Optimistically Compressed Hash Tables & Strings in the USSR

Tim Gubner
CWI
tim.gubner@cwi.nl

Viktor Leis
FSU Jena
viktor.leis@uni-jena.de

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

Modern query engines rely heavily on hash tables for query processing. Overall query performance and memory footprint is often determined by how hash tables and the tuples within them are represented. In this work, we propose three complementary techniques to improve this representation: *Domain-Guided Prefix Suppression* bit-packs keys and values tightly to reduce hash table record width. *Optimistic Splitting* decomposes values (and operations on them) into (operations on) frequently- and infrequently-accessed value slices. By removing the infrequently-accessed value slices from the hash table record, it improves cache locality. The *Unique Strings Self-aligned Region* (USSR) accelerates handling frequently occurring strings, which are widespread in real-world data sets, by creating an on-the-fly dictionary of the most frequent strings. This allows executing many string operations with integer logic and reduces memory pressure.

We integrated these techniques into Vectorwise. On the TPC-H benchmark, our approach reduces peak memory consumption by 2–4× and improves performance by up to 1.5×. On a real-world BI workload, we measured a 2× improvement in performance and in micro-benchmarks we observed speedups of up to 25×.

1. INTRODUCTION

In modern query engines, many important operators like join and group-by are based on in-memory hash tables. Hash joins, for example, are usually implemented by materializing the whole inner (build) relation into a hash table. Hash tables are therefore often large and determine the peak memory consumption of a query. Since hash table sizes often exceed the capacity of the CPU cache, memory latency or bandwidth become the performance bottleneck in query processing. Due to the complex cache-hierarchy of modern CPUs, the access time to a random tuple varies by orders of magnitude depending on the size of the working set. This means that shrinking hash tables does not only reduce memory consumption but also has the potential of improving query performance through better cache utilization [4,5,21].

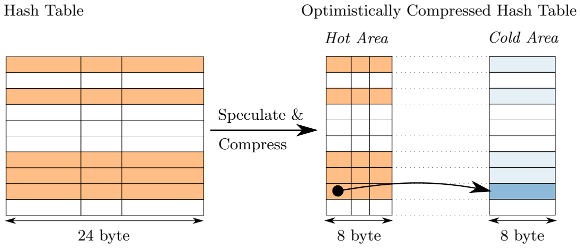


Figure 1: Optimistically Compressed Hash Table, which is split into a thin hot area and a cold area for exceptions

To decrease a hash table's hunger for memory and, consequently, increase cache efficiency, one can combine two orthogonal approaches: to increase the fill factor and to reduce the bucket/row size. Several hash table designs like Robin Hood Hashing [9], Cuckoo Hashing [19], and the Concise Hash Table [5] have been proposed for achieving high fill factors, while still providing good lookup performance. Here, we investigate how the size of each row can be reduced—a topic that, despite its obvious importance for query processing, has not received as much attention.

While heavyweight compression schemes tend to result in larger space savings, they also have a high CPU overhead, which often cannot be amortized by improved cache locality. Therefore, we propose a lightweight compression technique called *Domain-Guided Prefix Suppression*. It saves space by using domain information to re-pack multiple columns in-flight in the query pipeline into much fewer machine words. For each attribute, this requires only a handful of simple bit-wise operations, which are easily expressible in SIMD instructions, resulting in an extremely low per-tuple cost (sub-cycle) for packing and subsequent unpacking operations.

Rather than saving space, the second technique, called *Optimistic Splitting*, aims at improving cache locality. As Figure 1 illustrates, it splits the hash table into a hot (frequently accessed) and a cold (infrequently accessed) area containing exceptions. These exceptions may, for example, be overflow bits in aggregations, or pointers to string data. By separating hot from cold information, Optimistic Splitting improves cache utilization even further.

An often ignored, yet costly part of query processing is string handling. String data is very common in many real-world applications [18, 24]. In comparison with integers, strings occupy more space, are *much slower* to process, and are less amenable to acceleration using SIMD. To speed up

©IEEE 2020. This is a minor revision of the paper entitled "Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR" published in the Proceedings of the 2020 ICDE Conference, 2375-026X/20. DOI: 10.1109/ICDE48307.2020.00033

string processing, we therefore propose the *Unique Strings Self-aligned Region (USSR)*, an efficient dynamic string dictionary for frequently occurring strings. In contrast to conventional per-column dictionaries used in storage, the USSR is created anew for each query by inserting frequently occurring strings at query runtime. The USSR, which has a fixed size of 768 kB (ensuring its cache residency), speeds up common string operations like hashing and equality checks.

While each of the proposed techniques may appear simple in isolation, they are highly complementary. For example, using all three techniques, strings from a low-cardinality domain (few distinct strings) can be represented using only a small number of bits. Furthermore, our techniques remap operations on wide columns into operations on multiple thin columns using a few extra primitive functions (such as pack and unpack). As such, they can easily be integrated into existing systems and do not require extensive modifications of query processing algorithms or hash table implementations.

Rather than merely implementing our approach as a prototype, we fully integrated it into Vectorwise which originated from the MonetDB/X100 project [8]. Describing how to integrate the three techniques into industrial-strength database systems is a key contribution of the paper.

Based on Vectorwise, we performed an extensive experimental evaluation using TPC-H, micro-benchmarks, and real-world workloads. On TPC-H, we reduce peak memory consumption by 2–4× and improve performance by up to 1.5×. On a string-heavy real-world BI workload, we measured a 2.2× improvement in performance, and in micro-benchmarks we even observed speedups of up to 25×.

