Partitioned Bit-Packed Vectors for In-Memory-Column-Stores

Martin Faust
Hasso-Plattner-Institute
Potsdam, Germany
martin.faust@hpi.de

Pedro Flemming
Hasso-Plattner-Institute
Potsdam, Germany
pedro.flemming@hpi.de

David Schwalb
Hasso-Plattner-Institute
Potsdam, Germany
david.schwalb@hpi.de

Hasso Plattner
Hasso-Plattner-Institute
Potsdam, Germany
hasso.plattner@hpi.de

ABSTRACT

In recent database development, in-memory databases have grown more and more in popularity. The hardware development of the past years has made it possible to keep even larger data sets entirely in main memory of one or a few machines.

However, most applications on in-memory databases are memory-latency-bound rather than compute-bound. Combining strong compression techniques and efficient data structures is essential to fully utilize the hardware capabilities. A common data structure for efficient storing is the bit-packed vector. The bit-packed vector uses a fixed encoding length, which cannot be changed after initialization. Therefore it requires full re-initialization, when the encoding-length changes. In this paper we propose a new data structure, the partitioned bit-packed vector. Therein the encoding length of the stored elements may increase dynamically, while still providing comparable single-value access performance. This paper outlines the access to this data structure and evaluates its performance characteristics. The results suggest that the partitioned bitvector has the capabilities to improve the performance of existing in-memory column-stores for typical enterprise workloads.

1. INTRODUCTION

Over the last years several new databases have been developed, that aim to combine transactional (OLTP) and analytical (OLAP) workloads. These workloads have been traditionally kept separate because they require different optimization techniques. A key requirement to reunite both workloads is to store all data in main memory, as done in SAP HANA [8], [3] and our prototypical database system Hyrise [2]. Only by keeping all data in-memory we can achieve a performance that allows us to enable the flexibility in query processing that is needed to run analytical workloads on a transactional schema. Additionally, only when all data is in-memory a column store is able to answer transactional queries with reasonable and predictable latency.

The choice of the optimal data structure depends highly on the expected workload, however, in general a lightweight compression scheme can be beneficial for access speed and memory bandwidth savings [9]. Dictionary encoding together with bit-packed vectors are an efficient way to store data[7] by storing only the minimal amount of bits to represent the largest integer of the input range. However, bit-packed vectors hold several limitations due to the fact that they require a fixed number of bits per value. These limitations can be overcome through using the partitioned bit-packed vector, that is presented and evaluated in this paper.

There are two major limitations when using bit-packed vectors that are discussed in this paper and solved with the proposed data structure. The first problem is that bit-packed vectors can only hold a specific number of distinct values. Once an additional element has been added so that this upper bound is exceeded, the whole bit-packed vector has to be re-initialized with new parameters and all data has to be reloaded. On large datasets this results in a significant management overhead. This is unacceptable for use cases that require instant and predictable access performance.
The second limitation is that bit-packed vectors are bound to a specific encoding of the data that is stored. Every value that will be stored has to be encoded in the same number of bits and therefore takes the same amount of space in memory. In many cases this is not the best way to encode the data. Often enough an attribute in a database has a high number of distinct values but actually a very low number of those values accounts for far over 90% of the data that is actually stored. In those skewed cases it is beneficial to encode the values that occur with a higher frequency with less bits than those that occur with a lower frequency. While bit-packed vectors do not support this encoding, with partitioned bit-packed vectors it can be achieved.

IMDM '15, August 31 2015, Kohala Coast, HI, USA

DOI: http://dx.doi.org/10.1145/2803140.2803142

2. RELATED WORK

Column-stores have been introduced for the read-mostly business intelligence DBMS market [13]. Only recently, with the availability of large main memories at reasonable cost, in-memory column stores are being used to handle a mixed workload.

Schwalb et al. [12] show that dictionary encoding is most suited to handle a mixed workload. It is therefore a common encoding technology for in-memory column-stores [16]. SAP HANA, IBM BLU [10], Hyrise [2] and others use dictionary encoding. This encoding provides several advantages: (a) when compressing a dataset with dictionary encoding all distinct values will be stored in a separate data structure, the dictionary. The factor of compression is particularly high when there are only a few distinct values in a large dataset. Additionally, (b) for many operations on an encoded column, the dictionary does not need to be decoded for processing. Integer comparison is in general less expensive than a comparison of the actual string value. Further, (c) if the dictionary is sorted, range scans are done without decoding the encoded column-data. When, for example, looking for customers born after 1990 only the encoded value of 1.1.1990 needs to be compared to the encoded elements. We can compare the encoded integer value instead of a more costly comparion of date or string data types. Dictionary encoding is often used in combination with other compression technologies, that further reduce the size of the dictionary.

