Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation

Harald Lang[1], Tobias Mühlbauer[1], Florian Funke[2,*],
Peter Boncz[3,*], Thomas Neumann[1], Alfons Kemper[1]

[1]Technical University Munich, [2]Snowflake Computing, [3]Centrum Wiskunde & Informatica
{harald.lang,muehlbau,neumann,kemper}@in.tum.de, florian.funke@snowflake.net, boncz@cwi.nl

ABSTRACT

This work aims at reducing the main-memory footprint in high performance hybrid OLTP & OLAP databases, while retaining high query performance and transactional throughput. For this purpose, an innovative compressed columnar storage format for cold data, called *Data Block*s is introduced. Data Blocks further incorporate a new light-weight index structure called *Positional SMA* that narrows scan ranges within Data Blocks even if the entire block cannot be ruled out. To achieve highest OLTP performance, the compression schemes of Data Blocks are very light-weight, such that OLTP transactions can still quickly access individual tuples. This sets our storage scheme apart from those used in specialized analytical databases where data must usually be bit-unpacked. Up to now, high-performance analytical systems use either vectorized query execution or "just-in-time" (JIT) query compilation. The fine-grained adaptivity of Data Blocks necessitates the integration of the best features of each approach by an interpreted vectorized scan subsystem feeding into JIT-compiled query pipelines. Experimental evaluation of HyPer, our full-fledged hybrid OLTP & OLAP database system, shows that Data Blocks accelerate performance on a variety of query workloads while retaining high transaction throughput.

1. INTRODUCTION

In past years, a new database system architecture specialized for OLAP workloads has emerged. These OLAP systems store data in compressed columnar format and increase the CPU efficiency of query evaluation by more than an order of magnitude over traditional row-store database systems. The jump in query evaluation efficiency is typically achieved by using "vectorized" execution where, instead of interpreting query expressions tuple at a time, all operations are executed on blocks of values. The effect is reduced interpretation overhead because virtual functions implementing block-wise operations handle thousands of tuples per function call, and the loop over the block inside these function implementations benefits from many loop-driven compiler optimizations including the automatic generation of SIMD instructions. Examples of such systems are IBM BLU [30], the Microsoft SQL Server Column Index subsystem [18], SAP HANA [12] and Vectorwise [39]. An alternative recently introduced way to accelerate query evaluation is "just-in-time" (JIT) compilation of SQL queries directly into executable code. This approach avoids query interpretation and its overheads altogether. Recent analytical systems using JIT are Drill, HyPer [16, 25] and Impala [34].

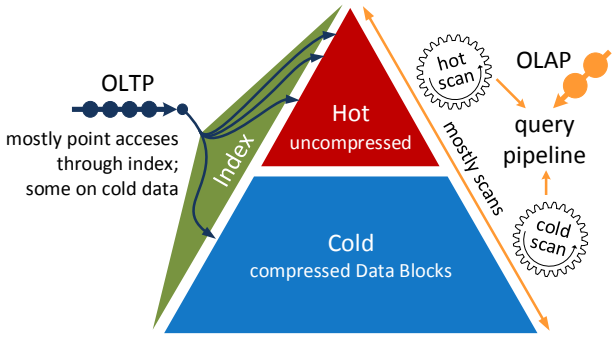


Figure 1: We propose the novel Data Block format that allows efficient scans and point accesses on compressed data and address the challenge of integrating multiple storage layout combinations in a compiling tuple-at-a-time query engine by using vectorization.

This paper describes the evolution of HyPer, our full-fledged main-memory database system, that was originally built with a JIT-compiling query engine to incorporate vectorization in the context of our novel compressed columnar storage format, "Data Blocks" (cf., Figure 1). HyPer differs from most of the aforementioned systems in that it aims to accomodate *both* high performance OLTP alongside OLAP running against the same database state and storage backend. Our primary goal is to reduce the main-memory footprint of HyPer by introducing compression while retaining the high OLTP and OLAP performance of the original system. To achieve this goal, we contribute (i) Data Blocks, a novel compressed columnar storage format for hybrid database systems, (ii) light-weight indexing on compressed data for improved scan performance, (iii) SIMD-optimized algorithms for predicate evaluation, and (iv) a blueprint for the integration of multiple storage layout combinations in a compiling tuple-at-a-time query engine by using vectorization.

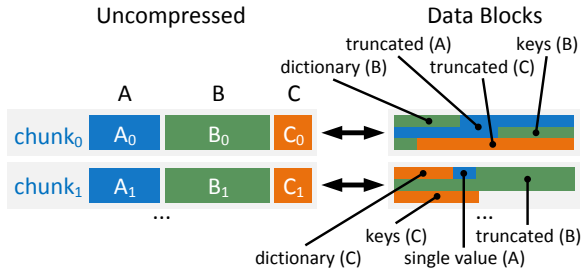
Designing hybrid database systems is challenging because the technical demands of both types of workloads are very

*Work done while at Technical University Munich.

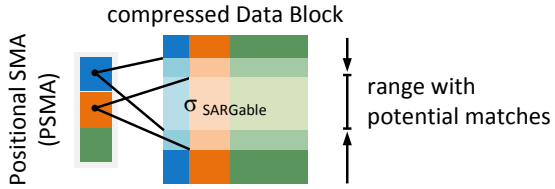
SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882925



(a) Optimal compression techniques are chosen for each column in each block.



(b) Data Blocks store not only min and max values of attributes to skip blocks as a whole, but also light-weight indexes called Positional SMAs (PSMAs) that narrow the scan ranges on a per-attribute basis.

Figure 2: Overview of compressed Data Blocks

different and many fundamental physical optimizations are contradictory. Compression, for instance, reduces memory consumption and can improve analytical query performance due to reduced bandwidth usage. High performance transactional systems, on the other hand, refrain from using compression in order to keep individual tuple access fast. The fundamental difference to specialized analytical systems is that hybrids need to manage hot and cold data efficiently within a single database instance. One approach to address this problem is to divide relations into a read- and a write-optimized partition [17]. Updates are exclusively performed in the latter partition, which is then periodically merged into the read-optimized (compressed) partition. However, the merge process has $\mathcal{O}(n)$ time complexity (where n is the relation's cardinality) and requires re-compressing of the entire relation. This merge-process is a heavy-weight operation, which is why we propose an alternative strategy: Relations in our OLTP and OLAP system are divided into fixed-size chunks, which are individually compressed into read-optimized immutable Data Blocks when they are identified as cold using the compression scheme optimal for that chunk (Figure 2(a)). Once a chunk has been packed into a Data Block, the contained data is immutable (frozen). Updates are still possible by invalidating the cold record and moving it to the hot region. Thus, updates are internally transformed into a delete followed by an insert.

In order to allow for highly efficient access to individual records by tuple position, we only use light-weight compression schemes in Data Blocks. While many OLAP systems employ bit-packing techniques to achieve higher compression ratios, we refrain from doing so. Our experiments in Section 5.4 show that even with recent SIMD algorithms [27] for bit-packed data, scans only work very fast if their early filtering is either so selective that there are no qualifying tuples and entire blocks can be skipped, or all tuples in a block qualify. In case early filtering produces a sparse set of qualifying tuple positions, the cost of bit-unpacking the scan columns dwarfs the performance gains made by these recent techniques in early filtering. Our Data Blocks are designed to not only keep positional access on compressed data cheap, for OLTP workloads, but also to allow scan-based OLAP workloads to reap the benefit of early filtering while not losing this benefit in sparse tuple decompression.

To speed up scans on Data Blocks, we introduce a new "Positional" type of Small Materialized Aggregates [23] (PSMAs). PSMAs are light-weight indexes that narrow the scan range within a block even if the block cannot be skipped based on materialized min and max values (see Figure 2(b)).

The strength of tuple-at-a-time JIT compilation is its ability to generate code that is highly efficient for both OLAP and OLTP queries, in terms of needing few CPU instructions per processed tuple. Where vectorization passes data between operations through *memory*, tuple-at-a-time JIT passes data through CPU *registers*, saving performance-critical load/store instructions. Vectorization brings no CPU efficiency improvement at all for OLTP as its efficiency depends on executing expressions on many tuples at the same time while OLTP queries touch very few tuples and typically avoid scans. Choosing different compression schemes on a per-chunk basis, however, constitutes a challenge for JIT-compiling tuple-at-a-time query engines. As each compressed block can have a different memory layout, the number of code paths that have to be compiled for a scan grow exponentially. This leads to compilation times that are unacceptable for ad-hoc queries and transactions.

In this case, vectorized scans come to the rescue because their main strength is that they remain interpreted and can be pre-compiled. Further, vectorized scans are amenable to exploit SIMD and can express *adaptive* algorithms, opening a path to future further optimizations. As a final contribution we thus show how the strengths of both worlds, JIT compilation and vectorization, can be fused together using an interpreted vectorized scan subsystem that feeds into JIT-compiled tuple-at-a-time query pipelines. For vectorized predicate evaluation, we contribute new SSE/AVX2 SIMD algorithms. Overall, we show that this novel query engine together with our Data Blocks format yields a system that can save large amounts of main memory while still allowing for *both* highest performance OLTP and OLAP.

2. RELATED WORK

The work most closely related to Data Blocks is the storage format of IBM DB2 BLU [4, 30], which also consists of blocks representing all columns of a sequence of tuples stored in compressed columnar format inside the block (a concept introduced as PAX [3]) and provides mechanisms for early evaluation of range filters inside the scan. The main differences are that HyPer avoids bit-packing to keep positional access fast and introduces the PSMAs that further improve the evaluation of early selections.

We also note that recent work on advanced bit-packing schemes [22, 13] and their SIMD implementation [27] focus on the benefits of early filtering and either ignore the high per-tuple cost of bit-unpacking, or just position these bit-packed formats as a secondary storage structure. Our choice for byte-aligned storage mostly avoids this penalty and makes early filtering beneficial in a broader range of query workloads (see Section 5.4).

Vectorwise [39] proposed the idea to decompress chunks of the data into the CPU cache and process the decompressed data while it is cached. Vectorwise does not do any early

filtering in scans and fully decompresses all scanned column ranges as it deems positional decompression too inefficient. In contrast, the efficient positional access of HyPer permits reaping the benefits of early evaluation of SARGable predicates inside the scan also in situations with moderate selectivities. Depending on the selectivity, early filtering can make scans in HyPer factors faster compared to Vectorwise (cf., Table 2); while on the other hand, byte-aligned compression on average increases space consumption $\leq \times 1.25$ (cf., Table 1).

Abadi et al. [1] thoroughly evaluated various compression schemes in column-oriented databases. They conclude that light-weight compression schemes should be preferred over heavy-weight schemes to operate directly on the compressed data. In contrast, we only consider filter operations and point-accesses on compressed data and do not pass compressed data to more complex operators like joins.

