Improving Tuple Reconstruction for Tiered Column Stores: A Workload-Aware Ansatz Based on Table Reordering

Martin Boissier, Alexander Spivak, Carsten Meyer
Hasso Plattner Institute, Potsdam, Germany
{firstname.lastname}@hpi.de

ABSTRACT

Improving memory consumption of main memory-resident databases by evicting infrequently accessed data to secondary storage layers can significantly reduce the TCO. But traditional approaches for data eviction such as paged-based LRU algorithms are unsuitable for OLxP systems with their scan-dominated workloads. In this paper, we present a data tiering ansatz for columnar in-memory databases that is workload-aware and retains full performance for analytical queries. To make our tiering ansatz also applicable for transactional workloads, we will focus on the topic of table reordering to improve tuple reconstruction, which is one of the the main bottlenecks for tiered columnar databases. We evaluate our approach using data and workload of a *production SAP ERP enterprise system* of a *Global 2000* company.

1. DATA TIERING FOR OLXP SYSTEMS

Main memory-resident databases have been in the focus of database research in recent years. One trend is the rise of columnar in-memory databases that are capable of handling both transactional (OLTP) and analytical (OLAP) workloads, so-called OLxP [7, 9]. Besides such software achievements, the rise of in-memory databases is caused by hardware achievements such as falling DRAM prices and increasing DRAM capacities per server. Nonetheless, main memory is still a scarce resource and comparatively expensive. Improving main memory utilization is thus a major objective for any in-memory database as more free memory can improve performance (e.g., using faster but more memory-intensive algorithms), allow larger systems to be stored on a single machine, or – the focus of this paper – to improve cost efficiency by evicting infrequently accessed data to less expensive storage layers. The process of evicting less relevant data to secondary storage layers while frequently accessed data is stored in main memory is called *Data Tiering.* Looking at production database workloads shows that data accesses are often skewed and frequently request a small fraction of the data [2]. This observation is in line with the working set model [5] that says that for each process there is a subset of pages that are accessed more frequently.

Our approach tries to exploit the working set model observation by improving the main memory footprint of columnar in-memory databases by partially evicting infrequently accessed tuples to secondary storage layers.

ACSW '17, January 31-February 03, 2017, Geelong, Australia

ISBN 978-1-4503-4768-6/17/01...$15.00
DOI: http://dx.doi.org/10.1145/3014812.3014838

1.1 Aspects of Data Tiering

To tackle the topic of data tiering for columnar in-memory databases end-to-end, we will first discuss the three main aspects of data tiering. Then, we briefly introduce our tiering implementation and discuss how we address each tiering aspect with our implementation.

Data Classification The task of data classification seeks to answer the question which data is actually relevant. For disk-resident row-stores, this question can be considered as solved using classical LRU approaches on page level. However, for columnar databases – with both transactional as well as analytical workloads [13] – efficient data classification remains an unsolved problem.

Data Access Given that the data has been classified into relevant (hot) and less relevant (cold) data and the cold data has been evicted to cheaper and slower storage tiers: *how can the database avoid unnecessary accesses to cold storage (thus significantly slowing down query run times)?* The aspects of data access include finding partition criteria that allow for efficient partition elimination as well as indexing on cold data to avoid unnecessary accesses.

Data Placement Modern server systems are no longer solely two-tiered architectures with fast volatile DRAM and slow non-volatile hard drives. In between these two tiers, various forms of solid-state drives have emerged, such as PCIe-connected NVMe devices with bandwidths of several GB/s. Additionally, non-volatile memory (NVDIMM) is on the horizon promising to revolutionize server computers. The challenge is to seamlessly move data between different tiers and store data on the tier with the best fitting characteristics in terms of storage costs, bandwidth, or latency.

Figure 1 depicts the tiering approach [1] we implemented in Hyrise[1], an open source hybrid in-memory database. Hyrise uses MVCC to ensure ACID properties and data modifications are done via an insert-only approach [8]. Columns are dictionary-encoded, whereby the *main partition* is read-optimized and regularly merged with the *write-optimized* delta partition, which handles data modifications.

[1]Hyrise on GitHub: https://github.com/hyrise/hyrise

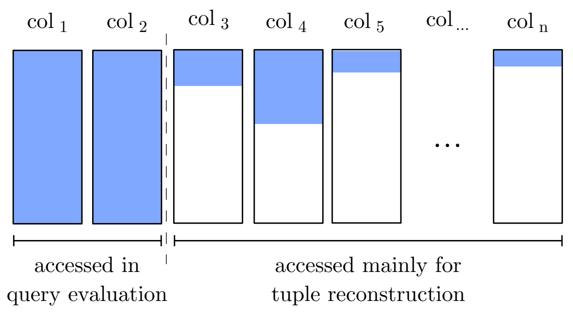

col 1
col 2
col 3
col 4
col 5
col ...
col n
accessed in query evaluation
accessed mainly for tuple reconstruction

Figure 1: Schematic visualization of our tiering ansatz. DRAM-allocated data shown in blue. Columns frequently used for query evaluation are DRAM-resident, while columns mainly accessed for tuple reconstruction are predominantly stored on secondary storage. As most applications have a small distinct working set of tuples that are regularly materialized, the challenge is to find an optimal reordering to gather such tuples and allocate them in DRAM.

Data Placement. To efficiently move data based on its relevance and access patterns, we use the tiering-aware *mmap* replacement discussed in [12]. This *mmap* replacement extends Linux's *mmap* with additional logic that uses so-called *data temperatures* in order to transparently move pages between different storage tiers. Linux's virtual file system is bypassed to improve random access latencies.

Data Access. As we will explain in Section 1.2, our approach differs from most published tiering approaches. For that reason, access to tiered data during query evaluation [3] is not hampered at the expense of lower eviction rates for narrow tables. For tuple reconstruction, data access is managed via our *mmap* replacement that transparently handles the access to disk in case of cold data materializations. This cold data is stored in an uncompressed format for further materialization improvements as presented.

1.2 Data Classification

Our approach to classify relevant data differs from most traditional approaches where wrongly classified data has a direct severe impact on query evaluation.

Traditional Approaches. There are various traditional approaches to determine whether data is considered to be relevant or not: *(i)* analyzing access frequencies given a workload, *(ii)* predefined business rules based on expert analysis or semantic knowledge, and *(iii)* caching approaches.

The idea behind the access frequency analysis is simple: the more often data is accessed, the more relevant it is and thus the higher the probability or recurring accesses is. There are different approaches to track data accesses in a fast and resource-efficient way. One example is logging system presented by Levandoski et al. [11] that uses sampling and smoothing techniques to reduce the size of memory consumption.

