Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance.

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

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

Jens Krueger
Hasso-Plattner-Institute
Potsdam, Germany
jens.krueger@hpi.uni-potsdam.de

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

ABSTRACT

In-memory column-oriented databases have become a major topic of interest in academia and commercial applications. The demand for analytics on up-to-the-minute data and the availability of systems with hundreds of gigabytes of main memory led to the proposal of combined systems, which provide a single database for operational processing and ad-hoc analytical queries on current data. Recent research has identified In-Memory Column-Stores as a possible database architecture to meet these requirements. They are claimed to be capable of delivering the analytical insights while providing sufficient transactional performance. Data therein is typically split up into a write-optimized partition, which gains speed from its small size and tree-structured indices, and a larger read-only partition. To enable fast transactional and analytical performance, an index on the large, read-only partition is advisable in many cases. In this paper we present an index structure for the read-only partition, describe its advantage over the column scan and present an algorithm for the maintenance of the index. The index drastically reduces the memory traffic during query execution, leading to faster lookups and joins, thereby providing benefits to transactional and analytical processing.

We analyze the memory traffic of index lookups in comparison with full column scans and the maintenance of the index structure. We develop formulas to determine the viability of an index lookup over a column scan at query runtime. While other research claimed that an index for in-memory systems should just be rebuild after every bulk-load, we show that a substantial performance increase can be achieved by reusing the former index to create an updated index.

1. INTRODUCTION

In-Memory column stores such as Sanssouci DB [6] and HYRISE [3] answer queries by scanning the whole column to apply a predicate. Although this operation is fast in comparison with disk based systems, since the data resides in main memory, it wastes memory bandwidth and processing power. We can decrease the response time and memory traffic by using a special inverted index, which we will call the Group-Key index, to replace column scans with index lookups.

Enterprise workload consists of many scan operations with a selectivity that favors an index, a speed-up thereof consequently leads to a better overall system responsiveness and decreases system utilization, which will allows to process more queries with an equally sized machine.

While most indexing schemes are built for frequently updated data partitions, many in-memory column store designs hold the majority of the data in a dictionary encoded read-only partition. Consequently, directly updatable indexing schemes, such as tree structures, consume more storage than necessary.

The Group-Key index increases the query performance, by providing a mapping to all positions for each value. The Group-Key index is only applied to the read-only part, and does not decrease the peak-insert performance [5], because index maintenance is only performed, when the writable partition is integrated into the read-optimized main partition.

In this paper we will explain the advantage of the index at runtime, and integrate the index maintenance into the creation of the read-only partition, thereby reducing the overhead of index rebuilding significantly.

All symbols that are used throughout the paper can be found in Table 1.

1.1 Background and Prior Work

We briefly summarize the database system, the used compression technique and refer to prior work.

1.1.1 Column Stores with Write-Optimized Partition

Column stores have been in the focus of research [9, 10, 11], because their performance characteristics make them a

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 $\mathbf{D}^j$	-	$\mathbf{T}^j$
Uncompressed Value-Length	bytes	$\mathbf{E}^j$
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 $\mathbf{M}^j$ / $\mathbf{D}^j$	-	$\mathbf{X}^j_\mathbf{M}$,$\mathbf{X}^j_\mathbf{D}$
Index Offsets / Postings	-	$\mathbf{I}^j$,$\mathbf{P}^j$
Cache Line size (typical: 64 bytes)	bytes	L
Memory Traffic	bytes	MT

Table 1: Symbol Definition. Entities annotated with $'$ represent the merged (updated) entry.

good fit for analytical (OLAP) processing.

Other researchers have proposed, that these systems can handle a mixed workload of transactional (OLTP) and analytical queries [6], and become the combined data source for systems that handle OLTP and OLAP workloads.

Hardware development, especially the availability of large main memory systems at economical prices, has enabled database vendors to construct systems that keep all data in memory, by using dictionary compressed [6] columns. Values are represented by bit-packed value-ids which reference the uncompressed values in a sorted dictionary, so the domain of each column is stored in the dictionary, and shorter value-ids are stored in the attribute vector.

To avoid inserting into the ordered dictionary and the costly re-encode of the bit-packed attribute vector, these systems typically split a table into a compressed read-only and a writable part [3, 6, 9]. The writable part, here called delta partition, is combined with the read-only main partition from time to time, to reduce the memory footprint, and keep the writable part small and fast. This merge process is covered in detail in Section 4.

1.1.2 Lightweight Bit-Packing

Data in main memory can be compressed to reduce the memory footprint and the memory traffic. If the overhead of decompression is smaller than the benefits gained through bandwidth savings, the overall system performance can increase through the use of compression. In this paper we use bit-packing [10] to encode the attribute vector and our index structures. Bit-packing encodes positive integers of a certain range in the minimum amount of bits that are needed to express the largest element of the range, e.g. let x be the rightmost element of the range, then each element of the range can be encoded in $\lceil \log_2 x \rceil$ bits.

Scanning, compression and decompression of bit-packed vectors can also gain performance through the use of the SIMD units of current CPUs [10]. With the recent increase of the SIMD-register size to 256 bits as part of the *Advanced Vector Extensions* we can expect further improvements in the performance of scanning, compression and decompression functions on bit-packed values

1.1.3 Dense-, Covering-, Lookup- Indices

Classical database indices are constructed to reduce the search scope during query execution. In a row oriented database a memory-resident index on a certain column might present an ordered list of value-position pairs, that allows for binary search on the index and a direct addressing of the data on disk. Dense indices contain entries for every record of the database table. Covering indices contain enough data to answer queries directly, without consulting the original table data, which is especially helpful in low selectivity scenarios.

Transactional workloads typically select only a few tuples in each query. Analytical workloads, although often operating on the complete table and domain, have to perform joins of the fact and dimension tables, for which an index can be used. Because column stores with a dictionary encoded domain use late materialization, so that for internal processing only positions are used, a covering index is the most efficient choice. It allows the database to replace the scan operation of the column with an index lookup, and to continue processing with the returned positions lists.

In our experimental database system the search scope for a single attribute is already reduced through the usage of the domain encoded column-layout. Additionally the index and the data reside in the same level on the storage hierarchy. We therefore present a dense and covering lookup index structure, that allows to answer queries directly and has an entry for each value.

1.1.4 Related Work

Important work on main-memory indices has been done by Rao and Ross [7], but their indexing method applies to the value-id lookup in the sorted dictionary 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.

