Leveraging Compression in In-Memory Databases

Jens Krueger, Johannes Wust, Martin Linkhorst, Hasso Plattner
Hasso Plattner Institute for Software Engineering
University of Potsdam
Potsdam, Germany
Email: {jens.krueger@hpi.uni-potsdam.de, johannes.wust@hpi.uni-potsdam.de,
martin.linkhorst@hpi.uni-potsdam.de, hasso.plattner@hpi.uni-potsdam.de}

***Abstract*—Recently, there has been a trend towards column-oriented databases, which in most cases apply lightweight compression techniques to improve read access. At the same time, in-memory databases become reality due to availability of huge amounts of main memory. In-memory databases achieve their optimal performance by building up cache-aware algorithms based on cost models for memory hierarchies. In this paper, we use a generic cost model for main memory access and show how lightweight compression schemes improve the cache behavior, which directly correlates with the performance of in-memory databases.**

Keywords-in-memory databases; database compression; dictionary compression.

I. INTRODUCTION

Nowadays, most database management systems are hard disk based and - since I/O-operations are expensive - therefore, limited by both the throughput and latency of those hard disks. Increasing capacities of main memory that reach up to several terabytes today offer the opportunity to store an entire database completely in main memory. Besides, the much higher throughput of main memory compared to disk access significant performance improvements are also achieved by the much faster random access capability of main memory and at the same time much lower latency. A database management system that stores all of its data completely in main memory - using hard disks only for persistency and recovery – is called an in-memory database (IMDB).

In earlier work, we have shown that in-memory databases perform especially well in enterprise application scenarios [12], [14]. As shown in [12], enterprise workloads are mostly reads rather than data modification operations; this has lead to the conclusion to leverage read-optimized databases with a differential buffer for this workloads [11]. Furthermore, enterprise data is typically sparse data with a well known value domain and a relatively low number of distinct values. Therefore, enterprise data qualifies particularly well for data compression as these techniques exploit redundancy within data and knowledge about the data domain for optimal results. We apply compression for two reasons:

- Reducing the overall size of the database to fit the entire database into main memory, and
- Increasing database performance by reducing the amount of data transferred from and to main memory.

In this paper, we focus on the second aspect. We analyze different lightweight compression schemes regarding cache behavior, based on a cost model that estimates expected cache misses.

A. The Memory Bottleneck

During the last two decades, processor speed increased faster than memory speed did [6]. The effect of this development is that processors nowadays have to wait more cycles to get a response from memory than they needed to 20 years ago. Since processors need to access data from memory for any computation, performance improvements are limited by memory latency time. As seen from a processor's perspective, main memory access becomes more and more expensive compared to earlier days – the Memory Gap widens. Nevertheless, it would be possible to manufacture memory that is as fast as a processor is but there is a direct trade-off between memory size and latency. The more capacity memory has, the longer is its latency time or - important as well - the faster memory is, the more expensive it gets. Since manufacturers concentrated on increasing capacity of main memory there wasn't much focus on improving latency times.

A solution to the problem found in modern processors is the use of a cache hierarchy to hide the latency of the main memory. Between the processors registers and main memory, a faster but smaller memory layer is placed that holds copies of a subset of data found in main memory. When a processor finds the needed data in the cache it will copy it from there waiting less processor cycles. The whole cache is usually much smaller and much faster than main memory. Since the Memory Gap widens with every new processor generation one layer of cache is not enough to fulfill both capacity and latency time demands. Therefore, modern CPUs have up to three layers of cache, each of which with more capacity but worse latency times than the one closer to the processor [8].

Since programs usually do not need to access the whole address space of main memory randomly there is the concept of locality. When a processor fetches a processor word from memory, it is very likely that it needs to fetch another word close by, so-called data locality. Leveraging that fact, processors do not only copy the requested data to its registers but also copy subsequent bytes to the cache. The amount of bytes that are copied at once to the cache is called a cache line or a cache block and usually is about four to 16 processor

 ISBN: 978-1-61208-185-4

words long depending on the specific CPU architecture. On a current 64 bit machine, it is between 32 and 128 bytes.

