Fast Column Scans: Paged Indices for In-Memory Column Stores

Martin Faust(✉), David Schwalb, and Jens Krueger

Hasso Plattner Institute, University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{martin.faust,david.schwalb,jens.krueger}@hpi.de

Abstract. Commodity hardware is available in configurations with huge amounts of main memory and it is viable to keep large databases of enterprises in the RAM of one or a few machines. Additionally, a reunification of transactional and analytical systems has been proposed to enable operational reporting on the most recent data. In-memory column stores appeared in academia and industry as a solution to handle the resulting mixed workload of transactional and analytical queries. Therein queries are processed by scanning whole columns to evaluate the predicates on non-key columns. This leads to a waste of memory bandwidth and reduced throughput.

In this work we present the Paged Index, an index tailored towards dictionary-encoded columns. The indexing concept builds upon the availability of the indexed data at high speeds, a situation that is unique to in-memory databases. By reducing the search scope we achieve up to two orders of magnitude of performance increase for the column scan operation during query runtime.

1 Introduction

Enterprise systems often process a read-mostly workload [5] and consequently in-memory columns stores tailored towards this workload hold the majority of table data in a read-optimized partition [9]. To apply predicates, this partition is scanned in its compressed form through the intensive use of the SIMD units of modern CPUs. Although this operation is fast when compared to disk-based systems, its performance can be increased if we decrease the search scope and thereby the amount of data that needs to be streamed from main memory to the CPU. The resulting savings of memory bandwidth lead to a better utilization of this scarce resource, which allows to process more queries with equally sized machines.

2 Background and Prior Work

In this section we briefly summarize our prototypical database system, the used compression technique and refer to prior work.

A. Jagatheesan et al. (Eds.): IMDM 2013/2014, LNCS 8921, pp. 15–27, 2015.
DOI: 10.1007/978-3-319-13960-9_2

2.1 Column Stores with a Read-Optimized Partition

Column stores are in the focus of research [10–12], because their performance characteristics enable superior analytical (OLAP) performance, while keeping the data in-memory still allows a sufficient transactional performance for many usecases. Consequently, Plattner [6] proposed, that in-memory column stores can handle a mixed workload of transactional (OLTP) and analytical queries and become the single source of truth in future enterprise applications.

Dictionary Compressed Column. Our prototypical implementation stores all table data vertically partitioned in dictionary compressed columns. The values are represented by bit-packed value-ids, which reference the actual, uncompressed values within a sorted dictionary by their offset. Dictionary compressed columns can be found in HYRISE [3], SanssouciDB [7] and SAP HANA [9].

Enterprise Data. As shown by Krueger et al. [5], enterprise data consists of many sparse columns. The domain of values is often limited, because there is a limited number of underlying options in the business processes. For example, only a relatively small number of customers, appears in the typically large order table. Additionally, data within some columns often correlates in regard to its position. Consider a column storing the *promised delivery date* in the *orders* table. Although the dates will not be ordered, because different products will have different delivery time spans, the data will follow a general trend. In this work, we want to focus on columns that exhibit such properties.

Related Work. Important work on main-memory indices has been done by Rao and Ross [8], but their indexing method applies to the value-id lookup in sorted dictionaries rather then the position lookup that we will focus on in this paper. Since they focus on Decision Support Systems (DSS), they claim that an index rebuild after every bulk-load is viable. In this paper we assume a mixed-workload system, where the merge-performance must be kept as high as possible, hence we reuse the old index to build an updated index.

Idreos et al. [4] present indices for in-memory column stores that are build during query execution, and adapt to changing workloads, however the integration of the indexing schemes into the frequent merge process of the write-optimized and read-only store is missing.

Graefe [2] evaluates a related indexing techniques, zone indexes with bit vector filters, in the context of row-oriented data warehouses.

In previous work, we presented the Group-Key Index, which implements an inverted index on the basis of the bit-packed value-id and showed that this index allows very fast lookups while introducing acceptable overhead to the partition-combining process [1].

