Instant Loading for Main Memory Databases

Tobias Mühlbauer
Technische Universität München
Munich, Germany
muehlbau@in.tum.de

Wolf Rödiger
Technische Universität München
Munich, Germany
roediger@in.tum.de

Robert Seilbeck
Technische Universität München
Munich, Germany
seilbeck@in.tum.de

Angelika Reiser
Technische Universität München
Munich, Germany
reiser@in.tum.de

Alfons Kemper
Technische Universität München
Munich, Germany
kemper@in.tum.de

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

eScience and big data analytics applications are facing the challenge of efficiently evaluating complex queries over vast amounts of structured text data archived in network storage solutions. To analyze such data in traditional disk-based database systems, it needs to be *bulk loaded*, an operation whose performance largely depends on the wire speed of the data source and the speed of the data sink, i.e., the disk. As the speed of network adapters and disks has stagnated in the past, loading has become a major bottleneck. The delays it is causing are now ubiquitous as text formats are a preferred storage format for reasons of portability.

But the game has changed: Ever increasing main memory capacities have fostered the development of in-memory database systems and very fast network infrastructures are on the verge of becoming economical. While hardware limitations for fast loading have disappeared, current approaches for main memory databases fail to saturate the now available wire speeds of tens of Gbit/s. With *Instant Loading*, we contribute a novel CSV loading approach that allows *scalable bulk loading at wire speed*. This is achieved by optimizing all phases of loading for modern super-scalar multi-core CPUs. Large main memory capacities and Instant Loading thereby facilitate a very efficient data staging processing model consisting of *instantaneous load*-work-unload cycles across data archives on a single node. Once data is loaded, updates and queries are efficiently processed with the flexibility, security, and high performance of relational main memory databases.

1. INTRODUCTION

The volume of data archived in structured text formats like comma-separated values (CSV) has grown rapidly and continues to do so at an unprecedented rate. Scientific data sets such as the Sloan Digital Sky Survey and Pan-STARRS are stored as image files and, for reasons of portability and debugability, as multi-terabyte archives of derived CSV files that are frequently loaded to databases to evaluate complex

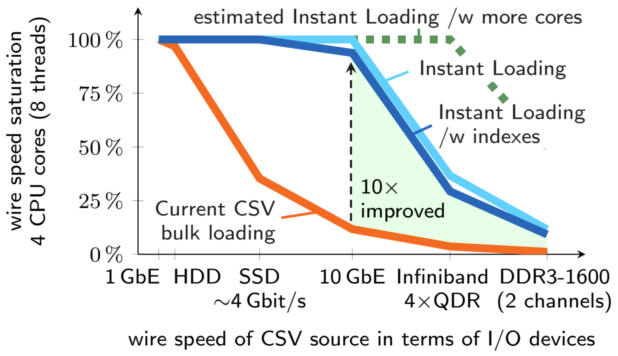


Figure 1: Pushing the envelope: wire speed saturation of current bulk loading vs. Instant Loading.

queries [27, 26]. Other big data analytics and business applications are equally faced with the need to analyze similar archives of CSV and CSV-like data [25, 26]. These archives are usually stored externally from the database server in a network-attached storage (NAS) or distributed file system (DFS) or locally in a SSD/RAID storage.

To efficiently analyze CSV archives, traditional databases can do little to overcome the premise of loading. The cost of parsing, deserializing, validating, and indexing structured text data needs to be paid either up front during a *bulk load* or lazily during query processing on external tables. The performance of loading largely depends on the wire speed of the data source and the speed of the data sink, i.e., the disk. As the speed of network adapters and disks has stagnated in the past, loading has become a major bottleneck and the delays it is causing are now ubiquitous.

But the game has changed: Ever increasing main memory capacities have fostered the development of in-memory database systems and modern network infrastructures as well as faster disks are on the verge of becoming economical. Servers with 1 TB of main memory and a 10 GbE adapter (10 Gbit/s ≈ 1.25 GB/s wire speed) already retail for less than $30,000. On this modern hardware, the loading source and sink are no longer the bottleneck. Rather, current loading approaches for main memory databases fail to saturate the now available wire speeds. With *Instant Loading*, we contribute a novel CSV loading approach that allows *scalable bulk loading at wire speed* (see Fig. 1). This makes the delays caused by loading unobtrusive and relational main memory databases attractive for a very efficient data staging processing model consisting of *instantaneous load*-work-unload cycles across CSV data archives on a single node.

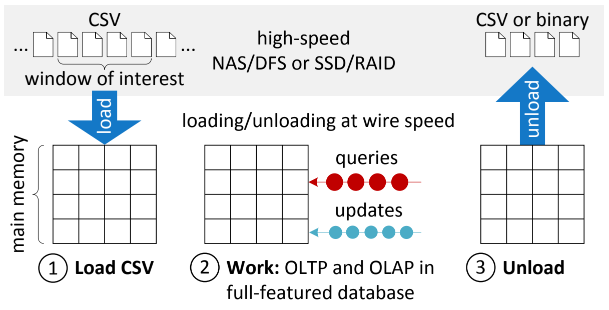
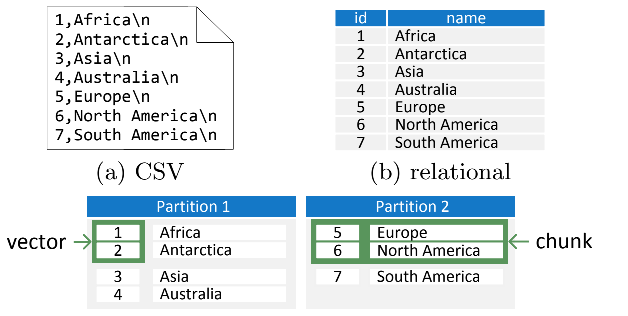


Figure 2: Instant Loading for data staging processing: load-work-unload cycles across CSV data.



```
1,Africa\n
2,Antarctica\n
3,Asia\n
4,Australia\n
5,Europe\n
6,North America\n
7,South America\n
```

(a) CSV

id	name
1	Africa
2	Antarctica
3	Asia
4	Australia
5	Europe
6	North America
7	South America

(b) relational

(c) physical (chunk-based column-store)

Figure 3: Continent names in three representations: (a) CSV, (b) relational, and (c) physical.

Contributions. To achieve instantaneous loading, we optimize CSV bulk loading for modern super-scalar multi-core CPUs by task- and data-parallelizing all phases of loading. In particular, we propose a task-parallel CSV processing pipeline and present generic high-performance parsing, deserialization, and input validation methods based on SSE 4.2 SIMD instructions. While these already improve loading time significantly, other phases of loading become the bottleneck. We thus further show how copying deserialized tuples into the storage backend can be sped up and how index creation can efficiently be interleaved with parallelized bulk loading using merge-able index structures (e.g., hashing with chaining and the adaptive radix tree (ART) [20]).

To prove the feasibility of our generic Instant Loading approach, we integrate it in our main memory database system HyPer [19] and evaluate our implementation using the industry-standard TPC benchmarks. Results show improvements of up to a factor of 10 on a quad-core commodity machine compared to current CSV bulk loading in main memory databases like MonetDB [4] and Vectorwise. Our implementation of the Instant Loading approach aims at highest performance in an in-memory computation setting where raw CPU costs dominate. We therefore strive for good code and data locality and use light-weight synchronization primitives such as atomic instructions. As the proportion of sequential code is minimized, we expect our approach to scale with faster data sources and CPUs with ever more cores.

Instant Loading in action: the *(lwu) data staging processing model.** Servers with 1 TB of main memory and more offer enough space to facilitate an in-memory analysis of large sets of structured text data. However, currently the adoption of databases for such analysis tasks is hindered by the inefficiency of bulk loading (cf., Sect. 3.1). With Instant Loading we remove this obstacle and allow a novel data staging processing model consisting of *instantaneous load*-work-unload cycles *(lwu)** across windows of interest.

