A Cost-Aware and Workload-Based Index Advisor for Columnar In-Memory Databases

Martin Boissier, Timo Djürken, Rainer Schlosser, and Martin Faust

Hasso Plattner Institute, Potsdam, Germany
{firstname.lastname}@hpi.de
https://epic.hpi.de

Abstract. Optimal index configurations for in-memory databases differ significantly from configurations for their traditional disk-based counterparts. Operations such as full column scans that have previously been prohibitively expensive in disk-based and row-oriented databases are now computationally feasible with columnar main memory-resident data structures and even outperform index-based accesses in many cases. Furthermore, index selection criteria are different for in-memory databases since maintenance costs are often lower while memory footprint considerations have become increasingly important.
In this paper, we introduce a workload-based and cost-aware index advisor tailored for columnar in-memory databases in mixed workload environments. We apply a memory traffic-driven model to estimate the efficiency of each index and to give a system-wide overview of the indices that are cost-ineffective with respect to their size and performance improvement. We also present our *Index Advisor Cockpit* applied to a real-world live production enterprise system of a Global 2000 company.

Keywords: column store, main memory, index advisor, live production system

1 Indices for Columnar In-Memory Databases

The evaluation of database indices typically boils down to three major aspects: (1) performance, (2) maintenance, and (3) storage costs. Indexing large database systems (particularly enterprise systems) is a thoroughly researched topic, especially for transactional enterprise workloads on row-oriented and disk-resident databases. However, recent hardware and software achievements introduced a completely new kind of database system into the field of enterprise systems: columnar in-memory databases. While in-memory databases have been used for decades for special purpose applications, modern servers with terabytes of DRAM allow storing entire enterprise systems completely in main memory. Furthermore, the same hardware developments as well as database research achievements made columnar databases – already established for analytical workloads – suitable for enterprise systems and their increasingly mixed workloads with transactional and analytical workloads on the same machine.

22nd International Conference, ICIST 2016, Druskininkai, Lithuania, October 13-15, 2016, Proceedings, CCIS 639, pp 285-299.
The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-46254-7_23.

For these new database systems, previous assumptions about indices are no longer true in many cases. For example, indices on row-oriented databases basically always improve the read performance, because full table scans are prohibitively expensive on disk-based row stores. For columnar in-memory databases in contrast, scanning compressed columns outperforms accesses via an index in many cases (see Section 2.1).

The picture looks similar for maintenance costs. Columnar in-memory databases usually deploy a multi-level partition hierarchy with small write-optimized partitions handling modifying queries while the remaining tuples are stored in read-optimized partitions (e.g., SAP HANA [3], HYRISE [7], or HyPer [9]). Both partitions are then merged periodically in most systems. In HYRISE and SAP HANA, e.g., indices of read-optimized partitions are solely written when partitions are merged [4]. In this way, the maintenance overhead of indexing has significantly shifted.

Last but not least, storage cost considerations gained more importance. While disk-resident indices with buffer management can almost be considered as free of charge, their main memory-resident counterparts incur high costs since DRAM is still a comparatively expensive resource.

Consequently, the aspects with which an index evaluation is done have shifted notably with the recent rise of columnar in-memory databases. In this paper, we propose a workload-based index heuristic that focuses on performance through evaluating memory traffic and costs. We decided against an automated index evaluation and selection. Instead, we created a tool set that guides database administrators and gives thorough insights into the performance impact of each index and the added costs. We think this tool is of importance for several reasons: First, indices are still one of the major means of performance optimization for database administrators as indices can be added and removed relatively fast compared to other performance optimizations, such as application changes or data model adaptations. Second, automated index selection approaches still have not gained any relevance for real-world systems, as many situations still require the domain knowledge of human experts. One example are indices that are required to meet service level agreements (SLAs) but which might be both expensive and only adding little performance advantage, thus being a good candidate for automated removal. As potential SLA violations can incur serious penalties, these indices have to be kept in the system. Third, existing index evaluations often fail to provide understandable and comparable results that can be interpreted without a comprehensive understanding of the database implementation.