Oracle Database [28] uses block-wise compression but uses fewer compression schemes and is not built for efficient processing on modern CPUs. SAP HANA [12] is optimized for hybrid workloads, but as mentioned, follows a different approach where the entire relation is optimized for memory consumption and scan performance, whereas updates are performed in a separate partition, which is then periodically merged [17]. Thus, the distinction between hot and cold data is only implicit and depends on the size of the write-optimized partition. The Siberia framework [11], which is part of Microsoft's Hekaton engine [10], provides an interface for managing cold data. Unlike Data Blocks, it aims to reduce RAM usage primarily by evicting cold data to disk.

On the issue of integrating JIT-compiled query evaluation and vectorization, there has been work in Vectorwise on JIT-compiling projections and hash-joins [31]. This work showed only modest benefits of JIT integrated into vectorized operators, which subsequently remained an experimental feature only. The Impala [34] JIT approach is very different from our tuple-at-a-time JIT system which fuses all operators inside a query pipeline into a tight single loop. For each physical relational operator, Impala provides a C++ template class that is compiled with LLVM [19] into intermediate code in advance. During JIT query compilation, only the methods to handle tuples and expressions inside these operators are compiled directly into intermediate code and linked into this pre-compiled template class. The code of the different operators in a query pipeline are thus not fused. The Impala operators communicate with each other by using tuple buffers: in that sense they can be considered vectorized operators. The problem with this approach is that the main strength of JIT in minimizing the amount of needed instructions, passing data through registers and avoiding load/stores, is lost; the only real benefit is a more easy-to-understand and -test templated execution engine.

Vectorization enables the use of SIMD and facilitates algorithms that access data belonging to multiple tuples in parallel. In the database context, SIMD instructions have been used for example, to speed up selection scans [37, 35, 26], for bit unpacking [35, 27], bulk loading [24], sorting [6], and breadth-first search on graphs [32].

3. DATA BLOCKS

Data Blocks are self-contained containers that store one or more attribute chunks in a byte-addressable compressed format. The goal of Data Blocks is to conserve memory

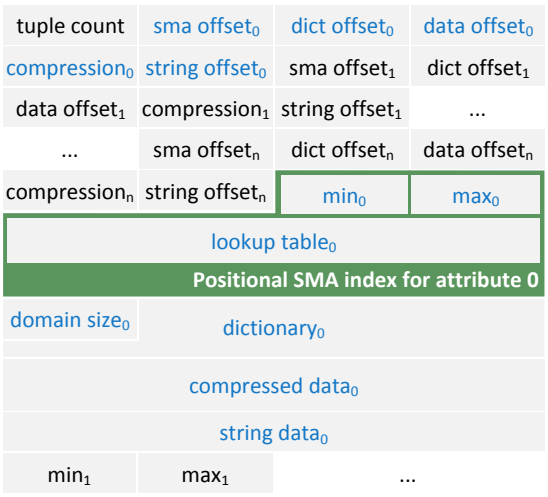


Figure 3: Layout of a Data Block for n attributes

while retaining the high OLTP and OLAP performance. By maintaining a flat structure without pointers, Data Blocks are also suitable for eviction to secondary storage. A Data Block contains all data required to reconstruct the stored attributes and our novel light-weight PSMA index structures, but no metadata, such as schema information, as replicating this in each block would waste space. Although orthogonal to this work, Data Blocks have further been designed with upcoming secondary storage solutions in mind, including non-volatile RAM (NVRAM) and byte-addressable flash storage. Data stored in Data Blocks on such storage devices can directly be addressed and read without bringing the whole Data Block into RAM. Several works have recently addressed the challenges of such storage devices [8, 33].

In HyPer, Data Blocks are used as a compressed in-memory storage format for cold data and for persistence. Identifying cold chunks of a relation is an orthogonal topic to the one addressed in this publication and has e.g., been addressed in [15, 9]. Once records have been packed into a Data Block, the contained data is immutable (frozen). Only delete operations are permitted where frozen records are marked with a flag. Updates are internally translated into a delete followed by an insert into a hot uncompressed chunk of the relation. In addition to immutability, Data Blocks have several convenient properties: (i) An optimal compression method is chosen based on the actual value distribution of an attribute within a chunk. This allows high compression ratios (cf., evaluation in Section 5). (ii) Only byte-addressable compression formats are used and the use of sub-byte encodings such as BitWeaving [22] is rejected in order to allow for efficient point accesses, which are required for OLTP and also for OLAP processing. (iii) SARGable scan restrictions, i.e., $=, \texttt{is}, <, \leq, >, \geq, \texttt{between}$, are evaluated on the compressed data representation using our aforementioned SIMD approach to find matches (cf., Section 4.2). As most compressed data in a Data Block is stored in either a 1-, 2-, or 4-byte integer, our SIMD algorithms provide even higher speedups than on uncompressed data. (iv) Data Blocks further contain Small Materialized Aggregates [23] (SMAs) that include a minimum and a maximum value for each attribute in the Data Block, which can be used to determine if a Data Block can be skipped during a scan. (v) Additionally, we include a novel light-weight index structure, PSMA,

that maps a value to a range of positions on the Data Block where this value appears. Using the PSMAs, scan ranges can further be narrowed, even in cases where the value domain is large and entire blocks cannot be skipped.

SMAs and the PSMA indexes are not restricted to our Data Blocks format and could also be used for uncompressed chunks. However, in HyPer we refrain from using these for hot uncompressed data as maintaining the minimum and maximum information as well as updating the PSMA indexes would have a very high negative impact on transaction processing performance. The hot part of the database is usually rather small, such that the additional overhead is not justified.

3.1 Storage Layout

The first value in a Data Block is the number of records it contains. Typically, we store up to 2^{16} records in a Data Block. The tuple count is followed by information about each of the attributes. Per attribute we store the compression method that is used and offsets to the attribute's Small Materialized Aggregates (SMA) [23], dictionary, compressed data vector, and string data. The attribute information is followed by the actual data beginning with the SMA and PSMA for the first attribute. Figure 3 shows an example layout of a Data Block.

Since Data Blocks store data in a columnar layout, but can store all attributes of a tuple in the same block, they resemble the PAX [3] storage format.

3.2 Positional SMAs

Data Blocks store SMAs for each attribute in the block. The basic SMAs consist of the minimum (min) and maximum (max) value of each attribute stored in a Data Block. Similar to the Optimized Row Columnar (ORC) Hadoop file format [14], we use this domain knowledge to rule out entire Data Blocks if a SARGable (Search ARGument) predicate does not fall within the min and max values of a Data Block. Such min/max SMAs work particularly well on sorted attributes. In real data sets this is often the case for date or incrementing key attributes. If, for example, the `lineitem` relation of TPC-H is sorted on `l_shipdate`, Q_6, which has a highly selective predicate on `l_shipdate`, can be answered by only looking at a few Data Blocks as most blocks can already be ruled out by the basic SMAs. If, however, values are uniformly distributed over the relation, SMAs usually do not help. A single outlier can increase the min/max range to such an extent that the SMA cannot rule out the block anymore. In particular, this is also true for TPC-H data as generated by the *dbgen* tool. For our TPC-H evaluation of Data Blocks, we kept the insertion order of the generated CSV files, which are only sorted according to the primary keys. As all relevant attribute values are uniformly distributed over the blocks in this case, no blocks were skipped during query processing. Within a Data Block, however, usually only a fraction of the records will qualify.

To further narrow the scan range we thus extend an attribute's basic min/max information with a light-weight index structure, called Positional SMA (PSMA). Internally, PSMAs consist of a concise *lookup table* that is computed when a cold chunk is "frozen" into a Data Block. Each table entry contains a scan range $[b, e)$ that points to the compressed data inside a Data Block with potential matching tuples. For fixed size data types, the lookup table contains

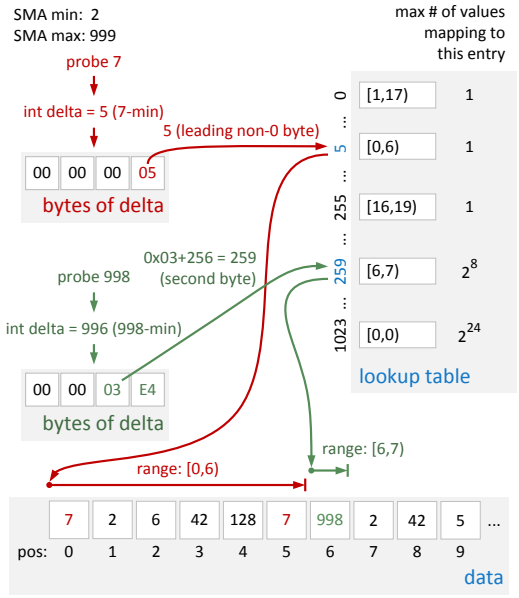


Figure 4: Examples of probing the lookup table of a Positional SMA index (PSMA)

2^8 entries for each byte of the data type, i.e., one entry for the values each byte can store. For example, for a 4-byte integer attribute, the lookup table contains $4 \times 2^8 = 1024$ entries. As such, multiple values map to the same lookup table entry stored in a single entry. If n values map to the same entry at index i and $[b_0, e_0), \ldots, [b_{n-1}, e_{n-1})$ are the ranges associated with these n values, then the range at entry i is widened to $[\min_{0 \le j < n} b_j, \max_{0 \le j < n} e_j)$. Thus, by design, the entries in the lookup table are more accurate for small values. Therefore, to improve the accuracy and the pruning power of the lookup table, we do not use the actual attribute value v itself, but its distance $\Delta(v)$ to the SMA's minimum value. The number of $\Delta(v)$ values whose range information is stored in the same lookup table entry increases with the size of the deltas. Delta values that fit into a single byte exclusively map to a single table entry (without collisions). For 2-byte delta values, 2^8 values map to the same table entry (2^{16} for 3-byte values, and 2^{24} for 4-byte values). Each entry points to a range on the compressed data where attribute values that map to this entry are stored. When looking for values equal to v, the associated scan range points to the position of first and one past the last element (right exclusive) where the most-significant non-zero byte equals the most-significant non-zero byte of v.

A lookup proceeds as follows: First, the delta between the probe value v and the SMA's min value is determined: $\Delta = v - \min$. Next, the highest non-0 byte of the delta, denoted as $\overline{\Delta}$, and the number of remaining bytes, denoted as r, are used to compute the index i into the lookup table: $i = \overline{\Delta} + r \times 256$. To determine the scan range for equality predicates, only a single lookup is required (Figure 4 illustrates example lookups and the implementation is sketched in Appendix B). For non-equality predicates like `between`, we perform multiple lookups and union the ranges of the entries. In the specific case of a `between a and b` predicate, we determine the lookup table indexes i_a of `a` and i_b of `b` and union the non-empty ranges for the indexes from i_a to i_b: range $= [\min_{i_a \le i \le i_b} b_b, \max_{i_a \le i \le i_b} e_i)$.

