A Case for Inverted Indices in In-Memory Databases

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

***Abstract*—Recent database research has focused on in-memory databases, which can be used in mixed workload scenarios, enabling OLTP and OLAP queries on the same database engine. These compressed column-oriented database systems use a differential store concept to enable fast inserting and require a merge process to compact the data periodically in a compressed main partition that is changed by the merge process only. This characteristic feature calls for a re-evaluation of the performance of inverted indices. In this paper we present a use case for an inverted index in a column-oriented in-memory database system to reduce the total costs of query processing in a mixed workload environment. We evaluate the benefits and drawbacks of using the index structure to answer queries and the costs of maintaining the index, especially during the merge process. An analytical model is introduced to compute the theoretical cost of index scans. Furthermore a comparison between different index maintenance strategies is presented. The theoretical findings are verified in a prototypic implementation within the HYRISE database system. Our contributions are an analytical framework to evaluate the benefit of an inverted index during query execution, the verification in an in-memory database system, and an evaluation of different index maintenance strategies. From the presented findings we conclude that indexing can be an efficient instrument to meet the performance requirements in a mixed-workload and main memory-based environment.**

***Keywords*-In-Memory Database; Inverted Index; Index Maintenance;**

I. Introduction

Over the last years the price of main memory has dramatically declined to a level on which it becomes competitive to prices for hard disk storage of only few years ago. This trend allows for servers with several gigabytes or even terabytes of main memory at a relatively low cost. The vast amount of fast accessible memory exceeds by far the space needed by traditional software, which mostly derives from times where main memory was a resource to be handled carefully. Consequently, in-memory databases build on this phenomenon and use the available space to store the application itself as well as the contained data entirely in the main memory, thus eliminating costly lookups on slow hard disk drives when accessing data. Despite all data now being present in the main memory at any time, processing the data still requires it to be loaded into the processor's cache, an operation that - after eliminating expensive reads from hard drives - represents the system's new bottleneck [1]. Due to this fact, reducing memory accesses is crucial to the overall performance of the database. Reading an entire table consisting of several million rows for the sake of a few records' values is an obvious waste, which can be eliminated by a lookup structure called inverted index. While inverted indices have been used for decades in traditional disk-based database systems or text retrieval systems, there are differences when using inverted indices in an in-memory database, such as HYRISE [2], [3], which impact the way an index is leveraged and maintained. In case dictionary compression is applied on a column, the inverted index is used to speed up read performance since the applied dictionary compression facilitates the inverted index structure as the compression can be leveraged to build up a compressed index. The impact is especially high for queries with a low selectivity. As shown in Figure 1 the workload of enterprise applications consist of $\sim$50 percent lookup queries and $\sim$30 percent of range queries. Consequently, in mixed workload environments an index is needed to efficiently support these kinds of read access.

This paper gives an overview over the benefits and tradeoffs, which result from the employment of inverted indices in the context of in-memory databases. The remainder of the paper is structured as follows: First, we will introduce the peculiarities of the used in-memory database HYRISE and provide additional architectural information. Section II gives an overview on the related work on inverted indices and describes the differences regarding an inverted index implemented in an in-memory database. In Section III, the inverted index is explained and theoretically analyzed regarding advantages and disadvantages, as well as other aspects such as the profitability of a delta partition index and a unique approach to index maintenance. The theoretical findings of this section are then compared against the actual HYRISE implementation in Section IV. The paper concludes with a summary and outlook.

A. Background

There are of course different implementation approaches to in-memory database systems. In this paper, we focus on HYRISE [2], that is an in-memory database, which facilitates a column-oriented data organization and dictionary compression with late materialization strategies during query execution [4].

 ISBN: 978-1-61208-185-4

The key features with impact on structure and performance of an inverted index encompass the following:

a) Column-oriented Data Storage: Typically there are two ways of storing the information contained in a table, either row-by-row or column-by-column. Which one to choose primarily depends on the expected workload. Sequentially accessing data on almost any storage medium is substantially faster than random access. It has been found that in many cases there are more analytical-style than transactional-style queries [5]. Typical analytical queries rarely access entire rows of a table but rather focus on the values of a small set of columns. Storing data in a columnar fashion saves the effort of reading unnecessary columns, thus reducing query execution time.

b) Dictionary Compression: Even if main memory is not a scarce resource with regards to the available size anymore, it still has to be used efficiently to reduce overall costs besides reducing the memory consumption only. The uncompressed size of a company's productive system may exceed even most-recent terabyte server setups. In HYRISE a column-based dictionary compression technique is used to reduce the memory footprint of each column. All distinct values are captured in a dictionary data structure (vector) that assigns each actual value of the column a unique value id, and provides bilateral translation. Now instead of the actual values in a column, each column consist of a dictionary vector and an attribute vector that stores the respective value ids. The larger those actual values and the smaller the number of distinct values, the more efficient the compression. For instance, applying dictionary compression to a column storing country names as strings: As the number of countries is naturally limited and in practice often narrows down to a handful that are actually used, storing that handful of strings once in the dictionary and using value ids in the column provides good compression rates.

Other lightweight compression techniques are not applied to HYRISE since fast tuple reconstruction is essential for the mixed workload environment HYRISE is build for.

c) Differential Store Concept: Due to the applied compression each column is separated into two parts, a primary, read-optimized, read-only part (referred to as "main partition") and a secondary, write-optimized part that serves as buffer for data altering operations (referred to as "delta partition") [3]. The actual state of a column is represented by a union of both partitions with a one-bit validity vector to handle record visibility in case of update and delete queries. Initially being empty, the delta partition gets filled by data modifying queries. Since the main partition is not altered, a number of optimizations regarding read performance can be applied while the delta partition focuses on insertion speed, thus the differential store copes efficiently with OLTP and OLAP style requests at the same time.

d) Merge Process: As stated, data changes occur only in the delta partition. Yet, main and delta partition form a unit and together represent the column, meaning that data retrieval of a column must respect both. Due to the write-optimized nature of the delta partition, it lacks the main partition's read performance and thus should not grow too large in size. Defining a threshold at which the delta partition becomes "too large" is not a trivial task and primarily depends on the individual workload and is discussed in [6]. However, once the delta partition size exceeds certain limits, a merge process is initiated. The merge process combines the main and delta partition to create a new main partition and a new empty delta partition. During this process the respective dictionary vectors are first merged into a new, sorted dictionary, secondly changes in the mapping of values onto value ids are applied to both main and delta partition content and last the contents are concatenated.