1.2 Paper Structure and Contributions

In the following section we define the Group-Key index and clarify its relation with the encoded column. We analyze the memory traffic savings during query execution. In Section 3 we present an algorithm to rebuild the index and analyze its memory traffic. It follows the definition of the column merge process in Section 4, and later on the integration of the index maintenance into the column merge. We compare the memory traffic of these algorithms in theory in Section 6,

and verify the conclusions in our prototype in the following section.

The contributions of this work include the first presentation of a Group-Key index-aware column merge process and the quantification of the performance impact.

2. GROUP-KEY INDEX

In this Section we present a dense, bit-packed Group-Key index for the main partition of an in-memory column j. The main partition contains two structures: the sorted dictionary $\mathbf{U^j_M}$ and the bit-packed attribute vector $\mathbf{V^j_M}$ of value-ids which refer to the position of the original value in the dictionary .

The Group-Key index consists of two separate structures: The offset structure $\mathbf{I}^j$ and the postings structure $\mathbf{P}^j$. $\mathbf{I}^j$ maps value-ids to offsets in $\mathbf{P}^j$, where positions in the attribute vector are recorded, at which the value-id is found. In particular, the interval in Equation 1 contains all positions for a value-id v.

$$\left[\mathbf{P}^j[\mathbf{I}^j[v]], ..., \mathbf{P}^j[\mathbf{I}^j[v+1]-1]\right] \quad (1)$$

Accordingly, all occurrences of a certain value-id can be found with the bounds from $\mathbf{I}^j$ and the resulting position list from $\mathbf{P}^j$.

The relationship between the involved structures is shown in Figure 1. As an example, the steps that are necessary to find all positions of *charlie* are marked. First, the value is translated to a value-id by performing a binary search on $\mathbf{U^j_M}$, then the according offsets in $\mathbf{I}^j$ are read (2,3), and the referenced right-open interval from $\mathbf{P}^j$ is retrieved (4). Note that the resulting list is ordered for a single-value predicate and can be streamed directly to the next operator in the query pipeline. For range queries a merging of the sorted position lists is necessary.

Both structures of the Group-Key index are bit-packed with $\mathbf{A}^j$ bits per value, which are needed to address a position in the main partition, as shown in Equation 2.

$$\mathbf{A}^j = \lceil \log_2(\mathbf{N}_M) \rceil \text{ bits} \quad (2)$$

Therefore, the sizes of $\mathbf{I}^j$ and $\mathbf{P}^j$ are given in Equations 3 and 4.

$$\textbf{sizeof}(\mathbf{I}^j) = (|\mathbf{U^j_M}| + 1) \cdot \frac{\mathbf{A}^j}{8} \text{ bytes} \quad (3)$$

$$\textbf{sizeof}(\mathbf{P}^j) = \mathbf{N}_M \cdot \frac{\mathbf{A}^j}{8} \text{ bytes} \quad (4)$$

Accordingly, after the delta and main partition are merged to form $\mathbf{M}'$, the resulting new index structure sizes depend on the size of the new main partition $\mathbf{N}'_M$ and the cardinality of the new dictionary $\mathbf{U'^j_M}$.

3. REBUILD OF THE GROUP-KEY INDEX

After the main and delta partitions have been combined to a new partition, the Group-Key index needs to be updated to reflect the changes. To obtain a Group-Key index for a column without an index, or for a newly merged main partition we run an indexing process after the execution of the merge process. The indexing process consists of three steps:

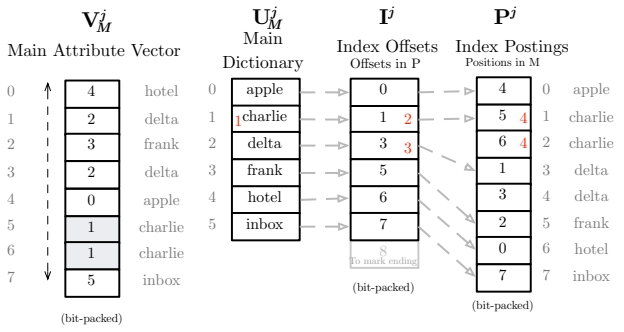


Figure 1: A Group-Key index on the Main Partition

1. **Counting occurrences of value-ids.** We create a new structure $\mathbf{I'}^j$ with size $|\mathbf{U'^j_M}| + 1$ and iterate once over $\mathbf{V'^j_M}$ and count the occurrences of value ids at their respective position in $\mathbf{I'}^j$.

2. **Creating offsets for $\mathbf{P'}^j$ in $\mathbf{I'}^j$.** By iterating once over $\mathbf{I}^j$ we create the prefix-sum of $\mathbf{I'}^j$, creating offsets for the $\mathbf{P'}^j$ structure. Additionally we create an exact copy $\mathbf{I'^j_{copy}}$.

3. **Creating postings in $\mathbf{P'}^j$.** We use $\mathbf{I'^j_{copy}}$ as a write counter. We iterate once again over all elements $i \in \mathbf{V'^j_M}$, obtain the write-offset from $\mathbf{I'^j_{copy}}$, by looking up $\mathbf{I'^j_{copy}}[i]$ and write the current position in $\mathbf{V'^j_M}$ to $\mathbf{P'}^j$. To ensure that further writes regarding the same value-id are directed to the next position in $\mathbf{P'}^j$, we increase the write-offset at $\mathbf{I'^j_{copy}}[i]$ after every read by 1.

$\mathbf{I'^j_{copy}}$ is deleted afterwards. $\mathbf{I'}^j$ and $\mathbf{P'}^j$ represent the new Group-Key index. Step 1 and Step 3 are dependent on the length of the new column, each executes in $\mathcal{O}(|\mathbf{V'^j_M}|)$. Step 2 is dependent on the length of the dictionary, $\mathcal{O}(|\mathbf{U'^j_M}|)$.

The memory traffic of Step 1 consists of reading the attribute vector and writing the positions. Because no inherent order between consecutive values can be assumed, every write-operation of $\mathbf{I'^j_{copy}}$ and $\mathbf{P}^j$ will possibly be directed towards a different cacheline. The memory traffic, including the fetch-for-write component, of Step 1 is given in Equation 5.