2. DOMAIN-GUIDED PREFIX SUPPRESSION

Domain-Guided Prefix Suppression reduces memory consumption by eliminating the unnecessary prefix bits of each attribute. This enables us to cheaply compress rows without affecting the implementation of the hash table itself, which makes it easy to integrate our technique into existing database systems. In particular, while our system (Vectorwise) uses a single-table hash join, Domain-Guided Prefix Suppression would also be applicable (and highly beneficial) for systems that use partitioning joins [4, 22]. Domain-Guided Prefix Suppression also allows comparisons of compressed values without requiring decompression.

2.1 Domain Derivation

A column in-flight in a query plan can originate directly from a table scan or from a computation. If a value originates from a table scan, we determine its domain based on the scanned blocks. For each block we utilize per-column minimum and maximum information (called ZoneMaps or Min/Max indices). This information is typically *not* stored inside the block itself as this would require scanning the block (potentially fetching it from disk) before this information can be extracted. Instead, the meta-data is stored "out-of-band" (e.g. in a row-group header, file footer or inside the catalog). By knowing the range of blocks that will be scanned, the domain can be calculated by computing the total minimum/maximum over the range.

On the other hand, if a value stems from a computation, the domain minimum and maximum can be derived bottom up according to the functions used, based on the minimum/maximum bounds on its inputs under the assumption of the worst case. Consider, for example, the addition of two integers $a \in [a_{\min}, a_{\max}]$ and $b \in [b_{\min}, b_{\max}]$ resulting in $r \in [r_{\min}, r_{\max}]$. To calculate $r_{\min}$ and $r_{\max}$ we have to assume the worst-case that means the smallest (r_{min}), respective highest ($r_{\max}$), the result of the addition. In case of an addition this boils down to $r_{\min} = a_{\min} + b_{\min}$ and $r_{\max} = a_{\max} + b_{\max}$.

Depending on these domain bounds, an addition of two 32-bit integer expressions could still fit in a 32-bit result, or less likely, would have to be extended to 64-bit. This analysis of minimum/maximum bounds often can allow implementations to ignore overflow handling, as the choice of data types *prevents* overflow, rather than having to *check* for it. For aggregation functions such as `SUM`, overflow avoidance is more challenging. In Section 3, we discuss Optimistic Splitting, which allows to do most calculations on small data types, also reducing the cache footprint of aggregates.

2.2 Prefix Suppression

Using the derived domain bounds, we can represent values compactly without losing any information by dropping the common prefix bits. To further reduce the number of bits and enable the compression of negative values, we first subtract the domain minimum from each value. Consequently, each bit-packed value is a positive offset to the domain minimum. We also pack multiple columns together such that the packed result fits a machine word. This is done by concatenating all compressed bit-strings and (if necessary) chunk the result into multiple machine words. Each chunk of the result constitutes a compressed column which can be stored just like a regular uncompressed column.

2.3 Compression and Decompression

Like many modern column-oriented systems, Vectorwise is based on vectorized *primitives* that process cache-resident vectors (=arrays of single column values). These primitives process items from multiple inputs in a data-parallel (SIMD-friendly) fashion in a tight loop. Consequently, modern compilers automatically translate such code into SIMD instructions for the specified target architecture (e.g. AVX-512). In our vectorized hash table implementation, `pack` primitives compress and "glue" multiple inputs together to produce one intermediate result. Later, this intermediate result is then stored inside the hash table. With all the inputs and the one output being cache-resident vectors, the compression itself happens in-cache. For bit-packing, our `pack` primitives look similar to the following pseudo-code:

```
void pack2_i32_i16_to_i32(i32* res, int n,
    i32* col1, i32 b1, int ishl1, int oshr1, i32 m1,
    i16* col2, i16 b2, int ishl2, int oshr2, i32 m2) {
  for (int i=0; i<n; i++) {
    // Select portion of input and cast to result's type
    i32 c1 = ((col1[i] - b1) >> ishl1) & m1;
    i32 c2 = ((col2[i] - b2) >> ishl2) & m2;
    // Move to output positions
    res[i] = (c1 << oshr1) | (c2 << oshr2);
} }
```

After bit-packing, we scatter the intermediate results into its final positions in the hash table. For improved cache locality, the hash table is stored in row-wise layout (NSM) [27].

When decompressing values, we fetch up to 4 columns from the hash table and directly decompress them. For decompressing a vector of `n` packed 16-bit integers from 32-bit and 16-bit integers at positions idx in the hash table, this leads to the following pseudocode (2-column example):

```
void unpack2_i32_i16_to_i16(i16* res, int n, int* idx, i16 b,
    i32* col1, int ishr1, int oshl1, i16 m1, int s1,
    i16* col2, int ishr2, int oshl2, i16 m2, int s2) {
  for (int i=0; i<n; i++) {
    // DSM (columnar) position -> NSM (row) position
    int idx1 = idx[i] * s1;
    int idx2 = idx[i] * s2;
    // Extract relevant bits from NSM record
    i16 c1 = (col1[idx1] >> ishr1) & m1;
    i16 c2 = (col2[idx2] >> ishr2) & m2;
    // Stitch back together
    res[i] = (c1 << oshl1) | (c2 << oshl2) + b;
} }
```

Notably compression and decompression operate in a non-intuitive fashion: Both process m inputs and produce *one* output. This particular approach has two advantages: (a) In contrast to approaches with multiple outputs, it allows decompressing specific columns without enforcing decompression of neighboring cells. This allows an efficient mix of key checks on compressed data together with key checks on bit-packed non-integer data, most notably strings. (b) We concatenate bit-strings directly in registers, as opposed to approaches that partially compress/decompress which require multiple rounds of reading/writing from/to output vectors to concatenate partial output vectors into the final output.