In contrast, rules-based approaches require manual work by application experts [14]. An example for such a rule could be: *"All paid receipts older than six months are cold"*. The benefits of business rules are manifold: domain knowledge can introduce high precision, especially considering the time parameter. In business applications, financial reports are often created monthly by analytical processes. A frequency-based system would probably not identify the needed data as hot, due to the fact that it is requested only monthly and the log history is usually shorter (the same is true for service level agreements, SLAs). But at the same time, experts potentially formulate rules too weak, losing optimization potential. Furthermore, modern OLxP systems are increasingly complex due to the consolidation of multiple platforms (e.g., operational reporting, warehousing, transaction processing, data mining applications, etc.) making rule-driven expert approaches not feasible in the long run.

Another approach to identify important data is caching. An example for such techniques is page buffer tracking using *Least Recently Used* or *Most Recently Used* policies [4]. Such methods can be faster and consume less memory than frequency approaches. On the other hand, they can be less accurate and have high book keeping overhead.

Our Approach. Transactional and analytical access patterns are different in various aspects. For the presented project in this paper, the most important aspects are query filtering (the most costly part of query evaluation) and tuple reconstruction. Anticipating analytical access patterns is challenging, because analytical applications do not follow predefined processes and are usually not automated. Further, the purpose of analytical queries is often gaining new insights or finding hidden patterns, which leads to a broad variety of query patterns hampering predictability. On the other hand, transactional workloads usually follow predefined business processes and are automated to a high degree. Also, OLTP accesses typically access only a very small subset (i.e., the working set). Thus, predicting upcoming accesses based on previous workloads is comparatively easy.

This has led to our tiering ansatz shown in Figure 1. All attributes that have been used for query evaluation are completely stored in DRAM. This way, we ensure that both analytical as well as transactional operations are as fast as they would be in a fully main memory-resident database. We think that any other approach will significantly thwart OLAP performance as analytical queries often access very large parts of the data and are hard to predict.

For OLTP accesses, we try to exploit the working set model assumption that only few data items are regularly accessed. All columns that are not part of the query evaluation are assumed to be accessed solely for tuple reconstruction. Each of these cold columns is horizontally partitioned into hot and cold partitions. Using Hyrise's variable data layouter, we partition each column on its own. If we would partition all hot/cold columns by a common partition criterion, some few columns would dominate the partitioning decision reducing the optimization benefit. This way, we are later able to allow query evaluation even on cold columns as long as we can guarantee that the most frequently accessed filter values are DRAM-located. The decision to use no more than two partitions per column was made to avoid hardly manageable complexities incurred by maintenance, query optimization, and query execution. After each column has been partitioned, the goal is to cluster tuples that are often materialized and store them together in the DRAM-allocated hot partition. Cold partitions are stored on secondary storage.

The reason for our model is the overhead of tuple reconstruction in column stores, which is further amplified when attributes are stored on disk. For typical ERP tables with over 300 attributes, a single `SELECT *` operation requires $300 * 2$ page reads for dictionary-encoded columns, reading 2.4 MB for a single tuple (assuming 4KB disk pages).

Consequently, our goal is to avoid reconstruction from secondary storage. The main challenge of our approach is to ensure that most tuple reconstructions can be processed without accessing secondary storage. This leads us to the main challenge discussed in this paper: *How can we reorder tuples to ensure fast reconstructions on tiered data?*

1.3 Table Reordering

Table reordering has been researched from multiple angles. Probably the most well known approach is the clustered index where a table is sorted by the index key to improve scanning [15]. Lemke et al. use heuristics to find a sorting order that both balances data compression and query performance [10]. A different approach called database cracking has been presented by Halim et al. [6]. Here, the database iteratively reorders the table to increasingly adjust to the current workload, improving query run times over time.

Our motivation for table reordering is different as the table ordering is orthogonal to query evaluation. The purpose of table reordering in our case is optimizing the data layout in a way that future tuple reconstructions access DRAM-allocated tuples with a high probability and only access slower storage tiers as a rare exception.

The presented reordering approach is completely independent of the extraction of relevance information. This way, our method can be applied in many typical database scenarios independent of their definition of data relevance and the information extraction methods. The basic idea of our reordering approach is to provide a simple but flexible definition of data importance. Therefore, we request as input for each data block the information whether a block can be evicted to disk or not. A data block is ideally a single table cell but can be also a group of cells (i.e., a chunk). We discuss this in more detail in Section 2.2. This input provides enough information to restructure the data dividing it in two partitions: *(i)* tuples kept in main memory and *(ii)* tuples evicted to disk. Throughout this paper, we refer to these two partitions hot and cold partition.

To tackle all described obstacles, we follow the approach of reordering table rows. Our goal is to find an optimal order of rows to minimize the number of cold data in hot partitions, while cold partitions evicted to disk are not allowed to contain hot data.

This paper focuses on the following topics:

- Formalization of the table reordering problem as an optimization problem.
- Introduction of rules and heuristics that significantly reduce complexity.
- Evaluation of presented techniques using real-world application access patterns and data.

1.4 SAP ERP Data

As a foundation for the experiments in this paper, we traced a live production SAP ERP enterprise system of a *Global 2000* company. We analyzed the system and executed a query replay to retrieve access frequencies as explained in [2]. The SQL workload trace was recorded over a period of three days and contains over 50 M queries. We use the trace data and the executed replay of the ERP system to identify data cells as hot or cold in dependence on their access frequencies. Section 2 describes in more detail how hot and cold are defined exactly. To provide meaningful results, we have chosen the `BSEG` table for our experiments. The `BSEG` is the central transactional table of the SAP financials module and stores accounting documents. It is one of the largest and most accessed tables in an SAP system and therefore representative for our purposes. Additionally, the `BSEG` table has the highest analytical load. In the regarded dataset, the table contains 345 columns and more than 100 million rows. Dealing with such large tables makes our task of finding an optimal row ordering even more challenging.

In the following sections, we show how to formalize the problem as an optimization problem and present various approaches to tackle this problem.

2. REORDERING ANSATZ

In this section, we formalize our ansatz of the separation of hot and cold data as an optimization problem and show the need for non-trivial algorithms due to the unacceptable performance of naïve approaches.

2.1 Task Formalization

Given a table and a workload, the goal of our system is to provide for each column in the table a separation in two partitions, the hot partition and cold partition. We define the cold partition as the partition containing cells that were never accessed during a given workload. The hot partition contains all column cells that are not in the cold partition. The separation is defined by a simple rule:

$$partition(cell) := \begin{cases} hot & row\_id(cell) < \tau \\ cold & row\_id(cell) \geq \tau \end{cases}$$

The reasoning for such a separation is given in Section 1.