Data staging workflows exist in eScience (e.g., astronomy and genetics [27, 26]) and other big data analytics applications. For example, Netflix, a popular on-demand media streaming service, reported that they are collecting 0.6 TB of CSV-like log data in a DFS per day [11]. Each hour, the last hour's structured log data is loaded to a 50+ node Hadoop/Hive-based data warehouse, which is used for the extraction of performance indicators and for ad-hoc queries. Our vision is to use Instant Loading in a single-node main memory database for these kinds of recurring load-work-unload workflows. Fig. 2 illustrates our three-step *(lwu)** approach. ①: A window of interest of hot CSV files is loaded from a NAS/DFS or a local high-performance SSD/RAID to a main memory database at wire speed. The window of interest can even be bigger than the size of the main memory as selection predicates can be pushed into the loading process. Further, data can be compressed at load time. ②: The full set of features of a relational main memory database—including efficient support for queries (OLAP) and transactional updates (OLTP)—can then be used by multiple users to work on the window of interest. ③: Prior to loading new data, the potentially modified data is unloaded in either a (compressed) binary format or, for portability and debugability, as CSV. Instant Loading is the essential backbone that facilitates the *(lwu)** approach.

Comparison to MapReduce approaches. Google's MapReduce [5] (MR) and its open-source implementation Hadoop brought along new analysis approaches for structured text files. While we focus on analyzing such files on a single node, these approaches scale jobs out to a cluster of nodes. By working on raw files, MR requires no explicit loading like relational databases. On the downside, a comparison of databases and MR [23] has shown that databases are, in general, much easier to query and significantly faster at data analysis. Extensions of MR and Hadoop like Hive [28] and HAIL [7] try to close this gap by, e.g., adding support for declarative query languages, indexes, and data preprocessing. As for comparison of MR with our approach, Instant Loading in its current state aims at accelerating bulk loading on a single database node—that could be part of a cluster of servers. We see scaleout of query and transaction processing as an orthogonal direction of research. Nevertheless, MR-based systems can as well profit from the generic high-performance CSV parsing and deserialization methods proposed in this work.

2. DATA REPRESENTATIONS

An important part of bulk loading is the transformation and reorganization of data from one format into another. This paper focuses on the comma separated values (CSV), relational, and common physical representations in main memory database systems; Fig. 3 illustrates these three.

CSV representation. CSV is a simple, yet widely used data format that represents tabular data as a sequence of characters in a human readable format. It is in many cases the least common denominator of information exchange. As such, tera-scale archives of CSV and CSV-like data exist in eScience and other big data analytics applications [27, 26, 25]. Physically, each character is encoded in one or several bytes of a character encoding scheme, commonly ASCII or UTF-8. ASCII is a subset of UTF-8, where the 128 ASCII characters correspond to the first 128 UTF-8 characters. ASCII characters are stored in a single byte where the high bit is not set. Other characters in UTF-8 are represented by sequences of up to 6 bytes where for each byte the high bit is set. Thus, an ASCII byte cannot be part of a multi-byte sequence that represents a UTF-8 character. Even though CSV is widely used, it has never been fully standardized. A first approach in this direction is the RFC 4180 [30] proposal which closely resembles our understanding of CSV. Data is structured in records, which are separated by a record delimiter (usually `'\n'` or `"\r\n"`). Each record contains fields, which are again separated by a field delimiter (e.g., `','`). Fields can be quoted, i.e., enclosed by a quotation character (e.g., `'"'`). Inside a quoted field, record and field delimiters are not treated as such. Quotation characters that are part of a quoted field have to be escaped by an escape character (e.g., `'\'`). If the aforementioned special characters are user-definable, the CSV format is highly portable. Due to its tabular form, it can naturally represent relations, where tuples and attribute values are mapped to records and fields.

Physical representations. Databases store relations in a storage backend that is optimized for efficient update and query processing. In our HyPer main memory database system, a relation can be stored in a row- or a column-store backend. A storage backend is structured in partitions, which horizontally split the relation into disjoint subsets. These partitons store the rows or columns in either contiguous blocks of memory or are again horizontally partitioned into multiple chunks (*chunked backend*, cf., Fig 3(c)), a technique first proposed by MonetDB/X100 [4]. The combination of these options gives four possibile types of storage backends: contiguous memory-based/chunked row-/column-store. Most, if not all, main memory database systems, including MonetDB, Vectorwise, and SAP HANA implement similar storage backends. Instant Loading is designed for all of the aforementioned types of storage backends and is therefore a generic approach that can be integrated into various main memory database systems.

This work focuses on bulk loading to uncompressed physical representations. Dictionary encoding can, however, be used in the CSV data or created on the fly at load time.

3. INSTANT LOADING

3.1 CSV Bulk Loading Analysis

To better understand how bulk loading of CSV data on modern hardware can be optimized, we first analyzed why it currently cannot saturate available wire speeds. The standard single-threaded implementation of CSV bulk loading in our HyPer [19] main memory database system achieves a loading throughput of around 100 MB/s for 10 GB of CSV data stored in an in-memory file system[1]. This is comparable to the CSV loading throughput of other state of the art main memory databases like MonetDB [4] and Vectorwise, which we also evaluated. The measured loading throughputs of 100 MB/s, however, do not saturate the available wire speed of the in-memory file system. In fact, not even a SSD (500 MB/s) or 1 GbE (128 MB/s) can be saturated. A `perf` analysis shows that about 50% of CPU cycles are spent on parsing the input, 20% on deserialization, 10% on inserting tuples into the relation, and finally 20% on updating indexes.

In our standard approach, parsing is expensive as it is based on a character at a time comparison of CSV input and special characters, where each comparison is implemented as an `if-then` conditional branch. Due to their pipelined architecture, current general purpose CPUs try to predict the outcome of such branches. Thereby, a mispredicted branch requires the entire pipeline to be flushed and ever deeper pipelines in modern CPUs lead to huge branch miss penalties [2]. For CSV parsing, however, the comparison branches can hardly be predicted, which leads to almost one misprediction per field and record delimiter of the CSV input.

Each value found by the parser needs to be deserialized. The deserialization method validates the string input and transforms the string value into its data type representation in the database. Again, several conditional branches lead to a significant number of branch miss penalties.

Parsed and deserialized tuples are inserted into the relation and are indexed in the relation's indexes. Inserting and indexing of tuples accounts for 30% of loading time and is not the bottleneck in our standard loading approach. Instead, our experiment revealed that the insertion and indexing speed of HyPer's partitioned column-store backend exceeds the speed at which standard parsing and deserialization methods are able to produce new tuples.

3.2 Design of the Instant Loading Pipeline

The aforementioned standard CSV bulk loading approach follows a single-threaded execution model. To fully exploit the performance of modern super-scalar multi-core CPUs, applications need to be highly parallelized [17]. Following Amdahl's law the proportion of sequential code needs to be reduced to a minimum to achieve maximum speedup.

We base our implementation of Instant Loading on the programming model of the Intel Threading Building Blocks (TBB) [24] library. In TBB, parallelism is exposed by the definition of *tasks* rather than threads. Tasks are dynamically scheduled and executed on available hardware threads by a run-time engine. The engine implements *task stealing* for workload balancing and reuses threads to avoid initialization overhead. Task-based programming allows to expose parallelism to a great extent.

Instant Loading is designed for high scalability and proceeds in two steps (see Fig. 4). ①st, CSV input is chunked and CSV chunks are processed by unsynchronized tasks. Each task parses and deserializes the tuples in its chunk. It further determines a tuple's corresponding partition (see Sect. 2 for a description of our partitioned storage backend) and stores tuples that belong to the same partition in a common buffer which we refer to as a partition buffer. Partition buffers have the same physical layout (e.g., row or columnar) as the relation partition, such that no further transformation is necessary when inserting tuples from the buffer into the relation partition. Additionally, tuples in partition buffers are indexed according to the indexes defined for the relation. In a ②nd step, partition buffers are merged with the corresponding relation partitions. This includes merging of tuples and indexes. While CSV chunk processing is performed in parallel for each CSV chunk, merging with relation partitions is performed in parallel for each partition.

[1]For lack of a high-speed network-attached storage or distributed file system in our lab, we used the in-memory file system `ramfs` as the loading source to emulate a CSV source wire speed of multiple GB/s.