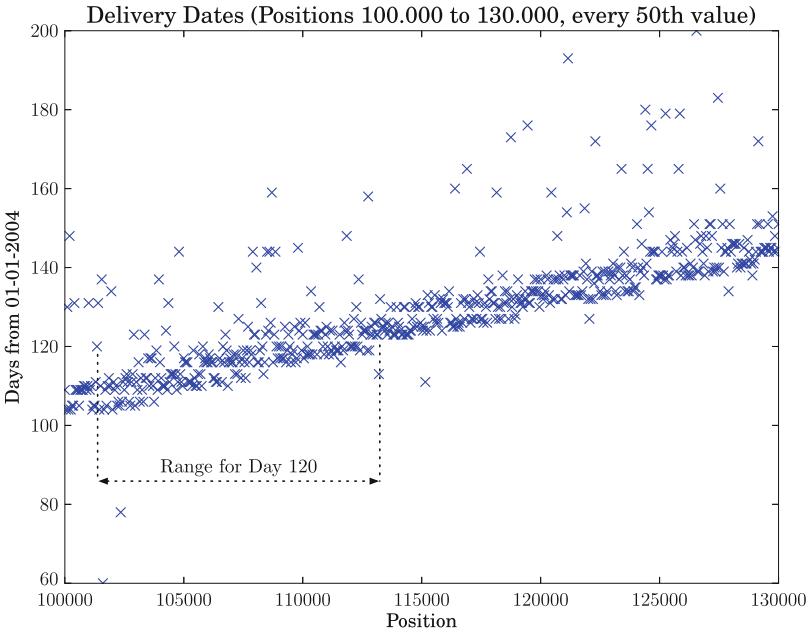

Fig. 1. Example for a strongly clustered column, showing delivery Dates from a productive ERP system. The values follow a general trend, but are not strictly ordered. The range for value 120 is given as an example.

2.2 Paper Structure and Contribution

In the following section we introduce our dictionary-compressed, bit-packed column storage scheme and the symbols that are used throughout the paper (Table 1). In Sect. 4 the Paged Index is presented. We explain its structure, give the memory traffic for a single lookup, and show the index rebuild algorithm. A size overview for exemplary configurations and the lookup algorithm is given as well. Afterwards, in Sect. 5, the column merge algorithm is shown, and extended in Sect. 6 to enable the index maintenance during the column merge process. In Sect. 7, we present the performance results for two index configurations. Findings and contributions are summed up in Sect. 9.

3 Bit-Packed Column Scan

We define the attribute vector $\mathbf{V}^j_\mathbf{M}$ to be a list of value-ids, referencing offsets in the sorted dictionary $\mathbf{U}^j_\mathbf{M}$ for column j. Values within $\mathbf{V}^j_\mathbf{M}$ are bit-packed with the minimal amount of bits necessary to reference the entries in $\mathbf{U}^j_\mathbf{M}$, we refer to the amount of bits with $\mathbf{E}^j_\mathbf{C} = \lceil \log_2(|\mathbf{U}^j_\mathbf{M}|) \rceil$ bits.

Consequently, to apply a predicate on a single column, the predicate conditions have to be translated into value-ids by performing a binary search on the

Table 1. Symbol definition. Entities annotated with ′ represent the merged (updated) entry.