Consequently, memory access is not truly random since always a complete cache line is fetched regardless the actual requested value. In the worst case, only one value out of the cache line are needed while the rest of the transferred date is polluting both the limited memory bandwidth and limited capacity on each cache. Data that are not found in the cache needs to be fetched from main memory, a so-called cache miss. There is a direct dependency between the performance of in-memory database algorithms and the number of issued cache misses as for instance described in [4], [15]. To gain significant performance improvements or to avoid performance loss, algorithms have to be cache conscious, which means that they have to efficiently use the cache and cache lines issuing as few cache misses as possible. This means data should be read sequentially from main memory instead of randomly.

Multicore processors that are supposed to work in parallel have to wait for other cores to finish their shared memory access before starting its own. Additionally, the physical distance between a processor and its cache also influences the latency time. Multicore processors' shared cache is normally placed in equal distance to each core resulting in less performance than possible on a single core chip. Intel has a solution called Non-Uniform Memory Access (NUMA), where the shared memory is logically the same but physically splitted on the chip. For example the first half of the address space is local to core one and the second half is local to core two resulting in better performance for core one when accessing addresses in the first half but worse performance for the other addresses. When core one requests data from main memory it will be fetched into an address in the first half of the address space if possible [16].

B. In-memory databases in Enterprise Application scenarios

Today's disk-based database management systems are either optimized for transactional record-oriented or analytical attribute-oriented workload, also called Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). The distinguishment arises from enterprises that have transactional systems to support their daily business and need to answer analytical queries on top of that data. OLAP style queries are typically slow on OLTP system; therefore, enterprises usually have a separate OLAP system, e.g., a data warehouse, that stores the same information in a different way and precomputes certain values up-front to improve query performance of analytical queries. The main reason for the performance loss is that OLAP queries are attribute-focused rather than entity-focused, usually reading only a few attributes but more records, e.g., read a whole column or apply a predicate on a complete column. Most OLTP systems store their data row-oriented: a record is stored sequentially on disk and then another record follows maintained by a page layout. Since OLAP queries read only a part of many records, e.g., one attribute of each record, the needed data is not stored sequentially on disk resulting in less read performance. Furthermore, the page layout determines the access pattern that read complete pages from disk as this is the finest granularity to read a record. Due to this fact lots of unnecessary data is transferred in case a few attributes of a relation are requested. Therefore, modern OLAP systems organize the data column-oriented to improve performance of accessing whole attributes [21].

With up to several terabytes of main memory available to applications as well as the increase of computing power with new multi core hardware architectures holding entire databases in main memory becomes feasible [17]; the application of these in-memory databases is especially promising in the field of enterprise applications.

In [14], we could show that Enterprise Applications typically reveal a mix of OLAP and OLTP characteristics. In order to combine both requirements for mixed workload scenarios, the introduction of a write optimized differential buffer together with a read-optimized main storage has been proposed [7], [11], [21]. The differential buffer stores all write operations in an uncompressed manner to allow fast appends. At regular intervals, the differential buffer is merged with the main database to maintain compression and query performance. During this process the buffered values are merge into the read-optimized store as described in [11].

The merge process essentially does two things: it merges the main dictionary with the delta dictionary and keeps track of value ids that may have changed along with their new value. Then, it merges the main attribute vector of the compressed read-optimized store and the attribute vector of the differential buffer while applying the old-value-id/new-value-id mapping from the step before. That second step is not needed if value ids cannot change like in the basic dictionary or hash map approach. However, in an order-preserving dictionary approach that mapping needs to be applied taking a significant amount of clock cycles of the overall merge process. The same happens if the value id are bit compressed and a new dictionary entry make an additional bit necessary in order to represent the values.