Throughout this paper, we make the following contributions:

- We reason why a memory traffic-based index evaluation provides a decent trade-off between precision and simplicity (Section 2).
- We argue that analyzing the database plan cache provides both thorough insights for the index evaluation on the basis of the actual workload while also making the analysis process simple and comparatively inexpensive, allowing to iteratively run analyses or react fast to sudden issues (Section 3).

– We present a simple and understandable but yet powerful *index coefficient* that simplifies the evaluation of particular indices as well as allowing to compare an index's performance-cost trade-offs (Section 4).
– We present exemplary results of our advisor applied on a live production enterprise resource planning system: the *index advisor cockpit.* This cockpit assists database administrators by analyzing the currently deployed indices and determining how much they are utilized under the current workload. Based on that information, the DBA can make an informed decision about which indices to keep, add, or remove from the system to improve performance or to reduce the memory footprint (Section 5).

2 Memory Traffic & Data Structures

Evaluating an index is a challenging task. First, an index's main characteristics performance and space consumption are often orthogonal. While it is trivial to calculate the space consumption, a thorough performance evaluation is often not feasible for large database systems. On the one hand, measuring the real-world performance is challenging as dynamically changing workloads often show high contention that is hard to measure and simulate accurately. On the other hand, estimating an index's performance is hard since there is a multitude of factors involved. Manegold et al. studied cost models for operations in an in-memory database [11]. They found that a good model is hardly possible without exact knowledge about the underlying hardware (e.g., the cache sizes). Even worse, modern system architectures with multi-hop NUMA and highly contending mixed workloads add additional unknowns.

For our index evaluations, we decided to concentrate on memory traffic as the dominating factor in order to evaluate performance and completely disregard any low-level hardware aspects. With steadily improving processing power (e.g., 15-core CPUs, vectorized instructions, such as SSE/AVX, et cetera) memory access has become the main bottleneck and dominating cost factor in modern database systems [10]. Especially systems without NUMA-optimized physical partitioning to optimize local data access (compare Kissinger et al. [8]) suffer from NUMA effects, making memory traffic increasingly important for steadily growing NUMA architectures. Following Manegold, our model combines both logical costs and physical costs.

2.1 Data Structures

Throughout this paper, we describe data structures and explain our calculations in the context of HYRISE [7]. HYRISE is an open source[1] columnar in-memory database for mixed workloads (OLxP). It shares several concepts with SAP HANA [3], e.g., the main delta architecture, insert-only modifications, dictionary-encoded columns, and MVCC concurrency control.

In this paper, we focus on the following two data structures of HYRISE:

[1] HYRISE on Github: https://github.com/hyrise/hyrise

Table 1. Symbols used in this paper

Symbol	Description
N_c	Length of column c
D_c	Length of the dictionary of column c
E_p	Execution count of plan p

Dictionary-Encoded Columns: HYRISE splits a table vertically into multiple columns. Each column consists of two data structures: a sorted dictionary that stores all distinct values of the column and an attribute vector that reflects the actual column. Instead of the actual values, the attribute vector stores the positional identifier (i.e., value ID) of the corresponding value in the dictionary. To reduce the memory footprint further, the attribute vector is fixed-width bit-packed [7].

Group-Key Indices: The group-key index is HYRISE's primary index structure [4]. The group-key index solely indexes the static main partition that is created periodically by merging the delta partition with it. Since the index is maintained during the merge phase as explained by Faust et al. [4], we do not consider maintenance costs throughout this paper.

The basic idea of the group-key index is to provide a mapping from dictionary values to all positions in the attribute vector that store this value. When creating a group-key index on a dictionary-encoded column, two structures will be created: Firstly, an Index Offset vector (IO) that contains the position in the Index Positions vector (IP) for each dictionary value. The IP vector contains all the positions in the attribute vector (AV) for each dictionary entry. An example is depicted in Figure 1 with a column storing countries and the corresponding index offset vectors and index position vectors.

The IO vector has one entry for each dictionary entry and follows the same order. For each value in the dictionary, the IO vector contains the offset in the IP vector at which the list of positions for that specific value ID starts. The IP vector is a list of all attribute vector positions (or row IDs) where the attribute vector has a certain value. There is no implicit indicator for the value IDs, but instead the IO vector is used as an index for the IP.