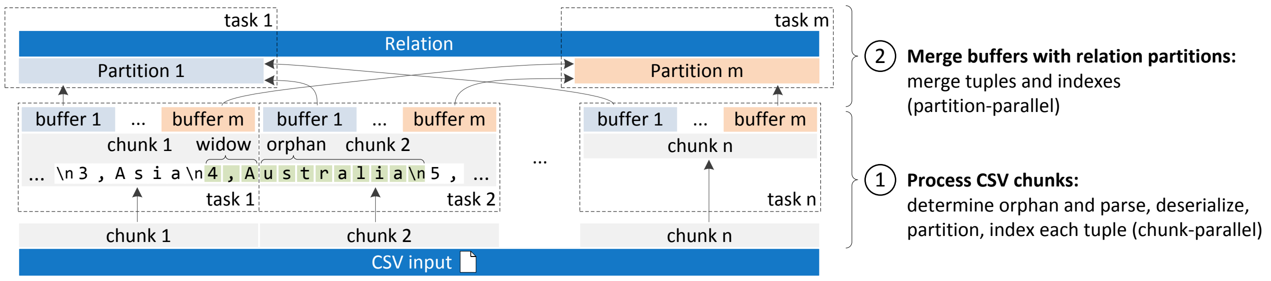

task 1
task m
Relation
Partition 1
Partition m
buffer 1
...
buffer m
chunk 1
widow
orphan
chunk 2
... \n 3 , A s i a \n 4 , A u s t r a l i a \n 5 , ...
task 2
chunk n
task n
CSV input
2 Merge buffers with relation partitions:
merge tuples and indexes
(partition-parallel)
1 Process CSV chunks:
determine orphan and parse, deserialize,
partition, index each tuple (chunk-parallel)

Figure 4: Schematic overview of Instant Loading: from CSV input to relation partitions.

3.3 Task-Parallelization

To allow synchronization-free task-parallelization of parsing, deserialization, partition classification, and indexing, we split CSV input into independent CSV chunks that can be processed in parallel. The choice of the chunk size granularity is challenging and impacts the parallelizability of the bulk loading process. The smaller the chunk size, the more chunk processing and merge steps can be interleaved. However, chunks should not be too small, as otherwise the overhead of dealing with incomplete tuples at chunk borders increases. Instant Loading splits the input according to a size for which it can at least be guaranteed that, assuming the input is well-formed, one complete tuple fits into a CSV chunk. Otherwise, parallelized parsing would be hindered. To identify chunk sizes that allow for high-performance loading, we evaluated our Instant Loading implementation with varying chunk sizes (see Fig. 13). The evaluation leads us to the conclusion that on a CPU with a last-level cache of size l and n hardware threads, the highest loading throughput can be achieved with a CSV chunk size in the range of $0.25 \times l/n$ to $1.0 \times l/n$. E.g., a good chunk size on a current Intel Ivy Bridge CPU with a 8 MB L3 cache and 8 hardware threads is in the range of 256 kB to 1 MB. When loading from a local I/O device, we use `madvise` to advise the kernel to prefetch the CSV chunks.

Chunking CSV input according to a fixed size produces incomplete tuples at CSV chunk borders. We refer to these tuples as widows and orphans (cf., Fig. 4):

***Definition* (Widow and orphan).** *"An orphan has no past, a widow has no future"* is a famous mnemonic in typesetting. In typesetting, a widow is a line that ends and an orphan is a line that opens a paragraph and is separated from the rest of the paragraph by a page break, respectively. Chunking CSV input creates a similar effect. A widow of a CSV chunk is an incomplete tuple at the end of a chunk that is separated from the part that would make it complete, i.e., the orphan, by a chunk border.

Unfortunately, if chunk borders are chosen according to a fixed size, CSV chunk-processing tasks can no longer distinguish between real record delimiters and record delimiters inside quoted fields, which are allowed in the RFC proposal [30]. It is thus impossible to determine the widow and orphan of a CSV chunk only by analyzing the data in the chunk. However, under the restriction that record delimiters inside quoted fields need to be escaped, widows and orphans can again be determined. In fact, as many applications produce CSV data that escapes the record delimiter inside quoted fields, we propose two loading options: a fast and a safe mode. The fast mode is intended for files that adhere to the restriction and splits the CSV input according to a fixed chunk size. A CSV chunk-processing task initially scans for the first unescaped record delimiter in its chunk[2] and starts processing the chunk data from there. When the task reaches the end of its chunk, it continues processing by reading data from its subsequent chunk until it again finds an unescaped record delimiter. In safe mode, a serial task scans the CSV input and splits it into CSV chunks of at least a certain chunk size. The task keeps track of quotation scopes and splits the input at record delimiters, such that no widows and orphans are created. However, the performance of the safe mode is determined by the speed of the sequential task. For our implementation, at a multiprogramming level of 8, the safe mode is 10% slower than the fast mode.

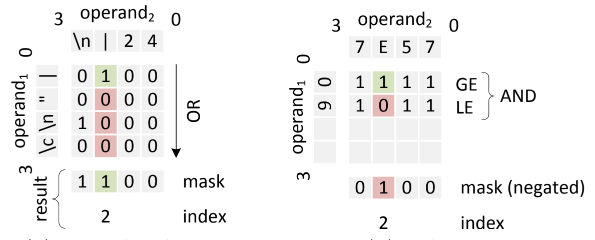
3.4 Vectorization

Parsing, i.e., finding delimiters and other special characters, and input validation are commonly based on a character at a time comparison of CSV input with certain special characters. These comparisons are usually implemented as `if-then` conditional branches. For efficient processing, current general purpose CPUs need multiple instructions in their instruction pipeline. To fill this pipeline, the hardware tries to predict upcoming branches. However, in the case of parsing and deserialization, this is not efficiently possible, which leads to a significant number of branch miss penalties [2]. It is thus desirable to reduce the number of control flow branches in the parsing and deserialization methods. One such possibility is data-parallelization.

Modern general purpose CPUs are super-scalar multi-core processors that allow not only parallelization at the task level but also at the data level—via *single instruction multiple data (SIMD)* instructions and dedicated execution units. Data parallelization is also referred to as *vectorization* where a single instruction is performed simultaneously on multiple operands, referred to as a vector. Vectorization in general benefits performance and energy efficiency [15]. In the past, SIMD extensions of x86 CPUs like SSE and 3DNow! mostly targeted multimedia and scientific computing applications. SSE 4.2 [15] adds additional byte-comparing instructions for string and text processing.

Programmers can use vectorization instructions manually via intrinsics. Modern compilers such as GCC also try to automatically vectorize source code. This is, however, restricted to specific code patterns. To the best of our knowledge, no compiler can (yet) automatically vectorize code using SSE 4.2 instructions. This is due to the fact that using these instructions requires non-trivial changes to the design of algorithms.

[2]This might require reading data from the preceeding chunk.



(a) EQUAL ANY (b) RANGES

Figure 5: SSE 4.2 comparisons: (a) searching for special characters and (b) validating characters.

Current x86 CPUs work on 128 bit SSE registers, i.e., 16 8 bit characters per register. While the AVX instruction set increased SIMD register sizes to 256 bit, the SSE 4.2 instructions still work on 128 bit registers. It is of note that we do not assume 16 byte aligned input for our SSE-optimized methods. Even though aligned loads to SIMD registers had been significantly faster than unaligned loads in the past, current generations of CPUs alleviate this penalty.

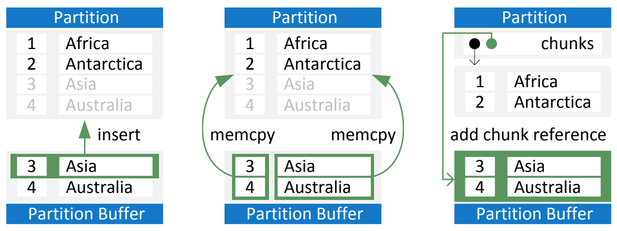
SSE 4.2 includes instructions for the comparison of two 16 byte operands of explicit or implicit lengths. We use the EQUAL ANY and RANGES comparison modes to speed up parsing and deserialization in Instant Loading: In EQUAL ANY mode, each character in the second operand is checked whether it is equal to any character in the first operand. In the RANGES mode, each character in the second operand is checked whether it is in the ranges defined in the first operand. Each range is defined in pairs of two entries where the first specifies the lower and the second the upper bound of the range. The result of intrinsics can either be a bitmask or an index that marks the first position of a hit. Results can further be negated. Fig. 5 illustrates the two modes. For presentation purposes we narrowed the register size to 32 bit.