To improve the impact of the hot and cold partitioning, we want to minimize the number of never accessed cold cells in the hot partition and therefore to minimize τ. Having no background information about the semantic meaning of underlying data, the only possible way is to change the order of table rows to sort rows with more cold cells to the bottom of the table while placing rows with often requested cells to the top. The alert reader has already noticed that we reorder complete rows and not only cells in each column. The reasoning is simple: if we reorder each column separately, we would need to store the related row id for each cell to be able to reconstruct tuples during the reconstruction or operation computation. This would introduce an intolerable, additional memory and computation overhead.

Therefore, we can define our task as the search of a table row permutation that minimizes the total number of cold cells in hot partitions for all columns. To measure the quality of the hot and cold separation on the permutation, we define a metric called *error*:

$$\begin{aligned} error_C(column) &= |\{cell \in column \mid state(cell) = cold \\ &\qquad \wedge row\_id(cell) < \tau\}| \\ error_T(table) &= \sum_{c \in columns} error_C(c) \end{aligned} \tag{1}$$

Using the error function, we can express our task as an optimization problem:

$$solution = \underset{p \in \pi(table)}{\arg\min}\ error_T(p)$$

A naïve solution is the examination of each possible row permutation, storing the best permutation found so far. The complexity of that naïve approach depends on the tables row count r and can be computed as $O(r) = r!$. Even providing the optimal solution, this approach cannot be used in real database systems, since the dimensions of table row count are in ranges of hundreds of millions. In Section 3, we are going to introduce some approaches which decrease the complexity significantly. But first, we need to find a way to formalize the terms hot and cold.

2.2 Hot and Cold Data

In this section, we present two methods to compute the *hotness* of a table cell and discuss advantages and disadvantages of both approaches. In this work, we do not discuss the workload analysis required to extract the information we use. For this paper, we assume that we have a log providing a timestamp for each distinct cell access and a query id of the query accessing the cell [2].

As already mentioned previously, we define cells never accessed by a query as cold, otherwise as hot:

$$state(cell) := \begin{cases} cold & log\_entries(cell) = \emptyset \\ hot & otherwise \end{cases}$$

For simplicity reasons in the following computations, we introduce the *hotness* function mapping the state of a cell to the interval $[0, 1]$. Using the upper definition, we retrieve the following mapping:

$$hotness_{state}(cell) := \begin{cases} 0 & state(cell) = cold \\ 1 & state(cell) = hot \end{cases}$$

In contrast to the state function, the hotness definition provides the possibility of taking the number of cell accesses into consideration. This can be useful for weighing cells: if a cell is accessed just once in the whole workload, it is colder than a cell accessed thousands of times. In this case, the hotness is computed in the following way:

$$hotness_{query}(cell) := \frac{\#log\_entries(cell)}{\max_{c \in cells} \#log\_entries(c)}$$

We use this definition in Section 3.4 to determine columns having similar query behavior. The state-based definition is used in Section 3 to reduce the computation complexity. To finish the consideration of the hotness definition, it must be mentioned that the query number-based definition can be transformed into the binary state-based by introducing a hotness threshold.

The `BSEG` table of the traced production system consists of 345 columns containing more than 100 millions rows, which result in 34.5 billion cells. These dimensions make it necessary to find heuristic simplifications to store and process hotness data. Examples for such simplifying approaches are sampling or summarization. In our research, we decided to use the second approach, because sampling would not take all cells into account, which could lead to worse results for our algorithms. Instead, we propose to summarize cells into equally sized *chunks*, based on their row ids. The computation of the hotness is also performed on the chunks accordingly. One efficient way to assign cells to a chunk, which is also supported by standard SQL methods, is to use grouping-based on integer division of the row id by a predefined chunk size:

$$chunk_i(column) = \{cell \in column \mid \left\lfloor \frac{row\_id(cell)}{chunk\_size} \right\rfloor = i\}$$

The introduction of chunks forces an adaption of the hotness definition. We follow both approaches, the state-based and query-based, resulting in following formulas:

$$hotness_{state}(chunk_i) = \frac{\sum_{cell \in chunk_i} hotness_{state}(cell)}{chunk\_size}$$

$$hotness_{query}(chunk_i) = \frac{\sum_{cell \in chunk_i} hotness_{query}(cell)}{chunk\_size}$$

By intuition, the state-based chunk hotness is the fraction of distinct accessed cells in the chunk. The query-based chunk hotness is the average count of query accesses to cells.

The summarization of cells in chunks improves the resource consumption, but influences the results of the partitioning algorithms. In Section 4, we investigate the impact of chunk sizes on the algorithms' performance and results.

Both approaches, the state- and the query-based one have different advantages and disadvantages. On the one hand, the state-based method does not require logging the query identifiers, just a flag per cell or a counter per chunk. Furthermore, it has less computational effort. On the other hand, the state-based approach does not differentiate between often and rarely accessed cells. This might be a problem considering workload traces over longer time periods. The probability of being accessed once only, at least at the row insertion moment, will increase with the increasing time period, so old data may be defined as hot even being cold in reality. The query-based approach can tackle these problems, providing more flexible data insight, which is paid with a significantly higher computational effort caused by the persistence of distinct query ids for each cell/chunk.

To specify the estimation of the hotness metric for workload traces over longer time periods, we propose to apply the already defined metrics on predefined time intervals. The intervals should be chosen depending on the task. For the partitioning task itself, a coarse time interval of multiple days is advisable to avoid the misclassification of hot data as cold. For the clustering task described in Section 3.4, a finer time interval of an hour provides more detailed information about the query behavior on each column.

3. SORTING PROBLEM

The problem of sorting the table via reordering tuples is a complex one as outlined in Section 2.1. In the following sections, we introduce rules that substantially decrease the complexity of finding an optimal solution and present heuristics based on these rules. These heuristics can be applied in cases when the complexity of finding an optimal solution would be too high. Furthermore, we discuss a simple estimation to distinguish cases when a heuristic or the optimal algorithm should be used. The considered algorithms are based on a binary hotness metric mapping the cold state to zero and the hot state to one. Ways to define such metrics were described in the previous section.