2.1 Insert-Only

The insert-only approach has important benefits for in-memory databases in the context of many real-world applications. This approach offers the possibility to keep an audit trail of the changes made to the data within the datatable. When using the insert-only approach, all updates and deletions to the dataset are executed by inserting additional tuples. The database system does not allow applications to actually modify physically stored tuples. Whenever an update or an deletion to an existing tuple is made, an new tuple is inserted that holds a reference to the value it is invalidating [8].
Using this approach allows the database system to keep a full history of the system. This makes it possible to execute "time-travel queries", which reflect the dataset at any previous point in time [1].

2.2 Bit-Packed Vectors

Bit-packed vectors use memory space very efficiently. If the values to be encoded are a continuous, monotonically increasing sequence it is the most efficient scheme that still allows for direct addressing via array-indexes. Due to dictionary encoding the value-ids form such a sequence. Wilhalm et al. [17] give details about the implementation of such bit packed vectors. Figure 1 illustrates the storage layout with an example.

3. PARTITIONED BIT-PACKED VECTOR

With the introduction of partitioned bit-packed vectors we want to solve the flexibility problems in bit-packed vectors while maintaining high positional-lookup and scan performance. The general concept is to hold multiple bit-packed vectors to eliminate the need for re-initialization when the encoding needs one more bit. We will allow an increase of the bits-per-value without sacrificing the storage savings of already compressed values.

This design of the partitioned bit-packed vector manages multiple instances of bit-packed vectors in a list. It focuses on flexibility and access performance. The most important benefit is that, when the length of the data-encoding increases, the vector does not need to be re-initialized. Instead, a new instance of a normal bit-packed vector is appended to the end of the vector-list. This new bit-packed vector is initialized so that it can hold the data with the new length, as shown in Figure 2. The handling of meta data such as the number of bits per value is left to the underlying bit-packed vector.

This design can only be used if the encoding of already stored values did not change when the encoding-length has increased, as it is the case when storing a column with an unsorted dictionary.

Inserts of new elements into the partitioned vector are always made into the last partition. This vector is called the *active* vector. This approach maintains the order of the elements. The index of an element E in the partitioned bit-packed vector (global index) is composed through the index of the element in its vector V_j (local index) and the number of elements in all previous partitions. Both, the local and the global index, are implicity calculated and not stored with the data.

$$GlobalIndex(E) = LocalIndex(V_j, E) + \sum_{k=0}^{j-1} size(V_k) \quad (1)$$

3.1 Single Access by Index

An element of the partitioned bit-packed vector is accessed by its global index. To retrieve an element by its global index, both the partition and the local index of the element in this partition need to be computed. A basic lookup strategy is shown in Listing 1 below.

In this algorithm we iterate over all existing partitions in order. If the given index is inside of the current vector, the value at the index in this vector will be returned. If not, the size of this vector will be subtracted from the given index. This will be repeated for all vectors. If the index does not point into any of the existing vectors, then the given index lies outside of the bounds of the partitioned bit-packed vector.

Listing 1: Accessing an element by its global index

```
function get(global_index):
  local_index = global_index
  for partition in listOfPartitions:
    if local_index < len(partition):
      return partition.get(local_index)
    else:
      local_index = local_index
                    - len(partition)
  raise Exception("Index out of bounds!")
```

This algorithm has linear complexity $O(|\#\text{ of partitions}|)$ because it needs to be iterated over the list of partitions. The access to a normal bit-packed vector has constant complexity $O(1)$. This strongly suggests that access to a single element in a partitioned bit-packed vector will be less efficient than