In Step 2, the index offsets structure is modified sequentially, and copied. Its size is $(|\mathbf{U'^j_M}| + 1) \cdot \mathbf{A'}^j$ bits, and the memory traffic is given in Equation 6. In Step 3 the attribute vector is read sequentially, and the postings-structure and the offset-write-counter are written randomly, leading to the read and write of a cacheline each. Equation 7 summarizes the costs of Step 3.

$$MT_{Step1} = 2 \cdot \mathbf{N}'_M \cdot \mathbf{L} + \mathbf{N}'_M \cdot \frac{\mathbf{A}^j}{8} \text{ bytes} \quad (5)$$

$$MT_{Step2} = 4 \cdot (|\mathbf{U'^j_M}| + 1) \cdot \frac{\mathbf{A}^j}{8} \text{ bytes} \quad (6)$$

$$MT_{Step3} = 4 \cdot \mathbf{L} \cdot \mathbf{N}'_M + \mathbf{N}'_M \cdot \frac{\mathbf{A}^j}{8} \text{ bytes} \quad (7)$$

Rebuilding the index by inserting all value-position pairs into a tree structure is much less performant than the here proposed rebuilding algorithm.

4. COLUMN MERGE

The 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. 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 Section 5.

In brackets we refer to the sub-steps shown in Figure 2. We perform the merge using the following two steps:

1. **Merging Main Dictionary and Delta Index, Creating 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 $\mathbf{X}^j_\mathbf{M}$(Mx, Dx, Bx) . 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 $\mathbf{V}^j_\mathbf{D}$. (D2,B3). 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 $\mathbf{X}^j_\mathbf{M}$ (UMx) as shown in Equation 8. Values from $\mathbf{V}^j_\mathbf{D}$ are copied without translation to $\mathbf{V}'^j_\mathbf{M}$(UD1). 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 Equation 9.

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

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.

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

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 attributevector, $\mathcal{O}(\mathbf{N}_M + \mathbf{N}_D)$.

4.1 Memory Traffic of the Column Merge

As far as the total amount of data read from the main memory is concerned, the total amount of memory required to store the tree is around 2X the total amount of memory consumed by the values themselves [8].

Algorithm 1 Extended Dictionary Merge

```
1: procedure ExtendedDictionaryMerge
2:     d, m, n, c = 0
3:     while d != |T^j| or m != |U^j_M| do
4:         processM = (U^j_M[m] <= T^j[d] or d == |T^j|)
5:         processD = (T^j[d] <= U^j_M[m] or m == |U^j_M|)
6:         I'^j[n] ← c                ▷ The Start of the position list.
7:         if processM then
8:             U'^j_M[n] ← U^j_M[m]
9:             X^j_M[m] ← n − m
10:            count = I^j[m + 1] − I^j[m]
11:            P'^j[c···] ← [P^j[I^j[m]] ··· P^j[I^j[m + 1]])
12:            c+ = count
13:            m ← m + 1
14:        end if
15:        if processD then
16:            U'^j_M[n] ← T^j[d]
17:            count = size(T^j[d].positions)
18:            [P'^j[c] ··· P'^j[c + count]) ← T^j[d].positions
   Store n in V^j_D at all positions from T^j[d].positions
19:            c+ = count
20:            d ← d + 1
21:        end if
22:        n ← n + 1
23:    end while
24:    I'^j[n] ← c                    ▷ The End-Marker
25: end procedure
```

Updating the tuples involves reading their tuple-id and a random write-access into $\mathbf{V}^j_\mathbf{D}$ to update the tuple. Since each access would read a cache-line ($\mathbf{L}$ bytes wide) the total amount of bandwidth required would be $(2 \cdot \mathbf{L} + 4)$ bytes per tuple (including the read for the write component).

To summarize, the memory traffic of our merge algorithm includes reading the main dictionary and the CSB+ tree, and all position lists therein; as well as writing the new dictionary, the main auxiliary structure and the delta attribute vector.

$$MT_{\mathbf{T}^j} = 2 \cdot \mathbf{E}^j \cdot |\mathbf{T}^j| \quad (10)$$

$$MT_{Step1Read} = MT_{\mathbf{T}^j} + \mathbf{E}^j \cdot (|\mathbf{U}^j_\mathbf{M}|) \quad (11)$$

$$MT_{Step1Write} = 2 \cdot \left(\mathbf{E}^j \cdot |\mathbf{U}'^j_\mathbf{M}| + \frac{\mathbf{E}'^j_C \cdot (|\mathbf{X}^j_\mathbf{M}|)}{8} + \mathbf{L} \cdot \mathbf{N}_D \right) \quad (12)$$

The exact memory traffic that incurs from reading the main aux structure is highly dependent on the actual data, randomized data will lead to many more misses than sorted data, however we will use Formula 13 to estimate the traffic for auxillary structures that do not fit the caches.

$$MT_{MainAux} = \mathbf{N}_M \cdot \mathbf{L} \quad (13)$$

$$MT_{Step2Read} = \mathbf{N}_M \cdot \mathbf{E}^j_C / 8 + MT_{MainAux} \quad (14)$$

$$MT_{Step2Write} = 2 \cdot \mathbf{N}_M \cdot \mathbf{E}'^j_C / 8 \quad (15)$$

5. EXTENDED COLUMN MERGE

We now integrate the index rebuild into the column merge process. This allows us to reduce the memory traffic, and