3.1 Optimization Rules

For the following discussion, we introduce two types of bit vectors: columnar and row vectors, containing information about the hotness of a column or a row. Due to our limitation to binary hotness metrics, bit vectors can be used for storing the hotness information. If we do not specify the type of the bit vector, we mean implicitly a row vector. Such a vector describes the hotness behavior for a row of cells:

$$\vec{r} = \begin{pmatrix} hotness(cell_{column_1,r}) \\ \dots \\ hotness(cell_{column_n,r}) \end{pmatrix}$$

The upper definition can be easily adopted to chunks, whereby the row means the range of rows described by the chunk. The presented solutions do not differ for cells or chunks, as they work only on the bit vectors. A columnar bit vector is defined in the same way as the row vector, but instead of iterating over columns, we iterate over rows.

Using the column bit vector definition, we can define the partition strategy as following:

- the last vector item points to the last hot column cell
- the column part up to this cell is the hot partition (])
- the column part after this cell is the cold partition ([)

This approach allows keeping the cold partition clean of hot cells and the hot partition as small as possible.

Blocking Rule

We found the trivial but important rule: "equal bit vectors should be ordered together". It is enough to find an ordering of distinct bit vectors to place all vectors. Due to the fact that the error function depends on cold cell counts, it is necessary to keep the information of how many equal bit vectors exist, even if we need to place just the distinct ones. Therefore, we introduce so-called *blocks*. Each block is described by a bit vector and the count of such bit vectors in the original table. The table from Figure 3 contains two blocks: $\{value : 101,\ count : 2\}$ and $\{value : 100,\ count : 2\}$. In the following explanations, we write a block shorter like 101×2. We call the bit vector of a block *value* and the count of such bit vectors *length*. The blocking rule is important to reduce the number of elements to be reordered. During our analyses of real-world enterprise applications, we saw that such systems exhibit only few distinct access patterns and have consequently only few distinct hotness vectors. For example, observing the `BSEG` table with its 345 columns and 100 million rows, we found only 1,053 distinct bit vectors, reducing the task complexity by a factor of 10,000.

In Section 2.1, we discussed the complexity of the naïve permutation approach for the sorting problem. Using the blocking rule, we can improve the naïve approach sorting only distinct bit vectors. The maximum number of differing row bit vectors in the database is: $2^{\#columns}$. Therefore, it would be enough to test all $2^{\#columns}!$ permutations instead of the previous $\#rows!$ permutations. Nevertheless, in cases of broad tables, the table can contain less rows than are needed to place all possible bit vectors. Therefore, we should adopt our complexity estimation to:

$$O(\#rows, \#columns) = \min(\#rows,\ 2^{\#columns})!$$

Trivial Blocks Rule

The trivial blocks rule implies that the block containing bit vectors full of ones should be always placed as first block in-

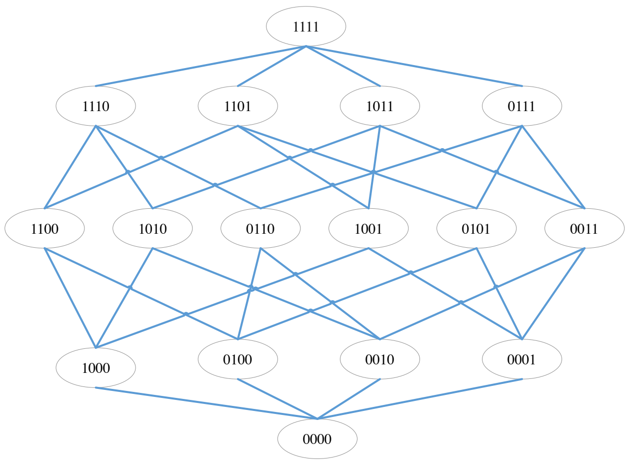


Figure 2: Hierarchical parent-child relations as lattice

to the ordered table. Respectively, the block containing bit vectors full of zeros should be always placed as last block. The block with ones represents rows where each cell is hot. If such rows are not placed on the top of a table, they always introduce an additional error. The block with zeros represents rows where each cell is cold. Such rows should be placed at the bottom of the reordered table, otherwise they would introduce additional errors.

The impact of this rule on the efficiency of heuristic methods is relatively low. However, it reduces the factorial complexity to find the optimal solution without heuristics. We estimate the complexity of the naïve approach as follows:

$$O(\#rows, \#columns) = \min(\#rows,\ 2^{\#columns} - 2)!$$

Parent-Child Rule

This rule provides the foundation for efficient reordering strategies. First, we define a *parent-child* relation between two bit vectors. A vector C is child of vector P if P has each 1 C has. For example, 0010 is child of 0011, but not a child of 0001. In other words, the parent vector has ones at least at the same positions like the child vector. We can use the logical disjunction to define a parent-child test function:

$$is\_child(C, P) = \begin{cases} child & C \vee P = P \\ not\ a\ child & else \end{cases}$$

Figure 2 provides a lattice of all possible bit vectors with the length of four. Each possible path downwards represents a parent-child relation. For example, 1110 is parent of 0010, but not of 0001.

Now, we can formulate the rule as: "Filling the reordered table from top: each child should be placed after all its parents. Filling the reordered table from bottom: each child should be placed before the according parent". To provide an example for this rule: in Figure 3, the vector 100 is placed after the vector 101, because 101 is a parent of 100.

We need to show that it is always better to place a parent before the child. To do so, we need to recall our partitioning strategy: the partition decision is made considering the last 1 in the column vector. Let us assume a parent block would be placed after its child in the table. Due to the definition of the parent-child relation, the parent has ones for the same columns as the child, but also for columns where the child has zeros. For these columns, this placement of the parent would introduce an additional error, because the cold cells of the child would reside in the hot partition.

The two topmost rows of the unsorted table in Figure 3 provide a good example for this case. The child 101 is placed before the parent 111. For column B, where the child has a cold cell and a parent a hot, the hot partition would finish at least at the parent's cell, introducing an additional error through the child's cold cell.

We have shown that it is bad to place a child before a parent, we need to show that it is never bad to place a parent before a child. This proof is trivial, because the child has only ones the parent also has. Therefore, the child can never introduce more errors as the parent and for that reason, the child should be placed always after the parent.