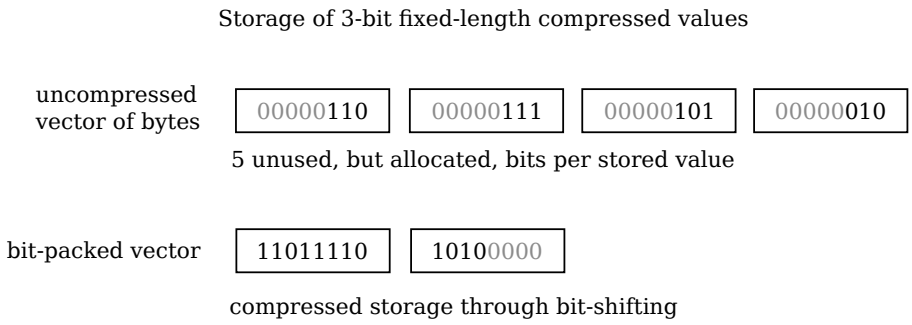


Figure 1: Storage of 3-bit values when using bit-packed vectors

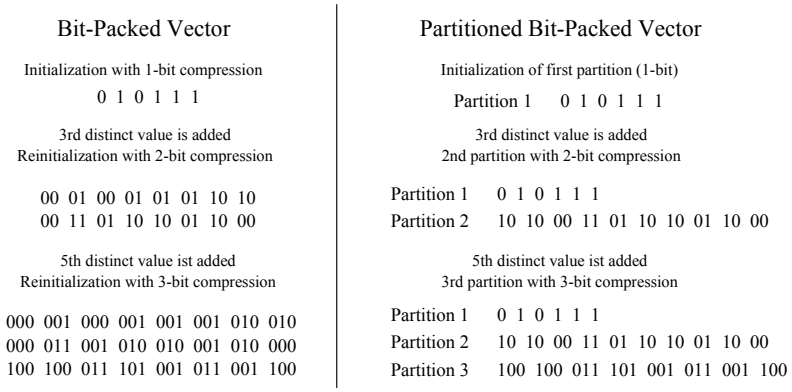


Figure 2: Construction of a partitioned bit-packed vector compared to a regular bit-packed vector

with a normal bit-packed vector. But it also needs to be pointed out that the number of bit-packed vectors that are stored inside of an instance of partitioned bit-packed vector will usually stay considerably low. We can safely assume that datasets on which bit-packed vectors are currently used, contain less than 2^{64} distinct values.

Consequently, in the worst case a partitioned bit-packed vector stores 64 sub-vectors. However, finding the right partition requires looping over the sizes of each subvector, which will stall the CPU for each comparison, as the following comparisons have to wait until the preceding operations are computed, thereby not making full use of instruction-level parallelism of modern CPUs. In Section 4 we will discuss how using prefix-sums increases the performance on accesses, by optimizing the algorithm for instruction level parallelism.

3.2 Scanning the Data

Besides the access to single elements, scanning over the entire data stored in the vector is an important operation. In this partitioned vector every sub-vector needs to scanned individually. The overhead of scanning a partitioned bit-packed vector compared to a normal bit-packed vector is significantly smaller than when accessing a single element. Each separate bit-packed vector can be scanned in the same way as they would in an unpartitioned store [6].

In Listing 2 a simple sequential scan algorithm for the partitioned vector is shown. Each sub-vector is scanned individually. The scan produces the local positions at which the predicate matched. To get the global positions the current offset is added to each local position. The offset of an vector is the global index of the first element the vector. After that the new fetched global positions are appended to the current result-set and the offset is increased by the length of the current vector to get the offset of the next vector.

Listing 2: Equals-Scan on a partitioned bit-packed vector

```
function eq_scan(scan_value):
  positions = []
  offset = 0
  for partition in listOfPartitions:
    # Scanning the sub-vector
    for index, value in partition:
      if scan_value == value:
        positions.push(offset + index)
    offset = offset + len(partition)
  return positions
```

Both the scan of the bit-packed vector and the scan of the partitioned bit-packed vector have linear complexity of $O(|\#\text{ of elements}|)$. It can be assumed that the possible loss of efficiency due to structural overhead being smaller in the scan than when accessing single items.

During a scan the partitioned bit-packed vector benefits

from its higher compression. Every sub-vector but the last uses a shorter encoding for its data than the normal bit-packed vector. The last vector will have the same encoding length as the normal bit-packed vector. Scanning over vectors that have a shorter encoding is faster because less data needs to be fetched from main memory to the CPU. This results in a gain in performance for the partitioned bit-packed vector. In Chapter 4 we will evaluate how this compression benefit compares to the management overhead during scans on various datasets.