Building a lookup table is an $\mathcal{O}(n)$ operation. First, the table is initialized with empty ranges. Then, a scan over the column is performed. For each value v_i of the column $C = (v_0, \ldots, v_{n-1})$ the associated table entry is determined. If the entry contains an empty range, then the entry is set to $[i, i+1)$, otherwise the range end is updated to $i+1$.

As shown in Figure 3, an individual lookup table is stored for each attribute. In the presence of multiple SARGable predicates on different attributes, the individual lookup tables are queried and the returned scan ranges are then intersected to further narrow the scan range. Even though we store a lookup table for each attribute, the PSMAs consume only a fraction of the space of a Data Block. Each entry in the PSMA lookup table contains a range which is represented as two unsigned 4-byte integers. When n-byte values are indexed, the table consists of $n \times 2^8$ entries in total. Thus, typical memory footprints are 2 KB, 4 KB and 8 KB for values of type 1-, 2- or 4-byte integers, respectively. The PSMA lookup table is thus significantly smaller than a tree-based index structure on the 2^{16} values of a Data Block. Since the PSMA only limits the range that is scanned and generates the same access path as a full scan, the lookup table is also more robust than traditional index lookups and does not incur a performance penalty in cases when the range of potentially qualifying values is very large (or the entire vector qualifies). The precision of PSMAs depend on both the domain of the values as well as their order in the vector. Scan range narrowing works particularly well for values that have a small distance from the minimum value as fewer delta values share a lookup table entry.

PSMA range pruning is particularly efficient for data sets where similar values are physically clustered, such that values which share an entry have similar ranges. In the case of Data Blocks, such a clustering can be created when the data is re-ordered during freezing. If workload knowledge exists or was collected while processing queries on the uncompressed chunks, Data Blocks can be frozen based on a sort criterion to improve accuracy of PSMAs for similar queries.

3.3 Attribute Compression

The main requirement for our compression schemes is that compressed data needs to remain byte-addressable for efficient point accesses. As shown in our evaluation, sub-byte encodings (e.g., BitWeaving [22]) would indeed allow for greater compression ratios but increase the cost for point accesses and scans with low selectivities by orders of magnitude compared to our byte-addressable encodings (cf., Section 5.4). The following compression schemes have proven themselves useful and suitable in terms of compression ratio and overall scan and point access performance in the context of Data Blocks: (i) single value compression, (ii) ordered dictionary compression, and (iii) truncation. For each attribute, the compression scheme is chosen that is optimal with regard to resulting memory consumption and data is only stored uncompressed in the rare case that no compression scheme is beneficial. Note that the choice of the compression scheme depends largely on the value domain of the attribute in the specific block. Thus, different blocks that store different ranges of an attribute of a relation might be compressed using many different schemes.

Single value compression is a special case of run-length encoding and is used if all values of an attribute in a block are equal. This includes the case where all values are `NULL`.

As Data Blocks are immutable data structures, our *ordered dictionary compression* does not need to be capable of handling insertions of new values or other kinds of updates. Immutability allows us to use an order-preserving dictionary compression, a scheme that is too expensive to use if dictionaries can be updated or grow. In our ordered dictionary, if $k < k'$ holds for two uncompressed values, k and k', then the same holds for their dictionary-compressed values $d_k, d_{k'} : d_k < d_{k'}$. Immutability also frees us from the burden of reference counting and free space management. The byte-width of the dictionary keys is chosen depending on the number of distinct keys, i.e., 8-bit integers for up to 256 keys, 16-bit integers for up to 2^{16} distinct keys, and 32-bit for up to 2^{32} distinct keys. A major advantage of preserving the order and using byte-truncated keys is that we can use our SIMD-optimized algorithms to find matches (cf., Section 4.2) on the compressed data. In fact, finding the matches is likely faster on compressed data than on uncompressed data because only the truncated values need to be compared and thus more values will fit in a SIMD register.

The *truncation* compression scheme reduces the memory consumption by computing the delta between each value and the attribute's minimum value: Let $A = (a_0, \ldots, a_m)$ denote the uncompressed data and $\min(A)$ be the smallest element in A, then $\Delta(A) = (a_0 - \min(A), \ldots, a_m - \min(A))$ is the compressed data. For these delta values, our truncation scheme again uses byte-truncation to either 8-bit, 16-bit, or 32-bit integers. Truncation is not used for strings and `double` types. An exception is the string type `char(1)` which is always represented as a 32-bit integer (such that it can store any UTF-8 character). Our truncation scheme is a special case of Frame of Reference (FOR) encoding where in our case the minimum value is the reference and each Data Block contains exactly one frame.

Even though the Data Block compression schemes are light-weight, we measured compression ratios of up to 5× compared to uncompressed storage in HyPer. Compared to Vectorwise's compressed storage, Data Blocks consume around 25% more space (see Table 1).

As each Data Block is self contained, the attribute compression is as well limited to a single block. This blockwise approach has drawbacks in terms of compression ratio. For example, dictionary compressed string data causes redundancies when identical strings appear in multiple different chunks. In that case, these strings have to be stored in multiple dictionaries. However, blockwise compression offers the aforementioned opportunity to chose the best suitable compression scheme for each column in each individual block, which can amortize this overhead. Depending on the value distribution, the blockwise approach can result in better compression ratios, compared to when relations are compressed as a whole. Another opportunity is that in a chunked relation, the system can easily migrate chunks between their compressed and uncompressed representation based on OLTP access frequency.

3.4 Finding and Unpacking Matches

Finding and unpacking matches in Data Blocks proceeds as follows: First, a Data Block's SMAs and PSMAs are used to narrow the scan range based on the scan restrictions. If the resulting scan range is not empty, further checks are performed before the actual scan starts. E.g., in case of dictionary compression and an equality predicate, a block

can be ruled out if a binary search on the dictionary does not find an entry. If the block cannot be ruled out, then the actual scan starts and all restrictions are evaluated on the compressed columns. The scan yields a vector which contains the *positions* (or offsets) of the matching tuples. These matches are unpacked by their positions before being pushed to the consuming operator. This vector-at-a-time processing is repeated until no more matches are found in the Data Block.

In contrast to a scan of an uncompressed chunk, restrictions are evaluated on compressed data. As we reject subbyte compression techniques in Data Blocks, the search for matches is thus (in most cases) reduced to an efficient scan over small integers with a simple comparison. Note that also string types are always compressed to integers. The predicate evaluation on compressed data only adds a small overhead on a per-block-basis as restriction constants have to be converted into their compressed representation.

Point-accesses to tuples residing in Data Blocks do not require any of the previous steps. Instead, required attributes of a record are uncompressed from a single position.

4. VECTORIZED SCANS IN COMPILING QUERY ENGINES

Data Blocks individually determine the best suitable compression scheme on a per-block and per-column basis. The resulting variety of physical representations improves the compression ratio but constitutes a challenge for JIT-compiling tuple-at-a-time query engines: Different storage layout combinations and extraction routines require either the generation of multiple code paths or the acceptance of runtime overhead incurred by branches for each tuple.

Our goal, thus, is to efficiently integrate multiple storage layouts, most specifically for our novel compressed Data Blocks layouts, into our tuple-at-a-time JIT-compiling query engine, that is optimized for both OLTP and OLAP workloads. HyPer uses a data-centric compilation approach that compiles relational algebra trees to highly efficient native machine code using the LLVM compiler infrastructure. The compilation times from LLVM's intermediate representation (IR) which we use for code generation to optimized native machine code is usually in the order of milliseconds for common queries.

Compared to a traditional query execution model where for each tuple or vector of tuples the control flow passes from one operator to the other, our query engine generates code for entire query pipelines. These pipelines essentially fuse the logic of operators that do not need intermediate materialization together. A query is broken down into multiple pipelines where each pipeline loads a tuple out of a materialized state (e.g., a base relation or a hash table), then performs the logic of all operators that can work on it without materialization, and finally materializes the output into the next pipeline breaker (e.g., a hash table). Note that compared to traditional interpreted execution, tuples are not pulled from input operators but are rather *pushed* towards consuming operators. In this context, scans are leaf operators that *feed* the initial query pipelines. The generated scan code (shown in C++ instead of LLVM IR for better readability) of two attributes of a relation stored in uncompressed columnar format looks as follows:

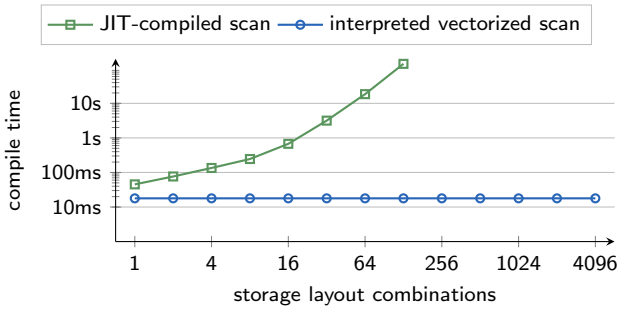


Figure 5: Compile times of a query plan with a scan of 8 attributes and a varying number of storage layout combinations of the base relation in HyPer.

```
for (const Chunk& c:relation.chunks) {
  for (unsigned row=0;row!=c.rows;++row) {
    auto a0=c.column[0].data[row];
    auto a3=c.column[3].data[row];
    // Check scan restrictions and push a0,a3
    // into consuming operator
    ...
} }
```

Note that for reasons of simplicity we omit multi-version concurrency control checks here. In order to perform the same scan over different storage layout combinations depending on the used compression techniques, the first possibility is to add a jump table for each extracted attribute that jumps to the right decompression method:

```
const Column& c0=c.column[0];
// Expensive jump table per attribute
switch (c0.compression) {
  case Uncompressed: ...
  case Dictionary: a0=c0.dict[key(c0.data[row])];
  ...
}
```

Since the outcome of the jump table's branch is the same within each chunk, there will not be a large number of branch misses due to correct prediction of the branches by the CPU. Yet, the introduced instructions add latency to the innermost hot loop of the scan code, and, in practice, this results in scan code that is almost 3× slower than scan code without these branches.