Description	Unit	Symbol
Number of columns in the table	-	$\mathbf{N}_C$
Number of tuples in the main/delta partition	-	$\mathbf{N}_M$,$\mathbf{N}_D$
Number of tuples in the updated table	-	$\mathbf{N}'_M$
For a given column $j; j \in [1 \dots \mathbf{N}_C]$:		
Main/delta partition of the j^{th} column	-	$\mathbf{M}^j$,$\mathbf{D}^j$
Merged column	-	$\mathbf{M}'^j$
Attribute vector of the j^{th} column	-	$\mathbf{V}^j_\mathbf{M}$,$\mathbf{V}^j_\mathbf{D}$
Updated main attribute vector	-	$\mathbf{V}'^j_\mathbf{M}$
Sorted dictionary of $\mathbf{M}^j/\mathbf{D}^j$	-	$\mathbf{U}^j_\mathbf{M}$,$\mathbf{U}^j_\mathbf{D}$
Updated main dictionary	-	$\mathbf{U}'^j_\mathbf{M}$
CSB+ Tree Index on $\mathbf{D}^j$	-	$\mathbf{T}^j$
Compressed Value-Length	bits	$\mathbf{E}^j_\mathbf{C}$
New Compressed Value-Length	bits	$\mathbf{E}'^j_\mathbf{C}$
Length of Address in Main Partition	bits	$\mathbf{A}^j$
Fraction of unique values in $\mathbf{M}^j/\mathbf{D}^j$	-	$\lambda^j_\mathbf{M}$,$\lambda^j_\mathbf{D}$
Auxiliary structure for $\mathbf{M}^j$ / $\mathbf{D}^j$	-	$\mathbf{X}^j_\mathbf{M}$,$\mathbf{X}^j_\mathbf{D}$
Paged Index	-	$\mathbf{I}^j_M$
Paged Index Pagesize	-	$\mathbf{P}^j$
Number of Pages	-	g
Memory Traffic	bytes	MT

main dictionary $\mathbf{U}^j_\mathbf{M}$ and a scan of the main attribute vector $\mathbf{V}^j_\mathbf{M}$. Of importance is here the scanning of $\mathbf{V}^j_\mathbf{M}$, which involves the read of MT_{CS} bytes from main memory, as defined in Eq. 1.

$$MT_{CS} = \mathbf{N}_M \cdot \frac{\mathbf{E}^j_C}{8} = \mathbf{N}_M \cdot \frac{\lceil \log_2(|\mathbf{U}^j_\mathbf{M}|) \rceil}{8} \text{ bytes} \tag{1}$$

Inserts and updates to the compressed column are handled by a delta partition, thereby avoiding to re-encode the column for each insert [5]. The delta partition is stored uncompressed and extended by a CSB+ tree index to allow for fast lookups. If the delta partition reaches a certain threshold it is merged with the main partition. This process and the extension to update the Paged Index will be explained in detail in Sect. 5.

4 Paged Index

While indices in classic databases are well studied and researched, the increase of access speed to data for in-memory databases allows to rethink indexing techniques. Now, that the data in columnar in-memory stores can be accessed at the speed of RAM, it becomes possible to scan the complete column to evaluate queries - an operation that is prohibitively slow on disk for huge datasets (Fig. 2).

We propose the Paged Index, which benefits from clustered value distributions and focuses on reducing the memory traffic for the scan operation, while adding as little overhead as possible to the merge process for index maintenance. Additionally the index uses only minimal index storage space and is built for a mixed workload. Figure 1 shows an example of real ERP customer data, outlining delivery dates from a productive system. Clearly, the data follows a strong trend and consecutive values are only from a small value domain with a high spatial locality. Consequently, the idea behind a Paged Index is to partition a column into pages and to store bitmap indices for each value, reflecting in which pages the respective value occurs in. Therefore, scan operators only have to consider pages that are actually containing the value, which can drastically reduce the search space.

4.1 Index Structure

To use the Paged Index, the column is logically split into multiple equally sized pages. The last page is allowed to be of smaller size. Let the pagesize be $\mathbf{P}^j$, then