3.3 Inserts and Updates

Inserts perform under the same conditions as they do when inserting into a non-partitioned bit-packed vector, because every insert is directly stored into the last bit-packed vector in the list. Whereas Updates have the same lookup overhead as single accesses.

When the insert-only approach is used, the lookup overhead for updates is eliminated. The concept of insert-only is that changes to existing data-tuples are made through appending new tuples to the dataset. When a database uses this concept, updates have the same low overhead as inserts. (See Section 2.1)

4. PREFIX-SUM TO IMPROVE READ PERFORMANCE

Using the prefix-sum of the sizes of the sub-vectors can increase the performance of the access to a single element. The prefix-sum of a sequence of numbers $X = x_0, x_1, ...$ is a sequence $P = p_0, p_1, ...$ with:

$$p_i = \sum_{j=0}^{i} x_j \tag{2}$$

The algorithm to calculate prefix-sums has linear complexity $O(n)$. It can be implemented in parallel to utilize multi-core processors more efficiently [5]. A prefix sum over the size of the sub-vectors in our partitioned bit-packed vector would indicate at what global indexes the next vector starts. (Example in Table 1). We can use this information to design a more efficient algorithm for accessing single values in our partitioned vector.

Listing 3: Accessing an element while using prefix-sums

```
function get(global_index):
  prefix_sums = [0] + calculatePrefixSums()
  for j in [0 .. num_partitions - 1]:
    local_index = global_index
                  - prefix_sums[j]
    if global_index < prefix_sums[j+1]:
      return partitions[j].get(local_index)
  raise Exception("Index out of bounds!")
```

In Listing 3 we can see an algorithm that uses prefix-sums to retrieve a value from the partitioned bit-packed vector. For now, we calculate the prefix sums on-demand, at each access. We prepend the start index 0 to the list of prefix-sums, so that the list now contains the start-indexes of all vectors. We then iterate over all sub-vectors. We calculate the local index that the given global index would have in a vector and then check if the global index lies within the vector. If yes we return the value at the local index in the current sub-vector.

This code can be parallelized more efficiently on instruction level by the CPU. The instructions to calculate the local indexes (Line 4) and the instructions to compare the global index to the prefix sum (Line 5) can be executed independently on every step of the loop.

In this design of a partitioned bit-packed vector, where inserts are only made into the last vector (active vector), we can permanently keep a list of the prefix-sums. On every insert, the last prefix-sum is incremented by 1. When a new sub-vector is created and therefore becomes the active vector, we append a new prefix-sum to the list. The new prefix-sum is the same as the prefix-sum of the previously active vector. On all new inserts, we only increment the new prefix-sum of the new active vector.

Listing 4: Optimized access to an element with prefix-sums

```
function get(global_index):
  for j in [num_partitions - 1 .. 0]:
    local_index = global_index
                  - prefix_sums[j]
    if global_index >= prefix_sums[j]:
      return partitions[j].get(local_index)
  raise Exception("Index out of bounds!")
```

We can optimize the algorithm from Listing 3 even further. The new algorithm shown in Listing 4 optimizes on the fact, that in partitioned bit-packed vectors the newest partitions usually hold more values than the first partitions. Therefore the accessed indexes are more often in the last partitions than in the first. The major change in this algorithm is that the list of partitions is traversed backwards. This allows the algorithm to terminate earlier on most of the accesses.

5. EVALUATION

In this Chapter the data structure that has been defined in Chapter 3 will be implemented and evaluated for a sample of database operations. It will be determined how strong the impact of partitioned bit-packed vectors is when using them instead of normal bit-packed vectors in a column-store scenario.

5.1 Implementation Details

The implementation of a partitioned bit-packed vector that is evaluated here was developed in C++. Accurate measuring of performance characteristics, such as CPU clock cycles, was achieved by using PAPI (version 5.0), the Performance Application Programming Interface [14] to access the perofmance counters of the CPU. All test were performed on an 2.66 GHz Core i5-750 system with 8 MB L3 cache running Ubuntu 12.04.