B. Enterprise Workload Characteristics

In order to give a background on realistic workloads in enterprise applications, this section presents results on analyzing actual workloads.

The common assumption that enterprise applications work row based and with many updates has driven decades of database research. For example the TPC-C benchmark, which incorporates the characteristics of a transaction processing system, issues around 45% data modification operations and 55% read queries.

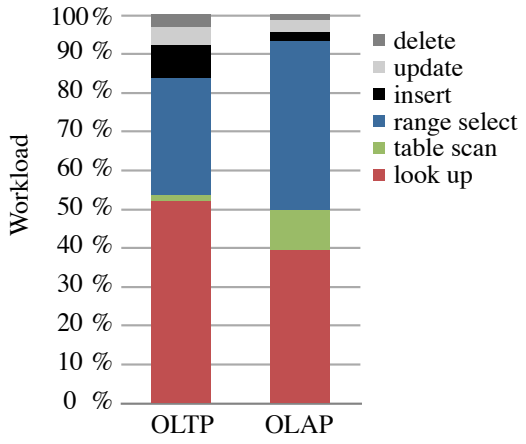

Figure 1. Analyzed Enterprise Workloads

However, our analyses of realistic database statistics derived from 65 customers have shown that more than 80% of all queries are read accesses — in OLAP systems even over 90% are read-only queries. Figure 1 shows the query distribution for transactional and analytical systems of key lookups, tables scans, range selects, inserts, updates and deletes. The analytical system differs in the distribution of inserts and table scans (which become column scans in column-oriented databases) with regards to the overall workload. While this is the expected result for OLAP systems, the high number of read queries on the transactional system is surprising. Consequently, the query distribution leads to the idea of using a read-optimized database for both transactional and analytical systems.

When implementing analytical functionality in a transactional system the support for fast tuple reconstruction based on key lookups still stays essential. At the same time, more

 ISBN: 978-1-61208-185-4

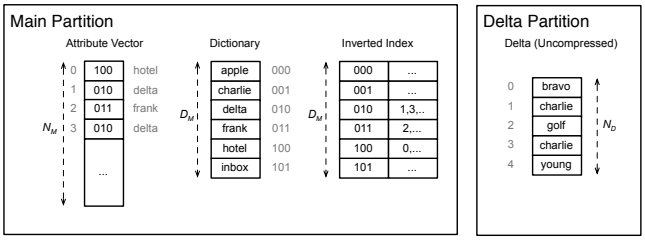

Figure 2. Inverted Index and Dictionary Encoding on a Single Column

complex queries with range requests and especially full column scans will increase. This trend will even further develop in the direction of more complex queries and full column scans as the direct reporting on transactional data enables new ”real-time analytics”. To summarize, highly selective queries are important in all workload variants and need to be supported efficiently. For this use case, indices have been introduced for any database system.

II. Related Work

Inverted indices have been used for decades and were initially used to reduce the amount of data read from disks for the sake of throughput. Inverted indices in the context of in-memory databases face a similar problem domain which led to examinations of main memory inverted indices before actual in-memory databases were feasible. For instance, Lehman and Carey [7] already compared possible implementation techniques in 1986, at the same time predicting the price decline of main memory.

Most research about main memory inverted indices concentrates on either performance of the index itself or its maintenance. For example Rao and Ross [8], and Raatikka [9] evaluate several index data structures, such as T-trees, B-trees and their descendants, optimizing them for cache-consciousness.

Another approach is taken by Transier and Sanders in [10] who evaluate compression techniques for inverted indices used in text retrieval. The topic of maintaining inverted indices has also been covered, mainly in the context of text retrieval systems by facilitating the nature of certain index implementations [11].

Lester et al. [12] discuss the maintenance strategies in-place, re-build and re-merge, which are close to the deliberations about maintenance in this work.

Several research on inverted files has taken compression into account, such as in [13], [14], [15], and [16].

The purpose of this paper is not to optimize the index in terms of performance or size, but rather to investigate the use inside an in-memory database, such as HYRISE [2], [3].

III. Inverted Index Structure

As stated before, considering an in-memory database I/O reduction is still the goal on main memory access. To eliminate the access of unnecessary data, the inverted index offers an inverse mapping of a column. As depicted in Figure 2, the inversion results in a map containing a list of record ids for every distinct value. When applying a predicate on a column the requested conditions have to be checked on every value of the column if no inverted index or sorting is in place. While this results in a sequential read access pattern, which is substantially faster than random access [1], still the complete column has to be read, creating a potential bottleneck. Compared to this, a lookup on the inverted index immediately offers the positional information about which records match the criteria and hence reduce the amount of data to be accessed. As described in Section I-A, HYRISE implements a column store without surrogate ids and therefore individual sorting of columns is prohibited. Furthermore, dictionary encoding is used as compression technique in HYRISE. Therefore, the inverted index can likewise leverage dictionary encoding to reduce memory footprint by mapping value ids instead of values to positions (see Figure 2). Since the dictionary for the attribute vector is already in place, there is no additional overhead for compressing the inverted index.