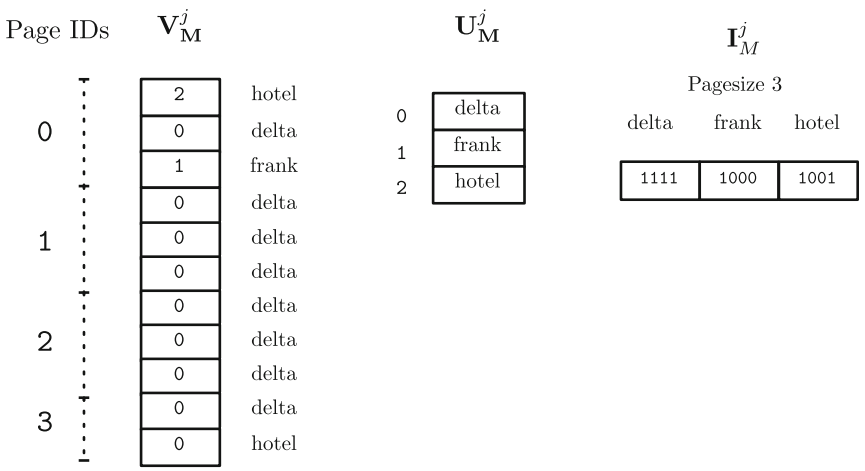

Fig. 2. An example of the Paged Index for $\mathbf{P}^j = 3$

$\mathbf{M}^j$ contains $g = \lceil \frac{\mathbf{N}_M}{\mathbf{P}^j} \rceil$ pages. For each of the encoded values in the dictionary $\mathbf{U}^j_\mathbf{M}$ now a bitvector $\mathbf{B}^j_v$ is created, with v being the value-id of the encoded value, equal to its offset in $\mathbf{U}^j_\mathbf{M}$. The bitvector contains exactly one bit for each page.

$$\mathbf{B}^j_v = (b_0, b_1 ... b_g) \tag{2}$$

Each bit in $\mathbf{B}^j_v$ marks whether value-id v can be found within the subrange represented by that page. To determine the actual tuple-id of the matching values, the according subrange has to be scanned. If b_x is set, one or more occurrences of the value-id can be found in the attribute vector between offset $x * \mathbf{P}^j$ (inclusive) and $(x+1) * \mathbf{P}^j$ (exclusive) as represented by Eq. 3. The Paged Index is the set of bitvectors for all value-ids, as defined in Eq. 4.

$$b_x \in \mathbf{B}^j_v : b_x = 1 \Leftrightarrow v \in \mathbf{V}^j_\mathbf{M}[x \cdot \mathbf{P}^j ... ((x+1) \cdot \mathbf{P}^j - 1)] \tag{3}$$

$$I_M = \left[\mathbf{B}^j_0, \mathbf{B}^j_1, ..., \mathbf{B}^j_{|\mathbf{U}^j_\mathbf{M}|-1} \right] \tag{4}$$

4.2 Index Size Estimate

The Paged Index is stored in one consecutive bitvector. For each distinct value and each page a bit is stored. The size in bits is given by Eq. 5. In Table 2 we show the resulting index sizes for some exemplary configurations.

$$\mathbf{s}(\mathbf{I}^j_M) = |\mathbf{U}^j_\mathbf{M}| * \lceil \frac{\mathbf{N}_M}{\mathbf{P}^j} \rceil \text{ bits} \tag{5}$$

Table 2. Example sizes of the Paged Index

$\mathbf{N}_M$	$\lvert\mathbf{U}^j_\mathbf{M}\rvert$	$\mathbf{P}^j$	$\mathbf{s}(\mathbf{I}^j_M)$	$\mathbf{s}(\mathbf{V}^j_\mathbf{M})$
100,000	10	4096	32 Byte	49 K
100,000	10	65536	3 Byte	49 K
100,000	100,000	4096	310 K	208 K
100,000	100,000	65536	31 K	208 K
1,000,000,000	10	4096	298 K	477 M
1,000,000,000	10	65536	19 K	477 M
1,000,000,000	100,000	4096	3 G	2 G
1,000,000,000	100,000	65536	182 M	2 G

4.3 Index Enabled Lookups

If no index is present to determine all tuple-ids for a single value-id, the attribute vector $\mathbf{V}^j_\mathbf{M}$ is scanned from the beginning to the end and each compressed value-id is compared against the requested value-id. The resulting tuple-ids, which equal to the position in $\mathbf{V}^j_\mathbf{M}$, are written to a dynamically allocated results vector. With the help of the Paged Index the scan costs can be minimized by evaluating only relevant parts of $\mathbf{V}^j_\mathbf{M}$.