5.2 Performance Evaluation

In this Section we will measure and evaluate the performance of the partitioned bit-packed vector. We will use different types of datasets and measure how some important

vector-size	8	23	95	420	1254
prefix-sum	8	31	126	546	1800
global-indexes	[0 .. 7]	[8 .. 30]	[31 .. 125]	[126 .. 545]	[546 .. 1799]

Table 1: Example of prefix-sums of vector sizes

operations perform on this proposed data structure. The overall runtime of an operation will be measured in CPU clock cycles.

All values that are handled by the vectors are dictionary encoded. We use a column structure that holds the dictionary. When a value is inserted it is encoded and its field code is passed to the underlying vector. The vector is notified if the length of the encoding has increased. In the case of a normal bit-packed vector, it needs to re-initialize itself with the new encoding. A partitioned bit-packed vector creates a new partition to handle all following values.

On every read access to the vector, the field code will be read from the vector, decoded by the dictionary and the real value returned. This allows a simulation that is close to a real use-case. Still it has to be pointed out, that the encoding and decoding of the values will create an overhead during the benchmarks. The runtimes that will be measured always include the access to the underlying vector as well as to the dictionary. The dictionary uses a binary-search tree to encode the values and a vector to decode the values. The results in logarithmic complexity ($O(log(|\text{number of elements}|))$ for encoding values and constant complexity ($O(1)$) for decoding field-codes.

5.3 Test Datasets

In the following benchmarks, the operations are evaluated on two different types of datasets (see Figure 3). They differ in how the distinct values are distributed across the dataset. Different data-distributions simulate the characteristics of real column-data. For example in a table with customer data the distribution of the value age can be similar to a gaussian distribution[15]. Very common in databases are columns in which only very few distinct values occur frequently. Other values occur only rarely. In a customer table of a German online-shop, it is plausible that the country of origin is most frequently "Germany". Many other values can occur, but they will occur significantly less often. This characteristic is simulated by the dataset with a pareto distribution [4].

The generation of gaussian and pareto distributions is based on [11]. To generate uniform distributed values which are also needed to create the other distributions, the implementation in the standard library of C++ is used.

5.4 Inserts

The first operation we evaluate is the insert of data into the vector. We run inserts with a dataset containing 10 million elements. A number of datasets is created with a varying number of distinct values. We evaluate, how the insert performance reacts to a change in the number of distinct values on the dataset. We compare the normal bit-packed vector from Section 2.2 to the partitioned bit-packed vector that was introduced in Section 3. In Figure 4 the

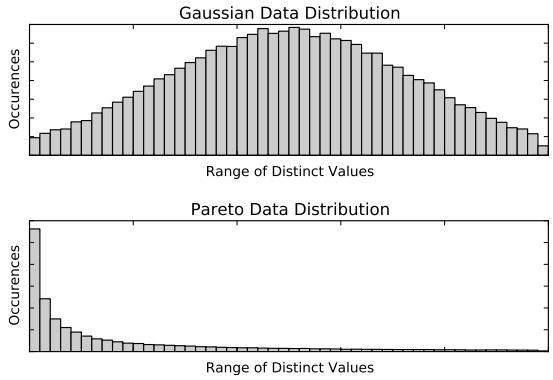


Figure 3: Distribution of distinct values in Test-Datasets

results of the insert-benchmark are shown. The base-line (dashed-line) shows the performance of a bit-packed vector that was already initialized with the number of bits, which were needed to encode the specific dataset. This line represents a base-line for the other implementations. The unpartitioned implementation is a bit-packed vector, that is initialized with a 1-bit compression. When the encoding length increases, the bit-packed vector is re-initialized with the longer encoding. All previously stored data needs to be copied and encoded with the new length. The partitioned implementation is a partitioned bit-packed vector as introduced in Section 2.2. The encoding length of its first partition is also 1 bit.

The results show that inserting into a partitioned vector instead of a unpartitioned vector offers a slight performance benefit of about 10 clock cycles on average, if we take the cost of re-encoding the unpartitioned vector into account. Comparing the paritioned vector with the base-line reveals only a moderate performance decrease of about 12 additional clock cycles per element.

5.5 Single Access by Index

The next operation we will evaluate is the read-access to a single item in the vector. This is a crucial operation when dealing with transactional workload (OLTP). In those workloads there are very frequently inserts and lookups of only a few rows. Having a low latency for those operations is essential to run a transactional application.