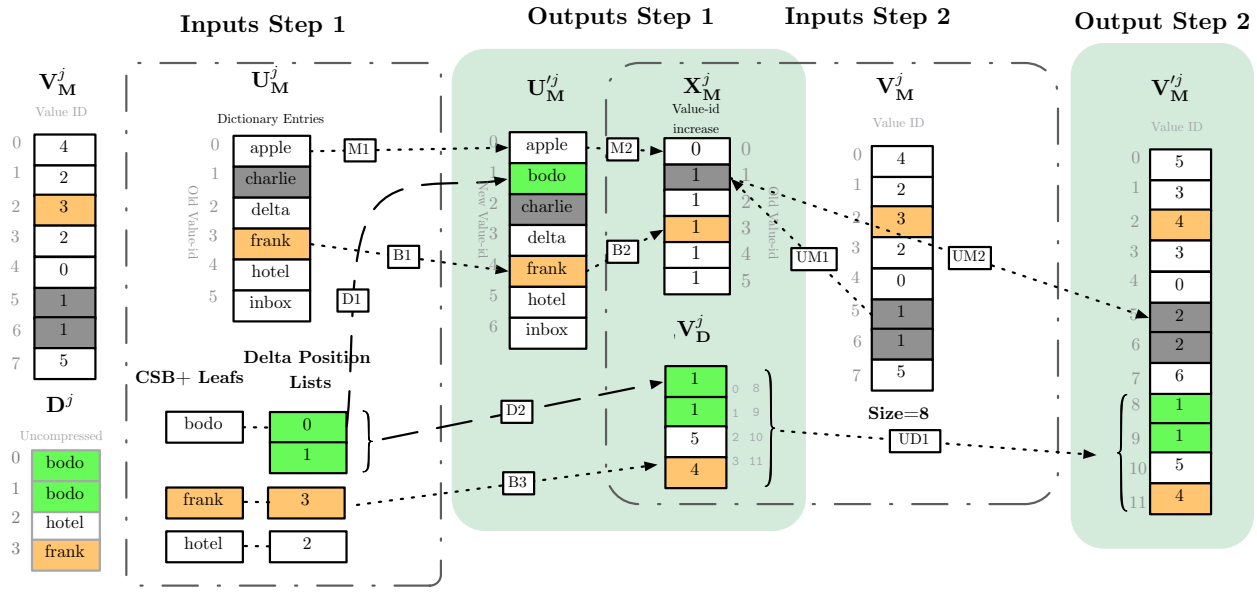


Figure 2: The Column Merge Process. The algorithm creates a new main partition by traversing the old main partition and the CSB+ Tree index of the Delta Partition. The uncompressed vector of $\mathbf{D}^j$ is shown for reference only.

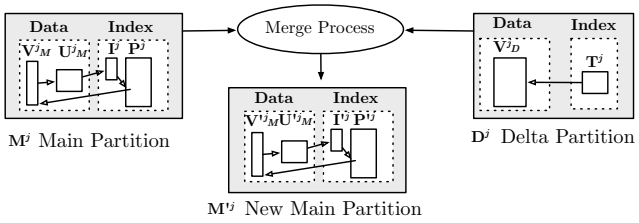


Figure 4: Extended Merge Process Concept

create a more efficient algorithm to merge columns with an Group-Key index.

We extend Step 1 of the column merge process from Section 4 to maintain the dense bit-packed index. The count of all records that correspond to already merged values is kept in a variable $\mathbf{C}$. During the dictionary merge we perform additional steps for each processed dictionary entry. Before processing the next dictionary entry, $\mathbf{C}$ is written to $\mathbf{I}'^j[n]$, with n denoting the new value-id. The substeps are extended as follows:

1. **For Dictionary Entries from the Main Partition** Read the begin and end offset from $\mathbf{I}^j$, copy all matching positions to $\mathbf{P}'^j$, starting at $\mathbf{C}$. Add the difference between start and end offset to $\mathbf{C}$.

2. **For CSB+ Index Entries from the Delta Partition** Read all positions from $\mathbf{T}^j$, increase them by $\mathbf{N}_M$, and write to $\mathbf{P}'^j$, starting at $\mathbf{C}$. Increase $\mathbf{C}$ by the amount of positions that were written.

3. **Entries found in both Partitions** Perform both steps sequentially.

After completing the dictionary merge, $\mathbf{C}$ is written to $\mathbf{I}^j[|\mathbf{U}'^j_\mathbf{M}|]$ to mark the end of the postings file, and avoid a special handling for the last value during query execution.

Algorithm 1 shows the detailed steps of the dictionary merge, where all index related operations are performed.

5.1 Memory Traffic of Augmented Merge

The modified Step 1 additionally reads $\mathbf{I}^j$ and $\mathbf{P}^j$ sequentially. During the process the new sturcutres $\mathbf{I}'^j$ and $\mathbf{P}'^j$ are written sequentially. Since Step 2 is unmodified, the difference regarding the memory traffic between the optimized merge process and the index-aware augmented merge process originates from these sequential reads and writes.

Therefore, we calculate the extra read memory traffic as given in Equation 16 and the extra write traffic (including the fetch-for-write component) in Equation 17.

$$MT_{ExtraR} = (|\mathbf{U}^j_\mathbf{M}| + \mathbf{N}_M) \cdot \frac{\mathbf{A}^j}{8} \tag{16}$$

$$MT_{ExtraW} = 2 \cdot (|\mathbf{U}'^j_\mathbf{M}| + \mathbf{N}'_M) \cdot \frac{\mathbf{A}'^j}{8} \tag{17}$$

6. EVALUATION

In this section we evaluate the memory traffic during query execution and the extra memory traffic during the merge process to determine the viability of the indexing scheme. We implemented the regular column merge and the index-aware merge process, as well as index lookup and column scan.

6.1 Impact on Read-Traffic during Query Execution

The Group-Key index helps only with query execution on the read-only main partition. Query execution on the delta partition is independent thereof and we therefore only consider the savings of the Group-Key index on main-partition query execution. Our proposed Group-Key index maps value-ids to positions, but queries include predicates on uncompressed values. The first step of query execution is therefore to translate the uncompressed value into a value-id by performing a binary search ($\mathcal{O}(\log(|\mathbf{U}^j_\mathbf{M}|))$) on the sorted dictio-