The core of the inverted index is a key-value-container with value ids as keys and record id lists as values. Since each column has presumably a unique dictionary, compound indices have to be implemented as higher-level functionality. Likewise, due to the distinction into a main partition and a delta partition with separate dictionaries on a single column, the inverted index cannot span an entire column but has to be implemented on the main and delta partition of the column respectively. The available options are to index the main partition only or to construct two inverted indices per column (one per partition). The remainder of this section examines benefits and tradeoffs accompanying the inverted index in theory, as well as whether or not to use an inverted index on the delta partition. For the index key-value-container an implementation of a balanced tree is assumed [17], guaranteeing at least $\mathcal{O}(logn)$ for search, insert and delete operations.

A. Cost Model

The scans with and without an index are examined for their assumed costs in order to provide a common base for comparison. The model is based on the assumption that the bandwidth is the theoretical optimum and that cost can be expressed as number of operations on data. These theoretical operations will not directly map to CPU cycles, because different operations will consume a different amount of CPU cycles, also varying among hardware platforms. Our model is suitable to theoretically define the impact of using an index in a column store. As we show in Section IV, the model and the measurements align in key properties, such as break even points and general cost factors between storage schemes, but also show, that theoretical operations and CPU cycles are not directly interchangeable.

Table I summarizes the symbols that will turn up in calculations later on.

The term “query selectivity” refers to the fraction of a table’s records that will be contained in the query result set. As a shortcut S_M equals $s * N_M$ and S_M^d equals $s * d_M * N_M$

Description	Unit	Symbol
Number of rows in table	-	N
Table width, i.e. number of columns in table	-	W
Number of rows in main partition	-	N_M
Number of rows in delta partition	-	N_D
Number of entries in main dictionary	-	D_M
Number of entries in delta dictionary	-	D_D
Fraction of unique values in main partition	%	d_M
Fraction of unique values in delta partition	%	d_D
Query selectivity	%	s
Selected entries in main partition	-	S_M
Selected entries in delta partition	-	S_D
Selected distinct entries in main partition	-	S_M^d
Selected distinct entries in delta partition	-	S_D^d
Length of a value id	bytes	L_{VID}
Length of a record id	bytes	L_{RID}
Memory bandwidth	bytes/cycle	B
Costs for operation X	operations	C_X

Table I
SYMBOLS USED IN CALCULATIONS

for the main partition, while the same applies to S_D and S_D^d for the delta partition. Regarding the lengths of value ids and record ids an implementation as 32 bit integers is assumed, meaning that $L_{VID} = L_{RID} = 4$. Furthermore, all scans are assumed to be non-materializing, i.e. for the result no translation of value ids into values will be done [18]. In order to align the results as close as possible to our actual implementation running on a system as described in Section IV-A, we compute the memory bandwidth. 1066 MHz DDR3 memory in dual channel configuration features a theoretical throughput of 17.066 gigabytes per second. Adding the CPU with a base clock of 2.266 GHz and a turbo boost frequency of 2.666 GHz, averaged to 2.466 GHz or 2.466 million cycles per second, allow for the following calculation:

$$B = \frac{17.066\ GB/s}{2.466\ GHz} = \frac{17{,}066{,}000{,}000\ bytes/s}{2{,}466{,}000{,}000\ cycles/s} \approx 7\ bytes/cycle \quad (1)$$

B. Performance Benefits

Performance benefits achieved by increased lookup speed are the key argument for inverted indices. Scanning an entire table requires $N_M + N_D$ memory accesses and comparison operations for a non-indexed column, resulting in a complexity of $\mathcal{O}(M + D)$. The actual number of memory accesses might be smaller because the data is read in chunks called cache lines [19] and the sequential processing reduces cache misses to a minimum and facilitates hardware based prefetching. Neither dictionary size nor the query's selectivity influence the performance of scanning the entire table, because each record is processed anyway. The total costs can be summarized as:

$$C'_{TableScan} = (N_M + N_D) * W * \frac{L_{VID}}{B} \quad (2)$$

This equation means that an entire row is loaded to validate a predicate on a single field's value of the row. A slightly improved variant runs a separate check on the column of interest to determine the desired rows and read them afterwards.