To improve parsing, we use EQUAL ANY to search for delimiters on a 16 byte at a time basis (cf., Fig. 5(a)). Branching is performed only if a special character is found. The following pseudocode illustrates our method:

```
procedure NEXTDELIMITER(input,specialChars)
  while !endOfInput(input) do
    special = _mm_set_epi8(specialChars)
    data = _mm_loadu_si128(input)
    mode = _SIDD_CMP_EQUAL_ANY
    index = _mm_cmpistri(special,data,mode)
    if index < 16 then
      // handle special character
    input = input+16
```

For long fields, e.g., strings of variable length, finding the next delimiter often requires to scan a lot more than 16 characters. To improve parsing of these fields, we adapted the method shown above to compare 64 characters at a time: First, 64 byte (typically one cache line) are loaded into four 128 bit SSE registers. For each of the registers a comparison mask is generated using the `_mm_cmpistrm` intrinsic. The four masks are interpreted as four 16 bit masks and are stored consecutively in one 64 bit integer where each bit indicates if a special character is found at the position of the bit. If the integer is 0, no special character was found. Otherwise, the position of the first special byte is retrieved by counting the number of trailing zeros. This operation is again available as a CPU instruction and is thus highly efficient.

To improve deserialization methods, we use the RANGES mode for input validation (cf., Fig. 5(b)). We again illustrate our approach in form of pseudocode:



(a) insert-based (b) copy-based (c) chunk-based

Figure 6: Merging buffers with relation paritions.

```
procedure DESERIALIZEINTEGERSSE(input,length)
  if length < 4 then
    deserializeIntegerNoSSE(input,length)
  range = _mm_set_epi8(0,...,0,'9','0')
  data = _mm_loadu_si128(input)
  mode = _SIDD_CMP_RANGES | _SIDD_MASKED_NEGATIVE_POLARITY
  index = _mm_cmpestri(range,2,data,length,mode)
  if index != 16 then
    throw RuntimeException("invalid character")
```

Experiments have shown that for string lengths of less than 4 byte, SSE optimized integer deserialization is slower than a standard non-SSE variant with current x86 CPUs. For integer deserialization we thus use a hybrid processing model where the SSE optimized variant is only used for strings longer than 3 characters. Deserialization methods for other data types were optimized analogously.

The evaluation in Sect. 5 shows that our vectorized methods reduce the number of branch misses significantly, improve energy efficiency, and increase performance by about 50% compared to non-vectorized methods.

3.5 Partition Buffers

CSV chunk-processing tasks store parsed and deserialized tuples as well as indexes on these tuples in partition buffers. These buffers have the same physical layout as the relation partitions in order to avoid further transformations of data during a merge step. In the following we discuss approaches to merge the tuples stored in a partition buffer with its corresponding relation partition in the storage backend (see Fig. 6). Merging of indexes is discussed in the next section. The insert- and copy-based approaches are viable for contiguous memory-based as well as chunked storage backends. The chunk-based approach requires a chunked storage backend (see Sect. 2).

insert-based approach. The insert-based approach constitutes the simplest approach. It iterates over the tuples in the buffer and inserts the tuples one-by-one into the relation partition. This approach is obviously very simple to realize as insertion logic can be reused. However, its performance is bounded by the insertion speed of the storage backend.

copy-based approach. In contrast to the insert-based approach, the copy-based approach copies all tuples from the buffer into the relation partition in one step. It is thereby faster than the insert-based approach as it largely only depends on the speed of the `memcpy` system call. We again task-parallelized `memcpy`ing for large buffers to fully leverage the available memory bandwidth on modern hardware. No additional transformations are necessary as the buffer already uses the physical layout of the relation partition.

chunk-based approach. For chunked storage backends the `memcpy` system call can be avoided entirely. A merge step then only consists of the insertion of a buffer reference into a list of chunk references in the backend. While merging time is minimal, too small and too many chunks negatively impact table scan and random access performance of the backend due to caching effects. In general, it is advantageous to have a small list of chunk references. Preferably, the list should fit in the CPU caches, so that it can be accessed efficiently. For Instant Loading, we are faced with the tradeoff between using small CSV chunk sizes for a high degree of task-parallelization (cf., Sect. 3.3) and creating large storage backend chunks to keep the backend efficient.

One way to meet this challenge is to store the partition buffer references of CSV chunk processing tasks in thread-local storage. Partition buffers are then reused as threads are reused by the TBB library. Hence, the expected mean size of relation partition chunks is the CSV input size divided by the number of hardware threads used for loading. Nevertheless, this is no panacea. If partition buffers are reused, merging of partition buffers with the relation can no longer be interleaved with CSV chunk processing. Furthermore, this approach requires CSV input to be of a respective size. For chunked storage backends it can thus also make sense to use copy-based merging or a hybrid approach. We intend to investigate further merge algorithms for various types of chunked storage backends in future work.

Buffer allocation. Allocation and reallocation of partition buffers on the heap is costly as, in general, it needs to be synchronized. Using scalable allocators that provide per-thread heaps is not an option as these are usually too small for loading purposes where huge amounts of data are moved. While an initial allocation of a buffer is unavoidable, reallocations can be saved by initially allocating enough memory for the tuples in a CSV chunk. The difficulty lies in the estimation of the number of tuples in a CSV chunk of a certain size. This is mainly due to nullable attributes and attributes of varying lengths. Our solution is to let CSV chunk processing tasks atomically update cardinality estimates for the partition buffers that serve as allocation hints for future tasks. For our implementation, at a multiprogramming level of 8, this allocation strategy increases performance by about 5% compared to dynamic allocation.

For hybrid OLTP&OLAP databases like HyPer, it further makes sense to allocate partition buffers on huge virtual memory pages. Huge pages have the advantage they have a separate section in the memory management unit (MMU) on most platforms. Hence, loading and mission-critical OLTP compete less for the transaction lookaside buffer (TLB).

3.6 Bulk Creation of Index Structures

Indexes have a decisive impact on transaction and query execution performance. However, there is a tradeoff between time spent on index creation and time saved during query and transaction processing. Using standard approaches, creating indexes during bulk loading can significantly slow down the loading throughput. Alternatives to the creation of indexes at load time such as database cracking [13] and adaptive indexing [14] propose to create indexes as a by-product of query processing and thereby allow faster data loading and fast query performance over time. However, if data is bulk loaded to a mission-critical OLTP or OLAP system that needs execution time guarantees immediately after loading, delayed index creation is not an option. This is especially true for our proposed data staging processing model where data is loaded, processed, and unloaded in cycles. Furthermore, to assure consistency, loading should at least check for primary key violations. We thus advocate for the creation of primary indexes at load time. With Instant Loading, it is our goal to achieve this at wire speed.

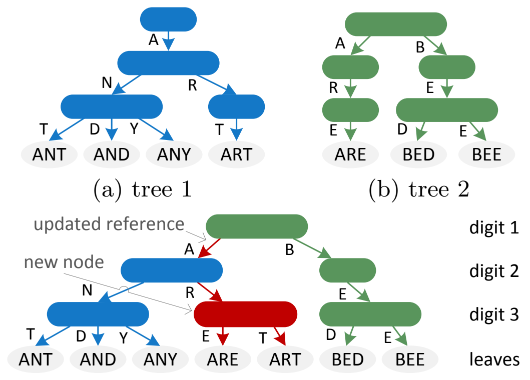
We identified different options regarding how and when to create indexes during loading. The first option is to always have a single index for the whole relation that is incrementally updated by inserting keys of new tuples after they have been added to the relation. The second option is to completely recreate a new index from scratch. The first option is limited by the insertion speed of the index structure. The second option could benefit from index structures that allow the efficient recreation of an index. However, depending on the size of the relation, this might impose a huge overhead. We thus propose a third way: each CSV chunk-processing task maintains indexes in its partition buffers. These indexes are then merged with the indexes in the relation partition during the merge step. We define indexes that allow our approach as merge-able index structures for bulk loading:

***Definition* (Merge-able index structures for bulk loading).** Merge-able index structures for bulk loading are index structures that allow the efficient and parallelized creation of the set of indexes $\mathcal{I} = \{I_1, \ldots, I_n\}$ over a set of keys $\mathcal{K} = \{k_1, \ldots, k_m\}$, where $\mathcal{K}$ is partitioned into n nonempty disjoint subsets $\mathcal{K}_1, \ldots, \mathcal{K}_n$ and I_j is an index over $\mathcal{K}_j$ for $1 \leq j \leq n$. Further, there exists an efficient parallelized *merge* function that, given $\mathcal{I}$, yields a single unified index over $\mathcal{K}$. The unified index creation time t is the aggregate of time needed to create $\mathcal{I}$ and time needed to merge $\mathcal{I}$. For merge-able index structures for bulk loading, t proportionally decreases with an increasing number n of key partitions assuming n available hardware threads.

In the following we show that hash tables with chaining and the adaptive radix tree (ART) [20] are merge-able index structures for bulk loading. Our evaluation (see Sect. 5) further demonstrates that parallelized forms of these indexes achieve a near-linear speedup with the number of key partitions and hardware threads used for bulk index creation.

