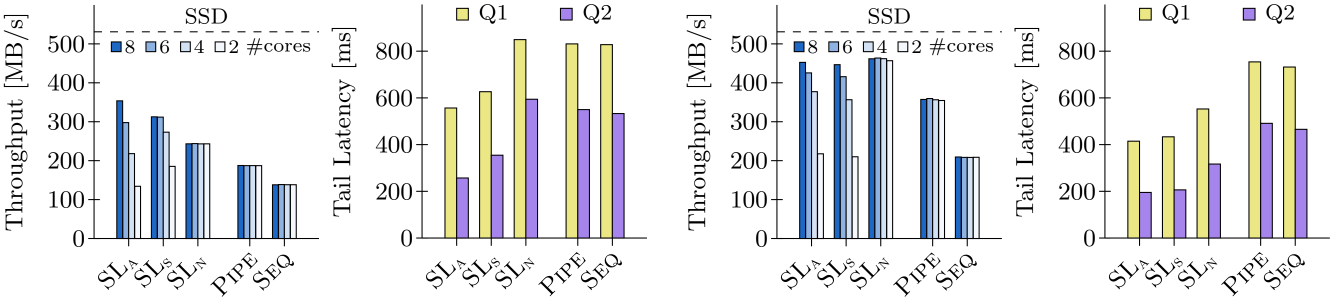
5.3 Loading With Concurrent Queries

To study whether *Shared Loading* improves throughput and query performance when the server is stressed, we perform bulk loading with queries running concurrently.

10-Gbit Network. Figure 9 shows the results for a network bandwidth of 10 Gbit/s. We notice that the results differ from previous results without query processing. When the server is stressed, *Shared Loading* performs best when the client transforms all data (SL$_A$). Compared to Pipe, SL$_A$ increases throughput by 89 % for the `lineitem` table and by 27 % 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, *Shared Loading* leaves the server more resources for query processing: When we compare Pipe with SL$_A$, tail latency improves by 33 % for Q1 and 53 % for Q2 for the `lineitem` table. For the `warehouse` table, tail latency improves by 45 % and 60 %, respectively. Thus, *Shared Loading* can reduce stress in peak load situations, which in return improves query performance and predictability.

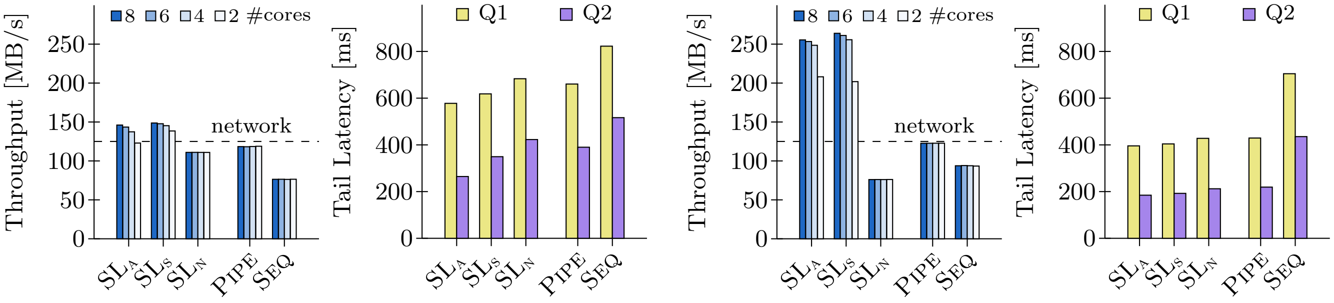
The client can be relatively weak: 4 cores suffice to transform and compress all data on the client while achieving a higher throughput than the state of the art. Average response times (not shown) improve by 19 % and 47 % for the `lineitem` table and by 13 % and 33 % for the `warehouse` table when comparing SL$_A$ with Pipe. Note that if the server is under heavy load, job scheduling cannot effectively reduce resource contention. *Shared Loading*, instead, mitigates a



(a) `lineitem`

(b) `warehouse`

Figure 9: Bulk loading over a 10-Gbit network with concurrent query processing.



(a) `lineitem`

(b) `warehouse`

Figure 10: Bulk loading over a 1-Gbit network with concurrent query processing.

high system load by leveraging the additional hardware resources of the client.

1-Gbit Network. Figure 10 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_S improves throughput by up to 26 % and 116 % compared to Pipe. In particular, we observe that throughput does not degrade even though the server is stressed. We attribute this to the low network bandwidth: it gives the server more time to process incoming partitions and makes merging the data into optimized storage less vulnerable to resource contention.

SL_A improves tail latency by 12 % for Q1 and by 32 % for Q2 compared to Pipe for the `lineitem` table. For the `warehouse` table, tail latency only improves by 8–16 %, 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. In addition, the approach is very robust. Throughput never degrades. The performance advantage in 10-Gbit environments becomes clear once the server is under load. *Shared Loading* increases throughput by up to 89 %. Tail latencies of the query workload improve by up to 60 %.

The results also demonstrate why work needs to be shifted *dynamically*: different configurations of *Shared Loading* perform best depending on network bandwidth, server load, and compute capability of the client: In fast network environments, when loading without query processing or when the client is weak, it is best to only deserialize on the client; in slow network environments, it is best to selectively transform data on the client to minimize transfer size; and when loading with concurrent query processing, it is best to transform all data on the client.