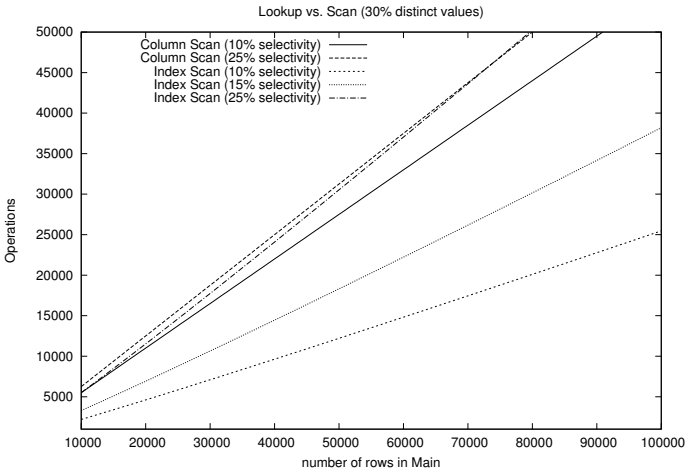

Figure 3. Column scan and index scan with varying main partition size

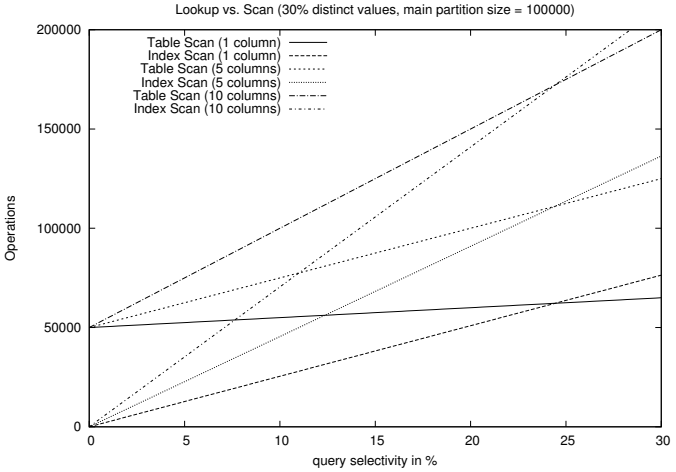

Figure 4. Column scan and index scan with varying selectivity

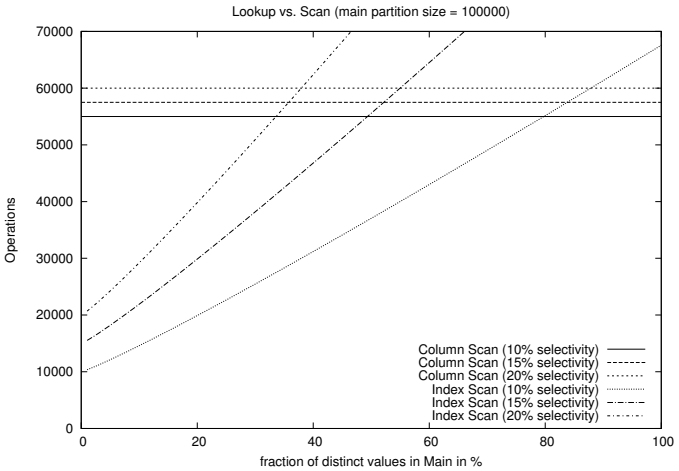

Figure 5. Column scan and index scan with varying fraction of distinct values

Therefore it suffers less from an increasing number of columns and will serve as the standard equation for a table scan in this Section and represents the characteristics of a column-oriented data storage. In this case of course query selectivity does have influence on the total costs:

$$C_{TableScan} = (N_M + N_D + (S_M + S_D) * W) * \frac{L_{VID}}{B} \quad (3)$$

Index-supported queries do not need to process the entire column first. While the delta partition is processed via a table scan as above, the positional information about the main partition is already at hand. Looking up the record id list for a given value id is in $\mathcal{O}(log\ D_M)$ as the number of keys in the index is equivalent to the number of distinct values in the column and therefore the number of dictionary entries. A value id lookup is necessary for each distinct value that meets the query conditions. The returned record id list contains $s * N_M$ entries and has to be read from memory. The actual $s*N_M*W$ can finally be read from the main partition by using the record id list. Data retrieval furthermore requires N_D values to be read from the delta partition to get the positions, followed by reading the $s * N_D * W$ values, thus the overall costs can be calculated by:

$$\begin{aligned} C_{IndexScan} &= (S^d_M * log(D_M) + N_D) * \frac{L_{VID}}{B} \\ &+ S_M * \frac{L_{RID}}{B} + (S_M + S_D) * W * \frac{L_{VID}}{B} \end{aligned} \quad (4)$$

As obvious by this equation, the selectivity has a major impact on the inverted indexes profitability. Figure 3 illustrates this with an one-column table. Neither this nor the other figures referenced in this section feature a delta partition because it influences scan costs for table and index scan alike. At 25% selectivity for a main partition size of about 70,000 rows, the table scan becomes more economic. Since the cost equation for an index scan is logarithmic, the two lines for 15% and 10% cross the table scan line as well, which grows linearly.

A more obvious demonstration is given in Figure 4, including graphs for one, five and ten columns. In contrast to the aforementioned influence of the selectivity, the number of columns does not affect the efficiency, because it again applies to both scan variants. The only other component with impact is the number of distinct values in the column. Its influence is quite obvious since it correlates with the inverted index's size. Less unique values allow for faster lookups in the index as illustrated in Figure 5. An index scan with 10% selectivity on a table with one column and 100,000 rows equals a table scan in terms of cost at about 80% distinct values. The break even point value for s can be calculated by equating $C_{TableScan}$ with $C_{IndexScan}$ (under the assumption that $L_{VID} = L_{RID}$) and reveals the following formula:

$$s_{breakeven} = \frac{1}{d_M * log(d_M * N_M) + 1} \quad (5)$$

Like observed in the graphs, the number of columns does not appear in the equation, neither does the delta partition as it is processed equally in both scan types. The only influencing factors are the number of distinct values and overall number of rows. If either row count or distinct value count increase, $s_{breakeven}$ logarithmically converges to 0, meaning that even when selecting just very few tuples an index scan is more costly than a table scan.

Of course this exact equation is only valid for a table with just one inverted index that can be used to resolve all query conditions. If this is not the case, either another inverted index has to be consulted or the resulting list of records has to be processed by a table scan. A second inverted index means an overhead of another $S^d_M * log(D_M) + S_M$ memory accesses. The returned record id list is intersected with the one from the first index before the respective rows are read from the table, keeping the performance impact low. Alternatively, the resulting rows from the index lookup are checked against all query predicates by using a column scan.

For a table scan the overhead is bigger, requiring another $N_M + N_D$ memory accesses for each column to be checked. If too many columns have to be checked, $C'_{TableScan}$ can become lower than $C_{TableScan}$. In summary the inverted index provides an excellent acceleration of query processing, as long as the query's selectivity does not exceed certain limits depending on table size and number of distinct values.

C. Tradeoffs

Besides the advantage of increased lookup speed, having an inverted index has some downsides as well. These affect both most important factors in almost any application, time and space.

1) Construction Costs: Before it can be used, the inverted index must be built. As there is no positional information besides the column itself, a complete scan of the main partition is required. Processing row by row, the found value id is used to find the corresponding record id list or to create a new one if it did not exist. The find operation is in $\mathcal{O}(log\ D_M)$ and has to be executed N_M times, resulting in a worst case construction cost of:

$$C(IndexConstruction) = N_M * log(d_M) * \frac{L_{VID}}{B} \quad (6)$$

Alongside the initial construction, maintenance costs arise when main and delta partitions are merged. The possibility of a change in the value id to value mapping renders the inverted index invalid and a rebuild is inevitable. Details about the inverted index maintenance including an alternative rebuild strategy are discussed in Section III-E.