2.4 Operating on Compressed Keys

Domain-Guided Prefix Suppression also allows comparing compressed values themselves (without having to decompress). Assume key value A is stored in the hash table and probe key B is compared to A. Normally one would just fetch the key A from the table and then compare it to B. In combination with compression, fetching A also requires decompressing A. We argue it is better to first bring B into the same representation as A, i.e., compressing B, and then directly compare the compressed values. This is especially true if keys A and B consist of multiple columns. For instance, a group-by on two columns can often be mapped into single-integer compressed key, reducing computational work (e.g. perform a single comparison, using fewer branches).

3. OPTIMISTIC SPLITTING

The goal of *Optimistic Splitting* is to exploit skewed access frequencies by separating the common case from exceptional situations. We physically split the hash table into two areas: The frequently-accessed *hot area* and the *cold area*, which is accessed rarely. This approach does not necessarily save space. However it shrinks the active working set, leading to lower memory access cost. Also, it converts operations on the final, widest, data type into operations on a potentially smaller data type. Specifically, if 128-bit operations become 64-bit or 32-bit; this can speed up computation noticeably. As we show in the following, Optimistic Splitting is especially important for data that is hard to compress such as aggregates and strings.

Aggregates are hard to compress with Domain-Guided Prefix Suppression as it is not possible to obtain tight bounds for aggregation results (for example SUMs). The reason is that one has to be pessimistic when deriving domain bounds to prevent integer overflows: Assuming a SUM of at most 2^{48} integers from, say, a 18-bit domain, would overflow 64-bit and thus need a 128-bit aggregate. If this type is used for the aggregate, on each addition in the sum this large 128-bit integer will be read, updated, and written back.

Table 1: Optimistic Aggregates

Aggregate	Common case	Exception
SUM	Small integer	Overflow counter
MIN	Small upper bound	Minimum
MAX	Small lower bound	Maximum
COUNT	Similar to SUM	
AVG	Rewritten into $\frac{\texttt{SUM}}{\texttt{COUNT}}$	

Using a 64-bit integer for the aggregate, on the other hand, would (a) reduce reads and writes by a factor 2 and (b) provide faster updates. Without sacrificing correctness, Optimistic Splitting allows one to do just that in the common case (i.e., when no overflow occurs): The 128-bit aggregate result is split into a frequently-accessed 64-bit sum and another, rarely-accessed 64-bit overflow/carry field, which is stored separately. In pseudocode, this looks as follows:

```
void opsum(u64* common, u64* except, int group, i32 value) {
  common[group] += value; // 64-bit unsigned addition
  // Overflow check
  bool overflow = common[group] < (u64)value;
  bool positive = value >= 0;
  if (!(overflow ^ positive)) { // Rare: handle overflow
    if (positive) except[group]++;
    else          except[group]--;
} }
```

Note that this is a generic implementation that handles positive as well as negative values. In combination with domain bounds (Min/Max information) it is possible to prove the absence of negative or positive values which leads to simplified logic and improved performance.

Similarly, it is possible to shrink the working set of other aggregates. Table 1 illustrates how to exploit Optimistic Splitting for these. We use the *associativity* of aggregates to provide a fast path for large aggregates and a smaller working set. MIN can be implemented using an upper bound (s) inside the hash table and storing the full minimum e as an exception ($s \geq e$). When calculating the aggregate, we first check against s and discard values that cannot become the new minimum. For the remaining values, we check against the full minimum and potentially update the full minimum e as well as the upper bound s. Similar is the implementation of MAX whereas the other aggregate functions, COUNT and AVG, can be implemented similar to SUM. However, in case of COUNT we can more aggressively reduce the common case to a 16-bit integer and after $2^{16} - 1$ iterations update both, the small optimistic counter as well as the exception.

Other Applications. Optimistic Splitting is a very general idea that we believe can be applied in many different use cases. It only requires that the entries of a hash table have different access patterns, and can be decomposed.

4. USSR: A DYNAMIC DICTIONARY

Strings are prevalent in many real-world data sets [13,18,24] and present additional challenges for query performance. In contrast to integers, any individual string generally does not fit into a single CPU register and requires multiple instructions for each primitive operation (e.g. comparison). Strings are also often larger than integers, which negatively affects memory footprint and cache locality. Furthermore, neither Domain-Guided Prefix Suppression nor Optimistic Splitting can directly be applied to strings. This section presents a dynamic data structure called *Unique Strings*

Self-aligned Region, which saves memory and enables processing strings at almost the same speed as integers.

4.1 The Problems with Global Dictionaries

To improve the performance of strings, some main-memory database systems—most notably SAP HANA [11]—represent strings using per-column dictionaries where codes respect the value order. Using these dictionaries, string comparisons and hashing operations can be directly performed on the dictionary keys, which are fixed-size integers, rather than variable-length strings. Unfortunately, global dictionaries have significant downsides, which have precluded their general adoption. First, because random access to the dictionaries is common, the dictionaries must fully reside in main memory. For systems that must manage data sets larger than main memory (e.g. analytical column stores), this is a major problem. Also, systems that support parallel and distributed execution, including those designed or optimized for the cloud, face the problem that bulk loading or updating tables in parallel would require continuous synchronization in order to maintain a consistent global dictionary. Another downside is that dictionaries incur significant overhead for inserts, updates, and deletes—in effect they are a mandatory secondary index on every string column. If, for instance, new values appear, extending the dictionary such that one additional bit is needed to represent a code, updates will no longer fit in previously encoded data. Deletes of no longer used strings leave holes in the code space that need to be garbage collected and inserts in sorted dictionaries often require re-coding (periodically rewriting all encoded columns).