Looking at Figure 2, it becomes clear why this rule is so important. Imagine we would try to find the best ordering by testing each possible permutation as we have presented in the naïve approach. Considering the rule, we know for example, that we do not need to try any permutation where 1010 would be placed before 1110, 1011 or 1111. Using the rule, it is possible to show that for a full lattice like in Figure 2, it would be enough to test each possible permutation of the blocks with only one 1 in the bit vector. In the presented figure, those are only elements from the second lattice row counting from the bottom. The number of such elements is equal to the number of table columns allowing us to reduce the complexity from Eq. 2 down to Eq. 3:

$$O(\#rows, \#columns) = \min(\#rows,\ 2^{\#columns} - 2)! \quad (2)$$

$$O(\#rows, \#columns) = \min(\#rows,\ \#columns)! \quad (3)$$

In the case of lattices with missing elements, the complexity increases again. In the worst case, only the top half of the lattice is given, forcing us to test all permutations of elements from the middle lattice row. This lattice row contains an exponential number of elements in dependence on the column count leading to similar complexity classes as shown earlier. Nevertheless, in real-world applications we observed only few distinct bit vectors with a high number of parent-child relations allowing to reduce the complexity in comparison to naïve approaches.

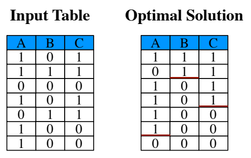
Bitmask Rule

The bitmask rule is a continuation of the parent-child rule. The reordered table is filled from the bottom: each child of the disjunction of all vectors already placed can be immediately placed. Let us first regard an example of the rule's application and discuss the impact on complexity afterwards.

Assuming we are going to reorder a table and have created a reordered table that we are filling with given blocks starting from the table's bottom. An example of such a table and elements to place is given in Figure 4. To apply the rule, we need to compute the disjunction of all vectors already placed. The result of the disjunction, we call *bitmask*. In this case, we compute the bitmask as:

$$bitmask = 0010 \vee 0011 \vee 0100 = 0111$$

The rule says that we are allowed to place all children of the bitmask immediately and in any order to the free table slots. In our example, we place elements 0101, 0111, and 0110 since they are children of the bitmask, but not 1110.



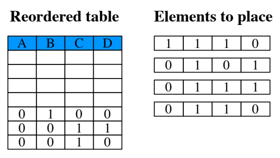
Input Table

A	B	C
1	0	1
1	1	1
0	0	0
1	0	1
0	1	1
1	0	0
1	0	0

Optimal Solution

A	B	C
1	1	1
0	1	1
1	0	1
1	0	1
1	0	0
1	0	0
0	0	0

Figure 3: Example of an input table and optimal reordering.



Reordered table

A	B	C	D
0	1	0	0
0	0	1	1
0	0	1	0

Elements to place

1	1	1	0
0	1	0	1
0	1	1	1
0	1	1	0

Figure 4: Example of placing new elements for bitmask rule application.

The reasoning for the rule correctness is similar to the proof of the parent-child rule. Recalling that the partitioning for each column depends only on the last 1 in the column bit vector, it is clear that if such 1 was already placed, values in upper cells do not matter. Constructed as the disjunction of all vectors already placed, the bitmask implicitly tracks the information for which columns the partitioning decision has already been made. This means, bit vectors having ones in a subset of such columns can be placed in an arbitrary order, because they only influence columns where the partition decision has already been made. Elements fulfilling this condition are by definition children of the bitmask.

As seen in the example, the bitmask rule can significantly influence the complexity in a positive way. Furthermore, this rule allows handling the worst case for the parent-child rule more efficiently. As pointed out earlier, this worst case was a table containing elements only from the top half of the lattice. Using the parent-child rule, it was still necessary to test all permutations of the middle row elements. In case of four columns, like in Figure 2, that would be six elements and accordingly $6! = 720$ permutations. Applying the bitmask rule, we can state that each time we choose two elements, their disjunction will be parent of a further element. Therefore, we need to test maximally $\binom{6}{3} = 20$ permutations.

3.2 Optimal Solution

In this section, we provide an optimized way to find the optimal solution for the sorting problem. As we already pointed out in the corresponding sections, rules are useful to significantly decrease the high complexity of the problem. Our algorithm for finding the optimal solution derives from the naïve approach. We try to test all permutations using rules preventing tests of clearly bad permutations.

The steps of our approach are the following:

1. Create blocks from row bit vectors (blocking rule).
2. Disregard the full ones and full zeros blocks in following computations (trivial blocks rule).
3. Construct all relevant permutations recursively starting from the bottom (parent-child and bitmask rules).
4. Compute the table error for each permutation and decide for the permutation with the lowest error.

The most interesting step is the third one. Due to its complexity, we explain it with an example. Assume we have the vectors 0010, 1001, 1010 and 0111. We use a recursive strategy to create permutations and start from the bottom of the table enabling the usage of the bitmask rule. To find candidates for the first element, we consider elements that have no children or whose children are already placed. In this case, no elements are placed and the vectors 0010 and 1001 have no children. We decide to start the first permutation with 1001 and place it at the bottom of the table. Following the bitmask rule, we compute the bitmask as 1001. Logically, we have no elements that are children of the bitmask in the first step. Now, we search candidates for the second element of our permutation. Remembering, that we are not allowed to place parents before children, we have only one candidate for this position: 0010. As 1010 and 0111 are parents of 0010. We recompute our bitmask with $1001 \vee 0010 = 1011$. Following the bitmask rule, we can place now all children of the bitmask in arbitrary order. In our case, vector 1010 is a child of the bitmask 1011 and should be placed as the third element of our permutation. The bitmask needs no adoption. Now, we search the fourth and last element to finish the creation of the first permutation. We have only one candidate: 0111. As we can see, all its children (0010) are already placed and therefore, we can place this candidate. The first permutation is: $(1001, 0010, 1010, 0111)$.

To create the second permutation we need to go back in our recursive trace until we reach a moment, where we had multiple candidates to choose and to select the next candidate. In the previous permutation, the last selection point was the choice of the first element. We chose between 1001 and 0010. To create the second permutation, we need to select now the element 0010. Our bitmask is 0010 accordingly. For the choice of the second permutation element, we have now three candidates: 1001, 0111 and 1010; due to the fact that the only child of 0111 and 1010 has already been placed. For the second permutation, we choose 0111 as the second element. The bitmask is consequently 0111, but we have no children of the bitmask that are not placed already. For the third element, we have two candidates: 1010 and 1001. We choose 1001 and retrieve as result the bitmask 1111. The last unplaced element is child of the bitmask and can be placed immediately resulting in the following permutation: $(0010, 0111, 1001, 1010)$. To create the third permutation, we need to search the moment of the last decision and decide for another candidate (i.e., another third element).

It is possible to prune permutations to reduce the complexity of the problem. In our small example, our approach would generate six permutations to test for being optimal. The naïve approach would generate 24 permutations instead. Nevertheless, for applications in real-world scenarios with a huge number of distinct vectors having only few parent-child relations, the optimal solution approach would fail due to high complexity as we show in Section 4. Therefore, we present heuristics in the next chapter, which address this problem.