We illustrate offloading transformations dynamically by employing a heuristic that optimizes transfer size. To further improve throughput for slow networks, we combined *Shared Loading* with additional compression (not shown), such as LZ4 [47]. Combining *Shared Loading* with LZ4 compression increases throughput by up to 91 %. Compared to state-of-the-art architectures with LZ4, *Shared Loading* with LZ4 increases throughput by up to 27 % with and by up to 35 % without query processing. Tail latency improves by up to 48 %. This demonstrates that *Shared Loading* works well with other compression methods.

Ultimately, we envision the client component of *Shared Loading* to be part of a lightweight SQL client. Its complexity remains low because the client performs only deserialization and data transformation. In addition, *Shared Loading* consumes a low amount of memory. In our implementation, the client buffers only one partition in-between processing steps resulting in 7 partitions in total. If we assume the chunk and partition sizes discussed in Section 4.4, the total memory consumption will not exceed the in-memory equivalent of $7 \cdot (50 \cdot 10\,\text{MiB}) \approx 3.5\,\text{GiB}$ 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.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. The work has received funding from the Deutsche Forschungsgemeinschaft (DFG), Collaborative Research Center SFB 876, project C5 (http://sfb876.tu-dortmund.de/).

7. REFERENCES

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *Proc. EDBT*, pages 1–10, 2013.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. In *Proc. SIGMOD*, pages 241–252, 2012.

[3] Amazon. EC2 instance types. https://aws.amazon.com/ec2/instance-types/.

[4] Amazon. Importing Data From Any Source to a MySQL or MariaDB DB Instance. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/MySQL.Procedural.Importing.AnySource.html.

[5] L. Andersen. JDBC 4.3 API Specification. *Oracle Corporation*, July 2017.

[6] J. Axboe. fio – Flexible IO Tester. http://git.kernel.dk/?p=fio.git.

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proc. SIGMOD*, pages 283–296, 2009.

[8] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *Proc. SIGMOD*, pages 385–396, 2014.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[10] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *Proc. SIGMOD*, pages 1287–1298, 2014.

[11] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *Proc. PODS*, pages 88–99, 2014.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[13] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, pages 205–220, 2007.

[15] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. DBMS data loading: An analysis on modern hardware. In *In Proc. ADMS/IMDM*, pages 95–117, 2016.

[16] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *Data Eng. Bull.*, 35(1):28–33, 2012.

[17] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. AOFA*, 2007.

[18] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, and D. Kossmann. Speculative distributed CSV data parsing for big data analytics. In *Proc. SIGMOD*, pages 883–899, 2019.

[19] K. Geiger. *Inside ODBC*. Microsoft Press, Redmond, WA, USA, 1995.

[20] Google. Virtual private cloud resource quotas. https://cloud.google.com/vpc/docs/quota.

[21] B. Gregg. *Systems Performance: Enterprise and the Cloud.* Prentice Hall Press, 1st edition, 2013.

[22] IBM Db2 Version 11.1 Data Movement Utilities and Reference. https://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.admin.dm.doc/com.ibm.db2.luw.admin.dm.doc-gentopic1.html.

[23] IDC. Worldwide Ethernet switch and router trackers, 2019. https://www.idc.com/getdoc.jsp?containerId=prUS45119319.

[24] Intel. Threading Building Blocks. https://github.com/01org/tbb.

[25] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[26] C. M. Kohlhoff. Asio C++ Library. https://github.com/chriskohlhoff/asio.

[27] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica

analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[28] G. Langdale and D. Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, Dec 2019.

[29] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. In *Proc. SIGMOD*, pages 1177–1184, 2011.

[30] Microsoft. The Data Loading Performance Guide, 2009. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070(v=sql.100).

[31] Microsoft SQL Server 2017 Documentation. https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017.

[32] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.

[33] I. Müller, C. Ratsch, and F. Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proc. EDBT*, pages 283–294, 2014.

[34] H. Ohno. C++ implementation of HyperLogLog algorithm and HIP (Historic Inverse Probability) Estimator. https://github.com/hideo55/cpp-HyperLogLog.

[35] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *PVLDB*, 10(10):1106–1117, 2017.

[36] Oracle Database 18c Documentation. https://docs.oracle.com/en/database/oracle/oracle-database/18/sutil/oracle-sql-loader.html.

[37] O. Peters. Pattern-defeating Quicksort. https://github.com/orlp/pdqsort.

[38] M. Poess and D. Potapov. Data compression in Oracle. In *Proc. VLDB*, pages 937–947, 2003.

[39] PostgreSQL 11 Documentation. https://www.postgresql.org/docs/11/sql-copy.html.

[40] M. Raasveldt and H. Mühleisen. Don't hold my data hostage: A case for client protocol redesign. *PVLDB*, 10(10):1022–1033, 2017.

[41] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *Proc. SIGMOD*, pages 475–486, 2000.

[42] S. J. Ross. C++ Implementation of Templated Hybrid String Sort Algorithm in Boost Framework. https://www.boost.org/doc/libs/1_69_0/libs/sort/doc/html/boost/sort/spreadsort/string_sort_idp52153312.html.

[43] S. J. Ross. The spreadsort high-performance general-case sorting algorithm. In *Proc. PDPTA*, pages 1100–1106. CSREA Press, 2002.

[44] A. Szalay, A. R. Thakar, and J. Gray. The sqlLoader data-loading pipeline. *Computing in Science and Engg.*, 10(1):38–48, 2008.

[45] Transaction Processing Performance Council (TPC). TPC Benchmark H (Decision Support) Standard Specification. Technical report, 2018.

[46] D. Xie, B. Chandramouli, Y. Li, and D. Kossmann. Fishstore: Faster ingestion with subset hashing. In *Proc. SIGMOD*, pages 1711–1728, 2019.

[47] Y. Collet et al. LZ4. https://lz4.org/.