Algorithm 1. Scanning the Column with a Paged Index

```
1: procedure PagedIndexScan (valueid)
2:     bitsPerRun = |I^j_M| / |U^j_M|
3:     results = vector < uint >
4:     for page = 0; page ≤ bitsPerRun; ++page do
5:         if I^j_M[bitsPerRun * valueid + page] == 1 then
6:             startOffset = page * P^j
7:             endOffset = (page + 1) * P^j
8:             for position = startOffset; position < endOffset; ++position do
9:                 if V^j_M[position] == valueid then
10:                    results.pushback(position)
11:                end if
12:            end for
13:        end if
14:    end for
15:    return results
16: end procedure
```

Our evaluated implementation additionally decompresses multiple bit-packed values at once for maximum performance. Algorithm 1 shows the simplified implementation. The minimum memory traffic of an index-assisted partial scan of the attribute vector for a single value-id is given by Eq. 7.

$$minPagesPerDistinctValue = \left\lceil \frac{\mathbf{N}_M}{\mathbf{P}^j * |\mathbf{U}^j_\mathbf{M}|} \right\rceil \tag{6}$$

$$MT_{PagedIndex} = \left\lceil \frac{\mathbf{N}_M}{\mathbf{P}^j \cdot 8} \right\rceil + \left\lceil \frac{\mathbf{N}_M}{\mathbf{P}^j \cdot |\mathbf{U}^j_\mathbf{M}|} \right\rceil \cdot \frac{\mathbf{P}^j \cdot \mathbf{E}^j_C}{8} \text{ bytes} \tag{7}$$

4.4 Rebuild of the Index

To extent an existing compressed column with an index, the index has to be built. Additionally, a straightforward approach to enable index maintenance for the merge of the main and delta partition is to rebuild the index after a new, merged main partition has been created. Since all operations are in-memory, Rao et al. [8] claim that for bulk-operations an index rebuild is a viable choice. We take the rebuild as a baseline for further improvements.

5 Column Merge

Our in-memory column store maintains two partitions for each column: a read-optimized, compressed main partition and a writable delta partition. To allow for fast queries on the delta partition, it has to be kept small. To achieve this, the delta partition is merged with the main partition after its size has increased beyond a certain threshold. As explained in [5], the performance of this merge process is paramount to the overall sustainable insert performance. The inputs to the algorithm consists of the compressed main partition and the uncompressed delta partition with an CSB+ tree index [8]. The output is a new dictionary encoded main partition.

The algorithm is the basis for our index-aware merge process that will be presented in the next section.

We perform the merge using the following two steps:

1. **Merge Main Dictionary and Delta Index, Create value-ids for $\mathbf{D}^j$.** We simultaneously iterate over $\mathbf{U}^j_\mathbf{M}$ and the leafs of $\mathbf{T}^j$ and create the new sorted dictionary $\mathbf{U}'^j_\mathbf{M}$ and the auxiliary structure $\mathbf{X}^j_\mathbf{M}$. Because $\mathbf{T}^j$ contains a list of all positions for each distinct value in the delta partition of the column, we can set all positions in the value-id vector $\mathbf{V}^j_\mathbf{D}$. This leads to non-continuous access to $\mathbf{V}^j_\mathbf{D}$. Note that the value-ids in $\mathbf{V}^j_\mathbf{D}$ refer to the new dictionary $\mathbf{U}'^j_\mathbf{M}$.
2. **Create New Attribute Vector.** This step consists of creating the new main attribute vector $\mathbf{V}'^j_\mathbf{M}$ by concatenating the main and delta partition's attribute vectors $\mathbf{V}^j_\mathbf{M}$ and $\mathbf{V}^j_\mathbf{D}$. The compressed values in $\mathbf{V}^j_\mathbf{M}$ are updated by a lookup in the auxiliary structure $\mathbf{X}^j_\mathbf{M}$ as shown in Eq. 8. Values from $\mathbf{V}^j_\mathbf{D}$ are copied without translation to $\mathbf{V}'^j_\mathbf{M}$. The new attribute vector $\mathbf{V}'^j_\mathbf{M}$ will contain the correct offsets for the corresponding values in $\mathbf{U}'^j_\mathbf{M}$, by using $\mathbf{E}'^j_C$ bits-per-value, calculated as shown in Eq. 9.