An alternative approach that does not add branches to the innermost loop is to "unroll" the storage layout combinations and generate code for each of the combinations. For each chunk, a computed goto can then be used to select the right scan code:

```
for (const Chunk& c:relation.chunks) {
  // Computed goto to specialized "unrolled"
  // code for the chunk's storage layout
  goto *scans[c.storageLayout];
  ...
  a0dicta3uncompressed:
  for (unsigned row=0;row!=c.rows;++row) {
    a0=c.column[0].dict[key(c0.data[row])];
    a3=c.column[3].data[row];
    ...
} }
```

Unrolling the combinations, however, requires the query engine to generate a code path for each storage layout that is used. The number of these layouts grows exponentially with the number of attributes n. If each attribute may be rep-

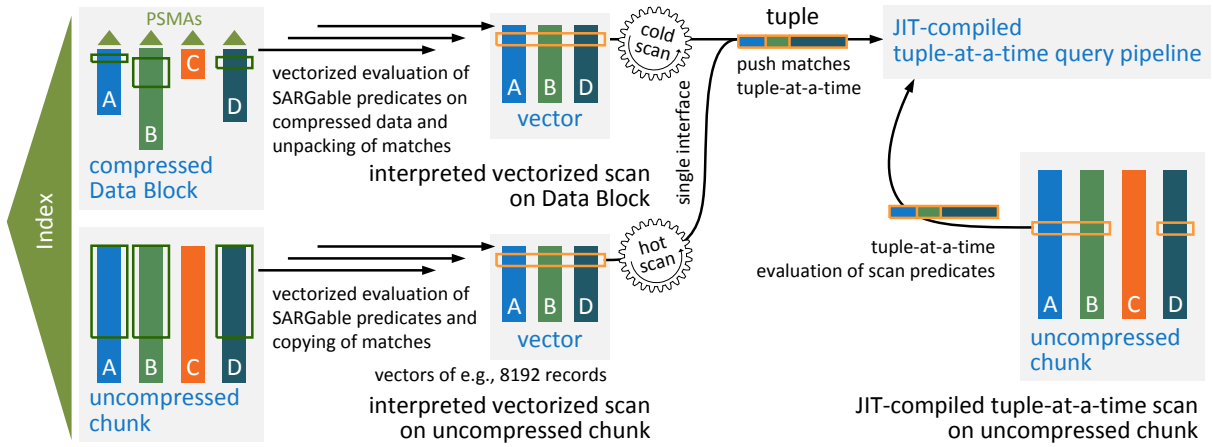


Figure 6: Integration of Data Blocks in our query engine: vectorized scans on Data Blocks and uncompressed chunks on the left share the same interface and evaluate SARGable predicates on vectors of records using SSE/AVX2 SIMD instructions (cf., Section 4.2). Matches are pushed to the query pipeline tuple at a time. The original JIT-compiled scan on the right evaluates predicates as part of the query pipeline.

resented in p different ways, the resulting number of code paths is p^n; e.g., for only two attributes and six different representations there are 36 generated code paths. While one can argue that not all of these combinations will actually occur in a relation[1], a small number of combinations will drastically increase code size and thus compilation time. This impact is shown in Figure 5, which plots the compilation time of a simple `select *` query on a relation with 8 attributes and a varying number of storage layout combinations.

Given the exploding compile time, we thus turned to calling pre-compiled interpreted vectorized scan code for vectors of say 8K tuples. The returned tuples are then *consumed* tuple-at-a-time by the generated code and pushed into the consuming operator:

```
while (!state.done()) {
  // Call to pre-compiled interpreted vectorized scan
  scan(result,state,requiredAttributes,restrictions);
  for (auto& tuple:result) {
    auto a0=tuple.attributes[0];
    auto a3=tuple.attributes[1];
    // Check non-SARGable restrictions and push a0,a3
    // into consuming operator
    ...
} }
```

Using the pre-compiled interpreted vectorized scan code, compile times can be kept low, no matter how many storage layout combinations are scanned (cf., Figure 5). Additionally, SARGable predicates can be pushed down into the scan operator where they can be evaluated on vectors of tuples.

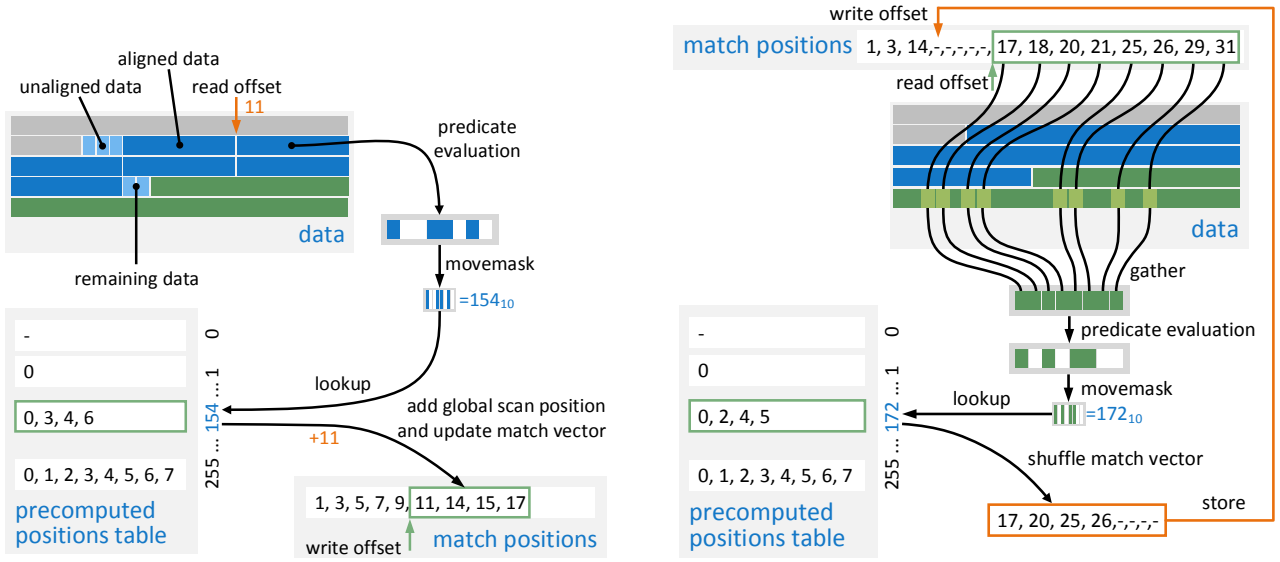
4.1 Integration in HyPer

Our JIT-compiling query engine is integrated in our full-fledged main-memory database system HyPer that supports SQL-92+ query processing and ACID transactions. As illustrated in Figure 6, vectorized scans on hot uncompressed chunks and compressed Data Blocks share the same interface in HyPer and JIT-compiled query pipelines are oblivious to the underlying storage layout combinations.

[1]Our proposed compressed Data Blocks use over 50 different layouts for the lineitem relation of TPC-H scale factor 100.

In HyPer, vectorized scans are executed as follows: First, for each chunk of a relation a determination is made as to whether or not the block is frozen, i.e., compressed. If yes, then a Data Block scan is initiated, if not, a vectorized scan on uncompressed data is initiated. Next, the JIT-compiled scan glue code calls a function that generates a match vector containing the next n positions of records that qualify restrictions. n is the vector size and determines how many records are fetched before each of these records are pushed to the consuming pipeline one tuple at a time. The rationale for splitting the scan into multiple invocations is cache efficiency: As the same data is accessed multiple times when finding the matches, potentially unpacking these matches, if compressed, and passing them to the consumer, the vector-wise processing in cache-friendly pieces minimizes the number of cache misses (see Appendix A for an experiment with different vector sizes). In HyPer, the vector size is set to 8192 records. After finding the matching positions, scan glue code on a cold compressed Data Block calls a function that unpacks the matches into temporary storage, and a scan on an uncompressed chunk copies the matching required attributes into temporary storage. Finally, the tuples in the temporary storage are pushed to the consuming operator tuple at a time. Even though vectorized scans are indeed copying more data, our evaluation of vectorized scans in our JIT-compiling query engine shows that most of the time the costs for copying can be neglected and vectorized predicate evaluation can outperform tuple-at-a-time evaluation.

In this respect, Q_1 and Q_6 of TPC-H exemplify two extremes: for Q_1 most tuples qualify the scan restriction and vectorized scans copy almost all of the scanned data. As such, the runtime of Q_1 suffers by almost 50% (cf., Appendix). Note that without predicates, our vectorized scan uses an optimization whereby it does not copy data if all tuples of a vector match and performance is not degraded; due to the uniform value distribution of the restricted attributes, this optimization is not helpful if predicates are SARGd. For Q_6, on the other hand, only a small percent of tuples qualify the scan restriction. On uncompressed data, the vectorized evaluation of predicates improves runtime with vectorized scans over JIT-compiled scans up to 2.3× (cf., Appendix F).



(a) The first applied restriction yields an initial vector of matching record positions. Positions for masks are stored in a lookup table to avoid costly computations.

(b) Additional restrictions reduce the initial match vector. Lookup table entries are used as shuffle control masks.

Figure 7: Evaluation of predicates on compressed and uncompressed data using SSE/AVX2 SIMD instructions

Using vectorized scans on our novel compressed Data Blocks, query runtimes on TPC-H improve by more than 2.3×: runtime of Q_6 improves by 6.7× and the geometric mean of the 22 query runtimes improves by 1.27×.

4.2 Finding Matches using SIMD Instructions

Vectorized processing enables us to employ SIMD instructions for evaluating SARGable predicates. As mentioned before, during the evaluation phase the *positions* (or offsets) of the qualifying elements are stored in a vector. We refer to the content of this vector as *match positions*[2]. For example, if the predicate $P(a) : 3 \leq a \leq 5$ is applied to the attribute vector $\vec{A}^T = (0, 1, 5, 2, 3, 1)$, then the resulting match positions are $\vec{M}^T = (2, 4)$. If additional conjunctive restrictions are present, then these restrictions need to be applied only to the tuples referred in M. At this low level, query engine only has to cope with conjunctive predicates, thus the match vector is shrunk with any additional predicate. Algorithmically, we distinguish between finding *initial* matches, which fill a match vector, and an optional *reduce* matches where an already existing match vector is shrunk.

In scalar (non-SIMD) code, restrictions are applied to a single attribute value at a time and evaluated to a single boolean value. When SIMD instructions are employed, the necessary comparisons are performed on multiple attributes in parallel. In contrast to the scalar code, a SIMD comparison yields a bit-mask which identifies the qualifying elements. When n elements are processed in parallel, then the resulting bit-mask is stored in a SIMD register where all bits of the n individual SIMD lanes are either set to 0 or 1. The challenging part is to compute the positions of matching records based on the given bit-mask. Possible solutions are (i) iterating over the bit-mask or (ii) performing a tree reduction where the complexity is $\mathcal{O}(n)$ or $\mathcal{O}(\log n)$, respectively.