2) Memory Consumption: The inverted index data structure uses space in memory. As said before, it contains one value id of size L_{VID} for each distinct value in the column, mapping onto a list of record ids (each L_{RID} in size). Consolidating all those lists yields N_M entries for a main partition's index.

 ISBN: 978-1-61208-185-4

Summed up, the inverted index size for one column is:

$$D_M * L_{VID} + N_M * L_{RID}\ bytes \quad (7)$$

If the assumption applies that $L_{VID} = L_{RID}$, the inverted index consumes at least as much space as the underlying attribute vector of the column ($N_M * L_{VID}$ bytes). In case the level of distinct values reaches 100%, the index is effectively double the size of the column.

D. Delta Partition Index

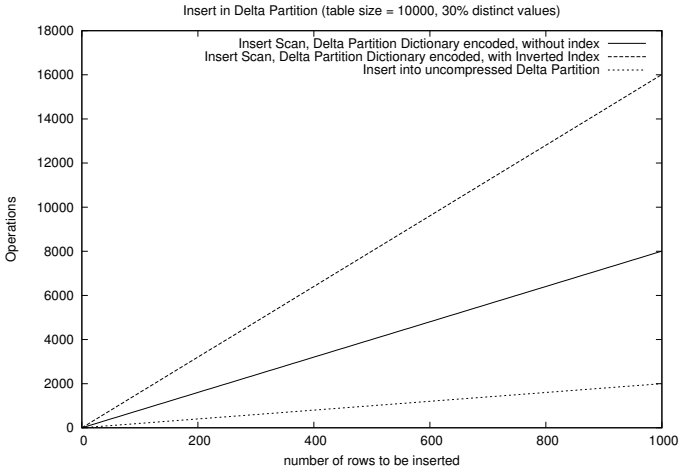

Figure 6. Insertion costs in delta partition.

An additional index on the delta partition can of course speed up lookups since the aforementioned inverted index on the main partition only leads to a retrieval of values from the delta partition with the help of a regular column scan. Unlike the main partition, the delta partition is constantly changed by insert, update and delete operations while the the employed insert only approach implements the logical update and delete queries as technical insert operations with timestamps. An index has to be always valid, therefore insertions into the delta partition entail an index update, which has to be part of the transaction.

In case of applied dictionary compression, an insertion is executed by retrieving the respective value id from the dictionary in $\mathcal{O}(log\ D_D)$, followed by a simple push back into the delta attribute vector in constant time. Having a delta index means another search in the indexes keys to find the appropriate bucket ($\mathcal{O}(log\ D_D)$) and inserting the new record id, which equals a regular insertion in terms of complexity and thus doubles insertion costs as shown in Figure 6. It also becomes obvious, that an uncompressed delta partition naturally offers the best insert performance.

Beneficial to an inverted index on the delta partition is the reduced lookup complexity when processing the delta partition. Using the delta index replaces the column scan with a regular index scan with another lookup. The costs are the same as for two index scans without a delta partition, one regular and the other one with $N_M = N_D$ and $D_M = D_D$. In practice, the delta partition will not grow large in size compared to the main partition, a priori reducing a delta indexes impact on the overall performance in general. Figure 7 demonstrates the theoretical cost reduction in a table containing 100000 rows in the main partition and 10000 rows in the delta partition with varying scan selectivity. The difference in scan costs between index scans with and without delta index constitutes about the ratio of delta to main size, in this case 10%. Recognizing the high insertion costs and the fact that in a productive environment the delta will be orders of magnitude smaller than the main, a delta index becomes fairly unprofitable. We therefore propose to leave the delta partition uncompressed and without an index.

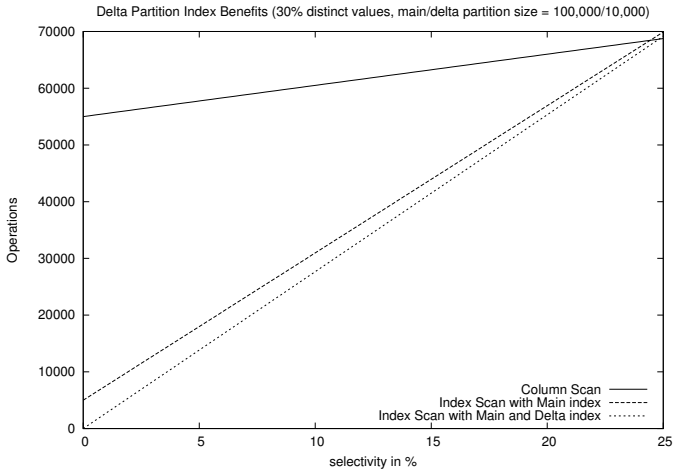

Figure 7. Performance enhancement through Delta index

E. Maintenance

One of the downsides to having an index are the mentioned maintenance costs. When the values of a table change, the index needs to be adapted too. Due to the fact that in in HYRISE a dedicated buffer called delta partition handles all data changes without affecting the main partition the inverted index of the main partition has to be adapted while the merge process compacts the delta and main partition to create a new main partition.

There are different strategies that can be used to retain the validity of the index, which can mostly be associated to one of three basic principles: rebuild, update and merge. A merge could be achieved by combining main and delta partition indexes, but as explained in Section III-D an inverted index on the delta partition is not in favor of the write-optimized characteristics, so we will not go into any more detail about this approach.

The two strategies presented are therefore rebuild and merge.

1) Rebuild Strategy: The rebuild strategy constitutes the most naive approach to inverted index maintenance. Basically, after merging the two storage partitions the former index is

discarded and a new one is created from scratch. There is no difference to the initial construction apart from the delta partition's rows now being indexed as well and consequently the costs are:

$$C_{IndexRebuild} = (N_M + N_D) * log(D_M + D_D) * \frac{L_{VID}}{B} \quad (8)$$

Although this may seem a less favorable option at first glance, rebuilding after merge process completed has the advantage of independence. First of all, it can be employed outside the merge process, a potentially costly and memory intensive operation. Therefore the merge process is not affected whether there is an index on a column or not.