We have measured the time of 1 million random read-accesses to single values the vector. The test dataset contains 10 million elements. The access-benchmark is repeated for different numbers of distinct values. We evaluate 2 implemen-

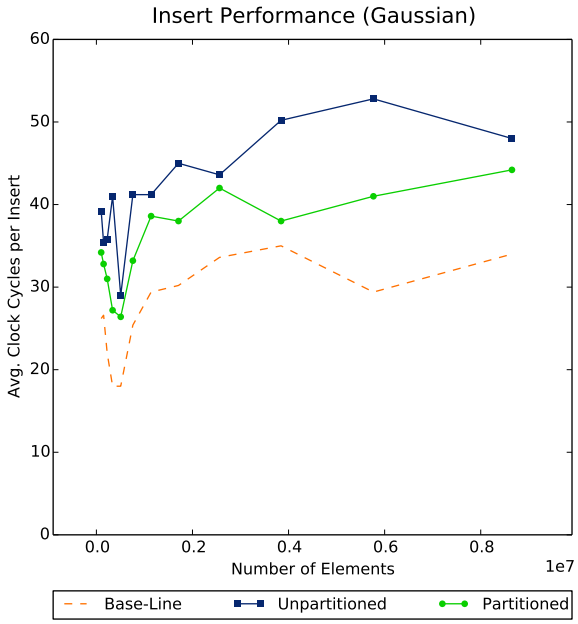


Figure 4: Insert performance of a partitioned bit-packed vector compared to a normal bit-packed vector

tations of a partitioned bit-packed vector. We compare the lookup algorithms presented in Listing 1 and Listing 4 and the unpartitioned bit-packed vector.

The partitioned bit-packed vectors we evaluate here all use a base encoding of 1-bit. This means, that they are initialized with an empty 1-bit partition. They then dynamically create partitions with a longer encoding when they are needed. When starting with a higher base encoding, fewer partitions will be created. Having less partitions can result in faster access to a single element, because the lookup-algorithm needs to loop over fewer partitions.

In Figure 5 we can see that the optimized lookup-algorithm is significantly faster than the basic algorithm. The lookup-speed of the partitioned bit-packed vector is slower than the speed of the normal bit-packed vector, due to the higher lookup costs. The difference lies at 35 clock cycles on average. Compared to the normal bit-packed vector this is a lookup-overhead of about 35 % on average.

5.6 Column Scan

The last operation we will look at is the full column scan. This operation is essential for any analytical workload (OLAP). All items of a column will be evaluated against a specific predicate. For example: Getting all values, where the value is smaller than 10. All values in the column need to be accessed. A high scan-performance allows analytical requests to be executed more efficiently.

We have measured the execution time of a scan over 10 million elements in a vector. The elements have been dictionary encoded and the partitioned bit-packed vector has built its partitions dynamically. Then an "Equals-Scan" is performed, where the vector is searched for all elements of a specific search-value. The field code of the search-value is calculated and every value of the vector is checked against the field code. After the first lookup no further interaction with the dictionary is needed.

In Figure 5.6 the results of this micro-benchmark are shown. We can see that there is little performance difference during scans between the normal bit-packed vector and the partitioned bit-packed vector. This can be explained through the fact that the scan over the partitioned vector is made up of multiple scans over normal bit-packed vectors. The overhead of iterating over these sub-vectors is relatively small compared to the runtime of the scans over the sub-vectors. We expect further improvements of the partitioned vector scan when the mememory bus is saturated, due to the decreased memory bandwidth utilization. We will evaluate this in future work.

5.7 Compression Benefit

The compression benefit of the partitioned bit-packed vector is the ratio of the memory-space used by the regular bit-packed vector to the memory-space of the partitioned bit-packed vector.

$$compr = \frac{\text{size of regular bit-packed vector}}{\text{size of partitioned bit-packed vector}} - 1.0 \quad (3)$$

This relative compression benefit depends on the number of distinct values. In Figure 6 the relative benefit for a varying number of distinct values is shown. A dataset of 10 million elements was used. The spikes in the compression ratio are located after exceeding a power of 2. This is because when a power of 2 is only slightly exceeded, the regular bit-packed vector needs to allocate one additional bit for all of its elements. The partitioned bit-packed vector only allocates one additional bit for the elements that are inserted after the number of distinct values exceeds this power of 2.