II. Compression

A. Motivation

As described in the previous section, main memory latency is a bottleneck for the execution time of computations: processors are wasting cycles while waiting for data to arrive. This is especially true for databases as described in [4]. While cache conscious algorithms are one way to improve performance significantly [3], [19], [20] another option is to reduce the amount of data transferred from and to main memory, which can be achieved by compressing data [22]. On the one hand, compression reduces I/O-operations between main memory and processor registers, on the other hand it leverages the cache hierarchy more effectively, because more data fits in each cache line.

The needed processor cycles to compress and decompress data and the less wasted cycles while waiting for memory result in increased processor utilization. This increases overall performance as long as memory access time is the bottleneck.

 ISBN: 978-1-61208-185-4

Once compression and decompression become so processor-intensive that the processor is limiting the performance instead of the memory, compression has a negative effect on the overall execution time. Therefore, most in-memory databases use light-weight compression techniques that have low CPU overhead [1].

In addition, some operators can operate directly on compressed data - saving decompression time. Abadi et al. illustrate [2] this concept of late materialization to further improve execution speed of queries.

In order to estimate the performance improvements achieved by different compression techniques, the cost model to estimate cache misses presented in [15] is extended by taking compression into account. The basic formula of the model for an uncompressed column scan is:

$$M(s\_trav(R)) = \left\lceil \frac{R.w \cdot R.n}{B} \right\rceil \quad (1)$$

where $M(s\_trav(R))$ are the estimated cache misses on one cache level while traversing a region R in memory sequentially for the first time. The parameters are $R.w$ being the width of one data item in bytes, $R.n$ which is the number of data items to traverse, as well as B which is the number of data items that fit into the cache. In case of an in-memory database $R.w$ is the width of a tuple while $R.n$ is the number of tuples.

As in [15], an inclusive Level 1 cache is assumed meaning that all data that is present in the Level 1 cache is also present in the Level 2 cache. This condition may only be violated temporarily when data in the L1 is changed and marked as dirty before being written back to L2. This assumption holds for Intel CPUs but not for AMD CPUs which have an exclusive cache, meaning that data can be either in L1 or in L2 bot not in both. Modeling exclusive caches is left for future work.

In the following, we provide the cost model for various light-weight compression techniques and compare their performance for a typical analytical query size. Other atomic data patterns as the conditional traversal read [12] can be extended the same way.

B. Run-Length Encoding

When using run-length encoding (RLE), subsequent equal values in memory are stored as a RLE-tuple of $(value, runLength)$, thus encoding a sequence of $(1, 1, 3, 3, 3, 4, 4, 4)$ as $((1, 2), (3, 3), (4, 3))$ reducing the size in memory the larger runs in the data exist. Whether a good amount of runs exist depends on two parameters: First, equal values need to be stored subsequently - this is usually the case if the column is stored in sort order of the values. Second, if the column is ordered, the number of distinct values in the data defines the number of tuples needed to be stored. However, having sorted data is much more important because a randomly ordered column with a few distinct values can contain no runs in worst case if the data is distributed equally. On the other hand, if the number of unique values is close to the number of data items, sorting the items has limited impact on compression, as there are only few runs in this case.

In a sorted run-length encoded column the main indicator of the size of the column is the cardinality of distinct values. Hence, the performance on aggregate operations in an in-memory database is mainly based on the amount of distinct values. Assuming a column's values are in sorted order and the number of distinct values of that column is given by $|D|$, the column can be encoded with $|D|$ RLE-tuples, each holding the value and the run-length. A defensive approach to determine the space needed for saving the run-length is to take the maximum run-length one value can span. Then, the maximum run-length is $R.n$, and therefore, can be encoded with $\lceil \log_2 R.n \rceil$ bits (for simplicity reasons. Actually, it is $runLength_{max} = R.n - |D| + 1$), while bit-compressing the run-length value.