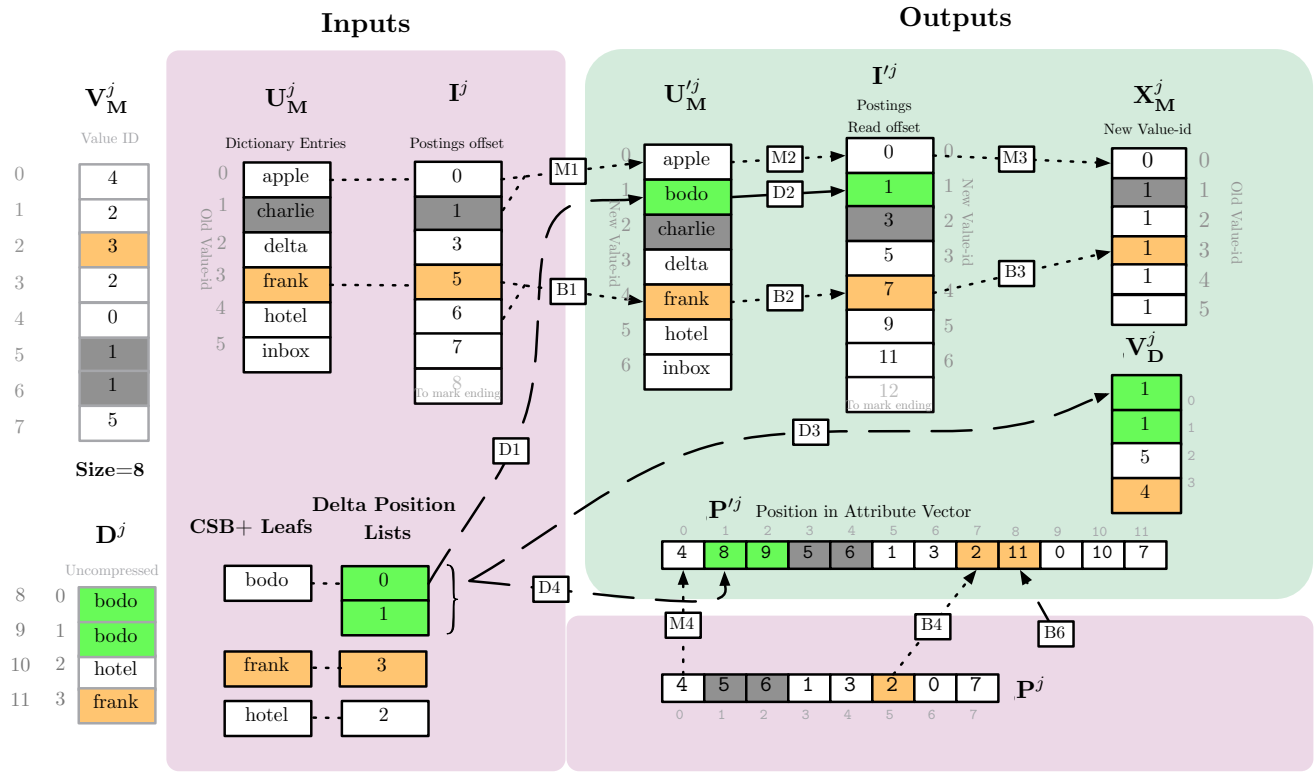


Figure 3: Step 1 of Merging the Group-Key index togther with the dictionaries, as explained in Strategy 3. $\mathbf{V}^j_\mathbf{M}$ and $\mathbf{D}^j$ are shown for reference only and are not used during this step of the index-aware merge-process.

nary $\mathbf{U}^j_\mathbf{M}$. Since this step has to be done for a full column scan as well, we can exclude this from the comparison.

Let λ^j be the distinct value fraction of column j. Then every value is expected to occur $\frac{\mathbf{N}_M}{|\mathbf{U}^j_\mathbf{M}|} = \frac{1}{\lambda^j}$ times in a uniform distribution. Hence, when evaluating a predicate on a single value, we have to read $\frac{1}{\lambda^j} \cdot \mathbf{A}^j$ bits, instead of the whole column, which has $\mathbf{N}_M \cdot \mathbf{E}'^j_C$ bits , excluding the dictionary lookup, which has to be performed for column scan and index lookup alike.

Note, that we typically need more bits to encode an address in the main partition, than a value-id, as a consequence of the implication given in Equation 18.

$$|\mathbf{U}^j_\mathbf{M}| \leq \mathbf{N}_M \Rightarrow \mathbf{E}'^j_C \leq \mathbf{A}^j \tag{18}$$

We calculate the saved memory bandwidth when using the Group-Key index for a single-value predicate as shown in Equation 19. The formulas first part is the complete scan of the attribute vector, the second part is the read-operation on the index structure of all positions plus the reading of a single cacheline to obtain the correct offset from $\mathbf{I}^j$.

$$\begin{aligned} MT_S =& \mathbf{N}_M \cdot \frac{\mathbf{E}'^j_C}{8} - \left(\frac{1}{\lambda^j} \cdot \frac{\mathbf{A}^j}{8} + \mathbf{L}\right) \\ =& \frac{\mathbf{N}_M \cdot \lceil \log_2(\lambda^j * \mathbf{N}_M) \rceil}{8} - \frac{\lceil \log_2(\mathbf{N}_M) \rceil}{8 * \lambda^j} - \mathbf{L} \end{aligned} \tag{19}$$

In Section 6.3 we will compare the estimated savings during query execution with the increase of memory traffic for the index maintenance.

6.2 Index Lookups vs. Column Scans

In this section we will compare the memory traffic of an index lookup with the costs of a column scan.

Because the Group-Key index stores positions in $\mathbf{V}^j_\mathbf{M}$, a single value typically uses more bits than a value-id, as shown in Equation 18. As a consequence, a column-scan can lead to less memory traffic, than a lookup in the Group-Key index, especially if $\mathbf{E}'^j_C << \mathbf{A}^j$. To find the minimum dictionary size for which a index lookup should consume less memory traffic for Single-Value predicates and with an assumed uniform distribution, we need to find the solutions to Inequality 20, which compares the memory traffic of a column scan to the memory traffic of the index lookup.

$$\begin{aligned} \mathbf{N}_M \cdot \frac{\mathbf{E}'^j_C}{8} \geq& \left(\frac{1}{\lambda^j} \cdot \frac{\mathbf{A}^j}{8} + \mathbf{L}\right) \\ \frac{\mathbf{N}_M \cdot \lceil \log_2(\lambda^j * \mathbf{N}_M) \rceil}{8} \geq& \frac{\lceil \log_2(\mathbf{N}_M) \rceil}{8 * \lambda^j} + \mathbf{L} \end{aligned} \tag{20}$$

Note that Inequality 20 assumes a uniform distribution, so every value occurs $\frac{1}{\lambda^j}$ times. As an example, queries on a column with $\mathbf{N}_M = 100,000$ benefit from the index, if more than five distinct values are present (with $\mathbf{L} = 64$ bytes). For very small columns, with less than 1,000 values, the theoretical break-even point is high, however, when the column takes only very few kilobytes of memory, the bandwidth will not be the limiting factor, but latency. In this paper we focus on columns that are larger than the CPU cache, and involve therefore significant amount of memory traffic. As it is shown in Figure 5, the break-even point for even the largest

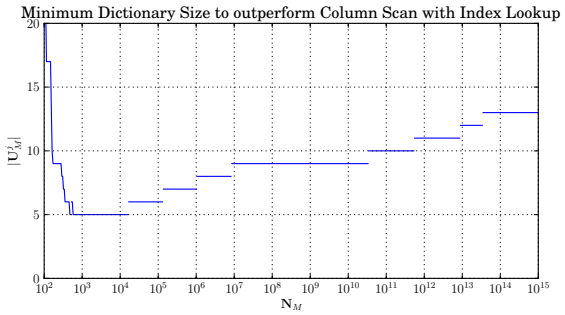


Figure 5: Index lookups vs. column scans in relation to dictionary size in a uniform-distributed column. Above the chartline are solutions to Formula 20.

columns is reached with relatively few distinct values. For columns with a size between millions and trillions of values, the Group-Key index lookup outperforms the column-scan, if the dictionary size is larger than about 8 to 10 values, so that a value in a uniform distribution appears in less than one tenth of the fields. Smaller dictionaries lead to smaller attribute vectors, and therefore the memory traffic of the column scan can be lower, than reading all associated positions from the index.