$$\mathbf{V}'^j_\mathbf{M}[i] = \mathbf{V}^j_\mathbf{M}[i] + \mathbf{X}^j_\mathbf{M}[\mathbf{V}^j_\mathbf{M}[i]] \quad \forall i \in [0...\mathbf{N}_M - 1] \tag{8}$$

Algorithm 2. Rebuild of Paged Index

```
1: procedure REBUILD PAGED INDEX
2:     bitsPerRun = (N_M + P^j - 1) / P^j
3:     I^j_M[0...(bitsPerRun * |U^j_M|)] = 0
4:     for pos = 0; pos ≤ N_M; ++pos do
5:         valueid = V^j_M[pos]
6:         run = valueid * bitsPerRun
7:         page = pos / P^j
8:         I^j_M[run + page] = 1
9:     end for
10: end procedure
```

Note that the optimal amount of bits-per-value for the bit-packed $\mathbf{V}'^{j}_{\mathbf{M}}$ can only be evaluated after the cardinality of $\mathbf{U}^{j}_{\mathbf{M}} \cup \mathbf{D}^{j}$ is determined. If we accept a non-optimal compression, we can set the compressed value length to the sum of the cardinalities of the dictionary $\mathbf{U}^{j}_{\mathbf{M}}$ and the delta CSB+ tree index $\mathbf{T}^{j}$. Since the delta partition is expected to be much smaller than the main partition, the difference from the optimal compression is low.

$$\mathbf{E}'^{j}_{C} = \lceil \log_2(|\mathbf{U}^{j}_{\mathbf{M}} \cup \mathbf{D}^{j}|) \rceil \leq \lceil \log_2(|\mathbf{U}^{j}_{\mathbf{M}}| + |\mathbf{T}^{j}|) \rceil \tag{9}$$

Step 1's complexity is determined by the size of the union of the dictionaries and the size of the delta partition. Its complexity is $\mathcal{O}(|\mathbf{U}^{j}_{\mathbf{M}} \cup \mathbf{U}^{j}_{\mathbf{D}}| + |\mathbf{D}^{j}|)$. Step 2 is dependent on the length of the new attribute vector, $\mathcal{O}(\mathbf{N}_M + \mathbf{N}_D)$.

6 Index-Aware Column Merge

We now integrate the index rebuild into the column merge process. This allows us to reduce the memory traffic and create a more efficient algorithm to merge columns with a Paged Index.

Algorithm 3. Extended Dictionary Merge

```
1: procedure ExtendedDictionaryMerge
2:     d, m, n = 0
3:     g = ⌈ N_M / P^j ⌉ (Number of Pages)
4:     while d != |T^j| or m != |U^j_M|  do
5:         processM = (U^j_M[m] <= T^j[d] or d == |T^j|)
6:         processD = (T^j[d] <= U^j_M[m] or m == |U^j_M|)
7:         if processM then
8:             U'^j_M[n] ← U^j_M[m]
9:             X^j_M[m] ← n − m
10:            I'_M[n ∗ g ··· n ∗ (g + 1)] = I_M[m ∗ g ··· m(g + 1)]
11:            m ← m + 1
12:        end if
13:        if processD then
14:            U'^j_M[n] ← T^j[d]
15:            for dpos in T^j[d].positions do
16:                V'^j_D[dpos] = n
17:                I'^j_M[n ∗ (|V^j_M|+|V^j_D|)/P^j + |V^j_M|+dpos/P^j] = 1
18:            end for
19:            d ← d + 1
20:        end if
21:        n ← n + 1
22:    end while
23: end procedure
```