The basic cost model can then be extended to take a run-length encoded column into account:

$$M(s\_trav(R)) = \left\lceil \frac{(R.w + \lceil \log_2 R.n \rceil) \cdot |D|}{B} \right\rceil \quad (2)$$

Since each tuple has the overhead of storing the run-length, run-length encoding becomes less effective as $|D|$ comes close to $R.n$. Hence, the number of distinct values is important. The break even point can be estimated with:

$$|D| = \left\lceil \frac{R.w \cdot R.n}{R.w + \lceil \log_2 R.n \rceil} \right\rceil \quad (3)$$

A generalized formula for unsorted columns encoded with run-length encoding depends on the average run-length of values in the collection which can be answered by examining the topology of the data. For example, imagine a customer table with a column of the customer's address' city name that is ordered in the sort order of another column with zip codes. Clearly the city names aren't in their sort order but they will contain a good amount of runs since equal and similar zip-codes map to the same city name. Given that average run-length $|r|$, one can estimate the cache misses with:

$$M(s\_trav(R)) = \left\lceil \frac{(R.w + \lceil \log_2 R.n \rceil) \cdot \left\lceil \frac{R.n}{|r|} \right\rceil}{B} \right\rceil \quad (4)$$

The break even point, i.e., the minimal number of the average run-length can be computed with:

$$r = \frac{R.w + \lceil \log_2 R.n \rceil}{R.w} \quad (5)$$

C. Bit-Vector Encoding

Bit-vector encoding stores a bitmap for each distinct value. Each bitmap has the length of the number of data items to encode in bits. The value 1 in a bitmap for a distinct value indicates that the data item with the same index has this particular value. As each data item can only have one value assigned, only one bitmap has the value 1 at a given index.

 ISBN: 978-1-61208-185-4

The compression size is therefore dependent on the number of data items and the number of distinct values to encode.

$$M(s\_trav(R)) = \left\lceil \frac{(R.w + R.n) \cdot |D|}{B} \right\rceil \quad (6)$$

For each distinct value, the value itself needs to be stored once plus a bitmap of the number of tuples in bits. The break even point can be estimated with:

$$|D| = \left\lceil \frac{R.w \cdot R.n}{R.w + R.n} \right\rceil \quad (7)$$

D. Null Suppression

Null Suppression stores data by omitting leading 0s of each value. The main indicator of how good the compression will be is the average number of 0s that can be suppressed. Think of an integer column that stores the number of products sold per month. Since only the less significant bits would equal to 1 and no negative values would appear one could get a good compression ratio with Null Suppression. Since each value has a variable length Null Suppression needs to store the length of each value. A good way to do that is to suppress only byte-wise, so a value can be stored with one to four bytes. To encode that length one needs two bits so the length-metadata for four values fits on one byte. Given an average number of 0s to suppress $|z|$ and the suppressable bits $|z_b| = 8 \cdot \left\lfloor \frac{|z|}{8} \right\rfloor$ an estimation of the cache misses is possible:

$$M(s\_trav(R)) = \left\lceil \frac{R.n \cdot (R.w - |z_b|) + 2 \cdot R.n}{B} \right\rceil \quad (8)$$

E. Dictionary Encoding

Dictionary Encoding is a widely used compression technique in column-store environments. A dictionary is created by associating each distinct value with a generated unique key - a value id - and replacing the original values in the attribute vector with their value id replacements. By combining the attribute vector with the dictionary entries the original values can be reconstructed. Each distinct value is stored only once while the smaller value ids are used as their references which saves space in memory as well as allowing compatible operators to directly work on the dictionary only, e.g find all distinct values, or vice versa operate on the attribute vector without accessing the dictionary. Usually, storing the value ids instead of the actual values take much less space in memory and their length is fixed allowing variable length values to be treated as fixed length values in the document vector, which leads to increased performance [9]. Given the dictionary fits into the cache the cache misses for a single column scan can be estimated with the following formula:

$$M(s\_trav(R)) = \left\lceil \frac{R.id \cdot R.n}{B} \right\rceil + \left\lceil \frac{|D| \cdot R.w}{B} \right\rceil \quad (9)$$