Given these problems with global dictionaries, most database systems therefore limit themselves to per-block dictionaries (e.g. one dictionary for every 10,000 strings). With this approach, dictionaries are a local feature, mainly used for compression rather than a global data structure. Per-block dictionaries are often almost as space-effective as per-column dictionaries without sharing their in-memory limitations and update overheads. For query processing, however, the advantage of per-block dictionaries is limited. While some systems evaluate pushed-down selections directly on the dictionary [14], all other operations require decompression and therefore do not benefit from the dictionary. The reason is that the dictionary is only available to the table scan operators. Materializing operators like hash join and group-by, therefore, typically allocate heap memory on the heap for every string. Needless to say, this is very inefficient, yet dealing with strings is only a sparsely researched topic.

4.2 Unique Strings Self-aligned Region (USSR)

The USSR is a query-wide data structure that contains the common strings of a particular query. In contrast to the heap, all strings within the USSR are known to be unique, which enables fast operations on these strings. To make it cache resident and efficient, the USSR has a limited size. Once it is full, strings need to be allocated on the heap as usual. By removing duplicates in this opportunistic fashion, the USSR reduces the number of heap allocations and therefore minimizes peak memory consumption.

By default both, heap-backed and USSR-backed, strings are represented as normal pointers, which means that query engine operators can treat all strings uniformly without any code modifications. This allows to retro-fit this idea easily into already existing engines. However, by exploiting the dictionary-like nature and artful implementation of the USSR, the following additional optimizations become possible for USSR-based strings: (a) String comparisons are almost as fast as integer comparisons. (b) Hashes are precalculated and stored within the USSR, speeding up hash-based operators like join and group-by. (c) Since the size of the USSR is limited, frequent strings can also be represented using small integer offsets, which can be exploited e.g. in Optimistic Splitting.

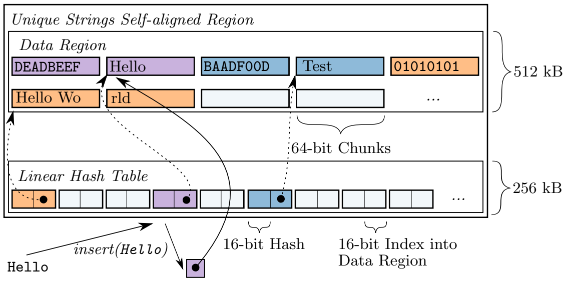


Figure 2: USSR data structure details

To summarize, the USSR is a lightweight, dynamic, and opportunistic string dictionary. It does not require changes to the storage level, but is implemented in the query processor, and speeds up queries with low to medium string cardinalities, which is where global string dictionaries excel.

4.3 Data Structure Details

Our USSR implementation limits its capacity to 768 kB: it consists of a *hash table* (256 kB) and a *data region* (512 kB). Figure 2 serves as an illustration of the USSR.

The 512 kB data region starts at a *self-aligned* memory address (i.e., the pointer has 0s in its lowest 19 bits). If one allocates 1 MB of data, there is always a self-aligned address in its first half for the data region; and there is always either 256 kB space before or after the data region for the hash table. The self-aligned memory address guarantees that all pointers inside the data region start with the same 45-bit prefix. This allows to very efficiently test whether a string pointer points inside the USSR (by applying a mask).

The data region stores the string data and materializes the string's hash value just before it. These numbers are stored aligned, so the data region effectively consists of 64k slots of 8 bytes where a string can start. Given that each string takes at least two slots (one for the hash and one for the string) the USSR can contain maximally 32k strings.

When inserting a string, the USSR needs to check whether that string is already stored, and if so, return its address rather than insert a new string. To do this in low $O(1)$, there is a fast linear probing hash table, consisting of 64k 4-byte buckets. Each bucket consists of a 16-bit hash extract and a 16-bit *slot number* that points into the data region to the start of the string. The lowest 16-bits of the string hash are used for locating the bucket, and the next 16-bits are the extract used to quickly identify collisions. The load factor is always below 50% (64k buckets for at most 32k strings).

4.4 Insertion

The purpose of the USSR is to accelerate operations on frequent strings. In the extreme, all strings could be part of the USSR. However, due to its limited size, the USSR can only fit a sample. The sampling happens during insertion into the data structure. Failure during insertion might hap-