In terms of memory consumption, the IP vector has N_c entries (Table 1) with a size of $\lceil \log_2(N_c) \rceil$ bits per entry as it points to a value in the attribute vector. The IO vector contains the same number of entries like the dictionary, i.e., D_c, and because it maps to the IP vector, each entry has a size of $\lceil \log_2(N_c) \rceil$ bits. Thus, we denote MI_c as the total memory consumption of an index for column c:

$$MI_c = N_c \cdot \lceil \log_2(N_c) \rceil + D_c \cdot \lceil \log_2(N_c) \rceil \tag{1}$$

The actual benefit of indices is the fast lookup of values. When using an index to look up the value "*Hungary*", the first step is to find the value ID in the dictionary. However, this step is also necessary when doing a regular

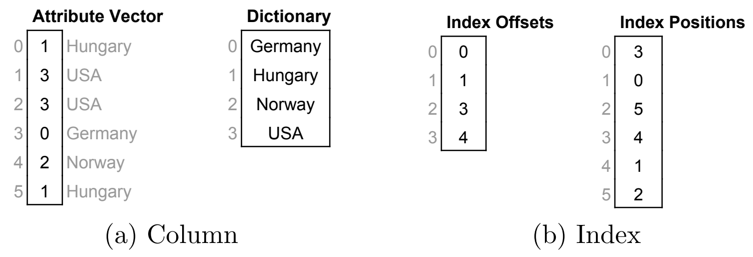


Fig. 1. Data structures for an indexed column storing countries. The two elements on the left show the default data structures in HYRISE to a table column: an (bit-packed) attribute vector that stores positional information referring to the (sorted) dictionary with all distinct column values. The two index structures on the right depict HYRISE's group-key index with an offset vector to find the row identifies in the position list.

column scan without an index, so this can be discarded when comparing the two approaches. The next step is to read the lower and upper bounds in the IO vector. With a complexity of $O(1)$ this step can also be safely ignored when analyzing the performance of an index look-up. The final step is reading the list of positions in the IP vector. Of course, the number of values to be read here depends on the value distribution in that column. However, on the assumption that the values are uniformly distributed, we need to read N_c/D_c entries. And since each entry in the IP vector has a size of $\lceil \log_2(N_c) \rceil$ the average amount of data that needs to be read is:

$$\frac{N_c \cdot \lceil \log_2(N_c) \rceil}{D_c} \tag{2}$$

In most cases, the memory to read using an index is significantly lower than the memory traffic of a regular column scan. Figure 2 shows a direct comparison between IP vector look-up and attribute vector scan (which is basically reading $N_c \cdot \lceil \log_2(N_c) \rceil$ bits). The graph shows that an index results in reduced memory traffic if the dictionary contains at least nine dictionary values (assuming uniform distribution).

3 Workload Analyses

The value of an index depends to a large extend on the characteristics of the actual workload. As a result, we built our analyses on top of the database plan cache[2] of a live production system to obtain both thorough insights into the workload as well as an overview of the system-wide load. The plan cache is a standard component of every major database product (e.g., Microsoft SQL

[2] The Plan Cache of SAP HANA contains frequently executed query plans (including prepared SQL statements) as well as a number of monitoring statistics per plan, such as the aggregated execution count or the minimal/average/maximal run times.

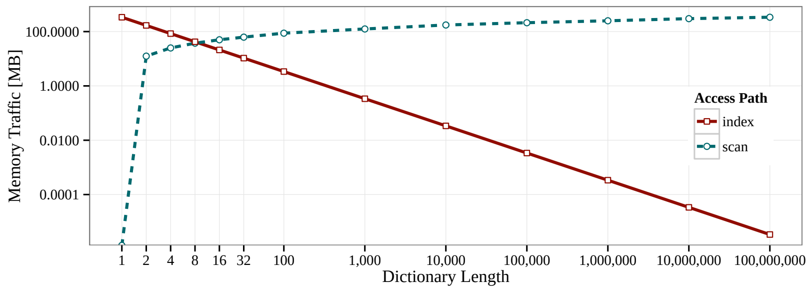


Fig. 2. Memory traffic for a single equi-predicate. Comparing scans on a dictionary-encoded and bit-packed column with index accesses using a group-key on a column with 100 million rows. For columns with fewer than nine distinct values, scanning a compressed column reads less memory from DRAM than an indexed access.