[2]This concept is also known as *selection vector* [5].

We found that these conversions of the bit-mask into a match position vector is too expensive. Therefore, we make use of a pre-computed table to map the bit-masks to positions whereby a `movemask` instruction provides the necessary offset. With this approach, which is illustrated in Figure 7(a), the mapping can be done in constant time. Each table entry reflects a possible outcome of an n-way SIMD comparison. For example, in Figure 7(a) an 8-way comparison is performed where the 0 th, the 3 rd, the 4 th and the 6 th element (counted from left to right) satisfy the predicate. All bits of the 4 corresponding SIMD lanes are set to 1, whereas the others are all set to 0. The `movemask` instruction extracts the most significant bit of each SIMD lane and stores the result in an integer value. In the example, this value is $10011010_2 = 154_{10}$. This value is used as an offset to access the corresponding entry in the lookup table. The table entry reflects the outcome of the 8-way comparison. In the example, the result is an 32-bit integer vector where the positions of the matching elements are $(0, 3, 4, 6)$. The global scan position is added to the (local) match positions before the match vector is updated.

As each of the n attribute values stored in a SIMD register may or may not be a match, we have to store each possible outcome in the table which results in 2^n entries. Naturally, the table can become extremely large. E.g., for attributes of type 8-bit integer processed in 256-bit AVX2 registers (using a 32-way comparison), the table would contain 2^{32} entries (= 512 GB). Therefore, we limit the table size to 2^8 entries and perform multiple lookups if necessary. A single entry in the lookup table can therefore contain the positions of up to 8 matches. Each match position is represented as an unsigned 32-bit integer, thus the size of the entire lookup table ($2^8 \times 8 \times 4\,\text{B} = 8\,\text{KB}$) is sufficiently small to fit into L1 cache (which is 32 KB on recent Intel CPUs).

Along with the match positions, each table entry also contains the number of matching elements (not depicted). These values are stored in the low-order bits to keep the

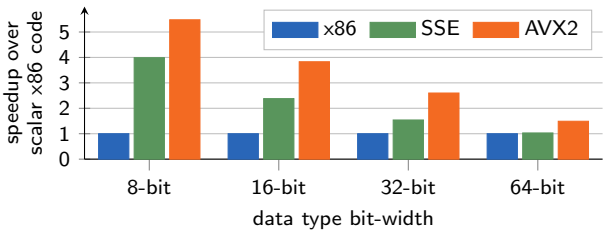


Figure 8: Speedup of SIMD predicate evaluation of type $l \leq A \leq r$ with selectivity 20%

memory footprint small and to avoid cache pollution. Thus, the extraction of the actual match positions requires an additional right-shift of the pre-computed position vectors. The number of matches per table entry are required to increment the write offset of the global match vector later on. We show the implementation details in Appendix C.

Applying additional restrictions is quite different to finding initial matches for two reasons: (i) The accessed elements are no longer contiguously stored in memory, as only elements are read that satisfy the first predicate, and (ii) a match vector already exists and needs to be shrunk. Nevertheless, our SIMD optimized implementation follows an approach similar to the one to find the initial matches. The most notable difference is that elements are directly *gather*ed from their respective memory location into SIMD registers as sketched in Figure 7(b). Again, we make use of our lookup table consisting of pre-computed position vectors. In contrast to the find initial matches implementation, the table entries are now used as *shuffle control mask*s to manipulate the match vector. In Figure 7(b), for example, the predicate is applied to the attributes at positions $(17, 18, 20, 21, 25, 26, 29, 31)$ where the elements in the SIMD lanes $(0, 2, 4, 5)$ qualify. The match positions are then shuffled according to the table entry: The zeroth element (value 17) remains in SIMD lane 0, the second element (value 20) is shuffled to the SIMD lane 1, and so on. Finally, the result $(17, 20, 25, 26, -, -, -, -)$ is written back to the match vector and the write offset is incremented by the number of matching elements. As only 4 of the 8 processed elements qualify the result contains *don't care* values (denoted as $-$) which are overwritten in the next iteration.

Unfortunately, the `gather` instruction is only available for 32- and 64-bit types. For 8- and 16-bit values we also draw on a 32-bit gather which reduces the degree of parallelism by a factor of 2 or 4, respectively. Thus, the performance improvements observed in our microbenchmarks are mostly independent from the underlying data types.

We observe speedups of almost $4\times$ with SSE and more than $5\times$ with AVX2 in microbenchmarks on a desktop Haswell CPU. Figure 8 shows the speedup of the evaluation of a *between* predicate where 20% of the tuples qualify. The degree of parallelism and therefore the performance gains highly depend on the bit-width of the underlying data type. For 64-bit integer attributes we observe speedups of $1.5\times$ with AVX2, whereas the degree of parallelism with SSE is too small to recognize performance benefits.

Our implementation for finding initial matches is insensible to varying selectivities due to the pre-computed positions table and varies only in the number of match positions written to the position vector. However, due to the fact that we process the input in a vectorized manner, the size of the

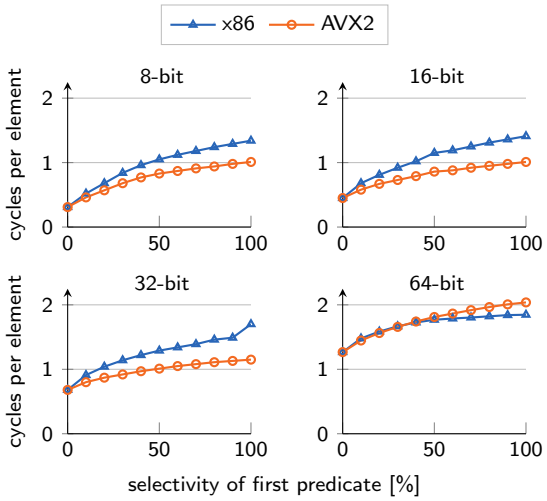


Figure 9: Applying an additional restriction with varying selectivities of the first predicate and the selectivity of the second predicate set to 40%

match vector is limited to the number of tuples processed at a time (which is set to 16 K in this experiment). Thus, the number of matches written to the vector has no notable effects on performance.

The implementation for reducing the match vector by applying additional restrictions is also insensible to varying selectivities of the applied restriction. However, scan performance highly depends on the selectivity of the preceding restrictions due to non-contiguous memory access pattern. When matching tuples are uniformly distributed, as they are in our experiment, the selectivity of the preceding predicates has the highest impact. Figure 9 shows these effects and compares the performance of our AVX2 implementation with a sequential version (branch-free scalar code). For integer attributes (up to 32-bit) we observe performance gains ranging from $1.0\times$ to $1.25\times$ with increasing selectivities. For 64-bit integer values, the reduction does not benefit from SIMD instructions. At higher selectivities, the SIMD implementation performs even worse than the scalar code.

Performance gains with other data types are almost identical, which is attributed to the missing `gather` instructions for 8- and 16-bit values. We would like to point out that the distribution of qualifying tuples is the crucial factor here, thus speedups of $1.25\times$ are also possible with highly selective predicates, e.g., when the data has a (natural) ordering.

The SIMD optimized algorithms presented here only work for integer data. For other data types and non-SARGable predicates we fall back to scalar implementations. However, our novel compressed Data Block format discussed in the previous Section compresses data into byte-addressable integer data, including string-like attributes. Thus, OLAP workloads highly benefit from the SIMD algorithms when large portions of the data are packed into Data Blocks.

5. EVALUATION

In this section we evaluate our implementation of interpreted vectorized scans, SIMD-optimized predicate evaluation, and the compressed Data Blocks format in our JIT-compiling query engine. The query engine we used for our experiments, is part of our full-fledged main-memory data-

	TPC-H SF100	IMDB cast info	Flights
uncompressed			
CSV	107 GB	1.4 GB	12 GB
HyPer	126 GB	1.8 GB	21 GB
Vectorwise	105 GB	0.72 GB	11 GB
compressed			
HyPer	66 GB	0.50 GB	4.2 GB
Vectorwise	54 GB	0.24 GB	3.2 GB

Table 1: Size of TPC-H, IMDB cast info, and a flight details database in HyPer and Vectorwise.

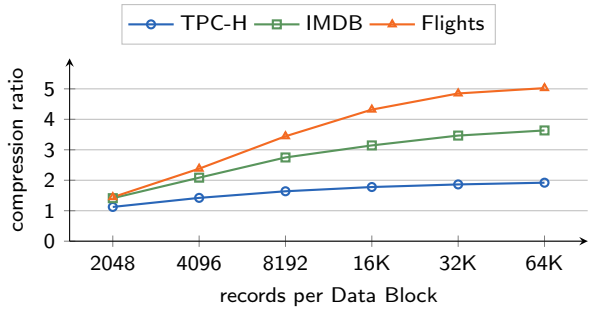


Figure 10: Compression ratio of TPC-H (scale factor 100), IMDB cast info, and a flight arrival and departure details data set for various Data Block sizes in HyPer (compared to uncompressed storage).

base system HyPer that supports SQL-92 queries and ACID transactions. We further show why we rejected sub-byte encodings of data by comparing our byte-addressable compression approach with SIMD-optimized *horizontal bit packing*.

Unless otherwise stated, the experiments were conducted on a 4-socket Intel Xeon X7560 2.27 GHz (2.67 .GHz maximum turbo) NUMA system with 1 TB DDR3 main memory (256 GB per CPU) running Linux 3.19. Each CPU has 8 cores (16 hardware contexts) and 24 MB of shared L3 cache. For experiments targeting the AVX2 ISA, we use an Intel Haswell i5-4670T CPU 2.3 GHz (3.3 GHz maximum turbo) with 16 GB DDR3 memory and 6 MB of shared L3 cache.

5.1 Compression

We evaluated the compression ratio of our compressed Data Blocks in an initial experiment. As input data sets we chose TPC-H scale factor 100, the largest relation of the Internet Movie Database (IMDB), which contains the casts of movies (cast info), and a data set consisting of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008[3]. Table 1 shows a comparison of uncompressed and compressed database sizes for these data sets in Vectorwise and HyPer.