The more distinct values the dictionary needs to hold the more likely it is that a lookup leads to a cache miss. Since the access is random previously unloaded cache lines need to be fetched again. Hence, the size of the dictionary matters. The cache misses can be estimated with the formula for a repetitive random access pattern rr_acc presented in [15], since the accessed position in the dictionary is random and can be the same multiple times.

$$M(s\_trav(R)) = \left\lceil \frac{R.id \cdot R.n}{B} \right\rceil + M(rr\_acc(|D| \cdot R.w)) \quad (10)$$

with:

$$M(rr\_acc(R)) = \left\lceil C + (\frac{r}{I} - 1) \cdot (C - \frac{\#}{C} \cdot \#) \right\rceil \quad (11)$$

and $C = \left\lceil \frac{I \cdot R.w}{B} \right\rceil$ where I is an approximation of the number of accessed tuples. Since the whole data is read, each value in the dictionary is read at least once and $I = R.n$. r is the number of access operations which is equal to $R.n$, too. $\#$ is the number of slots in the cache and therefore equals to $cacheSize/B$ in a fully associative cache.

F. Comparison

We compare the expected cash misses for a table with a size typical in Enterprise Data: Given one million 48 byte string values of which 50,000 are distinct an uncompressed column scan would issue $\left\lceil \frac{48 \cdot 10^6}{64} \right\rceil = 750,000$ cache misses with a 64 byte cache line. We calculate the expected cash misses for each algorithm and provide a sensitivity analysis on the number of distinct values.

1) Run-Length Encoding: A run-length encoded column when stored in sort order would issue only $\left\lceil \frac{(8 \cdot 48 + \log_2 10^6) \cdot 50{,}000}{8 \cdot 64} \right\rceil = 39,454$ cache misses.
The break even point would be at $\left\lceil \frac{8 \cdot 48 \cdot 10^6}{8 \cdot 48 + \log_2 10^6} \right\rceil = 950,496$ distinct values.

If the column was stored unsorted the number of expected cash misses would be in the worst case $\left\lceil \frac{(8 \cdot 48 + \log_2 10^6) \cdot 10^6}{8 \cdot 64} \right\rceil = 788,929$. This worst case would occur in the scenario of an average run-length of 1. The average run-length for each value has to be at least $\left\lceil \frac{8 \cdot 48 + \log_2 10^6}{8 \cdot 48} \right\rceil = 1,05$ to issue less cache misses compared to no compression.

2) Bit-Vector Encoding: For the example above a full scan would issue $\left\lceil \frac{(8 \cdot 48 + 10^6) \cdot 50{,}000}{8 \cdot 64} \right\rceil = 97,693,750$ cache misses. Bit-vector encoding clearly is not suitable for lots of distinct values or a small amount of bytes to compress. Since each column has a good number of 0-runs, run-length encoding on top of bit-vector encoding might help. The break even point is at $\left\lceil \frac{8 \cdot 48 \cdot 10^6}{8 \cdot 48 + 10^6} \right\rceil = 384$ distinct values.

On the other hand, the formulas show that the amount of bytes per value play a little role in the overall amount of cache misses. Thus, bit-vector encoding should be used when the values to compress have a certain size.

 ISBN: 978-1-61208-185-4

3) Null Suppression: Given that the average amount of bytes to suppress is 3 then the estimated cache misses are $\left\lceil \frac{10^6 \cdot (8 \cdot 48 - 8 \cdot 3) + 2 \cdot 10^6}{8 \cdot 64} \right\rceil = 707,032$.

4) Dictionary Encoding: Given a 4 byte value id the number of cache misses for a single column scan in a dictionary encoded column are $\left\lceil \frac{4 \cdot 10^6}{64} \right\rceil + \left\lceil \frac{50,000 \cdot 48}{64} \right\rceil = 100,000$

The following table shows the estimated number of cache misses for the example above with the 5% distinct cardinality.