3.6.1 Hash table with chaining

Hash tables are a popular in-memory data structure and are often used for indexes in main memory databases. Indexes based on hash tables only allow point queries but are very fast due to their expected lookup time of $\mathcal{O}(1)$. Hash tables inevitably face the problem of hash collisions. Strategies for conflict resolution include open addressing and chaining. Hash tables that use chaining for conflict resolution are particularly suitable as merge-able indexes for bulk loading. Our implementation of a merge-able hash table for bulk loading uses a fixed-sized hash table, where entries with the same hash value are chained in a linked list. For a given partitioned key range, equally-sized hash tables using the same hash function are, in parallel, created for each partition. These hash tables are then repeatedly merged in pairs of two by scanning one of the tables and concatenating each list entry for a specific hash value with the list for that hash value in the other hash table. The scan operation can thereby again be parallelized efficiently. It is of note that a space-time tradeoff is immanent in hash table-based index approaches. Our merge-able hash table with chaining allocates a fixed size hash table for each parallel task and is thus wasting space. In contrast to hash tables, the adaptive radix tree is highly space-efficient.



(a) tree 1 (b) tree 2

(c) merged tree

Figure 7: Two adaptive radix trees (ART) (a) and (b) and the result of merging the two trees (c).

3.6.2 *Adaptive Radix Tree (ART)*

The adaptive radix tree (ART) [20] is a high performance and space-efficient general purpose index structure for main memory databases that is tuned for modern hardware. Compared to hash tables, radix trees, also known as tries, directly use the digital representation of keys for comparison. The idea of a radix tree is similar to that of a thumb index of dictionaries, which indexes its entries according to their first character prefix. Radix trees use this technique recursively until a specific entry is found. An example of an ART index is shown in Fig. 7(a). ART is a byte-wise radix tree that uses the individual bytes of a key for indexing. As a result, all operations have a complexity of $\mathcal{O}(k)$, where k is the byte length of the indexed keys. Compared to hash tables, which are not order-preserving, radix trees store keys in their lexicographical order. This allows not only exact lookups but also range scans, prefix lookups, and top-k queries.

While other radix tree implementations rely on a globally fixed fanout parameter and thus have to trade off tree height against space efficiency, ART distinguishes itself from these implementations by using adaptively sized nodes. In ART, nodes are represented using four types of efficient and compact data structures with different sizes of up to 256 entries. The type of a node is chosen dynamically depending on the number of child nodes, which optimizes space utilization and access efficiency at the same time. The evaluation in [20] shows that ART is the fastest general purpose index structure for main memory databases optimized for modern hardware. Its performance is only met by hash tables, which, however, only support exact key lookups.

In this work we show that ART further belongs to the class of merge-able index structures for bulk loading by specifying an efficient parallelized merge algorithm. Fig. 7 illustrates the merging of two ART indexes. Radix trees in general are naturally suited for efficient parallelized merging: starting with the two root nodes, for each pair of nodes, children with common prefixes in the two trees are recursively merged in parallel. When all children with common prefixes have been merged, children of the smaller node that have no match in the bigger node are inserted into the bigger node. This bigger node is then used in the merged tree. Ideally, merging is thus reducible to a single insertion for non-empty trees. In the worst case, both trees contain only keys with common prefixes and nodes at maximum depth need to be merged. In general, merging of two radix trees t_1 and t_2 needs $\mathcal{O}(d)$ copy operations, where d is the minimum of $\mathit{diff}(t_1, t_2)$ and $\mathit{diff}(t_2, t_1)$, where $\mathit{diff}(x, y)$ is the number of inner nodes and leaves of y that are not present in x and are children of a node that does not already count towards this number.

Our parallelized merge algorithm looks as follows:

```
procedure MERGE($t_1$,$t_2$,depth)
  if isLeaf($t_1$) then insert($t_2$,$t_1$.keyByte,$t_1$,depth)
    return $t_2$
  if isLeaf($t_2$) then insert($t_1$,$t_2$.keyByte,$t_2$,depth)
    return $t_1$
  // ensure that $t_1$ is the bigger node
  if $t_1$.count > $t_2$.count then swap($t_1$,$t_2$)
  // descend trees in parallel for common key bytes
  parallel for each entry e in $t_2$ do
    c = findChildPtr($t_1$,e.keyByte)
    if c then c = MERGE((c,e.child,depth+1))
  // sequentially insert $t_2$'s unique entries in $t_1$
  for each entry e in $t_2$ do
    c = findChildPtr($t_1$,e.keyByte)
    if !c then insert($t_1$,e.keyByte,e.child,depth)
  return $t_1$
```

As mentioned before, we insert entries of key bytes of the smaller node that have no match in the bigger node sequentially and after all children with common prefixes have been merged in parallel. In ART, this separation into parallel and sequential phases is particularly due to the fact that nodes can grow when inserting new entries. For the biggest node type, which is essentially an array of size 256, insertions can further be parallelized using lock-free atomic operations. This kind of insertion parallelization is also applicable to other radix trees that work with nodes of a fixed size. It is indeed also feasible to implement a completely lock-free version of ART, which is, however, out of scope for this work, as we focused on an efficient merge algorithm.

4. INSTANT LOADING IN HYPER

4.1 The HyPer Main Memory Database

We integrated our generic Instant Loading approach in HyPer [19], our high-performance relational main memory database system. HyPer belongs to an emerging class of hybrid databases, which enable real-time business intelligence by evaluating OLAP queries directly in the transactional database. Using a novel snapshotting technique, HyPer achieves highest performance—compared to state of the art in-memory databases—for both, OLTP and OLAP workloads, operating simultaneously on the same database.

OLAP is decoupled from mission-critical OLTP using a snapshot mechanism with (almost) no synchronization overhead. The mechanism is based on the POSIX system call `fork()`: OLAP queries are executed in a process that is `fork`ed from the OLTP process (see Fig. 8). This is very efficient as only the virtual page table of the OLTP process is copied. The operating system uses the processor's memory management unit to implement efficient copy-on-update semantics for snapshotted pages. Whenever the OLTP process modifies a snapshotted page for the first time, the page is replicated in the `fork`ed process (see Fig. 8).

Transactions are specified in SQL or in a PL/SQL style scripting language and are compiled into machine code using the LLVM compiler framework [22]. Together with the elimination of ballast caused by buffer management, locking, and latching, HyPer can process more than 100,000 TPC-C

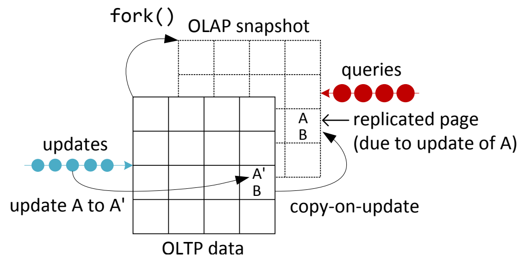


Figure 8: HyPer's snapshotting mechanism.

transactions per second in a single thread on modern hardware [19]. Similar to the design pioneered by H-Store [18] and VoltDB, HyPer also implements a partitioned execution model: The database is partitioned in a way that most transactions only need to access a single partition. Transactions that each exclusively work on a different partition can thus be processed in parallel without synchronization. Synchronization is only necessary for partition-crossing transactions.

Like transactions, SQL queries are compiled into LLVM code [22]. The data-centric compiler aims at good code and data locality and a predictable branch layout. LLVM code is then compiled into efficient machine code using LLVM's just-in-time compiler. Together with its advanced query optimizer, HyPer achieves superior query response times [19] comparable to those of MonetDB [4] or Vectorwise.

4.2 Instant Loading in HyPer

Instant Loading in HyPer allows *(lwu)** workflows but can indeed also be used for other use cases that require the loading of CSV data. This includes initial loads and incremental loads for continuous data integration.

The interface of Instant Loading in HyPer is designed in the style of the PostgreSQL `COPY` operator. Instant Loading takes CSV input, the schema it adheres to, and the CSV special characters as input. Except for `"\r\n"`, which we allow to be used as a record delimiter, we assume that special characters are single ASCII characters. For each relation that is created or altered, we generate LLVM glue code functions for the processing of CSV chunks and for partition buffer merging (cf., the two steps in Fig. 4). Code generation and compilation of these functions at runtime has the advantage that the resulting code has good locality and predictable branching as the relation layout, e.g., the number of attributes and the attribute types, are known. Searching for delimiters and the deserialization methods are implemented as generic C++ functions that are not tailored to the design of HyPer. Just like the LLVM functions HyPer compiles for transactions and queries [22], the Instant Loading LLVM glue code calls these statically compiled C++ functions. Such LLVM glue code functions can further be created for other CSV-like formats using the C++ functions similar to a library. Code generation of the LLVM functions for CSV data is implemented for the four storage backend types in HyPer (cf. Sect. 2).