Server's *Plan Cache Object*, Oracle's *Optimizer Statistics*, IBM DB2's *Query engine Plan cache*) and can easily be exported and analyzed offline. Many index advisors require a complete copy of the running system in order to evoke the plan optimizer and evaluate indices. While these approaches are highly accurate, the costs of a second system copy are immense. Other advisors rely on static and simplified assumptions. We decided for a trade-off between both approaches and evaluate our cost-based index heuristics using the database plan cache of the production system. In this way, analyses can be executed offline on an export of the plan cache with minimal additional costs while still preserving a detailed view on the current workload of the system. With this setup, we gain the advantage of having different plan caches for different server nodes. In this way, each node can independently create the indices that yield the best performance for the given workload. Especially systems that partition data along a time dimension see different workloads on server nodes storing recent data compared to server nodes storing historical data [13].

Comparing an index against the default selection mechanism of a database (i.e., column scanning) is helpful to obtain a first decent idea of the efficiency of an index. However, even the best index does not provide any advantages when the indexed column is never accessed during query execution. Even worse, for in-memory databases with limited DRAM resources a never or rarely used index wastes resources, because most in-memory databases deploy DRAM-resident indices to ensure optimal performance.

Consequently, we evaluate indices with their actual usage in the workload. The reason is simple: the later a selection on a column is executed in the query plan, the lower the advantages of an index are. We assume the traditional approach of query plan optimization. Here, first all selections on indexed columns are executed, then selections on non-indexed columns are performed. Both times the attributes are sorted by the selectivity of each attribute, beginning with the lowest selectivity (i.e., the smallest expected number of tuples to be returned).

While the costs of scanning increase linearly with the size of the vector, the costs for index accesses increase logarithmically. Let us consider a typical scan operation that is broken down into multiple chunks that are scanned in parallel on different cores. With a higher selectivity, the probability increases that fewer blocks of the following attribute are qualifying for further accesses. This simplified assumption is not always true. Even for selections with low selectivities there is the chance that all scanned blocks store qualifying tuples. However, for our model we assume that the first evaluated attribute is the one with the lowest selectivity and thus yields a smaller set of blocks including qualifying tuples.

In contrast, for the group-key index the position in the query plan does not matter. For every selection in the plan, the index will be accessed. In case that one or more selections have already been evaluated, the input position list from the previous operator will be merged with the result of the index access.

Consequently, we evaluate each attribute by looking at every query plan that executes a selection on that attribute and adapt the index coefficient depending on the selectivity of the attribute (i.e., the estimated position in the query plan).

4 Index Evaluation

In our model, the objective of an index is to reduce the memory traffic that is required to filter on an attribute. In this section, we present the *Index Coefficient* that quantifies the advantage of an index over the scan in respect to the required memory traffic and the *Index Cost Score* that puts the index memory traffic savings in relation with the index size.

4.1 Index Coefficient

It is an important index evaluation requirement that the coefficient is easy to understand and comparable. Thus, our index coefficient is a linear scalar representing the memory traffic saved by a particular index over the scan.

For example, index accesses on a particular attribute for the entire workload read on average 2 MB for a single equi-predicate to evaluate. The memory traffic for scan accesses on that attribute is 10 MB on average. In that example, the index coefficient would be $5\times$.

We decided for a two-phase approach for the evaluation of indices. In the first phase, the index is measured solely by comparing the index and its workload with the default non-index access (i.e., column scanning). The second phase adapts the coefficient according to the given workload to incorporate the frequency of potential accesses to the index and the order in the query plans. For simplicity, we will concentrate on single equi-predicates throughput this paper.

Columns are denoted by c while C denotes the set of all columns in the system. We consider a set of query plans P. For simplification and only assuming equi-predicates, a query plan p, $p \in P$, is characterized by a set of predicates on columns c, i.e., every plan p is a subset of C, $p \subseteq C$. Note, to evaluate the index coefficient for a given attribute c, it is sufficient to consider solely plans p that contain the column index c (i.e., evaluate a predicate on c).