We extend Step 1 of the column merge process from Sect. 5 to maintain the Paged Index. During the dictionary merge we perform additional steps for each processed dictionary entry. The substeps are extended as follows:

1. **For Dictionary Entries from the Main Partition.** Calculate the begin and end offset in $\mathbf{I}_M^j$ and the starting offset in $\mathbf{I}_M^{j\prime}$. Copy the range from $\mathbf{I}_M^j$ to $\mathbf{I}_M^{j\prime}$. The additional bits in the run are left zero, because the value is not present in the delta partition.
2. **For CSB+ Index Entries from the Delta Partition.** Calculate the position of the run in $\mathbf{I}_M^{j\prime}$, read all positions from $\mathbf{T}^j$, increase them by $\mathbf{N}_M$, and set the according bits in $\mathbf{I}_M^{j\prime}$.
3. **Entries found in both Partitions.** Perform both steps sequentially.

Algorithm 3 shows a modified dictionary merge algorithm to maintain the paged index during the column merge.

7 Evaluation

We evaluate our Paged Index on a clustered column. In a clustered column equal data entries are grouped together, but the column is not necessarily sorted by the value. Our index does perform best, if each value's occurrences form exactly one group, however it is not required. Outliers or multiple groups are supported by the Paged Index.

With the help of the index the column scan is accelerated by scanning only the pages which are known to have at least one occurrence of the desired value.

The benchmarks were performed on a two socket Intel Xeon X5650 system with 48 GB of RAM. In Fig. 3 the CPU cycles for the column scan and two configurations of the Paged Index are shown. We choose pagesizes of 4096 and 16384 entries as an example. The Paged Index enables an performance increase of two orders of magnitude for columns with a medium to high amount of distinct values through a drastic reduction of of the search scope. For smaller dictionaries,

Table 3. Example sizes of the evaluated Paged Index

$\mathbf{N}_M$	$\lvert\mathbf{U}_\mathbf{M}^j\rvert$	$\mathbf{P}^j$	$\mathbf{s}(\mathbf{I}_M^j)$	$\mathbf{s}(\mathbf{V}_\mathbf{M}^j)$
3,000,000	10	4096	917 byte	1.4 M
3,000,000	10	65536	58 byte	1.4 M
3,000,000	100,000	4096	8.7 M	6.1 M
3,000,000	100,000	65536	571.0 K	6.1 M
3,000,000	1,000,000	4096	87.4 M	7.2 M
3,000,000	1,000,000	65536	5.6 M	7.2 M
3,000,000	3,000,000	4096	262.3 M	7.9 M
3,000,000	3,000,000	65536	16.7 M	7.9 M