Secondly, the main partition is already merged and due to its nature of being read-only will undergo no further alterations until the next merge operation. This allows the inverted index to be rebuilt in parallel to regular execution. Queries reading from the main partition will be affected shortly after the completion of the merge, as long as the index is not yet ready, but their requests can be served with table scans. Once the index rebuild is finished, table scans can again be replaced by index scans. Further the index rebuild could be delayed or rather scheduled as, for example, explored in the context materialized view maintenance [20].

2) Update Strategy: The update strategy facilitates the special characteristics of the differential store concept that is implemented in the HYRISE database system, the merge process to be exact. During the merge process a new dictionary is created from the combination of the main partition's dictionary and the created dictionary of the delta partition's contents. For the subsequent conversion of table contents a mapping is created that assigns each former value id in both main and delta the respective value id in the newly created dictionary. With this mapping at hand main and delta partition are processed sequentially, applying the mapping whenever the currently read value id has been changed in the new dictionary. As a last step the delta partition's content is appended to the main partition, resulting in a new, empty delta partition, which is ready for new insertions.

Now, the idea behind the update strategy is that a major part of the table, the main partition, is already indexed. Merging main and delta partition also does not change row ids, i.e. the values in the inverted indexes key-value-map, but only the value ids, i.e. the keys since an insert only approach is applied in HYRISE. The value id mapping can be used to update the existing index by changing keys where necessary. The basic balanced tree structure is assumed as basis for the index sorts the keys, thus a key change might result in shifting half or more of the other keys in memory. But since value ids in the main partition are sorted by their respective value, it can be safely assumed that $ValueID_{old} \leq ValueID_{new}$. Parsing the inverted index from the end now will ensure that no values will be shifted or overwritten when applying the mapping, the keys are relocated at constant time, because no re-sorting is necessary. The costs for this update are:

$$C_{OldIndexUpdate} = D_M * \frac{L_{VID}}{B} \quad (9)$$

This update should happen right after creating the new dictionary and mapping. When the delta partition's content is processed in order to apply the mapping, each row has to be extracted, checked and changed if necessary. To avoid a second processing of the delta partition rows, indexing can happen right during this operation. The necessary information (value id and record id) are already present and can be inserted into the inverted index in $\mathcal{O}(logD_M)$. Altogether the additional costs of updating an index during merge are:

$$C_{IndexUpdate} = D_M * \frac{L_{VID}}{B} + N_D * log(D_M + D_D) * \frac{L_{RID}}{B} \quad (10)$$

Despite the fact that the update strategy has to be part of the merge process and may impact the cache friendly sequential processing of table data, the difference as illustrated in Figure 8 is immense in theory.

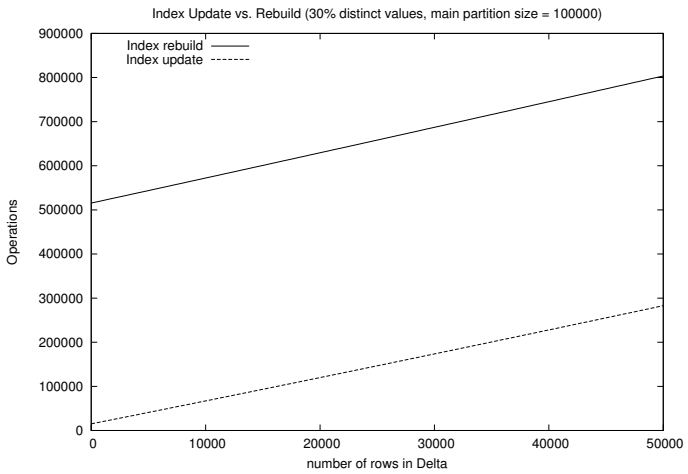

Figure 8. Comparison of index rebuild and update costs with varying delta partition size

IV. Implementation and Evaluation

A. Test Environment

The purpose of this section is to compare the theoretical findings with a working implementation. All tests were performed within HYRISE [2], a main memory hybrid storage engine prototype written in C++. While its main feature is the arbitrary partitioning of tables to accommodate mixed workload scenarios, it provides the necessary implementation of a differential store to allow for comparison with the assumed database layout of the previous sections such as a main and delta partition and dictionary compression. For the tests only the full column-oriented storage without horizontal partitioning were used.

All benchmarks were performed on an Intel Xeon X7560 native Octocore with a base clock of 2.26 GHz and a turbo

boost frequency of 2.66 GHz accompanied by 256 GB DDR3 1066MHz main memory.

B. Inverted Index Implementation

The implemented inverted indexes core is a Standard Template Library (STL) map. Despite offering better best-case complexity, an implementation as a hash-map was discarded due to its high worst-case complexity of $\mathcal{O}(N)$ [21]. Value ids and record ids are represented by 4 byte unsigned integers, the record id lists are STL vectors of the according type. To answer many types of requests, the inverted index provides methods for inserting single or multiple elements and retrieval of position lists for either one, multiple or a range of value ids.

As already proven, a larger number of columns has the same effect on table and index scan. Consequently the influence of the number of columns is excluded from the measurements, only one column is used.

C. Delta Index Insertion Costs

To verify the calculated double insertion costs into the indexed delta partition, a delta partition index was implemented. The graph shown in Figure 9 indicates, that indeed almost twice as many cycles are used when updating the delta partition's index on insert operations. The actual difference is a bit smaller due to overhead created by the software that adds onto both, but still high enough to confirm the theoretical findings.