Since $\mathbf{I}^j$ can be used to calculate the count of each distinct value, we can refine our approximation, to allow the query optimizer to decide at runtime between reading all positions from the Group-Key index or to perform a column scan, without assuming a specific distribution. Recall Figure 1: Since $\mathbf{I}^j$ contains offsets into the postings-structure $\mathbf{P}^j$, we can obtain the count of any value, represented by its value-id as shown in Equation 21. If a range-query is performed, the value-ids of the interval-limits are obtained, and the occurrence-count is calculated as shown in Equation 22. Calculating Equation 21 will typically result in reading one cacheline of $\mathbf{L}$ bytes, Equation 22 of two.

$$\mathbf{count}(value_{id}) = \mathbf{I}^j[value_{id}+1] - \mathbf{I}^j[value_{id}] \quad (21)$$

$$\mathbf{count}(v_{min}, v_{max}) = \mathbf{I}^j[v_{max}+1] - \mathbf{I}^j[v_{min}] \quad (22)$$

$$\mathbf{count}(value_{id}) = \mathbf{count}(v_{min}, v_{max}) \text{ with } v_{max} = v_{min} = value_{id}$$

We now modify Equation 20 accordingly.

$$\mathbf{N}_M \cdot \frac{\mathbf{E}'^j_C}{8} \geq \left(\mathbf{count}(v_{min}, v_{max}) \cdot \frac{\mathbf{A}^j}{8}\right) \quad (23)$$

With Inequality 23 a decision for or against the usage of the index can be made at runtime, and in the presence of a skewed distribution. The additional memory traffic for the evaluation thereof is limited to the read of two cachelines, to obtain the cardinality of the result set.

If a range query is executed, the position-lists have to be merged, to obtain a ordered position lists, which is typically required for the next step in a query pipeline. As shown in [2] this can be achieved by reading all lists only once from memory, hence the memory traffic can be calculated as shown.

6.3 Break-Even Point: Savings vs. Merge

With the Equations 16 and 17 given in Section 5.1 it is possible to calculate the amount of queries that are necessary to compensate the additional memory traffic of the extended merge process. Under the assumption that our in-memory column store will be bandwidth-bound, optimization goes towards reducing the overall traffic of the system, for the given workload. The first step to determine if maintaining a Group-Key index is worth the extra costs of the extended merge process, is to analyze if the extra traffic is offset by savings during query execution. Other dimensions that have to be considered are the increased memory-demand, and the associated initial and operational costs, but also latency demands. However, in this paper we focus solely on the bandwidth constraints.

$\mathbf{N}_M$	$\mathbf{N}'_M$	λ^j	ϕ^j_{Memory}
10000	11000	0.05	5.27
1000000	1100000	0.05	4.34
10000000	12000000	0.0001	8.18

Table 2: Queries between merges to offset elevated merge costs (ϕ^j_{Memory}) for selected configurations.

We will compare the estimated savings during query execution, with the extra-traffic of the augmented merge process. The amount of queries that offset the augmented merge costs will be referred to as ϕ^j.

$|\mathbf{U}^j_\mathbf{M} \cup \mathbf{U}^j_\mathbf{D}|$ is estimated as shown in Equation 24, e.g the fraction of distinct values before and after a merge process is assumed to be relatively constant.

$$|\mathbf{U}^j_\mathbf{M} \cup \mathbf{U}^j_\mathbf{D}| \approx \lambda^j \cdot (\mathbf{N}_M + \mathbf{N}_D) \quad (24)$$

We will focus on single-value predicates, and ignore range queries for now. In Equation 25 we put the runtime savings and elevated merge costs into relation to obtain ϕ^j.

$MT_{ExtraRead}$ and $MT_{ExtraWrite}$ are calculated as shown in Equations 16 and 17 and refer to the additional memory traffic of the extended merge process. MT_S is the saved memory traffic for each column scan that is replaced by an index lookup, defined in Equation 19.

$$\phi^j \approx \frac{MT_{ExtraRead} + MT_{ExtraWrite}}{MT_S} \quad (25)$$

The break-even point is therefore dependent on the distinct values, the frequency of the merge process and the column size. A few, selected results for Equation 25 are given in Table 2, and a plot in Figure 6. We can conclude, that for most common configurations, a moderate number (< 20) of index lookups, which are performed instead of column scans, already suffices to offset the increased memory-traffic of the augmented index-aware merge process. However, columns with only very few distinct values profit less, because the index lookup has to read many positions, and a position in $\mathbf{P}^j$ is in general encoded with more bits than a value-id in the main attribute vector (see also Equation 18).

6.4 Elevated Merge Costs

The performance of the merge-process is paramount to the overall sustainable system performance of in-memory column stores with a separate writable partition. Our augmented index-aware merge process raises the merge costs, and if the system is in fact bandwidth-bound, we can calculate the performance decrease of the merge process by calculating the relative increase in memory traffic, to which we refer as ρ^j and define in Equation 26.

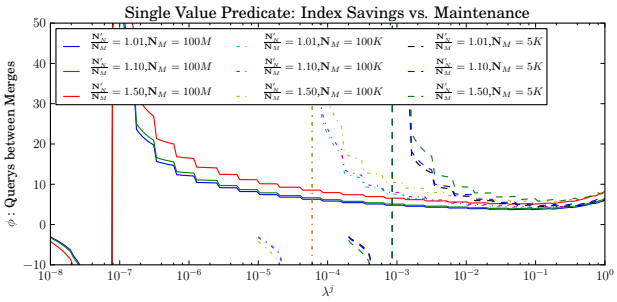


Figure 6: Amount of Index lookups that will offset the extra memory traffic of index maintenance during the augmented merge-process. If $\phi < 0$ the index lookup leads to more memory traffic than the column scan (e.g. $\mathbf{E}_C'^j << \mathbf{A}^j$), the respective peaks are at points from Figure 5.