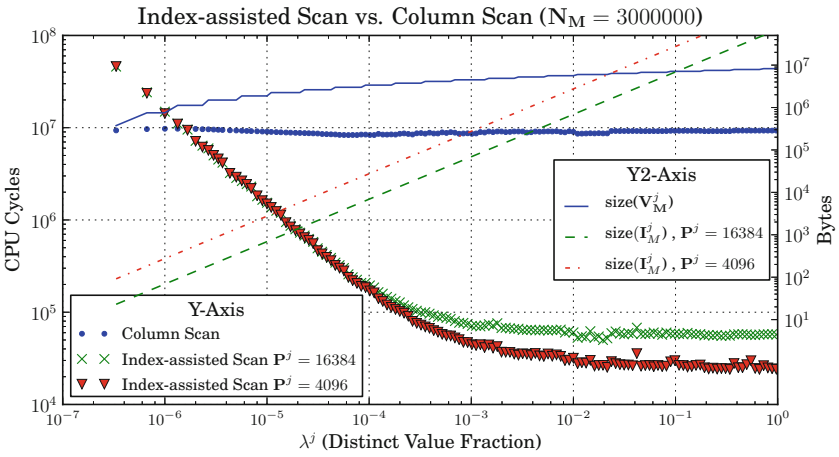

Fig. 3. Scan performance and index sizes in comparison

the benefit is lower. However an order of magnitude is already reached with $\lambda^j = 10^{-5}$, which corresponds to 30 distinct values in our example. For very small dictionaries with less than 5 values, the overhead of reading the Paged Index leads to a performance decrease. In these cases the Paged Index should not be applied to a column. In Table 3 the index and attribute vector sizes for some of the measured configurations are given. The Paged Index can deliver its performance increase for columns with a medium amount of distinct values for only little storage overhead. For the columns with a very high distinct value count the Paged Index grows prohibitively large. Note, that the storage footprint halves by each doubling of the pagesize. For the aforementioned delivery dates column the Paged Index decreases the scan time for a specific value-id by a factor 20.

8 Future Work

The current design of a bit-packed attribute vector does not allow a fixed mapping of the resulting sub-ranges to memory pages. In future work we want to compare the performance benefits if a attribute vector is designed, so that the reading of a sub-range leads to at most one transaction lookaside buffer (TLB) miss.

Other interesting topics include the automatic determination of the best page size, index compression and varying page sizes.

9 Conclusion

Shifted access speeds in main memory databases and special domain knowledge in enterprise systems allow for a reevaluation of indexing concepts. With the

original data available at the speed of main memory, indices do not need to narrow down the search scope as far as in disk based databases. Therefore, relatively small indices can have huge impacts, especially if they are designed towards a specific data distribution.

In this paper, we proposed the Paged Index, which is tailored towards columns with clustered data. As our analyses of real customer data showed, such data distributions are especially common in enterprise systems. By indexing the occurrence of values on a block level, the search scope for scan operations can be reduced drastically with the use of a Paged Index. In our experimental evaluation, we report speed improvements up to two orders of magnitude, while only adding little overhead for the index maintenance and storage. Finally, we proposed an integration of the index maintenance into the merge process, further reducing index maintenance costs.

References

1. Faust, M., Schwalb, D., Krueger, J., Plattner, H.: Fast lookups for in-memory column stores: Group-key indices, lookup and maintenance. In: ADMS'2012, pp. 13–22 (2012)
2. Graefe, G.: Fast loads and fast queries. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 111–124. Springer, Heidelberg (2009)
3. Grund, M., Krueger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: HYRISE-a main memory hybrid storage engine. Proc. VLDB Endowment **4**(2), 105–116 (2010)
4. Idreos, S., Manegold, S., Kuno, H., Graefe, G.: Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. Proc. VLDB Endowment **4**(9), 586–597 (2011)
5. Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., Zeier, A.: Fast updates on read-optimized databases using multi-core CPUs. Proc. VLDB Endowment **5**(1), 61–72 (2011)
6. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD '09 Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 1–8 (2009)
7. Plattner, H., Zeier, A.: In-Memory Data Management: An Inflection Point for Enterprise Applications. Springer, Heidelberg (2011)
8. Rao, J., Ross, K.: Cache conscious indexing for decision-support in main memory. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 78–89 (1999)
9. SAP-AG. The SAP HANA database-an architecture overview. Data Engineering (2012)
10. Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.: C-store: a column-oriented DBMS. In: Proceedings of the 31st International Conference on Very large data bases, pp. 553–564 (2005)

11. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endowment **2**(1), 385–394 (2009)
12. Zukowski, M., Boncz, P., Nes, N., Heman, S.: MonetDB/X100-A DBMS in the CPU cache. IEEE Data Eng. Bull. **28**(2), 17–22 (2005)

http://www.springer.com/978-3-319-13959-3

In Memory Data Management and Analysis
First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hongzhou, China, September 1, 2014, Revised Selected Papers
Jagatheesan, A.; Levandoski, J.; Neumann, Th.; Pavlo, A. (Eds.)
2015, VIII, 151 p. 57 illus., Softcover
ISBN: 978-3-319-13959-3