As already mentioned, we do not aim for an automated index selection that finds the best index configuration for a system. We consider a given global status Z of the system that is characterized by values $z_c \in \{0, 1\}, c \in C$. z_c denotes whether column c is indexed (i.e., $z_c = 1$) or not (i.e., $z_c = 0$). The index coefficient for column c evaluates the effect of having an index over not having an index, where Z with the exception of z_c is stable.

We define the selectivity for single equi-predicates as follows, $c \in C$:

$$S_c = \frac{1}{D_c} \tag{3}$$

To compare the memory traffic for a predicate on a particular attribute c in a given plan p, we define function M for the binary case that c is not indexed (denoted by $^{(0)}$) or is indexed (denoted by $^{(1)}$). In case c is not indexed, the expected data transfer (i.e., number of tuples to read) depends on all other columns that are indexed and thus are accessed first plus all non-indexed columns with a lower selectivity. Please note that the indexed case does not incorporate the position in the query plan by multiplying the selectivities of previous selections in contrast to the scanned access as explained in Section 3.

$$\begin{aligned} M_{c,p}^{(0)} &= \lceil log_2(D_c) \rceil \cdot N_c \cdot \prod_{\substack{i \in p \\ Z_i = 1}} S_i \cdot \prod_{\substack{j \in p \\ Z_j = 0 \\ S_j < S_c}} S_j \\ M_{c,p}^{(1)} &= \lceil log_2(N_c) \rceil \cdot \lceil N_c / D_c \rceil \end{aligned} \tag{4}$$

The next step is to calculate the memory traffic for a given workload (i.e., set P of query plans). Which set of query plans is considered depends on what aspects are of interest. If the user wants to evaluate the efficiency of a particular index, P only needs to include query plans accessing the indexed attribute. If the question is which indices of a particular table are the most/least efficient, P should include all query plans for that particular table. To obtain a system-wide overview and to determine the least efficient indices, P might even include all query plans. Depending on the current focus, the database administrator can adjust the current focus by defining what query plans to consider.

To denote the workload-adjusted traffic we define function MW for a given set of query plans P as follows (E_p denotes the execution count of plan p):

$$\begin{aligned} MW_c^{(0)} &= \sum_{p \in P} M_{c,p}^{(0)} \cdot E_p \\ MW_c^{(1)} &= \sum_{p \in P} M_{c,p}^{(1)} \cdot E_p \end{aligned} \tag{5}$$

With these functions at hand we can finally calculate the index coefficient IC_c for a given column c that determines the relative memory benefit:

$$IC_c = \frac{MW_c^{(0)}}{MW_c^{(1)}} \tag{6}$$

4.2 Index Cost Score

So far, the index coefficient helps to calculate the expected memory transfer reduction when having an index on a particular column. Although scans are comparatively efficient on in-memory column stores and outperform index accesses in many cases, for most cases indices will have a better coefficient. But since every main memory-resident index increases the memory footprint and thus potentially also the overall system costs, we need to compare the relative index benefit against the costs of that particular index. In our case, the cost is the main memory-allocated space to store the index.

The *Index Cost Score* ICS_c puts the overall saved memory traffic for a given set of query plans P in relation to the cost of the index for a given column c (see Equation (1) for the calculation of index size MI_c):

$$ICS_c = MW_c^{(0)} - MW_c^{(1)} - \alpha \cdot MI_c^{\beta} \tag{7}$$

We use two penalty parameters α and β. α is used to scale the index size accordingly to the size of the workload (i.e., the number of query plans and their execution counts), since the overall memory traffic reduction is relative to the size of the workload. The scaling parameter β is used to penalize large indices in a suitable way. Since performance gains and memory footprint reductions are usually orthogonal optimization objectives, α allows balancing the trade-off (also directly by the DBA in the front end to analyze the impact).

The model is well suited for dynamic programming approaches with which it is also possible to calculate (near) optimal index configurations with respect to given footprint constraints (see [12] for a such an approach). However, further use cases for the model are part of continued research.

5 Advisor Cockpit

In this section, we present the *Index Advisor Cockpit.* We also give a short overview of the production system we have run our analyses on. Then, we show exemplary screenshots of the HTML-based front end as well as a few insights from the analyzed real-world system.