Vectorwise is the commercial version of MonetDB/X100 [5] and uses a number of light-weight compression techniques that can handle non-compressible values (outliers) through a technique called "patching", which stands for the "P" in PFOR (Frame of Reference), PFOR-DELTA (delta encoding on PFOR), and PDICT (dictionary encoding) compression schemes [40, 38]. Run-length encoding is used where applicable. With these heavier compression schemes, Vectorwise

[3]http://stat-computing.org/dataexpo/2009/

scan type	geometric mean	sum
HyPer		
JIT (uncomressed)	0.586s	21.7s
Vectorized (uncompressed)	0.583s (1.01×)	21.6s
+ SARG	0.577s (1.02×)	21.8s
Data Blocks (compressed)	0.555s (1.06×)	21.5s
+ SARG/SMA	0.466s (1.26×)	20.3s
+ PSMA	0.463s (1.27×)	20.2s
Vectorwise		
uncompressed storage	2.336s	74.4s
compressed storage	2.527s (0.92×)	78.5s

Table 2: Runtimes of TPC-H queries (scale factor 100) using different scan types on uncompressed and compressed databases in HyPer and Vectorwise.

on average saves 25% more space than HyPer. However, this savings come at the cost of poorer query performance. Vectorwise's compression schemes were designed to work for situations where the database does not fit in memory and aims to accelerate disk load times. Processing compressed, memory-resident data is acceptably fast, but can be slower than processing uncompressed data [2]. In particular, we measured that TPC-H queries Q_1 and Q_6 were, respectively, 18% and 38% slower on compressed storage compared to uncompressed storage in Vectorwise (scale factor 10). Point accesses in Vectorwise are always performed as a scan; albeit on a scan range that is narrowed by indexes. Data Blocks, on the other hand, are designed around the assumption that most of the data fits into main memory. Performance-wise, our goal with Data Blocks is to be at least equally fast at query and transaction processing compared to uncompressed in-memory storage.

Figure 10 shows the compression ratio of the aforementioned data sets for various Data Block sizes from 2^{11} to our default of 2^{16} records per Data Block. As expected, if the number of compressed records in a block becomes too small, the overhead of block metadata worsens the compression ratio. In our experiments, 2^{16} records per Data Block proved to offer a good tradeoff between compression ratio and query processing speed as most values are still compressed into small integer types.

5.2 Query Performance

For our query performance evaluation, we first looked at TPC-H scale factor 100 and compared query runtimes in HyPer with JIT-compiled scans and vectorized scans on our uncompressed storage format, and vectorized scans on our compressed Data Block format. A summary of the results is shown in Table 2; full results are shown in Appendix F. 64 hardware threads were used and runtimes are the median of several measurements. Our results suggest that query performance does not change significantly with vectorized scans instead of JIT-compiled scans. This is also true if we push SARGable predicates (+SARG) into the vectorized scan. However, compilation times with vectorized scans is almost half of the time needed with JIT-compiled scan code (see numbers in parentheses in Appendix F). If we look at query performance on our Data Blocks format, we can see that if using only the compression aspect of Data Blocks, query performance also stays the same. If we push SARGable predicates (+SARG/SMA) into the scan on Data Blocks, we see a speedup of almost 26% in the geometric

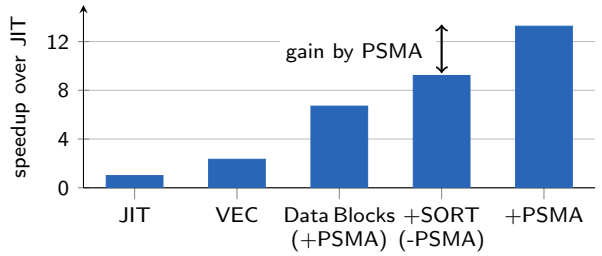


Figure 11: Speedup of TPC-H Q6 (scale factor 100) on block-wise sorted data (+SORT)

mean of query runtimes. We cannot measure the impact of SARGd predicates without SMA block skipping because our implementation of SARGing, by design, relies on block skipping. Nevertheless, no blocks were skipped in our experiment due to the uniform distribution of values in TPC-H and the performance improvement is only due to our efficient SIMD-optimized evaluation of predicates on vectors of the compressed data. As data is uniformly distributed, adding the PSMAs (+PSMA) does not provide a significant performance boost on TPC-H. We further compared HyPer to Vectorwise on its uncompressed and compressed storage formats. In Vectorwise, query processing on the uncompressed format is around 8% faster than on its compressed format when data is cached in memory. This suggests that even with light-weight compression techniques, such as those used in Vectorwise, it is actually very difficult to get a speedup from compression.

To further show the impact that PSMAs can have in more realistic scenarios than the default TPC-H, we conducted another TPC-H experiment for which we sorted each Data Block of the `lineitem` relation on `l_shipdate`. Due to the initial uniform distribution of dates, each block still contains data from every year of the data set, but inside the blocks the dates are sorted. Establishing this order can be done automatically when freezing a block (cf., Section 3.2). Figure 11 shows that a significant additional speedup can be gained by the PSMAs for TPC-H Q_6 in this scenario. Another query that profits even more from the SMAs and PSMAs is the following on the flights data set: select carriers and their average arrival delay in San Francisco for the years 1998 to 2008 (see Appendix D for the query text). As the relation is naturally ordered on date, most blocks are skipped due to the SMAs and on the remaining blocks the PSMAs narrow the scan range based on the restriction on the destination airport. Runtime of this query improves by more than 20×, compared to using a JIT-compiled scan on the uncompressed format.

5.3 OLTP Performance

To measure the overhead of accessing a record that is stored in a Data Block compared to uncompressed chunks, we performed a random point access experiment where we selected random records from a compressed TPC-H customer relation (scale factor 100, i.e., 15M records). Table 3 shows the measured lookup throughputs. With a primary key index (traditional global index structure), we measured an overhead of around 60% when accessing records in Data Blocks. Without the primary key index, all lookups are performed as scans. In this case the scans on Data Blocks can be faster than scanning uncompressed chunks because Data

		Ordered⋆	Shuffled°
uncompressed	PK index	551,268	545,554
(JIT)	no index	36	36
uncompressed	PK index	550,661	566,893
(Vectorized)	no index	26	26
Data Blocks	PK index	301,750	274,198
	no index	17,508	41
Data Blocks	PK index	276,014	294,291
+PSMA	no index	71,587	40

⋆ customer is ordered on `c_custkey` as generated by *dbgen*
° shuffled customer relation (no longer ordered on `c_custkey`): SMAs/PSMAs can no longer narrow scan range

Table 3: Throughput (in lookups per second) of random point access queries `select * from customer where c_custkey = randomCustKey()` on TPC-H scale factor 100 with and without a primary key (PK) index on `c_custkey` and using JIT-compiled and vectorized scans on uncompressed storage and Data Block scans with and without PSMAs.

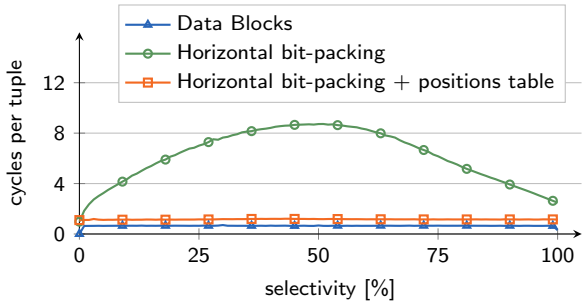
Blocks contain SMAs and PSMAs that can narrow the scan range. This is especially true if the customer relation is ordered on `c_custkey`, as is the case if the data is generated with the TPC-H *dbgen* tool. We also shuffled the data to show that without that order the SMAs and PSMAs cannot improve lookup performance. As such, for OLTP workloads, SMAs/PSMAs are not a general-purpose replacement for a traditional index structure. In Vectorwise, which uses no traditional index structure, we measured a random lookup throughput of 17 lookups per second.

We further ran TPC-C experiments with 5 warehouses. In a first experiment we only compressed old neworder records into Data Blocks, which reflects the intended use case as hot data should remain in uncompressed chunks. We measured a transaction throughput of 89,229 transactions per second on uncompressed storage and 88,699 transactions per second if the cold neworder records are stored in Data Blocks. The overhead stems from the additional switch on each access that determines if an uncompressed or a compressed chunk is accessed. In a second experiment, we executed only the read-only TPC-C transactions order status and stock level on an uncompressed TPC-C database and a TPC-C database that is completely stored in Data Blocks. On uncompressed storage we measured a transaction throughput of 119,889 transactions per second, on Data Blocks we measured 109,649 transactions per second; a difference of 9%. This shows that even if the individual record lookup throughput is decreased by 60%, in real transactional workloads this translates to a low percentage overhead if cold compressed data is accessed.

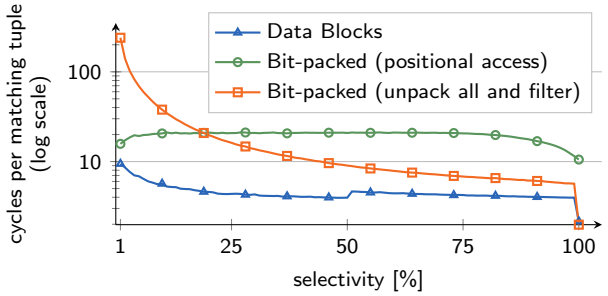
As Data Blocks primarily target OLAP workloads, we omit the mixed workload CH-benCHmark [7]. Our thorough evaluation with TPC-H and TPC-C covers the two extremes of the CH benchmark and we expect the results to be somewhere in between.

5.4 Advantages of Byte-Addressability

As previously mentioned, sub-byte encodings of values (e.g., BitWeaving [22]) can achieve higher compression ratios compared to byte-aligned truncation, which we use in



(a) Cost of evaluating a SARGable predicate of type $l \leq A \leq r$ with varying selectivities



(b) Cost of unpacking matching tuples (3 attributes)

Figure 12: Horizontal bit-packing compared to Data Blocks with byte-addressable compression schemes

Data Blocks. In this microexperiment, we show why we consciously decided against using sub-byte encodings in HyPer and how point accesses and scans with low selectivities suffer from these encodings.

For this experiment we use three columns A, B and C populated with 2^{16} integer values, whereas the domain of the values in column A and B is $[0, 2^{16}]$ and $dom(C) = [0, 2^8]$. Thus, the columns can be horizontally bit-packed into 9 bits or 17 bits, respectively. Intentionally, the domains exceed the 1-byte and 2-byte truncation by one bit which represents the worst case for Data Blocks as they are forced to use 2- and 4-byte codes. Therefore, the compression ratio of bit-packing is almost two times higher in this scenario. As workload, we evaluate a `between` predicate of the form $l \leq A \leq r$ and unpack the matching tuples into an output buffer. The benchmark is conducted on a Haswell i5-4670T CPU with the AVX2 instruction set. The codebase that we used for horizontal bit-packing is the highly SIMD-optimized implementation of [27]. The only bit-packing-related extension is a function that allows us to unpack a single tuple at a given position.