	Cache Misses
No compression	750,000
RLE (sorted)	39,454
RLE (unsorted, worst case)	788,929
Bit-Vector Encoding	97,693,750
Null Suppression	707,032
Dictionary Encoding	100,000

Figure 1 shows a sensitivity analysis with regards to the number of distinct values in order to investigate the influence of a changing cardinality of those.

G. Evaluation

The usefulness of compression algorithms depends on the data profile of a column. The following table describes the applicability of each compression technique for several data profiles.

	few distinct	many distinct
No compression	- -	- -
RLE (sorted)	+ +	+
RLE (unsorted)	- -	- -
Bit-Vector Encoding	+	- -
Null Suppression	-	-
Dictionary Encoding	+	+

Our goal is to find a compression technique that performs best under OLTP as well as OLAP workloads in an enterprise environment. Based on our findings of enterprise data characteristics in [12], we focus on a sparse data set with a vast amount of columns but most of them storing a small number of distinct values. We use a column-oriented store since it performs better under an OLAP workload than a row store does [21]. However, in an OLTP scenario most queries fetch only few complete records. The column store finds each respective entry in all columns separately and then reconstructs the record. Since there are lots of columns finding those entries has to be fast.

Run-length encoding on a sorted column issues by far the fewest cache misses of all presented compression techniques. However, applying run-length encoding requires sorting each column before storing it. In order to reconstruct records correctly we would have to store the original row number as well, called the surrogate id. When reconstructing records each column needs to be searched for that id resulting in a complexity of $O(R.n)$ per column. As Enterprise Applications typically operate on tables with up to millions records we cannot use surrogate ids and prefer direct or implicit offsetting instead ($O(1)$).

Basic dictionary encoding allows for direct offsetting into each column and also benefits from a sparse data set as enterprises have it. In addition the compaction process' performance increases when using dictionary encoding compared to run-length encoding. Therefore, dictionary encoding fits our needs best and is our compression technique of choice for a mixed workload. Furthermore, it still can be optimized as described in the following section.

III. Dictionary Compression Techniques

In this section, we discuss various optimization of the basic dictionary approach introduced in section II-E. We evaluate their applicability for different data access profiles.

A. Order-Indifferent Dictionary

The basic dictionary approach described in the last chapter did not care about how the values in the dictionary are ordered. That makes finding a value in the dictionary, e.g., for an insert - one needs to find out whether the value is already in the dictionary - an expensive operation ($O(|D|)$). A possible solution is to store the values based on their hash value. Given a good hash function one can find a value in the dictionary in $O(1)$ as well as finding a value for a specific value id in $O(1)$. This clearly depends on a good hash function and may increase compression size.

B. Order-Preserving Dictionary

In comparison to the basic dictionary a hash value supported dictionary approach could speed up finding a value in the dictionary but still does not enforce ordered data items in memory. That becomes a disadvantage when executing range queries like finding all records that begin with the letter K, e.g., in a column that stores names. An order-indifferent dictionary needs to traverse the whole dictionary filtering all values that begin with K and returning their associated value ids. This has a complexity of $O(|D|)$. In a sorted dictionary, one could find the first occurrence of a value starting with K and L with binary search and then return the lower and upper bound for all value ids that are associated with values beginning with K without actually checking their values. The complexity is $O(\log_2 |D|)$. The downside of that approach is that if new values need to be added to the dictionary they can destroy the sort order invalidating possibly all value-id/value associations and resulting in a complete rewrite of the attribute vector $O(R.n)$.