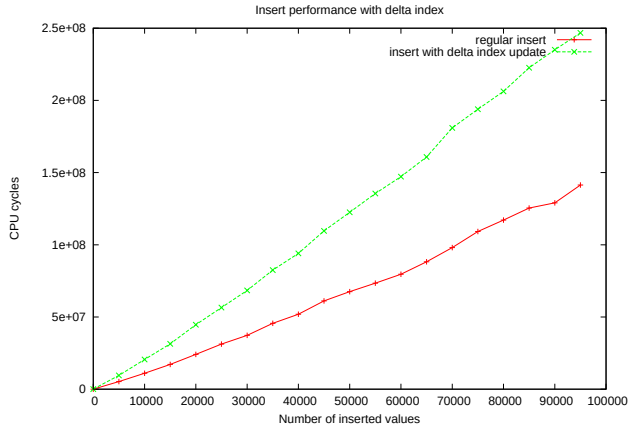

Figure 9. Insertion cost increase with Delta index

D. The Inverted Index in Practice

As done in Section III-B, the influence of variations in main partition size, number of distinct values and selectivity were examined, but this time in practice.

Figure 10, depicts the variations in main partition size with two column scans and two index scans, both fetching once 10% and once 40% of the values. At a lower selectivity level, the index scan outperforms the column scan as expected, at 40% the difference is remarkably smaller. In contrast to Figure 3, the graphs seems not to be logarithmic. An explanation could be the fact that the STL map stores its keys sequentially in memory, allowing for a smaller number of cache misses and in consequence an effective lookup time that is below the theoretically average $log(N)$.

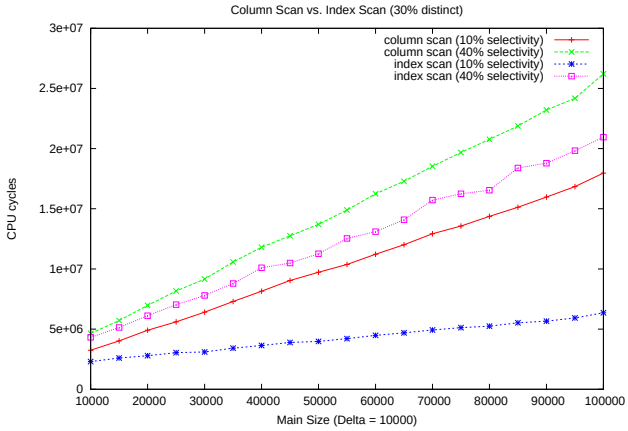

Figure 10. Actual index scan performance with varying main partition size

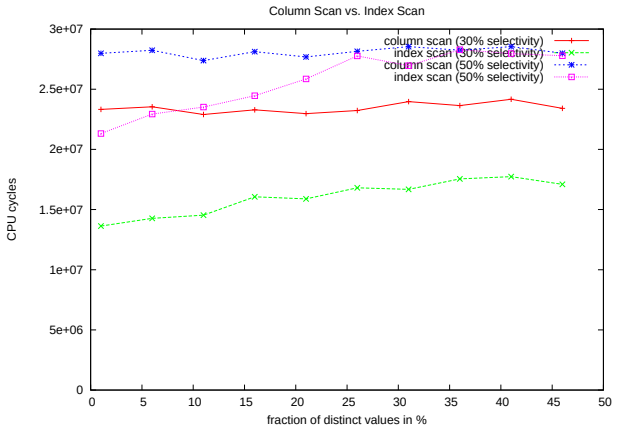

Figure 11. Actual IndexScan performance with varying levels of distinct values

A different perspective is offered in Figure 11. The theoretical analyses have proven that a column scan is not affected by the number of distinct values in a columns, because it scans all values anyways. This theory is justified by the more or less horizontal lines depicting the column scans' costs. The slight deviation can be explained by unpredictable variations in the overall system load, apart from that the scan costs are about

 ISBN: 978-1-61208-185-4

constant.

In contrast, the costs for an index scan increase once reaching 20%-25% selectivity. On 50% distinct values it is outrun by the column scan. The graphs for the remaining two scans will meet somewhere around 70%-80%, almost exactly matching the theoretical graph in Figure 5.

Figure 12 underlines the selectivity's impact on table and index scan performance. The costs for an index scan grow almost twice as fast as the column scan costs, on 50% distinct values the break even point is reached at a selectivity of about 45%, and 65% on 20% distinct values respectively. Also here the curves' shapes resemble to the prediction in Figure 4.

In summary, the measurements and observations of the implementation largely align with the expectations that emerged from the theoretical discussion.

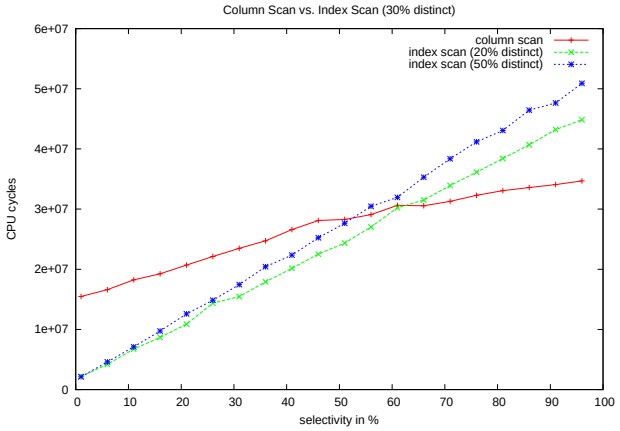

Figure 12. Actual index scan performance with varying selectivity

E. Maintenance

Both the rebuild and update strategy were implemented in HYRISE to verify the theoretical findings. To make the actual difference more obvious, the regular merge was included in the graphs.

Figure 13 shows a result similar to Figure 8 from the previous chapter. The costs for the mere merge process grow almost equally to the number of rows in the delta partition, whereas a parallel update or subsequent rebuild means a linear growth with a greater factor. The update strategy's advantage of not having to re-index the main is pretty obvious and marks a constant offset from the rebuild costs. Yet, as the delta partition grows, the ratio of indexed to non-indexed rows diminishes and changes the overhead reduction from about 70% in the beginning to about 40% at a main-delta-ratio of 1:2.