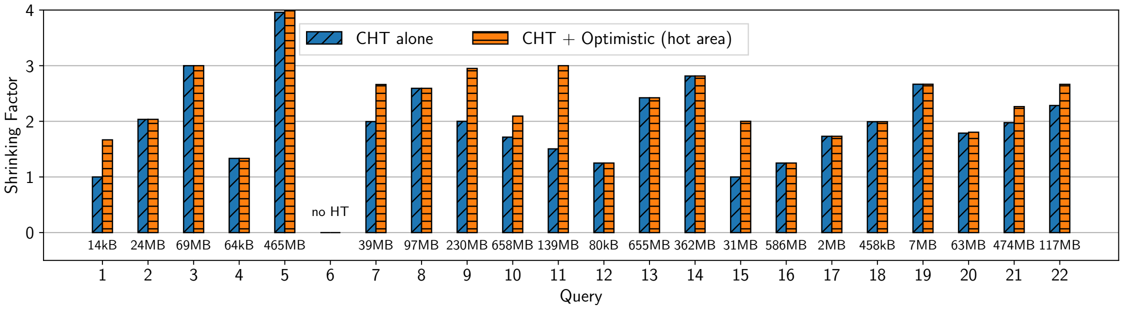


Figure 3: Reduction in hash table memory footprint over TPC-H with baseline hash table memory footprint (below the bar)

pen because (a) the string is rejected based on our sampling strategy or (b) a probing sequence of longer than 3 in the linear hash table is detected (due to the low load factor, this is highly infrequent, yet keeps negative lookups fast).

Our sampling strategy gives priority to *string constants* that occur in the query text; these are inserted first. After that, scans will insert strings until the USSR is full. We argue that the fact that a string column is dictionary-compressed, indicates that strings stem from a domain with a small cardinality. Therefore, these strings are good candidates for insertion into the USSR.

Vectorwise stores and buffers data in compressed form and decompresses column slices on the fly in the table scan operator. When reading a new dictionary-compressed block, the scan needs to set up an in-memory array with string pointers. Strings are represented as pointers in-flight in a query and decompression means looking up dictionary codes into this array. Rather than pointing into the dictionary inside the buffered block, when setting up this array, the scan inserts all dictionary strings into the USSR, so (most of) these pointers will point into the USSR instead. Insertion may fail, in which case the pointers still point into the block.

The sampling strategy further tries to optimize usage of the limited data region, by failing inserts of long strings that occupy $> \min(F, \max(2, \lfloor \frac{F}{64} \rfloor))$ 8-byte slots, where F is the free space in the data region (in slots). The idea is that it is better to accept more small strings than a few large strings, in case space fills up.

4.5 Accelerating Hashing & Comparisons

The USSR can be used to speed up hash computations. After testing whether a given string resides in the USSR using a bit-wise `and` operation, one can directly access the pre-computed hash value, which physically precedes the string:

```
inline uint64_t hash(char* s) {
  if (((uintptr_t)s & USSR_MASK) != ussr_prefix)
    return strhash(s); // compute hash
  return ((uint64_t*) (s))[-1]; // exploit pre-computed hash
}
```

The USSR also speeds up string comparisons when both compared strings reside in it. We exploit the fact that all strings within the USSR are unique. Hence, if the pointers are equal, the strings themselves are:

```
inline bool equal(char* s, char* t) {
  if ((((uintptr_t)s & USSR_MASK) != ussr_prefix) |
        (((uintptr_t)t & USSR_MASK) != ussr_prefix))
    return strcmp(s, t)==0; // regular string comparison
  return s==t; // in the USSR pointer equality is enough
}
```

4.6 Optimistic Splitting & the USSR

Optimistic Splitting and the USSR are complementary. The idea is to store USSR-backed strings, as small integers, compactly in the hot area and heap-backed strings in the cold area. Specifically, rather than storing string pointers in the hot area, we store slot numbers, pointing into the USSR. As mentioned earlier, these slot numbers are limited to 2^{16}, so they can be represented as unsigned 16-bit integers.

During packing, we represent exceptions using the invalid slot number 0 in the hot area of the hash table, and store the full 64-bit pointer in the exception area. Whenever a string needs to be unpacked, we first access the hot area and unpack the slot number. For non-zero slot numbers we can directly reconstruct the pointer of the string (`base address of USSR data region + slot*8`). However, we can further accelerate equality comparisons on strings by first comparing the slot numbers and, only if they are 0, comparing the full strings. A USSR encoded string `p` can be translated into a slot number quickly using `(p >> 3) & 65535`.

5. EXPERIMENTAL EVALUATION

In this section, we provide an experimental evaluation of our contributions to show that our techniques improve performance as well as memory footprint.

For this evaluation, we integrated Domain-Guided Prefix Suppression, Optimistic Splitting, and the USSR into Vectorwise. Besides generating all necessary function kernels, we had to extend the domain derivation mechanism and implement our greedy packing algorithm. In addition, we modified the existing hash table implementation, extended the hash join operator to take advantage of compressed key and payload columns, as well as the hash aggregation (group-by) operator to support Optimistic Aggregates.

We first evaluate the end-to-end performance on the TPC-H benchmark. We then present a high-level comparison on a real BI workload Afterwards we move to micro-benchmarks, analyze and discuss the impact of the USSR on string-intensive queries. Then we evaluate the hash probe performance over varying hash table sizes and the influence of different domains on hash table performance.

All experiments were performed on a dual-socket Intel Xeon Gold 6126 with 12 physical cores, 19.25 MB L3 cache each and is equipped with 384 GB of main memory. All results stem from hot runs using single-threaded execution.