C. Bit-Compressed Attribute Vector

The size of the attribute vector, hence the read performance, is also affected by the compression ratio between the original values and the value id replacements. However, the number of distinct values in the uncompressed values collection is important as well. Firstly, because the entries in the dictionary increase with every unique value - for every value-id/value compression ratio, there is a number of distinct values hat dictionary encoding becomes useless - secondly, the more unique values need to be encoded, the more unique value

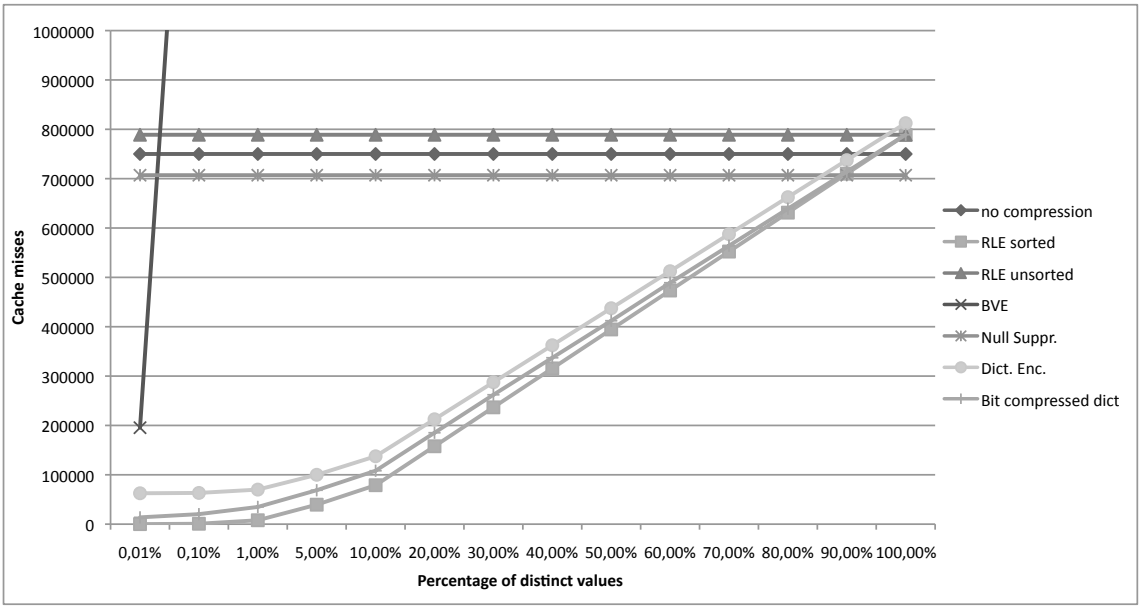

Fig. 1. Compression techniques with regards to cache misses and distinct values.

ids are needed. In the basic dictionary example above a lot of compression opportunities are wasted by reserving 32 bits to define the value id space. Dictionary encoding with bit-compressed value ids varies the length of the value ids and reserves only the needed number of bits to encode all distinct values but still guarantees fixed-length value ids. Given 200 values in the dictionary, the attribute vector that needs to be compressed needs only one byte to store each value id, allowing 64 value ids to fit on a 64 byte cache line. Similar to the order-preserving dictionary the disadvantage of this approach is that the bit-length of the value ids needs to be increased and all values in the attribute vector need to be rewritten when the number of distinct values in the dictionary exceeds the amount of values that can be encoded with the current number of bits - $O(R.n)$.

The cost model can then be extended with:

$$M(s\_trav(R)) = \left\lceil \frac{\lceil \log_2 |D| \rceil \cdot R.n}{B} \right\rceil + \left\lceil \frac{|D| \cdot R.w}{B} \right\rceil \quad (12)$$

Taking the same parameters from the previous section the issued cache misses then are $\left\lceil \frac{\lceil \log_2 50{,}000 \rceil \cdot 10^6}{8 \cdot 64} \right\rceil + \left\lceil \frac{50{,}000 \cdot 48}{64} \right\rceil = 68,750$ which is almost half the amount of the basic dictionary.

D. Bit-Compressed Order-Preserving Dictionary Encoding

The last two described dictionary compression techniques have the same problem of rewriting the whole attribute vector for different reasons. However, the two problems are connected. A reordering of the dictionary can only happen if new distinct values are added to the collection or when deleting values. Furthermore, an extension of the value id space can only be a result of adding new distinct values to the dictionary. Using both approaches together can lower the cost of inserts and updates. When new values are added to the dictionary and the amount of values exceeds the value id space then the rewriting of the attribute vector can do both, updating to new value ids with the new bit-length, in one step.