Besides obvious requirements like accurate performance indicators and valuable insights for DBAs, we think a suitable user interface is crucial for any advisor tool. Nowadays, there is a vast array of tools for DBAs to control and monitor their databases. Unfortunately, many of them lack simple and easy understandable metrics that provide valuable insights without the need to fully understand the underlying database engines.

An easy to understand user interface is especially important for systems, such as enterprise systems. The analyzed SAP ERP installation consists of ∼112,000 tables with over 240,000 indices. We think our tool set with a workload-driven approach and a straightforward index evaluation model is a promising approach helping database administrators to handle such large systems.

5.1 Plan Cache Analysis

All data in this demo has been extracted from the live production enterprise system of a *Global 2000*[3] company:

- the uncompressed data amounts to over seven terabytes
- one of the most recent versions of an SAP ERP system including operational reporting on transactional data (i.e., OLxP)
- database system handles ∼1.5 billion queries each day

For our analyses, we have exported the plan cache table as well as several statistic/administrative tables (e.g., column and index information of the system). The extracted plan cache included ∼300,000 query plans accounting for ∼5 billion query executions. All analyses shown here can also be executed directly on the production data. The computation overhead on the production is manageable, because most queries simply request data that has already been aggregated by the database system itself (e.g., overall plan executions of a particular query plan, average run times, et cetera).

An important capability of our tool set is the analysis of query plans that access logical database views. The inclusion of database views is increasingly important as modern enterprise systems use them, e.g., to ease the transition to new systems [13] and to disassemble sophisticated queries into multiple layers of logical views.

5.2 Table Information

The user has the opportunity to examine specific tables in detail. The *table detail page* displays details about the table in general, its columns and indices as well as information about primary key indices. To give the user a fast overview of the memory footprint of the table, the top of the page displays a graph about the overall memory footprint broken down in the table's elements (Figure 3).

5.3 Index Information

Figure 4 shows two aspects of the index detail page. This page presents information about the currently viewed index, i.a., the ratio between all queries accessing the corresponding table and queries on the corresponding table that furthermore potentially access this index (see Figure 4a). Most importantly, the

[3] Global 2000: http://www.forbes.com/global2000/

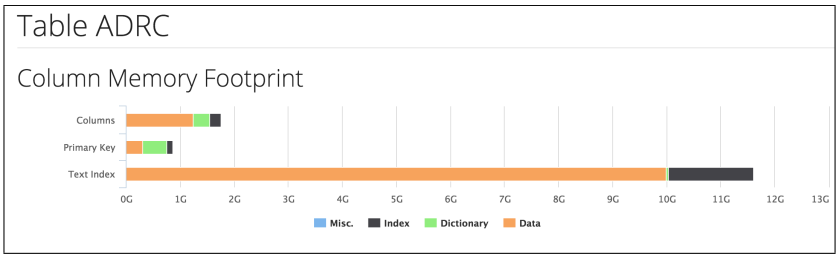
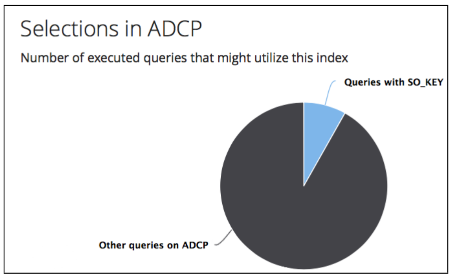
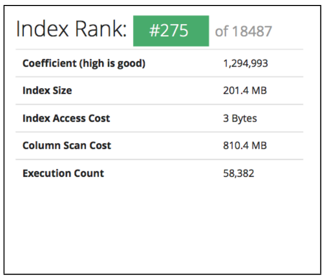


Fig. 3. Memory consumption overview for table `ADRC` storing customer address information. It is shown that text indices are responsible for most of the DRAM consumption of that – comparatively small – master data table.

user can analyze the index's ranking. In this example, the index ranks comparatively well with its rank 275 out of the 18,487 indices (see Figure 4b) that have been ranked (we limited the ranking to indices that have been accessed at least once). Additionally, all factors used to calculate the coefficient are shown to ease understanding of the resulting coefficient. In this particular case, the evaluation of a single equi-predicate using a scan reads over 800 MB from DRAM, while an access via the index only reads three Bytes from DRAM.