5.1 TPC-H Benchmark

We evaluated the impact of Domain-Guided Prefix Suppression, Optimistic Splitting and the USSR on the widely used TPC-H benchmark with scale factor 100. We executed

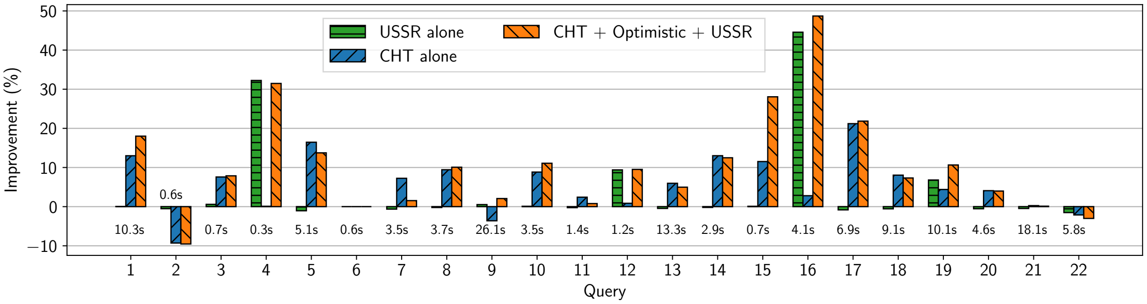


Figure 4: Improvement over TPC-H power run with baseline times (under the bar)

all 22 queries on our modified Vectorwise with and without our optimizations. We measured hash table memory footprint, as well as query response time.

Memory Footprint: In Vectorwise the memory consumption of many queries, particularly the TPC-H queries, is dominated by the size of hash tables. Therefore, during the TPC-H power run, we measured hash table sizes. Figure 3 shows the compression ratios we measured.

Domain-Guided Prefix Suppression (*CHT alone*), without Optimistic Splitting and USSR, was able to reduce hash table size by up to 4×. However, due to certain hurdles the compression ratio is often limited to 2×:

(a) Aggregates are not compressible without Optimistic Splitting. (b) Without the USSR, each string has to be a 64-bit pointer into a string heap. On recent hardware, this requires storing at least 48 bits with Domain-Guided Prefix Suppression. (c) As CHT does not make sense for CPU cache-resident hash tables, we do not enable it if the hash table is small, based on optimizer estimates. The impact of (a) and (b) on the active working set will be reduced using Optimistic Splitting and the USSR.

Optimistic Splitting aims at improving performance through more efficient cache utilization by separating the hash table into a thin frequently-accessed table (hot area) and a rarely accessed table (cold area). In combination with the USSR we measured a 2–4× smaller hot area (*CHT + Optimistic (hot area)*) in many TPC-H queries.

However, Optimistic Splitting in fact *increases* (rather than reduces) the memory consumption as it introduces additional data. For example, splitting a 128-bit `SUM` aggregate will introduce an additional aggregate with a smaller size but the full 128-bit aggregate will still reside in cold area.

Query Performance: To demonstrate the performance benefits of the USSR, Domain-Guided Prefix Suppression and Optimistic Splitting, we visualize the query response times of all 22 TPC-H queries in Figure 4. We split our analysis into three stages. First, we evaluate the impact achieved by only using the USSR. Then we discuss the effects of only using Domain-Guided Prefix Suppression. Finally, we discuss the influence of the combination of all three techniques.

The idea of the USSR is to boost operations on frequent strings. However, TPC-H is not an extremely string-intensive benchmark. Nonetheless, by using the Unique Strings Self-aligned Region (*USSR alone*) three queries (Q4, Q12 and Q16) showed significant performance gains. All three benefit from faster string hashing and equality comparisons provided by the USSR and improve by up to 45%.

Apart from the string-specific USSR, Domain-Guided Prefix Suppression aims at shrinking hash tables and providing operations on compressed data. Domain-Guided Prefix Suppression accelerates most queries (*CHT alone*) by up to 30%. In most queries we noticed an improvement of at least 10%. This is caused by the more efficient expression evaluation that smaller data types provide and the more cache-efficient hash table that allows equality comparisons directly on compressed keys. Notably, the regression in Q2 was caused by type casting overhead due to opportunistic shrinking of data types. We highlight that the purpose of Domain-Guided Prefix Suppression is mostly to reduce the memory footprint and not necessarily to speedup query evaluation.

When combining all three techniques (Domain-Guided Prefix Suppression, USSR and Optimistic Splitting) we measured gains up to 40% (*CHT + Optimistic + USSR*). We measured additional improvements from 5%, in Q1, up to 10%, in Q15. Both queries benefited from the Optimistic `SUM` aggregate which boosted the aggregate computation.

5.2 Public BI Benchmark

It has been noted that synthetic benchmarks like TPC-H do not capture all relevant aspects of real workloads [7, 10]. Recently, a workload study was published [24] based on the Tableau Public [1] Business Intelligence (BI) free cloud service. It analyzes its workbooks (data and queries generated by the Tableau BI tool) and specifically notes that users make extensive use of string data types (i.e. strings are by far the most common data type; used for 49% of all values). Not only is text data prevalent in these workbooks, but it is also observed that date columns, numeric and decimal columns are often stored as strings; arguably sub-optimally, but often this is related to data cleaning issues. Regrettably, this study did not publish the data and queries as an open benchmark, also upon our request to Tableau. Inspired by this work, we manually downloaded the 48 biggest Tableau Public workbooks (400 GB data) and extracted the SQL statements from its query log. This workload is now available in open-source as the *Public BI Benchmark* [2]. As a representative example, we focus on one of its workbooks:

CommonGovernment. We extracted all 43 queries and all 13 tables. Each table contains around 8 GiB of data in CSV format. Unlike TPC-H, each table contains many string columns and columns that contain `NULL` values are common. We executed each query sequentially and Table 2 shows the measured effects on the runtime.