We first compare the costs for predicate evaluation without unpacking. The bit-packing implementation yields a bitmap where set bits identify matching positions. In contrast, our Data Blocks API returns a vector of 32-bit integers populated with the corresponding match positions. For our experiment we transform the bitmap in the case of bit-packing into a position vector to make the results comparable. Figure 12(a) shows that Data Blocks are robust with respect to varying selectivities. Horizontal bit-packing on the other hand is not. Bit-packing suffers from branch mispredictions in the aforementioned conversion of the bitmap. Therefore, we applied our *pre-computed position table* approach (cf., Section 4.2) to the bit-packing implementation, which makes bit-packing also robust to changing selectivities. Overall, predicate evaluation on Data Blocks is still 1.8× faster than predicate evaluation on the horizontal bit-packed format.

We compare Data Blocks to two bit-packing alternatives to evaluate unpacking performance: (i) *Positional access*, where all matching tuples are unpacked sequentially with scalar code, and (ii) *Unpack all and filter*, where all tuples are first unpacked using SIMD code and then sequentially filtered using the positions vector. Figure 12(b) shows the costs per extracted tuple for all three implementations with varying selectivities. Data Blocks outperform bit-packing in almost all cases, except when all tuples qualify where bit-packing is approximately 9% faster. The bit-packing implementation that extracts individual tuples based on the match vector performs well for selectivities less than 20%. If more than 20% of the tuples qualify, the *unpack all and filter* strategy performs better than positional accesses due to SIMD. The costs for unpacking are significantly higher for queries with moderate selectivities due to unpacking mostly non-qualifying tuples. Compared to the SARGing costs, unpacking clearly dominates in all cases. E.g., if 10% of the tuples qualify, then 10% of the cycles are spent in predicate evaluation with Data Blocks and only 5% for bit-packing, whereas 90% and 95% of the time is used for unpacking.

When a sparse set of tuples is selected by the SARGable predicate, we only benefit from this early selection iff individual tuples can be accessed and decompressed fast. On the other hand, if larger (dense) sets of tuples are selected, then early evaluation of predicates becomes increasingly pointless. In the above example (selecting 10% uniformly), extracting the selected tuples from a Data Block is more than 3× faster, while selection is 1.8× faster. As HyPer has to provide fast access to individual compressed records and also needs to deliver robust performance for scans with modest selectivities, we cannot use bit-packing in our Data Blocks.

We intentionally do not show results for *vertical bit-packing* as it trades faster SARGing for even higher decompression costs and is therefore even less suitable for our high performance OLTP and OLAP database system. Nevertheless, the recent work of Li et al. [21] shows how SARGing on vertical bit-packed data can be significantly improved, which might offer new applications of vertical bit-packed data, such as using it as secondary index.

6. CONCLUSION

The goal of this work was to reduce the main-memory footprint in high performance hybrid OLTP & OLAP database systems without compromising high query **and** transactional performance. To achieve this goal, we developed a novel compressed columnar storage format for hybrid database systems, termed *Data Blocks*. This compressed data format was further improved by light-weight intra-block indexing, called *Positional SMA*, for improved scan performance and SIMD-optimized predicate evaluation. The fine-grained adaptivity of Data Blocks necessitated the integration of JIT-compiled query execution with vectorized scans in order to achieve highest possible performance for compilation as well as query execution. This paper serves as a blueprint for integrating all these innovative techniques into a full-fledged hybrid OLTP & OLAP system. Further optimization potentials are outlined in Appendix E.

7. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, 2006.

[2] Actian Corporation. *Actian Analytics Database – Vector Edition 3.5*, 2014.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[4] R. Barber, G. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *ICDE*, 2015.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[6] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, et al. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *PVLDB*, 1(2), 2008.

[7] R. L. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, page 8, 2011.

[8] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, et al. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. *PVLDB*, 7(14), 2014.

[9] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB*, 6(14), 2013.

[10] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[11] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB*, 7(11), 2014.

[12] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record*, 40(4), 2011.

[13] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *SIGMOD*, 2015.

[14] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. *PVLDB*, 7(12), 2014.

[15] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *PVLDB*, 5(11), 2012.

[16] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *ICDE*, 2011.

[17] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, et al. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1), 2011.

[18] P. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, et al. SQL Server Column Store Indexes. In *SIGMOD*, 2011.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[20] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, 2014.

[21] Y. Li, C. Chasseur, and J. M. Patel. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *SIGMOD*, 2015.

[22] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[23] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, 1998.

[24] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14), 2013.

[25] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9), 2011.

[26] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*, 2015.

[27] O. Polychroniou and K. A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *DaMoN*, 2015.

[28] M. Pöss and D. Potapov. Data Compression in Oracle. In *VLDB*, 2003.

[29] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro Adaptivity in Vectorwise. In *SIGMOD*, 2013.

[30] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, 6(11), 2013.

[31] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. Compilation in Query Execution. In *DaMoN*, 2011.

[32] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, et al. The More the Merrier: Efficient Multi-source Graph Traversal. *PVLDB*, 8(4), 2014.

[33] S. D. Viglas. Data Management in Non-Volatile Memory. In *SIGMOD*, 2015.

[34] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *DEBU*, 37(1), 2014.

[35] 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), 2009.

[36] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

[37] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, 2002.

[38] M. Zukowski and P. A. Boncz. Vectorwise: Beyond Column Stores. *DEBU*, 35(1), 2012.

[39] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.

[40] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*, 2012.

APPENDIX

A. IMPACT OF VECTOR SIZE ON QUERY PERFORMANCE

Figure 13 shows the runtime of the TPC-H 100 benchmark with varying vector sizes. Query runtimes slightly increase for small vector sizes due to interpretation overheads (e.g., function calls). On the other hand, when the records stored in a vector exceed the cache size, query performance decreases as records are evicted to slower main memory before they are pushed into the JIT-compiled query pipeline.

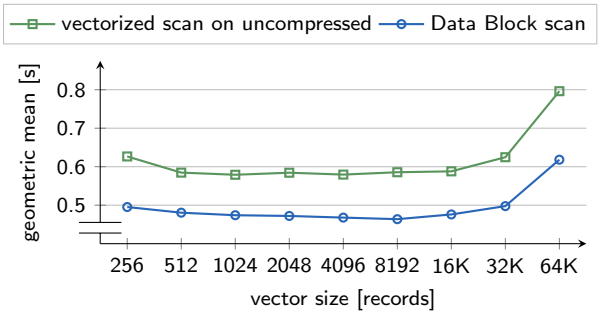


Figure 13: Geometric mean of TPC-H 100 query runtimes depending on vector size.

B. PSMA IMPLEMENTATION

The following listing sketches the code for constructing and probing the PSMAs. Before any PSMA access, the values are first converted into deltas relative to the smallest value in the current data block (*min*). The build scans the data block once and remembers the first and last occurrence of a value in the respective slot.

```
// Compute the PSMA slot for a given value
uint32_t getPSMASlot(T value,T min) {
  // d = delta
  uint64_t d=value-min;
  // r = remaining bytes (note: clz is undefined for 0)
  uint32_t r=d ? (7-(__builtin_clzll(d)>>3)) : 0;
  // m = most significant non-zero byte
  uint64_t m=(d>>(r<<3));
  // return the slot in PSMA array
  return m+(r<<8);
}

// Initialize all slots to empty ranges
for (auto& entry : psma)
  entry={0,0};

// Update ranges for all attribute values
for (uint32_t tid=0; tid!=values.size(); ++tid) {
  auto& entry=psma[getPSMASlot(values[tid],min)];
  if (entry.empty())
    entry={tid,tid+1};
  else
    entry.end=tid+1;
}
```

At query processing time, the potential range of tuples can now immediately be looked up in the PSMA:

```
// value = query constant of an equality predicate
auto scanRange=psma[getPSMASlot(value,min)];
```

C. SIMD IMPLEMENTATION OF FINDING INITIAL MATCHES

The following code listing shows the details of our find-matches implementation that makes use of the pre-computed table to map bit-masks to match positions. As the match table is limited to 256 entries, each entry can store 8 match positions. In most cases, we compare more elements in parallel which means multiple lookups are necessary. In the listing below, we evaluate a predicate on 32 8-bit integers at a time which results in 4 lookups.

Declarations:

```
// Vector of 8 32-bit integers.
typedef union {
 int32_t cell[8];
 __m256i reg256;
 __m128i reg128[2];
} vector8_int32;

using matchTableEntry = vector8_int32;

const matchTableEntry matchTable[256]{
 {{-256,-256,-256,-256,-256,-256,-256,-256}},
 {{1,-255,-255,-255,-255,-255,-255,-255}},
 {{257,-255,-255,-255,-255,-255,-255,-255}},
 // ...
 {{263,519,775,1031,1287,1543,1799,-249}},
 {{8,264,520,776,1032,1288,1544,1800}}
};
```

Find matches function:

```
if (reinterpret_cast<uintptr_t>(&column[from])%32) {
 // Process non-32-byte aligned elements sequentially
 ...
 // Recurse
} else {
 // Process 32-byte aligned elements (using SIMD/AVX2)
 const uint32_t simdWidth=32;
 const uint32_t numSimdIterations=(to-from)/simdWidth;
 const __m256i comparisonValueVec=set(comparisonValue);
 const __m256i vec16=_mm256_set1_epi32(16);
 uint32_t* writer=matches;
 for (uint32_t i=0;i!=numSimdIterations;i++) {
  uint32_t scanPos=from+(i*simdWidth);

  // Load and compare 32 values
  __m256i attributeVec=_mm256_load_si256(
      reinterpret_cast<__m256i*>(&column[scanPos]));
  __m256i selMask=cmp(attributeVec,comparisonValueVec);
  int bitMask=_mm256_movemask_epi8(selMask);

  // Lookup match positions and update positions vector
  auto& matchEntry0=matchTable[bitMask&0xFF];
  __m256i scanPosVec0=_mm256_set1_epi32(scanPos);
  _mm256_storeu_si256(
    reinterpret_cast<__m256i*>(writer),
    _mm256_add_epi32(scanPosVec0,
      _mm256_srai_epi32(matchEntry0.reg256,8)));
  writer+=static_cast<uint8_t>(matchEntry0.cell[0]);

  auto& matchEntry1=matchTable[(bitMask>>8)&0xFF];
  __m256i scanPosVec1=_mm256_set1_epi32(scanPos+8);
  _mm256_storeu_si256(
    reinterpret_cast<__m256i*>(writer),
    _mm256_add_epi32(scanPosVec1,
      _mm256_srai_epi32(matchEntry1.reg256,8)));
  writer+=static_cast<uint8_t>(matchEntry1.cell[0]);

  auto& matchEntry2=matchTable[(bitMask>>16)&0xFF];
  __m256i scanPosVec2=
      _mm256_add_epi32(scanPosVec0,vec16);
```