6. CONCLUSION

We have evaluated the partitioned bit-packed vector as primary column storage datastructure for an in-memory column-store database.

The partitioned vector is designed for use in combination with unsorted dictionary encoding. We showed its benefits compared to regular bit-packed vectors for the most common database operations. The major benefit is that since it can deal with multiple encoding lengths, it is more flexible than the unpartitioned vector, thereby offering the possibility to avoid the merge process [4] altogether.

For now it is hard to say, whether this data structure can replace the bit-packed vector in a database system. But the data structure should be further explored. In our test environment the partitioned vector performed well. Further work includes testing with additional operations (e.g. join) and in an actual database workload scenario. The data structure has the capabilities of improving the overall performance of our database system, because it eliminates the need for a differential store, as updates and inserts can be handled directly.

Modern optimization techniques to improve the perfor-

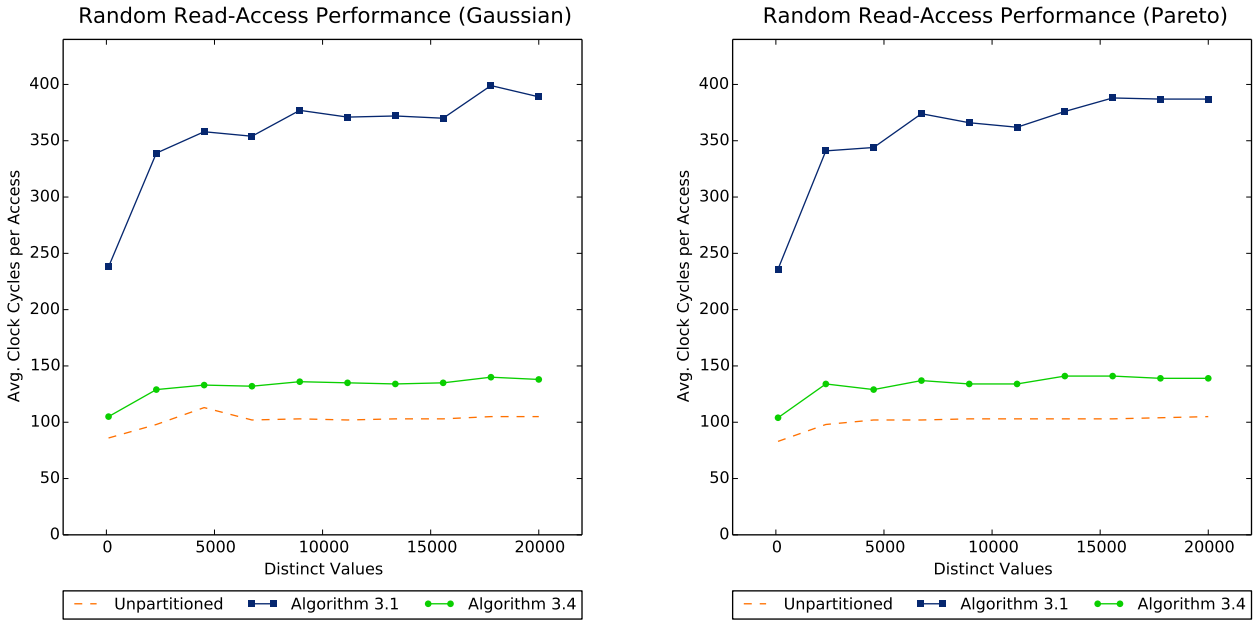


Figure 5: Comparison of partitioned bit-packed vector implementations to a normal bit-packed vector on single access performance. Algorithm 3.1 refers to the basic access algorithm in Listing 1. Algorithm 3.4 refers to the optimized algorithm in Listing 4.

mance need to be implemented. This includes better exploitation of instruction level parallelism (e.g. through branch-free algorithms). Also the utilization of data level parallelism (e.g. with SIMD) can increase performance.

To conclude, this data structure has the capability of improving the overall performance of current in-memory database systems. The results of this evaluation suggest that exploring the partitioned bit-packed vector in more detail can be beneficial for the future development of in-memory column-stores.