$\mathbf{N}_M$	$\mathbf{N}'_M$	λ^j	$\mathbf{E}^j$	ρ^j_{Memory}
10000	11000	0.05	4	1.07
1000000	1100000	0.05	4	1.11
10000000	12000000	0.0001	16	1.11
600000	660000	0.0001	4	1.10

Table 3: Theoretical increase of memory traffic with index-aware extended column merge algorithm.

$$\rho^j_{Memory} = 1 + \frac{\text{Extra Traffic for Index Maintenance}}{\text{Merge Traffic}} \quad (26)$$

ρ^j is dependent on the main size, the delta size, the distinct value count and the size of the uncompressed values. If the main size grows, costs for index maintenance and merge costs both grow. However, since the index-merge copy runs sequentially, and the merge-step needs to make random lookups for each element, the relative costs for maintaining the index decrease for large main sizes. The delta size has a similar influence, although the random access takes place when updating the attribute vector. The index-aware merge adds only sequential access.

If the distinct value count grows, the dictionary merge step becomes more expensive. Since the length of the runs that are copied sequentially is inversely proportional to the distinct value count, the access patterns of the index-aware merge degrades more and more into a random access pattern.

The uncompressed value size affects only the shared steps (the index operates on value-ids), and therefore bigger values decrease the relative indexing costs.

Figure 7 shows a plot of the theoretical merge costs for different distinct-value configurations, and selected results are given in Table 3. Memory traffic is typically increased by about ten percent for columns with few distinct values, and by up to about 25 percent for columns with high distinct value counts.

7. MEASUREMENT

We implemented a prototype version of the merge process as presented in Section 4 and the index-aware merge process from Section 5. A scalar version of the Sanssouci DB [6] bit-packed attribute vector implementation was used. All tests were performed on an dual Intel X5650 (Caches: L1d/L2i: 32KB, L2: 256KB, L3: 12MB) system with 48 GB of DDR3-1066 RAM. We used the Intel C++ compiler in version 12.1.

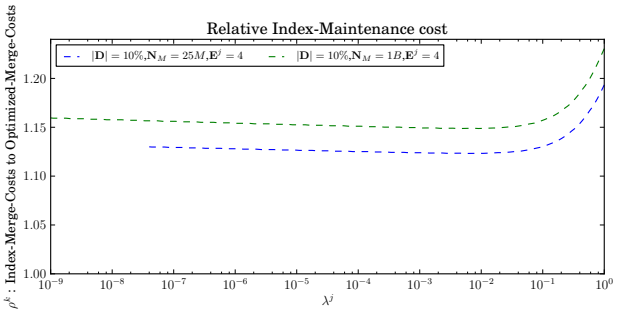


Figure 7: Relative Increase of memory traffic for Index-Aware merge process.$|\mathbf{D}| = 10\%$ denotes that the delta partitions size is one percent of the main partitions size.

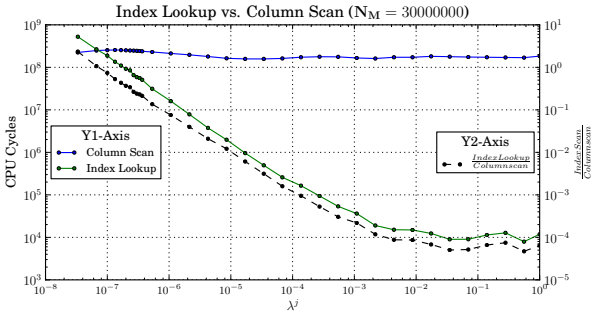


Figure 8: Costs of Index Lookup vs. Column Scan. Note the Log-Log scale.

In Figure 8 we verify our estimation about the index viability during query execution. We note, that the measured break-even point is indeed reached within the expected area of λ^j, representing a few distinct values. Note that the benefit of the index stagnates for $\lambda^j > 10^{-2}$, since other costs rather than memory access begin to dominate the index lookup.

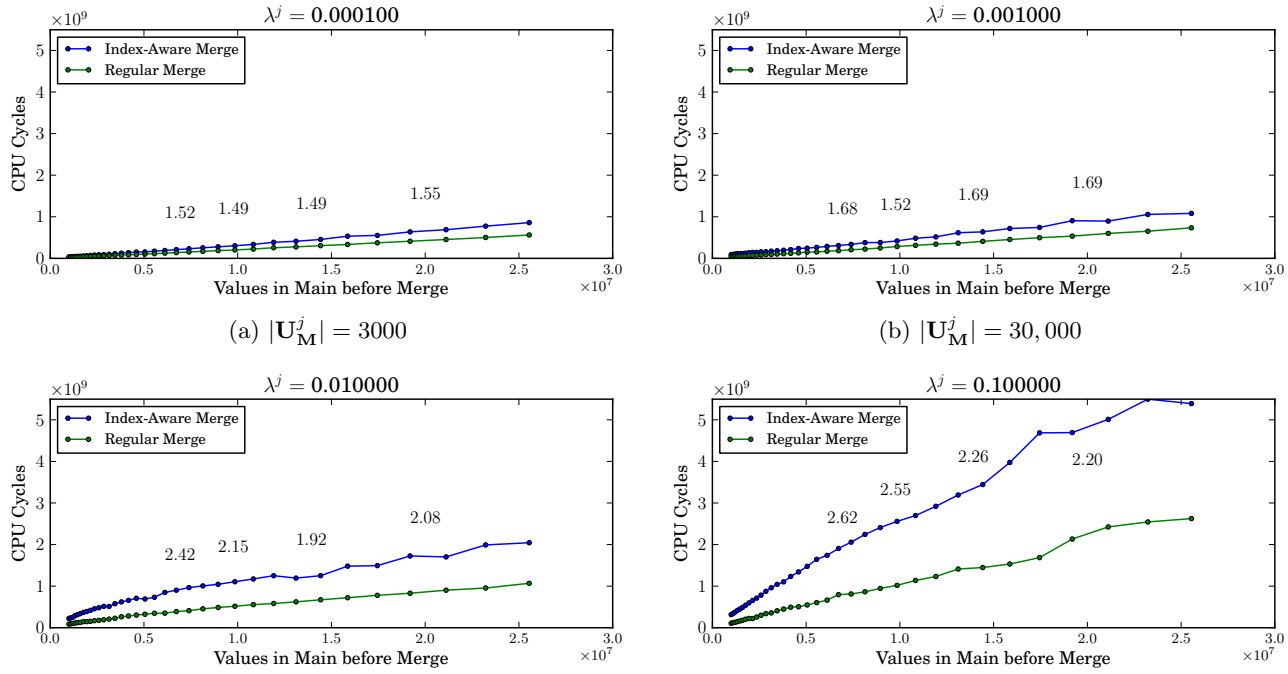
We implemented and measured our column merge algorithm and compared it with the index-aware implementation thereof. We define

$$\rho^j_{CPUCycles} = \frac{\text{CPU Cycles for Index Aware Merge}}{\text{CPU Cycles for Regular Merge}} \quad (27)$$

and analyze how it corresponds with the theoretical statements drawn from the memory traffic. Since we expect bandwidth to be a limiting factor, $\rho^j_{CPUCycles}$ should correlate with ρ^j_{Memory}.