The workbook *CommonGovernment* is string-intensive: using only the USSR, we measured a speedup of up to ≈ 2× (55% improvement). These speedups are caused by (a) many strings residing in the USSR, because they originate from a small domain of unique strings, and (b) many strings

Table 2: Speedup and USSR statistics for workbook *CommonGovernment*

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Speedup	**2.1**	**1.4**	**2.2**	**1.4**	**1.3**	**1.0**	**1.2**	**1.0**	**1.5**	**1.8**	**1.1**	**2.2**	**2.2**	**1.8**	**1.5**	**2.1**	**1.4**	**1.1**	**1.4**	**1.1**
USSR Size (kB)	1.8	0.5	2.0	0.3	66.1	512.0	83.2	512.0	12.7	7.2	112.4	1.9	1.8	7.2	1.8	2.0	1.8	110.3	0.3	512.0
Rejection Ratio (%)	0.0	0.0	0.0	0.0	0.0	18.3	0.0	32.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	21.1
#Rejected	0	0	0	0	0	37627	0	30204	0	0	0	0	0	0	0	0	0	17	0	13742
#Candidates	1312	2720	1056	1504	73200	205440	77296	92208	656	6768	110704	1072	1232	6752	3792	1360	3808	99744	1728	65222
#Strings in USSR	46	16	49	11	2251	12227	2835	12218	252	165	3041	48	45	167	51	49	51	2990	11	21343
Average String Length	23	3	20	5	18	26	19	29	21	25	23	22	22	25	18	22	19	23	5	11
Baseline Runtime (s)	0.15	0.27	0.13	0.17	0.37	3.48	0.39	0.54	0.18	0.16	0.29	0.14	0.14	0.16	0.18	0.14	0.17	0.27	0.18	0.51
Baseline HT size (MB)	0.05	0.09	0.05	0.09	0.14	82.11	0.14	9.12	0.07	0.05	0.11	0.05	0.05	0.05	0.05	0.05	0.05	0.11	0.09	8.11

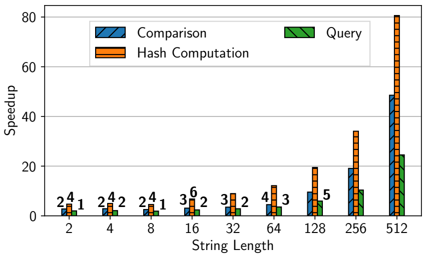


Figure 5: Group-By on string keys: Speedup vs. length

are long enough to significantly impact string operations to cause a speedup of the whole query.

Q6, Q8 and Q20 show no significant benefits from the USSR, mainly because the string columns have a large unified dictionary (that does not even fit fully in the USSR). While dictionary-coded decompression in Vectorwise has a sub-cycle per/tuple cost, the effort of setting up the dictionary array when the scan moves to a new disk block increases, when the per-block dictionary size increases. With the USSR, this setting-up effort becomes significantly higher as all dictionary strings must be looked up in the USSR linear hash table. Also, with larger dictionaries per block, each dictionary string has a lower repetition count during execution; so the amortization of the setting-up investment by faster hashing and comparison decreases. Still, we see that we make a good trade-off, as queries Q8 and Q20 still get (marginally) faster, and only Q6 is marginally slower.

In general, the Public BI workload is characterized by few joins and many aggregations [24], where these aggregations produce small results—few or in the thousands, but almost never in the millions of tuples. This means that the hash tables needed for aggregation are often CPU cache-resident. Therefore, CHT is not triggered and that the USSR is what most matters in this workload, so we focus only on that.

5.3 Micro-Bench: USSR and Group-By

We now move to a number of micro-benchmarks to focus on individual performance aspects of string processing with the USSR. We start with the performance on a `SELECT COUNT(*) FROM T GROUP BY s` query. These strings came from a domain of 10 unique strings, all strings had the same length. Figure 5 shows the speedups that can be achieved using the USSR. We profiled the time spent on string comparisons when checking the keys inside group by's hash table. This results show significant speedups reaching from a 2× to 50× faster string comparison. Similarly, we profiled the time spent on computing hash of the string keys. This results in speedups reaching from 4× for small strings, to 80× for large strings. Besides the significant speedup in terms of string comparison and hash computation, we also noticed significant speedup of the whole query up to ≈ 25×.

5.4 Micro-Bench: Join Probe Performance

We now micro-benchmark Domain-Guided Prefix Suppression, with respect to hash table lookup performance. Our experiment consists of a simple join query where we vary the size of the inner/build relation and the domains of the key columns. We experimented with two and four key columns, four payload columns with values $v \in [0, 10]$.

Figure 6 visualizes the speedup, as well as, the L3/last-level cache (LLC) misses measured. We observe an up to 2.5× faster hash probe including the tuple reconstruction cost. The measured speedups tend to increase with hash table size (size of inner relation). For large hash tables with more than 10^6 rows, the speedups were caused by the significantly smaller and, consequently, more cache-resident hash table. For hash tables with less than 10^6 rows, the performance was mostly affected by the more efficient comparisons directly on compressed data.