(a) Workload information.


Index Rank: #275 of 18487

Coefficient (high is good)	1,294,993
Index Size	201.4 MB
Index Access Cost	3 Bytes
Column Scan Cost	810.4 MB
Execution Count	58,382

(b) Index ranking.

Fig. 4. Index detail page: (a) ratio of queries on table `ADCP` that potentially access this particular index, (b) the ranking information.

5.4 Index Ranking and Issues

The *Index Ranking* page lists all indices that have been ranked according to their coefficient. The table can be searched and sorted. For each index, additional information, such as the column size, number of query executions and the source table are displayed as well.

The *Issues* page prepares two lists of indices that could be of interest to the database administrator. The first list displays all indices that are not used by the traced workload at all. That means that not even a single query with a selection on the indexed column has been found.

Secondly, a list of the least economical indices is provided for the user. Amongst these indices are typically indices that are automatically created by the ERP system, because they are advantageous for row stores. More importantly, for main memory column stores these indices are often wasting resources. Furthermore, they might even slow down the entire system in case the query optimizer does not recognize that scanning incurs a lower memory transfer.

Workload analyses have shown that over 5,000 indices in the system have not been accessed once during the recorded plan cache period. The footprint of these indices is 18.1 GB. We think this information is important for database administrators as those indices waste resources. Nevertheless, we do not propose to automatically remove these indices since even though they might be accesses rarely, they can still be necessary for a variety of reasons (e.g., to avoid SLA violations for certain processes).

6 Related Work

In this section, we briefly discuss related work in the areas of cost models for in-memory databases and index selection.

6.1 Cost Models for In-Memory Databases

Manegold et al. published an extensive low-level cost model for database operators for in-memory databases [11]. Using their cost model, the authors estimate the cost for accesses to different levels in the memory hierarchy. They compare the effects of different memory access patterns to create building blocks that can be used to estimate the costs of query plan alternatives. In contrast to our comparatively simple approach, Manegold et al. can adjust their model to a given hardware setup using a calibrator tool.

Schwalb et al. published a cost model for estimating query costs for different physical column organizations (e.g., uncompressed or bit-packed columns, sorted or unsorted dictionaries, et cetera) [15]. In contrast to Manegold et al., who focus on join operators, Schwalb et al. compare scan and lookup operators on different column layouts. Furthermore, the authors focus on mixed workload with the additional evaluation of inserts into read-optimized data structures. However, similar to Manegold et al. the work by Schwalb et al. does not investigate more complex systems with highly concurrent mixed workloads that falsify several assumptions made throughout this work.

6.2 Index Selection

Finkelstein et al. worked on the topic of index selection as early as 1988 [5]. Similar to our solution, they analyzed the workload queries to gather information about how tables and their attributes are accessed. Based on these access statistics and the execution frequencies of workload queries, the authors try to find an optimal index configuration.

Another tool that allows the analysis of existing indices is *AutoAdmin* by Chaudhuri et al. [1]. AutoAdmin aims to be a multi-purpose tool set to analyze currently deployed indices, assist the database administrator in index selection and also perform what-if analysis for hypothetical indices. The tool integrates into the Microsoft SQL Server and gathers information about the workload during run time. This information can then be used to assess the benefits of the current indices, propose new (better) indices and do what-if analyses for theoretical changes to the current index setup.

The index selection approach by Finkelstein et al. has a clear focus on the workload similar the tools proposed in this paper, however, they focus on indices that are about to be created instead of evaluating the ones that are already in place. AutoAdmin, on the other hand, is much closer to what we are trying to achieve and offers workload-driven analysis of indices that are currently in place. However, since AutoAdmin is tightly integrated into the SQL Server, it only allows live analysis and no offline analysis.

The aforementioned approaches as well as most other approaches in this field are designed for disk-resident row stores. And since the index selection for columnar in-memory database differs considerably in many cases, those approaches cannot be simply adapted by changing expected access latencies and block sizes.

7 Future Work

There are several additional aspects we want to cover with our index coefficient and the index cost score. One aspect is the inclusion of additional data structures and approaching new hardware technologies, such as non-volatile memory (NVM). Another aspect is a declarative language than could narrow the gap between application requirements and the database.