3.3 Heuristics

In this section, we present four heuristics differing in their quality, complexity and ordering approaches. Two heuristics try to reorder the table placing elements starting from the top. In contrast, both other heuristics start from the bottom.

The first heuristic we are going to discuss is called *Fulls First*. The idea is quite simple. Starting the placement from the top of the table, we iteratively place the block that has the most ones in the bit vector. If we have found multiple blocks with the same number of ones, we decide on the block with the highest length value.

The idea of this heuristic is to make the smallest possible local error. Placing as many ones as possible at the top of the table, we gather hot cells at the table's top and approximate the best solution this way. The Fulls First heuristic makes use of the blocking, trivial and implicitly of parent-child rules. The complexity of the heuristic is the same as the complexity of the best sorting approach with blocks as input elements. Consequently, using Quicksort Fulls First has a complexity of $O(n) = n \log n$, where n is the number of distinct bit vectors in the table.

The second heuristic we discuss is called *Shortest Zero Chain First* (SZCF). The basic idea here is that we iteratively place the block with the shortest number of zeros starting from table's top. The number of zeros in a block can be computed as the product of number of zeros in the bit vector and the length of the block. For example, for the block 01011×4, we compute the number of zeros with $\#_0 01011 \cdot 4 = 2 \cdot 4 = 8$. The reasoning for this heuristic is similar to the Fulls First ones. We try to choose blocks keeping the error local as small as possible. In contrast to the Fulls First heuristic, SZCF considers the block length in each case and not only if multiple blocks have the same number of ones.

The complexity of the second heuristic is the same as for the first one. Since each placement decision is done only comparing all blocks, it is enough to sort the blocks using the heuristic criteria. The complexity for sorting is $n \log n$. It must be mentioned that in contrast to the Fulls First heuristic, SZCF does not implicitly fulfil the parent-child rule and it needs to be considered while sorting.

A more sophisticated heuristic is called Best Local Chain heuristic (BLC). The idea is to place elements iteratively starting from the table's bottom while trying to make local the smallest possible error. To reach this goal, we try to defer the partitioning decision per column as long as possible. To do so, we choose the block with largest number of zeros in each placement round, which adds as few new ones as possible to the bitmask. While searching for the next elements, we always take the parent-child and the bitmask rules into account. The name of the heuristic describes the principle: the approach tries to create zero chains for the column bit vectors at the bottom of the table as long as possible.

Before we analyze the complexity, we provide an example. Let us assume that we have to reorder the blocks:

$$0010 \times 3 \quad - \quad 0100 \times 4 \quad - \quad 0110 \times 7 \quad - \quad 1000 \times 2$$

The first step of the heuristic is to find the block with the largest number of zeros. In our example case, it is the block 0100×4, hence 0110 cannot be placed until all children are placed. Now, we search for a continuation that has the most zeros and adds as few new ones as possible. Each of the remaining three elements would add a new 1 to the bitmask and the element with the best length, having no unplaced children, is 0010. After the placement, we can place 0110 following the bitmask rule. The fourth element to place is the last element: 1000. Since we have placed elements from the bottom, we need to revert our ordering resulting in: $(1000, 0110, 0010, 0100)$. As we can see, BLC makes use of all rules to reach better performance.

The complexity of this heuristic is higher than of both previous ones. In contrast to the other heuristics, BLC considers the previous placements and needs to recheck all unplaced elements in each round. Therefore, the complexity of this heuristic is in the worst case: $O(n) = n^2$.

The last and most complex heuristic we are going to present is called *Deep Dive*. The method is a derivative of BLC and is inspired by N-gram models and chess programs. The basic idea is the following: similar to BLC we try to find the best local chains from the table's top but instead considering only one element at a time, we consider all legal permutations with k elements. After finding the best k-deep permutation, we choose its first element and place it in the reordered table. Then we repeat the procedure until all elements are placed. The process is comparable to chess programs: calculating on a predefined depth, the best-ranked move is found, executed and the ranks of next moves recalculated.

Like the BLC heuristic, Deep Dive also makes use of all rules. The complexity of the approach depends on the choice of k. In the worst case, we can estimate the complexity with:

$$O(n) = \sum_{i=1}^{n} \left(\begin{array}{c} i \\ k \end{array} \right) < n \cdot \left(\begin{array}{c} n \\ k \end{array} \right) < n^{k+1}$$

We found that choosing k with 2 or 3 provides acceptable results keeping acceptable performance. In Section 4, we discuss the impact of k on results quality and runtime.

3.4 Clustering

In this section, we want to discuss a heuristic approach to reduce the underlying complexity of the sorting problem. As already mentioned, the complexity heavily depends on the number of distinct bit vectors. The number of distinct bit vectors in turn depends on the number of columns and the query behavior. If we were able to reduce the number of columns, we would also reduce the problem's complexity. In our research, we discovered that many columns have similar querying behavior and could be regarded during sorting as single column without a significant loss of hotness information. For that reason, we propose to cluster columns with similar hotness behavior and to combine columnar vectors of each cluster into a single vector using logical disjunction. The reasoning for the use of the disjunction operation is the same as for chunks: we do not lose information about the hot state, at most about the cold state. To follow this approach, it is necessary to discover similar hotness behavior of columns. We see three possibilities for similarity metrics:

- Similarity between sets of templates affecting columns
- Distances between state-based column vectors
- Distances between frequency-based column vectors

The choice of an appropriate similarity metric is a field of further research. The first approach is complex in implementation and parameter tuning. The second approach also has an important disadvantage: it is not possible to differentiate key columns from columns requested in OLAP style. For these reasons, we prefer to use the latter named approach. Nevertheless, it must be mentioned that this method requires the computation of frequency-based column vectors, which cannot be directly used for the sorting problem. But as already pointed out, each frequency-based vector can be easily transformed into a state-based one.

In addition, we propose using k-Means as clustering approach for a simple reason. The parameter k allows controlling the number of resulting columns and consequently the number of potentially distinct bit vectors. Thus, k-Means allows controlling the worst-case complexity.

During our research we applied the clustering approach to the `BSEG` table and found that 6 artificial column vectors would be enough to represent all 345 real columns in acceptable quality.

4. EXPERIMENTS

In this section, we examine the performance and the quality of the presented methods: the optimal solver and the heuristics. For solid reasoning, we use two test data sets. The first data set is the one described in Section 1.4. The second data set is generated randomly to simulate different parameters not occurring in our real world example, but whose impact should be determined. We assume that each random decision is made following a uniform distribution.