Deductively it can be said that the update strategy always has a benefit over discarding and rebuilding, even though the difference decreases when the delta partition grows too large.

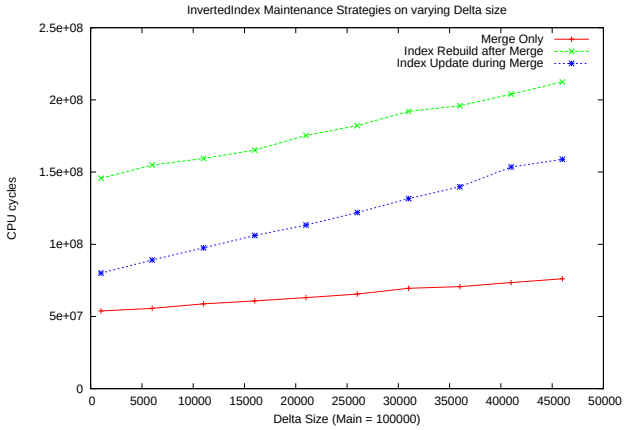

Figure 13. Comparison of implemented index rebuild and update with varying delta partition size

V. Conclusion

In this paper, we showed that an inverted index can dramatically speed up value lookups in a table with a differential store as long as certain conditions apply. If either selectivity or the number of distinct values in an indexed column exceed limits that depend on the actual implementation, the performance gain is lost. As a consequence, the typical profiteers of such an index are selective queries fetching only small subsets or even just a single tuple from the database, which is the standard case in transactional-style queries. Although this paper concentrates on inverted indices for the sole purpose of data retrieval as a distinct query, many different kinds of typical operators in a database could make use of it. When for instance a merge requires only a handful of value ids to be changed in an indexed column, the respective rows could easily be found using the index, which is the case when choosing the so called in-place merge strategy that implements a sparse dictionary.

Besides, it has been presented that an inverted index used for the delta partition adds benefit to the overall performance. While it provides speedup capabilities similar to the main partition index, even though with less impact on the overall performance, the delta partition inverted index doubles the insertion costs. Due to the fact of the delta partition acting as a buffer and thus predictably being orders of magnitude smaller, the minor reduction in scan costs does not justify the high insertion costs as the delta partition is designed as a write-optimized partition.

The last conclusion regards maintenance and how the merge process's nature can be exploited for it. The two strategies explained both have advantages and disadvantages. While updating the index during merge saves memory accesses, it is encapsulated in the merge which makes parallelization difficult. Being more costly in total, this does not apply to the rebuild strategy. Whether or not an inverted index makes sense

 ISBN: 978-1-61208-185-4

is subject to the individual case, i.e. the general pattern of the expected workload. While they cannot outperform full column scans for OLAP style queries, they are beneficial for answering OLTP style queries, which request typically only a few rows. Thereby the inverted index helps to avoid congestion on the main memory interface in a mixed workload environment.

REFERENCES

[1] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access," in *VLDB*, 1999, pp. 54–65.

[2] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden, "HYRISE - A Main Memory Hybrid Storage Engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[3] 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*, vol. 5, no. 2, 2011/2012.

[4] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.

[5] J. Krüger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner, "A Case for Online Mixed Workload Processing," in *DBTest in conjunction with SIGMOD'10*, 2010.

[6] F. Huebner, J.-H. Boese, J. Krueger, F. Renkes, C. Tosun, A. Zeier, and H. Plattner, "A Cost-Aware Strategy for Merging Differential Stores in Column-Oriented In-Memory DBMS," in *BIRTE - in conjunction with VLDB'11*, 2011.

[7] T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," in *VLDB*, 1986, pp. 294–303.

[8] J. Rao and K. A. Ross, "Making B+- trees cache conscious in main memory," *SIGMOD Rec.*, vol. 29, pp. 475–486, May 2000. [Online]. Available: http://doi.acm.org/10.1145/335191.335449

[9] V. Raatikka, "Cache-Conscious Index Structures for Main-Memory Databases," Master's thesis, University of Helsinki, 2004.

[10] F. Transier and P. Sanders, "Compressed Inverted Indexes for In-Memory Search Engines," in *ALENEX*, 2008, pp. 3–12.

[11] R. Guo, X. Cheng, H. Xu, and B. Wang, "Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree," in *Conference on information and knowledge management*. ACM, 2007, pp. 751–760.

[12] N. Lester, J. Zobel, and H. E. Williams, "In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems," in *ACSC*, 2004, pp. 15–23.

[13] I. H. Witten, T. C. Bell, and C. G. Nevill-Manning, "Models for Compression in Full-Text Retrieval Systems," in *Data Compression Conference*, 1991, pp. 23–32.

[14] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel, "Data Compression in Full-Text Retrieval Systems," *JASIS*, vol. 44, no. 9, pp. 508–531, 1993.

[15] A. Trotman, "Compressing Inverted Files," *Inf. Retr.*, vol. 6, no. 1, pp. 5–19, 2003.

[16] H. E. Williams and J. Zobel, "Compressing Integers for Fast File Access," *Comput. J.*, vol. 42, no. 3, pp. 193–201, 1999.

[17] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, June 1979.

[18] M. Grund, J. Krüger, M. Kleine, A. Zeier, and H. Plattner, "Optimal Query Operator Materialization Strategy for Hybrid Databases," in *DBKDA*, 2011.

[19] U. Drepper, "What Every Programmer Should Know About Memory," 2007.

[20] J. Zhou, P.-Å. Larson, and H. G. Elmongui, "Lazy Maintenance of Materialized Views," in *VLDB*, 2007, pp. 231–242.

[21] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003, pp. 3–3.

 ISBN: 978-1-61208-185-4