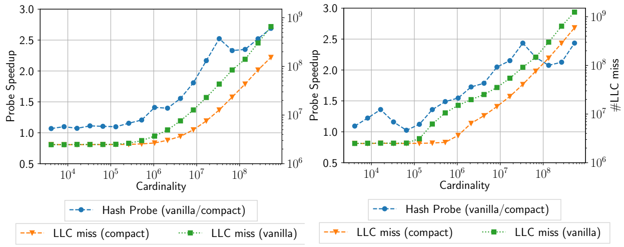
6. RELATED WORK

In this paper we aim at improving cache efficiency of hash tables through compression. An alternative approach is to increase the fill-rate using techniques such as Robin Hood Hashing [9] and Cuckoo Hashing [19]. Similarly, Concise Hash Tables [5] are optimized linear hash tables which try to omit the storage of empty rows. A major disadvantage of Concise Hash Tables is the restriction to linear hash tables, whereas our techniques can be applied to linear as well as bucket-chained hash tables. An extension of Concise Hash Tables are Concise Array Tables [5] which avoid storing the keys by providing a collision-free hash table. These four techniques are orthogonal to our approach of treating the hash table as a compressed table. We highlight that these approaches can be applied, in addition to our techniques, to achieve even more cache-efficient hash tables.

The use of compression is widely spread among database systems. Commonly, analytic databases utilize lightweight compression schemes [26] to elevate effective memory and disk bandwidth during table scans. Although they used similar techniques as ours, notably bit-packing, frame-of-reference and dictionary compression, in this work, we apply compression to hash tables, data structures used *during query evaluation* rather than in data storage. An application of compression which has not received much attention.

The possibility of exploiting compressed data inside data processing pipelines, i.e. compressed execution, is investigated by Abadi et al. [3]. Evaluating predicates on compressed data has been explored deeply for table scans [12, 14, 17, 25]. However, these works focus on scans in isolation.

Lee et al. [16] pioneered joins on data encoded using per-column on-the-fly dictionaries. These dictionaries might seem closely related to the USSR. However there are two major differences: (1) The use of multiple dictionaries necessitates a translation from one encoded join column to the



(a) 4 keys $k_1, ..., k_4 \in [0, 1.000]$ whereas schema suggests 64-bit integers

(b) 2 keys $k_1, k_2 \in [0, 10^6]$ whereas schema suggests 128-bit integers

Figure 6: Hash probe speedup vs. build-side cardinality using 4 payload columns $p_1, ..., p_4 \in [0, 10]$

encoding of the other. The USSR, being a unified query-wide dictionary, does not require such a translation. (2) To fit into cache the USSR has a small fixed size, as opposed to the on-the-fly dictionaries that can grow very large.

Many database systems allow evaluating simple predicates directly on compressed data. Notable examples are IBM BLU [21], SQLServer [15], Quickstep [20], and HyPer [14]. But only very few systems exploit compressed data inside query pipelines. Most notably IBM BLU [21] supports operations on compressed data. It performs joins on encoded and partitioned data similar to Lee et al. [16] and, hence, suffers from the same disadvantages.

Shatdal et al. [23] proposed to optimize algorithms for cache efficiency. One of their techniques, key extraction in sorting/partitioning shares some similarity with Optimistic Splitting. However, Optimistic Aggregates are continuously updated, whereas extracted keys stay constant.

Encoding strings inside dictionaries has been explored by Färber et al. [11] and Binnig et al. [6]. However, both assume global dictionaries, which we do not require (Section 4.1). In contrast, the USSR is a small query-wide on-the-fly dictionary which only encodes frequent strings and avoids expensive update and delete operations.

7. SUMMARY

Hash tables are crucial data structures for modern query engines. However, in analytical queries, they consume a significant amount of memory. Shrinking hash tables not only lowers the memory footprint, but also leads to faster access by fitting more data into faster memory. We present three composable techniques to shrink hash tables: Domain-Guided Prefix Suppression (extremely lightweight compression), Optimistic Splitting (decomposition into hot and cold value slices), and the USSR (opportunistic dictionary compression). We implemented these techniques in the industrial-strength DBMS Vectorwise. In our experiments, our techniques improved query performance by up to 25× in string-intensive queries. On the synthetic TPC-H benchmark we noticed up to 4× smaller hash tables and improved query runtime by up to 50%. On the realistic Public BI workload we achieved improvements of up to 2.2×.

8. REFERENCES

[1] `https://public.tableau.com`.

[2] `https://github.com/cwida/public_bi_benchmark`.

[3] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[5] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

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

[7] P. Boncz, A.-C. Anatiotis, and S. Kläbe. JCC-H: Adding join crossing correlations with skew to TPC-H. In *TPCTC*, pages 103–119, 2017.

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

[9] P. Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.

[10] A. Crolotte and A. Ghazal. Introducing skew into the TPC-H benchmark. In *TPCTC*, pages 137–145, 2012.

[11] 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*, pages 45–51, 2012.

[12] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, pages 857–872, 2018.

[13] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year SQL-as-a-service experiment. In *SIGMOD*, pages 281–293, 2016.

[14] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, 2016.

[15] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[16] J.-G. Lee, G. Attaluri, R. Barber, N. Chainani, O. Draese, F. Ho, S. Idreos, M.-S. Kim, S. Lightstone, G. Lohman, K. Morfonios, K. Murthy, I. Pandis, L. Qiao, V. Raman, V. K. Samy, R. Sidle, K. Stolze, and L. Zhang. Joins on encoded and partitioned data. *PVLDB*, 7(13):1355–1366, 2014.

[17] Y. Li and J. Patel. BitWeaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

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

[19] R. Pagh and F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[20] J. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[21] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. Kulandaisamy, J. Leenstra, S. Lightstone, S. Liu, G. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[22] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976, 2016.

[23] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB, pages 510–521, 1994.

[24] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTEST*, 2018.

[25] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[26] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

[27] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, pages 47–54, 2008.