We have outlined in Section 3.2 that the presented strategies have less effect on the performance the fewer parent-child relations can be found in the lattice. To provide fair measurements, we use the following strategy for the generation of the random data. We define two parameters: the column count n and the fraction p describing the number of randomly chosen lattice elements allowed to be used. For example, setting $n = 5$ and $p = 0.25$, we retrieve a table with 5 columns and $0.25 \cdot 2^5 = 8$ randomly chosen blocks. The length of each block is set to one, ensuring a better comparability between generated tables.

4.1 Optimal Solver

Relating to the optimal solution's search, two questions are of importance. How does the column count influence the runtime and how does the number of distinct elements influence the runtime? To answer both questions we used the following experiment. For each column count in the interval $[4, 7]$, we generated for each fraction $p \in \{0.25, 0.5, 0.75, 1.0\}$ 100 random tables and measured the minimal, maximal and average runtime. The computation of the optimal solution was aborted when the runtime exceeded one minute and these timeouts were counted.

Table 1 shows the results of the experiment. As expected, we see that the average runtime grows significantly with the number of columns. One interesting observation is that the runtime difference between four and five columns broad tables is marginal but between five and six the difference is significant. The reason for this behavior is the following one. Tables with only four or five columns have so few distinct blocks that the parent-child rule and the bitmask rule can be applied nearly for each table reducing the task complexity. The second interesting observation is that the fraction parameter influences not only the average runtime but also the deviation of runtime values. As we can see from measurements with 6 columns, fraction values of 0.5 and 0.75 cause a high variation in the runtime behavior. The reason for this behavior has already been outlined: the fewer elements from the lattice are in the table, the lower the chance for parent-child relationships.

From the experiments, we can conclude that the optimal solution can be computed only for tables not broader than 6 columns in an acceptable time. Using better computational resources and accepting longer waiting times, the number of columns could be increased but not by an order of magnitude. Nevertheless, this low number of columns may be sufficient if the column vectors represent bit vectors build from business rules or created via the clustering approach, which was described in Section 3.4.

4.2 Heuristics

We performed three experiments to measure the quality and the efficiency of different heuristics. For these measurements, we used both data sets: the data set derived from

Table 1: Run time measurements for the optimal solver.

columns	**fraction**	**min. [s]**	**avg. [s]**	**max. [s]**	**timeouts**
4	0.25	0.24	0.29	0.55	0
4	0.5	0.24	0.25	0.26	0
4	0.75	0.25	0.32	0.56	0
4	1.0	0.30	0.32	0.37	0
5	0.25	0.29	0.32	0.45	0
5	0.5	0.32	0.38	1.22	0
5	0.75	0.36	0.52	0.96	0
5	1.0	0.50	0.54	0.83	0
6	0.25	0.40	0.69	2.35	0
6	0.5	2.00	4.87	22.98	0
6	0.75	4.20	12.76	54.25	0
6	1.0	7.93	9.08	12.34	0
7	0.25	8.18	30.20	57.66	5
7	0.5	55.65	55.65	55.65	99
7	0.75	-	-	-	100
7	1.0	-	-	-	100

the BSEG table data and random generated tables of appropriate size. The experiments on the BSEG table show how the heuristics perform on distinct bit vectors appearing in a real world application. Such vectors cannot be randomly generated because the underlying data distribution is unknown. The main purpose of these experiments is to show that the heuristics can be applied with success in business applications. In contrast, experiments with smaller random tables allow analyzing the behavior of the heuristics on tables not independent of any data distributions. These experiments are necessary to make statements about quality and efficiency valid for all varying scenarios.

The BSEG table has 345 columns and therefore too many elements to provide an optimal solution. To observe the impact of the chunking method on the data, we applied the heuristics to the BSEG table, creating chunks for each order of magnitude between 1 and 10,000. Table 3 provides the measurement results. The error rates of the heuristic are computed as the fraction of the error after sorting comparing to the original error. It must be mentioned that the original error is computed directly on tables resulting from chunking because we want to measure the quality of the heuristics and not the quality of chunking. The runtime is provided in seconds. For the Deep Dive heuristic the depth parameter was set to two because higher values led to unacceptable runtimes.

The chunk size influences the number of distinct elements significantly. The original table contains 1,053 distinct elements. We would expect that this number decreases with the size of the chunks but the opposite is the case. Each chunk of size b consists of b bit vectors. The higher b becomes, the higher is the probability that the disjunction of these b creates a new bit vector differing from previous ones. As we can see from the table, this reasoning is not always valid. For $b = 1,000$ and $b = 10,000$ the number of elements decreases again. Therefore, we need to take into account that previously only thousand distinct elements existed. Combining them to chunks of such large sizes, the probability increases to have all distinct vectors in each chunk always producing the same vector after the disjunction. Consequently, the number of distinct vectors decreases after chunking. The low number of distinct vectors for the chunk size of $10,000$ suggests that chunk sizes should be chosen smaller. Otherwise, the result of the sorting will be less expressive because too many cold table cells would be masked as hot.

The Deep Dive heuristic performs best closely followed by the Shortest Chain First approach (Table 3). Expectably, the

Table 2: Comparison of error rates for different heuristics on a 6 columns wide table.

heuristic	**fraction**	**minimal error [%]**	**average error [%]**	**maximal error [%]**
$DD_{k=3}$	0.25	0.00	3.10	27.27
$DD_{k=3}$	0.5	0.00	10.25	32.00
$DD_{k=3}$	0.75	0.00	11.64	34.48
$DD_{k=3}$	1.0	0.00	0.00	0.00
$DD_{k=2}$	0.25	0.00	7.46	36.36
$DD_{k=2}$	0.5	0.00	14.94	37.93
$DD_{k=2}$	0.75	0.00	13.58	34.48
$DD_{k=2}$	1.0	0.00	0.00	0.00
BLC	0.25	0.00	27.38	62.50
BLC	0.5	16.00	49.95	105.56
BLC	0.75	28.85	55.57	100.00
BLC	1.0	48.33	59.92	90.48
SZCF	0.25	15.62	44.07	92.31
SZCF	0.5	35.48	64.63	113.64
SZCF	0.75	37.74	77.21	127.59
SZCF	1.0	66.67	75.03	105.00
FF	0.25	9.09	45.23	92.86
FF	0.5	35.56	74.07	135.29
FF	0.75	36.00	73.14	128.57
FF	1.0	70.00	76.41	120.00

Fulls First heuristic shows the worst error values because it does not regard block lengths in contrast to other methods. The runtimes of all approaches depend on the number of blocks. For the chunk size of $10,000$ the runtimes are the lowest, for a size of 100 the highest. This behavior confirms our theoretical assumptions presented in previous sections. The measured runtimes also show three different complexity classes: the similar runtimes of SZCF and FF, the runtimes of BLC and the runtimes of DD.