In Figure 9 the advantage of the augmented column merge becomes visible. Rebuilding the index after the merge process is significantly more expensive than augmenting the column merge. In our measurements rebuilding the index after the merge on average triples the merge time , while the augmented column merge only doubles merge time.

When the distinct value count is very small, and only few entries exist, our models for the memory traffic are misleading, because the dictionary, the CSB+ index for the delta partition, the index offset structure and the aux structure can reside in cache, and the algorithm becomes computation bound in some phases.



(a) $|\mathbf{U}^j_\mathbf{M}| = 3000$

(b) $|\mathbf{U}^j_\mathbf{M}| = 30,000$

(c) $|\mathbf{U}^j_\mathbf{M}| = 300,000$

(d) $|\mathbf{U}^j_\mathbf{M}| = 3M$

Figure 10: Merge performance for a regular column and a column with an index. The delta was merged at 10 percent size of the main partition. The annotated values in the chart show $\rho^j_{CPUCycles}$.

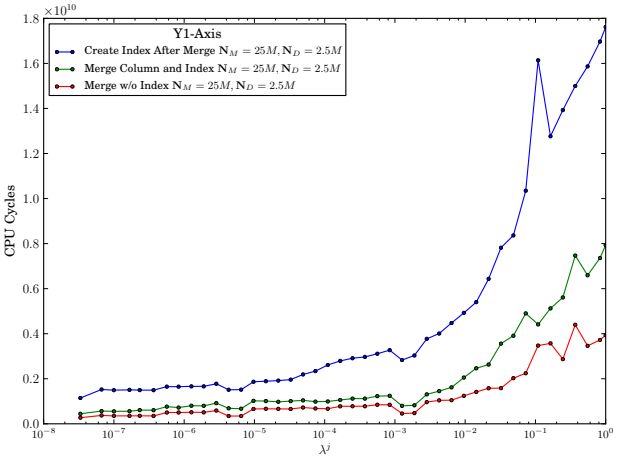


Figure 9: Merge Costs for different λ^j , $\mathbf{N}_M = 25M$, Delta Partition 10 percent of main partition.

Although the results show, that the extra index costs follow the trend defined by the memory traffic, the precision is not sufficient for a detailed model. However, the results verify, that the index overhead can be estimated , and that we can determine an estimate for the CPU-cycle costs from the calculated memory traffic.

The drops in Figure 9 can be explained by certain favorable configurations, at which the distinct-value levels result in byte-aligned values (e.g $\mathbf{E}^j_C = 8$ / 16 / 24 bit) for the compression of the value-id, which carry less processing overhead, than other configurations.

We also note, that the differences between our memory traffic based estimation and the measured values should decrease, with further optimizations of the code, such as SIMD compression and decompression.

The changes to the merge-algorithm increase the code size, and thereby decrease locality of access not only to the data that is merged, but also to instructions. Integrating the index-merging step into the dictionary step would profit from a fine grained cache-control, to avoid that data that is going to be used is evicted by data that should be streamed in or out, without polluting the caches. In general, putting more work into the *dictionary merge loop* will lead to less optimal usage of write combining buffers. We tested to push the work in a work list and produce the index after the dictionary merge, however, the changes were only minimal. The results indicate, that the index-aware merging step is slightly less bandwidth bound. We therefore plan to explore how parallelization across cores can help to maximize performance.

Figure 9 and Figures 10(a)-10(d) reveal, that the index-aware merge step in practice decreases the merge performance (in terms of CPU cycles) by up to a factor of 3 for columns

with rather high distinct value counts.

Evaluating the Cycles-per-Instruction (CPI) ratio reveals, that the current implementation does not achieve small CPI values for high distinct value counts, likely because the dictionary merge step can not benefit as much from out-of-order processing and other CPU features as the monotonous attribute vector update, and the sequential access to the index structures.

However, augmenting the merge process is always faster than rebuilding the index after every merge process.

8. CONCLUSION

Indices can decrease runtime memory demands significantly. In fact, scanning the whole column to answer queries with a low selectivity leads to excessive waste of memory transfer capacity and processing power.

The index comes at a price, not only does it increase the memory footprint of the column significantly, but it also increases the maintenance costs for the column, when the index has to be updated.

While earlier research [7] stated, that the index should simply be rebuild for analytical systems, we showed that this is not optimal. Since the merge process is crucial for the performance of in-memory column stores [5], we showed how the cost of index maintenance can be reduced by combining the index rebuild with the column merge process and the reuse of the former index. We showed the advantages with a theoretical view of the involved memory traffic and verified the findings with an experimental implementation. The proposed augmented column merge in our prototype implementation is on average about 30 percent faster than rebuilding the index after the column merge.

9. FUTURE WORK

Currently we assume that there are more columns to be merged than processors available, however this might not hold true in the future. As outlined in [1], future chip designs may feature thousands of cores, and therefore intra-column parallelization would become necessary to make use of many more cores. The same logic applies to the main memory subsystem - we introduced the index maintenance in a NUMA- agnostic manner in this paper, but if the main memory bandwidth is the limiting factor, the placement of data structures on different NUMA nodes may increase the performance by utilizing all memory channels in parallel.

10. REFERENCES

[1] S. Borkar. Thousand core chips: a technology perspective. In *the 44th annual conference*, pages 746–749, New York, New York, USA, 2007. ACM Press.

[2] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, Aug. 2008.

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

[4] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, June 2011.

[5] 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. *Proceedings of the VLDB Endowment*, 5(1):61–72, Sept. 2011.

[6] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. 2011.

[7] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[8] J. Rao and K. A. Ross. *Making B+- trees cache conscious in main memory*, volume 29. ACM, June 2000.

[9] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, and E. O'Neil. C-store: a column-oriented DBMS. *Proceedings of the 31st international conference on Very large data bases*, pages 553–564, 2005.

[10] 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.

[11] M. Zukowski, P. Boncz, N. Nes, and S. Heman. MonetDB/X100—A DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.