Offline loading. In offline loading mode, loading has exclusive access to the relation, i.e., there are no concurrent transactions and queries; and loading is not logged. Processing of CSV chunks and merge steps are interleaved as much as possible to reduce overall loading time. If an error occurs during the loading process, an exception is raised but the database might be left in a state where it is only partially loaded. For use cases such as *(lwu)** workflows, in-situ querying, and initial loading this is usually acceptable as the database can be recreated from scratch.

Online transactional loading. Online transactional loading supports loading with ACID semantics where only the merge steps need to be encapsulated in a single merge transaction. Processing of CSV chunks can happen in parallel to transaction processing. There is a tradeoff between overall loading time and the duration of the merge transaction: To achieve online loading optimized for a short loading time, chunk processing is interleaved with merge steps. The duration of the merge transaction starts with the first and ends with last merge step. No other transactions can be processed in that time. To achieve a short merge transaction duration, first all chunks are processed and then all merge steps are processed at once.

5. EVALUATION

The evaluation of Instant Loading in HyPer was conducted on a commodity workstation with an Intel Core i7-3770 CPU and 32 GB dual-channel DDR3-1600 DRAM. The CPU is based on the Ivy Bridge microarchitecture and supports the SSE 4.2 string and text instructions, has 4 cores (8 hardware threads), a 3.4 GHz clock rate, and a 8 MB last-level shared L3 cache. As operating system we used Linux 3.5 in 64 bit mode. Sources were compiled using GCC 4.7 with `-O3 -march=native` optimizations. For lack of a high-speed network-attached storage or distributed file system in our lab, we used the in-memory file system `ramfs` as the CSV source to emulate a wire speed of multiple Gbit/s. Prior to each measurement we flushed the file system caches.

5.1 Parsing and Deserialization

We first evaluated our task- and data-parallelized parsing and deserialization methods in isolation from the rest of the loading process. CSV data was read from `ramfs`, parsed, deserialized, and stored in heap-allocated result buffers. We implemented a variant that is SSE 4.2 optimized (SSE) as described in Sect. 3.4 and one that is not (non-SSE). As a contestant for these methods we used a parsing and deserialization implementation based on the Boost Spirit C++ library v2.5.2. In particular, we used Boost Spirit.Qi, which allows the generation of a recursive descent parser for a given grammar. We also experimented with an implementation based on Boost.Tokenizer and Boost.Lexical_Cast but its performance trailed that of the Boost Spirit.Qi variant. Just like our SSE and non-SSE variants, we task-parallelized our Boost implementation as described in Sect. 3.3.

As input for the experiment we chose TPC-H CSV data generated with a scale-factor of 10 (~10 GB). While the SSE and non-SSE variants only require schema information at run-time, the Spirit.Qi parser generator is a set of templated C++ functions that require schema information at compile-time. For the Boost Spirit.Qi variant we thus hardcoded the TPC-H schema information into the source code.

Fig. 9 shows that SSE and non-SSE perform better than Boost Spirit.Qi at all multiprogramming levels. SSE outperforms non-SSE and shows a higher speedup: SSE achieves a parsing and deserialization throughput of over 1.6 GB/s with a multiprogramming level of 8 compared to about 1.0 GB/s with non-SSE, an improvement of 60%. The superior performance of SSE can be explained by (i) the exploitation of vector execution engines in addition to scalar execution units across all cores and (ii) by the reduced number of branch

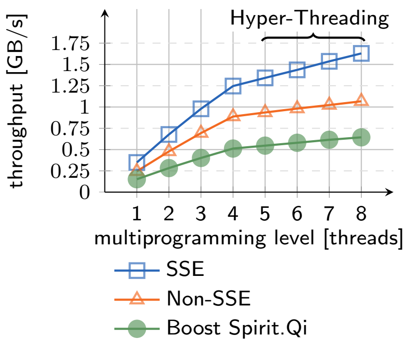


Figure 9: Speedup of parsing and deserialization methods with heap-allocated result buffers.

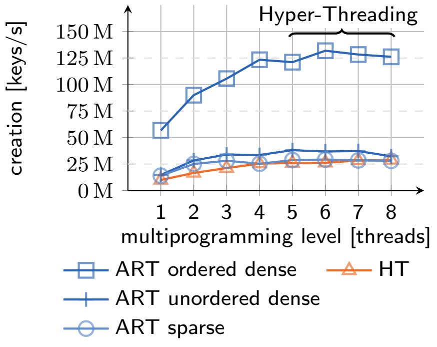


Figure 10: Speedup of merge-able HT and ART parallelized index building and merging for 10 M 32 bit keys.

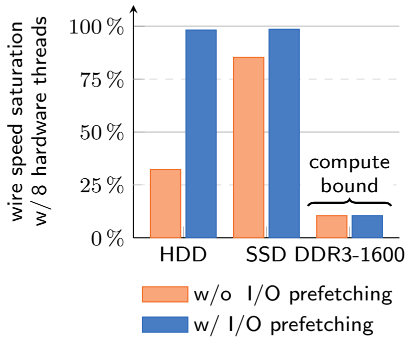


Figure 11: Wire speed saturation of Instant Loading (cf., Fig. 1) and the impact of I/O prefetching.

	insert	copy	chunk
column-store	7841 ms	6939 ms	6092 ms
row-store	6609 ms	6608 ms	6049 ms

Table 1: Loading of TPC-H CSV data (scale-factor 10) to a column- and row-store using insert-, copy-, and chunk-based partition buffer merging.

misses compared to non-SSE. Performance counters show that the number of branch misses is reduced from 194/kB CSV with non-SSE to just 89/kB CSV with SSE, a decrease of over 50%. Using all execution units of the CPU cores also allows SSE to profit more from Hyper-Threading. This comes at no additional cost and improves energy efficiency: Measuring the Running Average Power Limit energy sensors available in recent Intel CPUs reveals that SSE used 388 J compared to 503 J (+23%) with non-SSE and 625 J (+38%) with Boost Spirit.Qi.

5.2 Partition Buffers

We evaluated Instant Loading for the column- and row-store storage backend implementations in HyPer (cf., Sect. 2) and the three partition buffer merging approaches we proposed in Sect. 3.5. For the insert- and copy-based merging approaches we used storage backends based on contiguous memory, for the chunk-based approach we used chunked storage backends. Table 1 shows the benchmark results when loading a TPC-H CSV data set with a scale-factor of 10. For the column-store backends, copy was around 12% faster than insert. The chunk-based approach improved performance by another 12%. For the row-store backend, insert and copy performed similarly; chunk-based merging was 8.5% faster.

5.3 Bulk Index Creation

We evaluated the parallelized creation of hash tables with chaining (HT) and adaptive radix trees (ART) on key range partitions and the parallelized merging of these indexes to create a unified index for the total key range.

Fig. 10 shows the speedup of index creation for a key range of 10M 32 bit keys. For ordered dense keys, i.e., ordered keys ranging from 1 to 10M, ART allows a faster creation of the index than the HT for all multiprogramming levels. Merging of ART indexes is, in the case of an ordered dense key range, highly efficient and often only requires a few pointers to be copied such that the creation time of the unified index largely only depends on the insertion speed of the ART indexes that are created in parallel. The lower speedup of ART ($\times$2.2) compared to HT ($\times$2.6) with a multiprogramming level of 4 is due to caching effects. The performance of ART heavily depends on the size of the effectively usable CPU cache per index [20]. In absolute numbers, however, ART achieves an index creation speed of 130M keys per second compared to 27M keys per second with HT. While the performance of HT does not depend on the distribution of keys, an ordered dense key range is the best case for ART. For unordered dense, i.e., randomly permuted dense keys, and sparse keys, i.e., randomly generated keys for which each bit is 1 or 0 with equal probability, the performance of ART drops. The index creation speed is still slightly better than with HT. For unordered key ranges merging is more costly than for ordered key ranges because mostly leaf nodes need to be merged. For a multiprogramming level of 4, merging accounted for 1% of loading time for ordered dense, 16% for unordered dense, and 33% for sparse keys.