E. Comparison

The following table shows the different dictionary encoding variants under different workloads.

	Basic	Hash Map	Bit-Compr.	Order-Pres.
few inserts, many equal queries	+	+	+ +	-
few inserts, many range queries	-	- -	+	+ +
many inserts, many equal queries	+	+ +	-	-
many inserts, many range queries	-	-	- -	+

The following table lists the advantages and disadvantages of the different dictionary encoding variants.

	ADVANTAGES	DISADVANTAGES
Basic Dict.	fixed length, compression time	compression size
Hash Map	compression time	execution time
Bit-Compr.	compression size	compression time
Order-Pres.	execution time	compression time
Order-Pres. & Bit-Compr.	execution time, compression size	compression time

IV. RELATED WORK

In the area of database management systems, compression is also used to improve query speed as described in [22] as work

focused on reducing the amount of data only. That becomes especially useful when data is stored column-wise such as in C-Store, a disk-based column-oriented database, see [21]. The work presented in [1] describes how compression can be integrated into C-Store and shows the impact on read performance. In a real world scenario one has to consider the negative impact on write performance when using compression. [10] comes to the conclusion that column stores perform better than row stores in most cases.

However, data compression can limit the applicability to scenarios with frequent updates leading to dedicated delta structures to improve the performance of inserts, updates and deletes. The authors of [7] and [18] describe a concept of treating compressed fragments as immutable objects, using a separate list for deleted tuples and uncompressed delta columns for appended data while using a combination of both for updates. In contrast, the work of [11] maintains all data modification of a table in one differential buffer that is write-optimized and keeps track of invalidation with a valid bit-vector. Later work of the same authors shows how to enable fast updates on read-optimized databases by leveraging multi-core CPUs [13].

The work of [5] depicts a technique of maintaining a dictionary in a order-preserving way while still allowing inserts in sort order without rebuilding the attribute vector due to changed value id's.

In the area of in-memory databases with the focus on OLTP and real-time OLAP, the customer study presented in [12] show a very high amount of read queries compared to write queries supporting the fact that a compressed read-optimized store is useful.

V. Conclusion

In this paper, we showed and explained the positive impact on read performance for an in-memory database when using data compression. In order to compare different kinds of lightweight compression techniques under different data distributions we extended the generic cost model to take compression into account. We also presented several lightweight compression techniques and different optimizations regarding dictionary compression as well as trade-offs that have to be made in favor of late materialization and write performance. The paper also described why focussing on read performance is necessary and how a sufficient write performance can be achieved as well. It concludes that under most circumstances - especially for column stores - dictionary compression is the best choice when it comes to optimizing read and write performance under a mixed workload.

References

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475, 2007.

[3] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[5] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 283–296, 2009.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[8] U. Drepper. What every programmer should know about memory, 2007.

[9] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[10] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 487–498, 2006.

[11] J. Krüger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber. Optimizing Write Performance for Read Optimized Databases. In *Database Systems for Advanced Applications, 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II*, pages 291–305, 2010.

[12] J. Krüger, M. Grund, A. Zeier, and H. Plattner. Enterprise Application-Specific Data Management. In *Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitoria, Brazil, 25-29 October 2010*.

[13] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[14] J. Krüger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner. A case for online mixed workload processing. In *Proceedings of the Third International Workshop on Testing Database Systems, DBTest 2010, Indianapolis, Indiana, USA, June 7, 2010*, 2010.

[15] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 191–202, 2002.

[16] D. E. Ott. Optimizing software applications for numa.

[17] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1–2, 2009.

[18] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *VLDB J.*, 12(2):89–101, 2003.

[19] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89, 1999.

[20] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486, 2000.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564, 2005.

[22] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55–67, 2000.

 ISBN: 978-1-61208-185-4