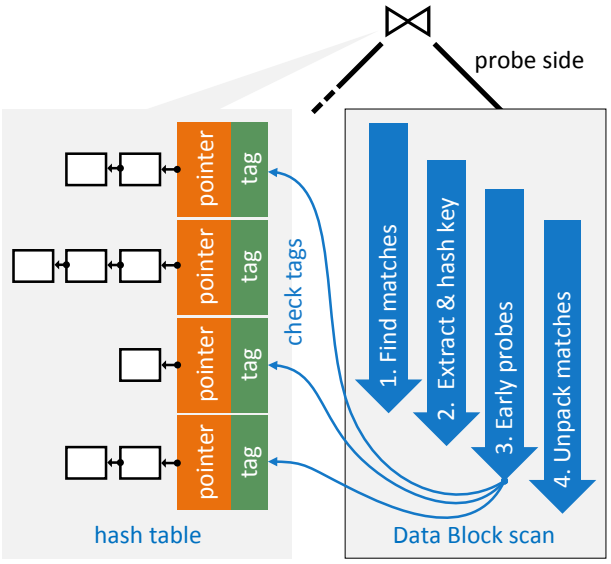


Figure 14: Early probing of vectors of matching keys from DataBlocks: pointer tags in the hash table are used to filter out keys that are not in one of the buckets (cf., tagged hash table pointers in [20]).

```
  _mm256_storeu_si256(
    reinterpret_cast<__m256i*>(writer),
    _mm256_add_epi32(scanPosVec2,
      _mm256_srai_epi32(matchEntry2.reg256,8)));
  writer+=static_cast<uint8_t>(matchEntry2.cell[0]);

  auto& matchEntry3=matchTable[(bitMask>>24)&0xFF];
  __m256i scanPosVec3=
      _mm256_add_epi32(scanPosVec1,vec16);
  _mm256_storeu_si256(
    reinterpret_cast<__m256i*>(writer),
    _mm256_add_epi32(scanPosVec3,
      _mm256_srai_epi32(matchEntry3.reg256,8)));
  writer+=static_cast<uint8_t>(matchEntry3.cell[0]);
 }
 // [...] Process remaining elements sequentially
 return writer-matches; // Number of matches
}
```

D. FLIGHT DATA SET QUERY

```
select
   uniquecarrier as carrier,
   avg(arrdelay) as avgdelay
from
   flights
where
   year between 1998 and 2008
   and dest = 'SFO'
group by
   uniquecarrier
order by
   avgdelay desc
```

E. FURTHER OPTIMIZATIONS

Vectorized scans that feed JIT-compiled query pipelines, as implemented in our hybrid OLTP and OLAP system, enable a large space of further optimization opportunities.

One research direction is to integrate more SIMD processing besides SARGable predicate evaluation in the vectorized scans. While SIMD processing is generally impossible in a tuple-at-a-time compiled loop produced by our JIT engine, vectorized scans will enable this possibility. One idea is to extend the vectorized scan with *eager aggregation* [36] functionality. This functionality would be able to calculate (arithmetic) expressions and evaluate aggregates only on the data inside one chunk. The resulting pre-aggregated data would then be re-aggregated by a true aggregation operator transforming the (non-holistic) aggregates in a way usually done for distributed, parallel, and indeed eager aggregation. This optimization mostly targets aggregates that just depend on a scan and are expected to have few groups, such as in TPC-H Q_1 and Q_6. In future work we intend to implement such early aggregation for vectorized scans on uncompressed chunks and compressed Data Blocks.

Another optimization opportunity is *early probing* of upstream hash-joins in the JIT-compiled query pipeline. When a vectorized scan of the probe side starts running in HyPer, the build side of the hash-joins in the JIT-compiled query pipeline has already been materialized. There are many situations in data warehousing workloads where such hash-joins can be very selective. For instance, this is the case when a fact table is used to probe a restricted dimension table. Scanning and decompressing all records of the fact table is a waste of CPU resources; hence, if the probe side is probed early inside the vectorized scan, touching attributes that would be eliminated later can be avoided. To enable such probing, systems like Vectorwise use bloom filters [29].

Probing a bloom filter is cheaper than performing a full hash lookup, for two reasons: First, a bloom filter is a much smaller memory object than a full hash table and thus typically fits in a lower-level CPU cache. Second, the JIT-compiled query pipeline contains the compiled hash lookup that performs the work in very few instructions; however, the loop in which it is placed also contains code for all subsequent operators in that query pipeline. The fact that this code is all part of the same code path means that this path potentially contains a large number of instructions and may thus be too complex to be executed efficiently. As such, even though modern out-of-order CPU cores can speculate deeply into a stretch of instructions if they hit stalls, it is improbable that the CPU is able to speculate through such a complex code path all the way up to the next loop iteration, i.e, up to the next input tuple. This means that in a JIT-compiled bloom filter, there is typically only one tuple at a time being looked up in the bloom filter and if this bloom filter is larger than the CPU cache, there will be only one outstanding memory load at any time. This is problematic because modern hardware is capable of handling multiple concurrent memory misses and limiting the number of outstanding loads available to the CPU core will make it impossible to use the available memory bandwidth. In contrast, vectorized bloom filter probing as part of a vectorized scan can be implemented as a simple loop that computes a boolean lookup result without any inter-tuple dependencies, and trivially generates the maximum number of parallel memory loads to saturate the memory bandwidth.

HyPer has a form of early probing, very similar to bloom filters, built into its hash table pointers (similar to tagged hash table pointers in [20]). This early probing is already used for tuple-at-a-time early probing in JIT-compiled pipelines. An implementation of this early probing for a vector

of keys inside the vectorized scan (see Figure 14 for an example of vectorized early probing of matching keys from DataBlocks), directly after evaluating the SARGable predicates, allowed us to gain significant performance benefits on TPC-H Q_3–Q_5, Q_7–Q_8, Q_{10}–Q_{14}, and Q_{21}–Q_{22} in preliminary experiments. Overall, the geometric mean of query runtime improved by 1.2×. However, performing the early test for *all* joins inside the scan also slowed down a significant number of other queries. This provides a third optimization opportunity enabled by vectorization, namely to make query processing *adaptive*. Vectorwise has proposed *Micro Adaptivity* [29] where different implementations of particular vectorized functions ("flavors") are tried at runtime and performance is monitored. Given that inside a query a vectorized function might still be called millions of times, the micro-adaptive expression evaluator can experiment with the different flavors and stick most of the time with the flavor that performs best at that point in time. Micro-adaptivity captures the choice of flavors automatically and makes query performance more robust. Using such adaptive algorithms in a tuple-at-a-time JIT-compiled query pipeline is not possible, since every binary decision opportunity in the query pipeline duplicates the amount of possible code paths, which again leads to high compilation times (cf., Section 4). In vectorized scans, it is possible to employ such adaptive behavior and we experimented using micro-adaptivity to guide the decision whether or not to use the early join test inside the vectorized scan which allowed us to avoid performance penalties.

F. TPC-H RESULTS

	Uncompressed					Compressed			
	JIT scan		Vectorized scan		+SARG	Data Block scan	+SARG/SMA	+PSMA	over JIT
Q1	0.388s	(45ms)	0.373s	(29ms)	0.539s	0.431s	0.477s	0.478s	0.81×
Q2	0.085s	(177ms)	0.097s	(89ms)	0.086s	0.092s	0.086s	0.086s	1.00×
Q3	0.731s	(64ms)	0.723s	(34ms)	0.812s	0.711s	0.634s	0.627s	1.17×
Q4	0.491s	(50ms)	0.508s	(27ms)	0.497s	0.502s	0.457s	0.454s	1.08×
Q5	0.655s	(120ms)	0.662s	(57ms)	0.645s	0.691s	0.658s	0.655s	1.00×
Q6	0.267s	(20ms)	0.180s	(11ms)	0.114s	0.188s	0.040s	0.040s	6.70×
Q7	0.600s	(124ms)	0.614s	(62ms)	0.659s	0.632s	0.557s	0.548s	1.09×
Q8	0.409s	(171ms)	0.420s	(78ms)	0.401s	0.505s	0.458s	0.460s	0.89×
Q9	2.429s	(121ms)	2.380s	(59ms)	2.357s	2.423s	2.439s	2.453s	0.99×
Q10	0.638s	(96ms)	0.633s	(50ms)	0.691s	0.614s	0.521s	0.512s	1.25×
Q11	0.094s	(114ms)	0.092s	(56ms)	0.092s	0.087s	0.082s	0.081s	1.16×
Q12	0.413s	(58ms)	0.447s	(32ms)	0.430s	0.381s	0.305s	0.305s	1.35×
Q13	6.695s	(45ms)	6.766s	(27ms)	6.786s	7.260s	7.132s	7.098s	0.94×
Q14	0.466s	(41ms)	0.410s	(22ms)	0.438s	0.213s	0.145s	0.140s	3.33×
Q15	0.441s	(48ms)	0.440s	(37ms)	0.434s	0.359s	0.278s	0.275s	1.60×
Q16	0.831s	(99ms)	0.836s	(55ms)	0.842s	0.662s	0.669s	0.664s	1.25×
Q17	0.427s	(74ms)	0.439s	(41ms)	0.436s	0.504s	0.490s	0.487s	0.88×
Q18	2.496s	(91ms)	2.418s	(49ms)	2.401s	2.379s	2.366s	2.394s	1.04×
Q19	1.061s	(70ms)	1.119s	(34ms)	1.125s	0.682s	0.528s	0.521s	2.04×
Q20	0.602s	(108ms)	0.596s	(54ms)	0.610s	0.577s	0.529s	0.530s	1.14×
Q21	1.223s	(129ms)	1.176s	(65ms)	1.166s	1.212s	1.142s	1.136s	1.08×
Q22	0.265s	(81ms)	0.321s	(48ms)	0.261s	0.391s	0.278s	0.277s	0.96×
Sum	21.708s	(1945ms)	21.649s	(1016ms)	21.822s	21.497s	20.271s	20.179s	
Geometric mean	0.586s	(78ms)	0.583s	(42ms)	0.577s	0.555s	0.466s	0.463s	1.27×

Table 4: Query runtimes and compilation times (in parentheses) of TPC-H queries on scale factor 100 with (i) JIT-compiled tuple-at-a-time scans on uncompressed data, (ii) vectorized scans on uncompressed data, (iii) vectorized scans on uncompressed data with SARG-able predicate evaluation (+SARG), (iv) vectorized compressed Data Block scans, (v) vectorized compressed Data Block scans with SARG-able predicate evaluation and SMAs (+SARG/SMA), and (vi) (v) with Positional SMA indexes (+PSMA).