5.4 Offline Loading

To evaluate the end-to-end application performance of offline loading we benchmarked a workload that consisted of (i) bulk loading TPC-H CSV data with a scale-factor of 10 (~10 GB) from `ramfs` and (ii) then executing the 22 TPC-H queries in parallel query streams. We used an unpartitioned TPC-H database, i.e., only one merge task runs in parallel, and configure HyPer to use a column-store backend based on contiguous memory. Partition buffers were merged using the copy-based approach. We compared Instant Loading in HyPer to a Hadoop v1.1.1 Hive v0.10 [28] cluster consisting of 4 nodes of the kind described at the beginning of Sect. 5 (1 GbE interconnect), SQLite v3.7.15 compiled from source, MySQL v5.5.29, MonetDB [4] v11.13.7 compiled from source, and Vectorwise v2.5.2.

Fig. 12 shows our benchmark results. Instant Loading achieves a superior combined bulk loading and query processing performance compared to the contestants. Loading took 6.9 s (HyPer), unloading the database as a LZ4-compressed binary to `ramfs` after loading took an additional 4.3 s (HyPer /w unload). The compressed binary has a size of 4.7 GB (50% the size of the CSV files) and can be loaded again in 2.6 s (3$\times$ faster than loading the CSV files). In both cases, the queries were evaluated in just under 12 s. Our unloading and binary loading approaches in HyPer are again highly parallelized. We further evaluated the I/O saturation when loading from local I/O devices. Fig. 11 shows that Instant Loading fully saturates the wire speed of a traditional HDD (160 MB/s) and a SSD (500 MB/s). When the

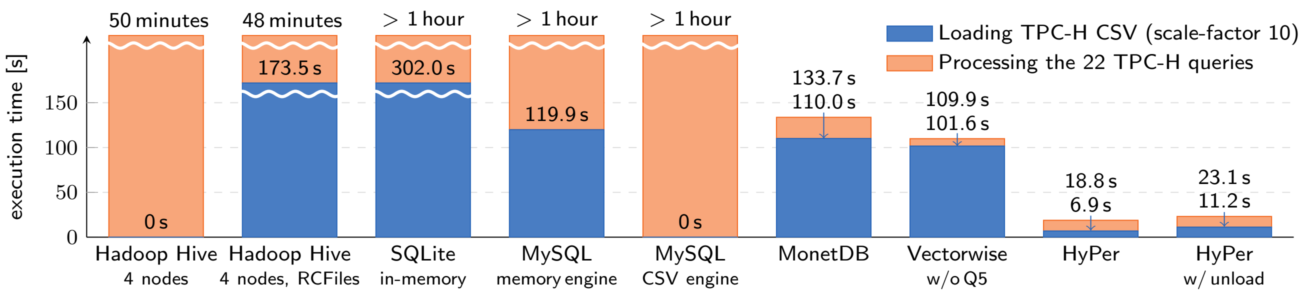


Figure 12: Offline CSV bulk loading and query processing performance in HyPer with Instant Loading, other main memory databases, and a Hadoop Hive cluster with 4 nodes.

memory is used as the source and the sink, only 10% of the available wire speed are saturated (CPU bound). Fig. 11 further shows that advising the kernel to prefetch data from the local I/O device (using `madvise`) is necessary to achieve a near-100% saturation of local devices.

Hive is a data warehouse solution based on Hadoop. For our benchmark, we used 4 Hadoop nodes. Hadoop's distributed file system (HDFS) and Hive were configured to store data in `ramfs`. Other configuration settings were untouched, including the default replication count of 3 for HDFS. This means that each node in the setup had a replica of the CSV files. We did not include the HDFS loading time (125.8 s) in our results as we assume that data is ideally already stored there. To evaluate the query performance, we used an official implementation of the TPC-H queries in HiveQL[3], Hive's SQL-like query language. Even though no explicit loading is required and 4 nodes instead of a single one are used, Hive needed 50 minutes to process the 22 queries. We also evaluated Hive with record columnar files (RCFiles). Loading the CSV files into RCFiles using the BinaryColumnarSerDe, a transformation pass that deserializes strings to binary data type representations, took 173.5 s. Query processing on these RCFiles was, however, only 5 minutes faster than working on the raw CSV files.

SQLite was started as an in-memory database using the special filename `:memory:`. For bulk loading, we locked the tables in exclusive mode and used the `.import` command. Query performance of SQLite is, however, not satisfactory. Processing of the 22 TPC-H queries took over 1 hour.

For MySQL we ran two benchmarks: one with MySQL's memory engine using the `LOAD DATA INFILE` command for bulk loading and one with MySQL's CSV engine that allows query processing directly on external CSV files. Bulk loading using the memory engine took just under 2 minutes. Nevertheless, for both, the memory and CSV engine, processing of the 22 TPC-H queries took over 1 hour again.

We compiled MonetDB with MonetDB5, MonetDB/SQL, and extra optimizations enabled. For bulk loading we used the `COPY INTO` command with the `LOCKED` qualifier that tells MonetDB to skip logging operations. As advised in the documentation, primary key constraints were added to the tables after loading. We created the MonetDB database inside `ramfs` so that BAT files written by MonetDB were again stored in memory. To the best of our knowledge MonetDB has no option to solely bulk load data to memory without writing the binary representation to BAT files. Bulk loading in MonetDB is thus best compared to Instant Loading with binary unloading (HyPer w/ unload). While loading time is comparable to the MySQL memory engine, queries are processed much faster. The combined workload took 133.7 s to complete.

For Vectorwise, we bulk loaded the files using the `vwload` utility with rollback on failure turned off. Loading time is comparable to MonetDB while queries are processed slightly faster. TPC-H query 5 could not be processed without the prior generation of statistics using `optimizedb`. We did not include the creation of statistics in our benchmark results as it took several minutes in our experiments.

We would have liked to further compare Instant Loading to MonetDB's CSV vault [16] but couldn't get it running in the current version of MonetDB. We would have also liked to evaluate the NoDB implementation PostgresRaw [3] in the context of high-performance I/O devices and main memory databases, but its implementation is not (yet) available.

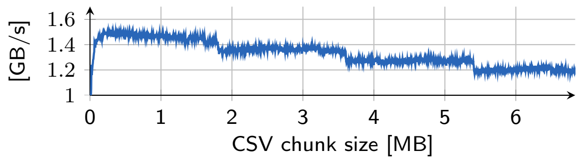


Figure 13: Throughput as a function of chunk size.

scale-factor	loading throughput	query time
10 (∼10 GB)	1.14 GB/s (∼9 Gbit/s)	16.6 s
30 (∼30 GB)	1.29 GB/s (∼10 Gbit/s)	57.9 s
100 (∼100 GB)	1.36 GB/s (∼11 Gbit/s)	302.1 s

Table 2: Scaleup of Instant Loading of TPC-H data sets on a server with 256 GB main memory.

Optimal chunk size. Fig. 13 shows Instant Loading throughput of a TPC-H data set as a function of chunk size. Highest throughputs were measured between 256 kB and 1 MB, which equals a range of 0.25–1.0 times the L3 cache size divided by the number of hardware threads used.

Scaleup of Instant Loading. We evaluated the scaleup of Instant Loading on a server machine with an 8 core Intel Xeon X7560 CPU and 256 GB of DDR3-1066 DRAM and bulk loaded TPC-H CSV data with scale-factors of 10 (∼10 GB), 30 (∼30 GB), and 100 (∼100 GB). We then again executed the 22 TPC-H queries in parallel query streams. As shown in Table 2, Instant Loading achieves a linear scaleup.

`perf` analysis of Instant Loading. A `perf` analysis of Instant Loading of a TPC-H scale-factor 10 lineitem CSV file shows that **37%** of CPU cycles are used to find delimiters, **11.2%** to deserialize numerics, **9.1%** to deserialize dates, **6.5%** to deserialize integers, **5.5%** in the LLVM glue

[3]`http://issues.apache.org/jira/browse/HIVE-600`

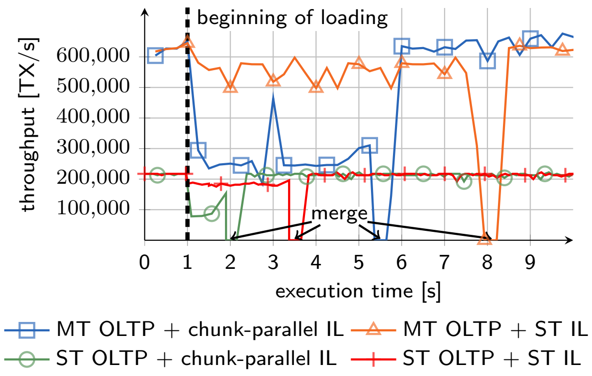


Figure 14: Chunk-parallel and single-threaded (ST) online CSV Instant Loading (IL) of 1M item and 4M stock entries and single-threaded (ST) and multi-threaded (MT) TPC-C transaction processing.

code that processes CSV chunks, and **5%** in the LLVM glue code that merges partition buffers. The remaining cycles are mostly spent inside the kernel. In more detail, the costs of deserialization methods and the method to find delimiters are dominated by the instructions that load data to the SSE registers and the SSE comparison instructions.