7.1 Additional Data Structures

Besides the bit-packed and (sorted) dictionary-encoded columns and the group-key indices, we want to include additional data structures. On the one side, there are several alternatives to store columns, e.g., uncompressed columns as used in HyPer [6] or run-length encoded columns [14]. Both approaches have a significant impact on the expected memory traffic for a column scan. On the other side, we want to incorporate additional indices into our cost model, e.g., block indices [14] and compressed group-key indices (e.g., using *Golomb* or *Simple9* compression as used in SAP HANA).

7.2 New Storage Technologies

Upcoming non-volatile memory (NVM) promises to provide a persistent and byte-addressable DRAM alternative with larger capacities than DRAM. With the expected capacity increase of 5× at a lower price point and an expected latency orders of magnitudes better than PCIe-connected devices [2], NVM is a natural fit for index cost considerations. At the moment, data access to other storage tiers than DRAM is considered as too slow for applications with low latencies requirements. This is even the case when data is stored on high performance PCIe-connected NVMe NAND flash drives. However, the expected performance of NVM adds a new layer to the cost model for indices that provide performance advantages but are currently too large to be stored in DRAM.

While it is simple to add complexity to a cost model, let us emphasize that we explicitly decided for a model yielding results that are simple to interpret. We think that the main challenge will not be to find a cost model incorporating NVM, additional data structures and more. The main challenge will be to find a simple, yet powerful, and applicable cost model.

7.3 Declarative Languages for Business Requirements

As of now, one of the main tasks of a database administrator is to ensure that the database meets business requirements. Such requirements include highly prioritized processes, such as end-of-quarter closings or service-level agreements (SLAs). As already mentioned, we think that a completely automated index selection approach is not feasible for real-world systems since the database is not aware of external application requirements. We think a declarative language could narrow this gap. Using such a language, the application developer can define requirements of the application, e.g., certain processes that need to finish within a defined time frame. The language can be parsed and interpreted by the database to automatically optimize itself within the given constraints.

8 Conclusion

We presented a workload-aware heuristic to evaluate indices of columnar in-memory databases. The proposed model is cost-aware. This is increasingly important for main memory-resident indices that have a direct impact on the DRAM footprint and thus on the overall costs of a system.

The index coefficient is a straightforward linear scalar helping database administrators to understand and interpret the efficiency of a particular index without requiring a comprehensive understanding of the implementation details of the database system. The coefficient ranking allows making informed decisions about the benefits and costs of an index configuration. We think that the presented heuristic with its focus on memory traffic is simple, but yet powerful.

Furthermore, we presented our *index advisor cockpit* applied on a production enterprise system of a Global 2000 enterprise. Particularly interesting was – as soon as the actual workload was incorporated – the unexpected high number of apparently unused indices that exist in a standard ERP installation and that waste main memory without further index tiering concepts. Furthermore, we determined indices that are no longer necessary using columnar data structures.

References

1. Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, 1998, Seattle, Washington, USA*, pages 367–378, 1998.
2. Subramanya Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 15:1–15:16, 2016.
3. Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.
4. Martin Faust, David Schwalb, Jens Krüger, and Hasso Plattner. Fast lookups for in-memory column stores: Group-key indices, lookup and maintenance. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012*, pages 13–22, 2012.
5. Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.
6. Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid OLTP & OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.
7. Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.
8. Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A NUMA-aware in-memory storage engine for analytical workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014*, pages 74–85, 2014.
9. Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *International Conference on Management of Data, SIGMOD 2016, San Francisco, CA, USA*, 2016.
10. Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.
11. Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, 2002*, pages 191–202, 2002.
12. Stratos Papadomanolakis and Anastassia Ailamaki. An integer linear programming approach to database design. In *ICDE 2007, 15-20 April 2007, Istanbul, Turkey*, pages 442–449, 2007.
13. Hasso Plattner. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, 7(13):1722–1729, 2014.
14. Hasso Plattner and Alexander Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications.* Springer, 1st edition, 2011.
15. David Schwald, Martin Faust, Jens Krüger, and Hasso Plattner. Physical column organization in in-memory column stores. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013*, Lecture Notes in Computer Science, pages 48–63. Springer, 2013.