Summarizing the experiments, we can state that SZCF and DD are both applicable for the sorting problem in tables similar to the BSEG. Deep Dive allows slightly better results but requires significantly more time.

In the next experiment, we compare the heuristics' quality. For this experiment, we use random data generated with the previously described method. We use six columns broad tables to be able to find the optimal solution and to compare the heuristics against. We vary the number of distinct elements setting the fraction parameter as we have done in Section 4.1. Table 2 shows the results. As we can see, the Deep Dive heuristic performs best. With increasing depth, the heuristics seems to perform better. We investigate this in a further experiment. In contrast to the previous experiment, SZCF shows a very low success rate rarely finding a good solution. This means that SZCF shows good performance in special cases, but is not a good heuristic in average. The majority of distinct vectors of the BSEG are sparse or almost full. Therefore, placing the full vectors at the top of the table, the SZCF heuristic can achieve very good results, but in other cases the results are worse.

A further insight we gain from this experiment is that a fraction of distinct elements in the interval $[0.5, 0.75]$ leads to worst results for all heuristics. This means that such fractions provide enough elements for a high probability of a wrong decision in a heuristic step but not enough elements to make sufficient use of parent-child rules. For fraction values of 0.25 and 1.0 the heuristics perform at best. Especially the Deep Dive heuristic is able to find the optimal solution

Table 3: Heuristical error rate for the full BSEG table (error as percent of original error, runtime in seconds).

chunk size	blocks	error SZCF	runtime SZCF	error FF	runtime FF	error BLC	runtime BLC	error $DD_{k=2}$	runtime $DD_{k=2}$
10,000	121	5.26%	0.735	10.18%	0.872	8.45%	1.015	6.65%	10.253
1,000	1,233	8.51%	64.920	12.44%	75.178	11.54%	99.410	5.77%	755.656
100	2,495	6.16%	229.684	11.99%	284.282	7.78%	397.048	4.10%	1,822.058
10	2,219	3.39%	192.575	8.78%	235.940	3.84%	340.963	2.66%	2,182.305
1	1,053	2.71%	51.240	11.12%	59.598	2.90%	73.799	2.41%	864.433

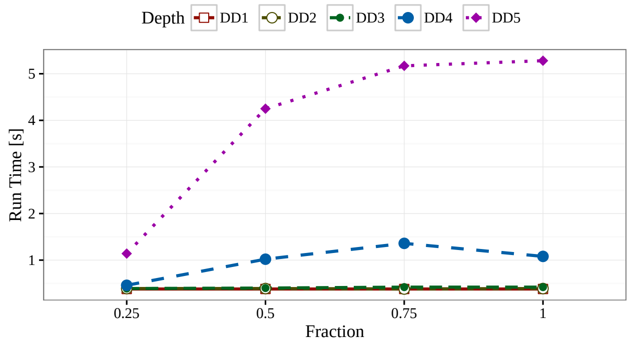


Figure 5: Run times of the Deep Dive heuristic on varying depths.

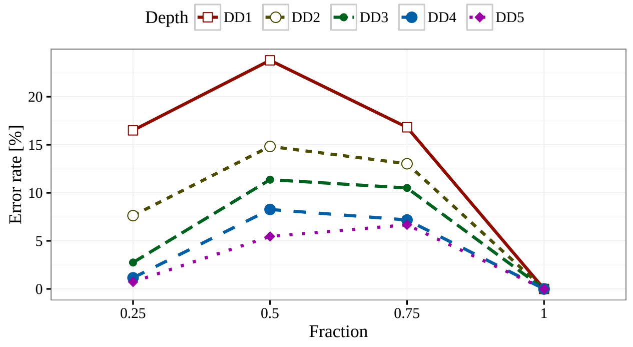


Figure 6: Error rates of the Deep Dive heuristic on varying depths.

in each case of a full lattice.

In the last experiment, we observe the influence of the depth parameter k on quality and runtime of the Deep Dive heuristic. Again, we use the randomly generated data set with six columns to measure the error rate in comparison to the optimal ordering. As expected, Figure 5 shows that the runtime of the heuristics increases with the depth. Furthermore, we can state that the number of distinct elements influences the runtime for higher depth values stronger than for lower values. As we can see in Figure 6, the quality of the solution is also strongly depending on the depth. But as we have already observed in the first experiment, for larger problems a depth of three causes runtime problems.

5. CONCLUSION

We presented our data tiering ansatz and discussed how table reordering can improve tuple reconstruction for tiered in-memory column stores. We implemented multiple heuristics to tackle the sorting problem for real-world applications with acceptable run time overheads. Our reordering approaches outperform the naïve method significantly. Especially the Deep Dive and SZCF heuristics have shown to be feasible approaches finding accurate table orders with run times of few seconds for a production ERP database table with over 100M tuples. Chunking and clustering are two approaches to further reduce complexity of the sorting problem.

6. REFERENCES

[1] M. Boissier. Optimizing main memory utilization of columnar in-memory databases using data eviction. In *Proc. VLDB Ph.D. Workshop*, pages 1–6, 2014.

[2] M. Boissier et al. Analyzing data relevance and access patterns of live production database systems. In *Proc. CIKM 2016*, pages 2473–2475, 2016.

[3] P. A. Boncz et al. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[4] S. Dar et al. Semantic data caching and replacement. In *Proc. VLDB 1996*, pages 330–341, 1996.

[5] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, 1968.

[6] F. Halim et al. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[7] A. Kemper et al. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE 2011*, pages 195–206, 2011.

[8] J. Krüger et al. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[9] J. Lee et al. High-performance transaction processing in SAP HANA. *Data Eng. Bull.*, 36(2):28–33, 2013.

[10] C. Lemke, K. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores - A case for compression. In *DAWAK 2010*, pages 117–129, 2010.

[11] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *Proc. ICDE 2013*, pages 26–37, 2013.

[12] C. Meyer et al. Dynamic and transparent data tiering for in-memory databases in mixed workload environments. In *Proc. ADMS 2015*, pages 37–48.

[13] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. SIGMOD 2009*, pages 1–2, 2009.

[14] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, 7(13):1722–1729, 2014.

[15] P. G. Selinger et al. Access path selection in a relational database management system. In *ACM SIGMOD 1979*, pages 23–34, 1979.