7. REFERENCES

[1] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database. *ACM SIGMOD Record*, 40(4):45–51, Jan. 2012.

[2] M. Grund, J. Krueger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE—A Main Memory Hybrid Storage Engine. *VLDB '10*, 2010.

[3] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Mühe. Hyper: Adapting columnar main-memory data management for transactional AND query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[4] J. Krueger, 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.

[5] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, Oct. 1980.

[6] S. Manegold, P. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. page 27, Mar. 2002.

[7] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *ACM Sigmod Records*, pages 1–8, June 2009.

[8] H. Plattner. SanssouciDB: An In-Memory Database for Processing Enterprise Workloads. In T. Härder, W. Lehner, B. Mitschang, H. Schöning, and H. Schwarz, editors, *BTW*, volume 180 of *LNI*, pages 2–21. GI, 2011.

[9] H. Plattner, A. Zeier, J. Schaffner, Y. Boshmaf, T. Willhalm, and N. Popovici. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units . *VLDB '09*, pages 1–9, Mar. 2009.

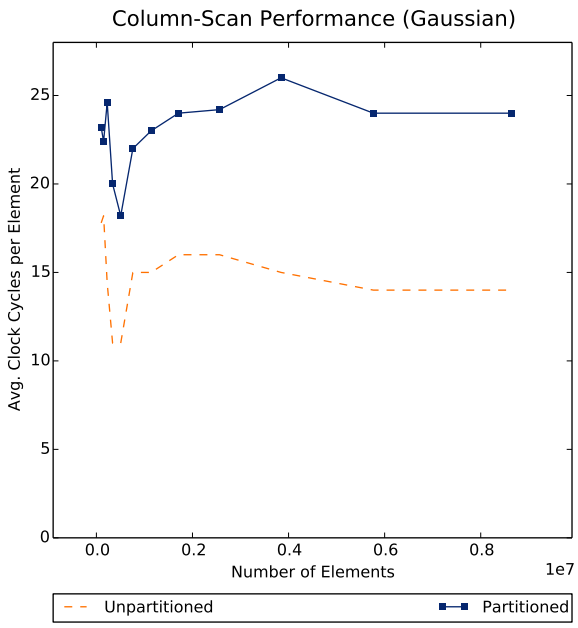
[10] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*,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

[11] R. Saucier. Computer generation of statistical distributions. Technical report, DTIC Document, 2000.

[12] D. Schwalb, M. Faust, J. Krueger, and H. Plattner. Physical Column Organization in In-Memory Column Stores. pages 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] M. Stonebraker et al. C-store: A column-oriented dbms. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *VLDB*, pages 553–564. ACM, 2005.

[14] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel,



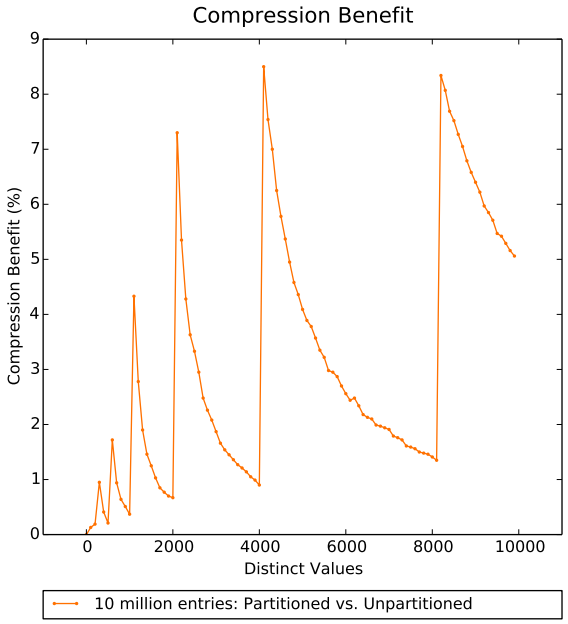


Figure 6: Compression benefit of partitioned vector compared to unpartitioned

editors, *Parallel Tools Workshop*, pages 157–173. Springer, 2009.

[15] E. Union et al. *The 2012 Ageing Report: Underlying Assumptions and Projection Methodologies*, chapter Overall results of the EUROPOP2010 population projection, pages 51–56. European economy. Office for Official Publications of the European Communities, 2011.

[16] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[17] 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. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.