5.5 Online Transactional Loading

Finally, we evaluated Instant Loading in the context of online transactional loading with ACID semantics. In particular, we benchmarked the partitioned execution of TPC-C transactions in a TPC-C database partitioned by warehouse with 4 warehouses. In parallel to transaction processing, we bulk loaded a new product catalog with 1M new items into the item table. In addition to the 1M items, for each warehouse, 1M stock entries were inserted into the stock table. The storage backend was a chunked row-store and we used chunk-based partition buffer merging. Fig. 14 shows the TPC-C throughput with online bulk loading of the aforementioned data set (∼1.3 GB), which was stored as CSV files in `ramfs`. In our benchmark, loading started after 1 second. We measured transaction throughput in four scenarios: single- (ST) and multi-threaded (MT) transaction processing combined with single-threaded and CSV chunk-parallel Instant Loading. In case of ST transaction processing, a throughput of 200,000 transactions per second was sustained with ST Instant Loading; with chunk-parallel Instant Loading throughput shortly dropped to 100,000 transactions per second. Loading took around 3.5 s with ST Instant Loading and 1.2 s with chunk-parallel Instant Loading. Merge transactions took 250 ms. In case of MT transaction processing, transaction processing and Instant Loading compete for hardware resources and throughput decreased considerably from 600,000 to 250,000 transactions per second. With ST Instant Loading, the additional load on the system is lower and transaction throughput barely decreases. With chunk-parallel Instant Loading, loading took 4.6 s; with ST Instant Loading 7.0 s. Merge transactions took 250 ms again.

To the best of our knowledge, none of our contestants supports online transactional loading yet. We still compared our approach to the MySQL memory engine, which, however, has no support for transactions. We thus executed the TPC-C transactions sequentially. MySQL achieved a transaction throughput of 36 transactions per second. Loading took 19.70 s; no transactions were processed during loading.

6. RELATED WORK

Due to Amdahl's law, emerging multi-core CPUs can only be efficiently utilized by highly parallelized applications [17]. Instant Loading highly parallelizes CSV bulk loading and reduces the proportion of sequential code to a minimum.

SIMD instructions have been used to accelerate a variety of database operators [31, 29]. Vectorized processing and the reduction of branching often enabled superlinear speedups. Compilers such as GCC and the LLVM JIT compiler [22] try to use SIMD instructions automatically. However, often subtle tricks, which can hardly be reproduced by compilers, are required to leverage SIMD instructions. To the best of our knowledge no compiler can yet automatically apply SSE 4.2 string and text instructions. To achieve highest speedups, algorithms need to be redesigned from scratch.

Already in 2005, Gray et al. [10] called for a synthesis of file systems and databases. Back then, scientists complained that loading structured text data to a database doesn't seem worth it and that once it is loaded, it can no longer be manipulated using standard application programs. Recent works addressed these objections [12, 3, 16]. NoDB [3] describes systems that "do not require data loading while still maintaining the whole feature set of a modern database system". NoDB directly works on files and populates positional maps, i.e., index structures on files, and caches as a by-product of query processing. Even though the NoDB reference implementation PostgresRaw has shown that queries can be processed without loading and query processing profits from the positional maps and caches, major issues are not solved. These, in our opinion, mainly include the efficient support of transactions, the scalability and efficiency of query processing, and the adaptability of the paradigm for main memory databases. Instant Loading is a different and novel approach that does not face these issues: Instead of eliminating data loading and adding the overhead of an additional layer of indirection, our approach focusses on making loading and unloading as unobtrusive as possible.

Extensions of MapReduce, e.g., Hive [28], added support for declarative query languages to the paradigm. To improve query performance, some approaches, e.g., HAIL [7], propose using binary representations of text files for query processing. The conversion of text data into these binary representations is very similar to bulk loading in traditional databases. HadoopDB [1] is designed as a hybrid of traditional databases and Hadoop-based approaches. It interconnects relational single-node databases using a communication layer based on Hadoop. Loading of the single-node databases has been identified as one of the obstacles of the approach. With Instant Loading, this obstacle can be removed. Polybase [6], a feature of the Microsoft SQL Server PDW, translates some SQL operators on HDFS-resident data into MapReduce jobs. The decision of when to push operators from the database to Hadoop largely depends on the text file loading performance of the database.

Bulk loading of index structures has, e.g., been discussed for B-trees [8, 9]. Database cracking [13] and adaptive indexing [14] propose an iterative creation of indexes as a by-product of query processing. These works argue that a high cost has to be paid up-front if indexes are created at load-time. While this is certainly true for disk-based systems, we have shown that for main memory databases at least the creation of primary indexes—which enable the validation of primary key constraints—as a side-effect of loading is feasible.

7. OUTLOOK AND CONCLUSION

Ever increasing main memory capacities have fostered the development of in-memory database systems and very fast network infrastructures with wire speeds of tens of Gbit/s are becoming economical. Current bulk loading approaches for main memory databases, however, fail to leverage these wire speeds when loading structured text data. In this work we presented *Instant Loading*, a novel CSV loading approach that allows *scalable bulk loading at wire speed*. Task- and data-parallelization of every phase of loading allows us to fully leverage the performance of modern multi-core CPUs. We integrated the generic Instant Loading approach in our HyPer system and evaluated its end-to-end application performance. The performance results have shown that Instant Loading can indeed leverage the wire speed of emerging 10 GbE connectors. This paves the way for new *(load-work-unload)** usage scenarios where the main memory database system serves as a flexible and high-performance compute engine for big data processing—instead of using resource-heavy MapReduce-style infrastructures.

In the future we intend to support other structured text formats and include more data preprocessing steps such as compression, clustering, and synopsis generation. E.g., small materialized aggregates [21] can efficiently be computed at load time. Another idea is to port our scalable loading approach to coprocessor hardware and general-purpose GPUs.

8. ACKNOWLEDGEMENTS

Tobias Mühlbauer is a recipient of the Google Europe Fellowship in Structured Data Analysis, and this research is supported in part by this Google Fellowship. Wolf Rödiger is a recipient of the Oracle External Research Fellowship. This work is further sponsored by the German Federal Ministry of Education and Research (BMBF) grant HDBC 01IS12026.

9. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? *VLDB*, pages 266–277, 1999.

[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, pages 241–252, 2012.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[5] J. Dean. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[6] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, et al. Split Query Processing in Polybase. In *SIGMOD*, pages 1255–1266, 2013.

[7] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.

[8] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, 2006.

[9] G. Graefe and H. Kuno. Fast Loads and Queries. In *TLDKS II*, number 6380 in LNCS, pages 31–72, 2010.

[10] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[11] Hive user group presentation from Netflix. `http://slideshare.net/slideshow/embed_code/3483386`.

[12] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, pages 57–68, 2011.

[13] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[14] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):586–597, 2011.

[15] Extending the worlds most popular processor architecture. *Intel Whitepaper*, 2006.

[16] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDM*, volume 7338 of *LNCS*, pages 485–494, 2012.

[17] R. Johnson and I. Pandis. The bionic DBMS is coming, but what will it look like? In *CIDR*, 2013.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[19] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[20] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, pages 38–49, 2013.

[21] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. *VLDB*, pages 476–487, 1998.

[22] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[23] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, et al. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, pages 165–178, 2009.

[24] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* 2007.

[25] E. Sedlar. Oracle Labs. Personal comm. May 29, 2013.

[26] A. Szalay. JHU. Personal comm. May 16, 2013.

[27] A. Szalay, A. R. Thakar, and J. Gray. The sqlLoader Data-Loading Pipeline. *JCSE*, 10:38–48, 2008.

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, et al. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[29] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[30] Y. Shafranovich. IETF RFC 4180, 2005.

[31] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.