Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements

Martin Boissier, Rainer Schlosser, Matthias Uflacker

Hasso Plattner Institute, Potsdam, Germany

{fistname.lastname}@hpi.de

***Abstract*—Recent developments in database research introduced HTAP systems that are capable of handling both transactional and analytical workloads. These systems achieve their performance by storing the full data set in main memory. An open research question is how far one can reduce the main memory footprint without losing the performance superiority of main memory-resident databases. In this paper, we present a hybrid main memory-optimized database for mixed workloads that evicts cold data to less expensive storage tiers. It adapts the storage layout to mitigate the negative performance impact of secondary storage. A key challenge is to determine which data to place on which storage tier. We introduce a novel workload-driven model that determines Pareto-optimal allocations while also considering reallocation costs. We evaluate our concept for a production enterprise system as well as reproducible data sets.**

Publication: This paper has been published in the proceedings of the 34th IEEE International Conference on Data Engineering 2018. The final publication is available at IEEE Xplore via http://dx.doi.org/10.1109/ICDE.2018.00028.

I. Hybrid Data Layouts for HTAP

Modern enterprise systems are no longer separated into traditional transaction-dominated systems and analytics-dominated data warehouses. Modern mixed workload (HTAP or OLxP) systems are expected to handle transactional OLTP workloads as well as analytical OLAP workloads, both on a single system. Until today it is argued which storage format is the best for mixed workloads. Columnar table layouts incur a large overhead for write-intensive OLTP workloads, especially inserts and wide tuple reconstructions (cf. [1], [2]), while performing well for analytical tasks. Row-oriented table layouts – the defacto standard for enterprise databases – have shown to be insufficient for increasing analytical workloads of upcoming enterprise systems [3]. Hence, commercial database vendors added columnar storage engines [4]–[6] to their row stores. To combine both worlds, research has proposed hybrid systems that combine both row- and column orientation in a single storage engine and adjusting a table's layout depending on the workload. Recent work includes, e.g., H_2O [7] and Peloton [8].

Most hybrid systems show improvements over pure row- or column-oriented variants. Peloton and most other hybrid database systems presented improvements by factors of $3-4\times$ over non-hybrid counterparts. We see two problems with these results. First, most hybrid research prototypes evaluate the performance of hybrid structures against homogeneous layouts using the same execution engine. Hence, homogeneous implementations pay the price for hybrid data layouts that usually incur indirections and abstractions that affect database performance negatively. Second, well-known optimizations for homogeneous layouts (e.g., SIMD for sequential operations [9]) have not been fully exploited. We think that none of the proposed systems have proven yet that the advantages brought by hybrid designs justify the added complexity.

The added complexity of a hybrid storage engine introduces new complexity in higher-level functionalities of the database since both, query optimization as well as query execution, have to deal with additional variables and uncertainty. It remains questionable whether the gained flexibility is worth the added complexity that further impedes optimal decisions, e.g., during query plan building. While the overhead of hybrid abstractions and indirections can partially be mitigated by query compilation, the process of compiling efficient query plans for diverse data formats is highly sophisticated as Lang et al. have shown in [10]. Thomas Neumann stated that the query optimizer's impact is often a factor 10 improvement while tuned runtime systems might bring 10% improvements [11]. Based on our experience with Hyrise, we think this is not only true for the runtime system, but for the physical design too. As a consequence, the project presented in this paper focuses less on further optimizing the physical layout of hybrid DRAM-resident databases, but instead on the tiering aspect and how hybrid data structures can be used to improve data eviction and loading. Our goal is to evict data to secondary storage without impacting performance for HTAP workload patterns. While the performance impact of data eviction for main memory-resident databases can be disastrous, also the benefits are manifold. Reduced main memory footprints can lower the costs for hardware, allow larger systems on a single server, improve elasticity, reduce recovery times, or simply allow for more memory-intensive algorithms to be executed.

A straightforward tiering approach for column stores that evicts attributes can cause disastrous performance losses (e.g., for tuple reconstructions of wide tables with dictionary-encoded attributes from disk), rendering hybrid data layouts a promising idea. Hence, we do not solely use hybrid layouts in order to evict data to secondary storage, but rather to optimize it towards the characteristics of each storage layer (cf. [12]).

A. Workload-Driven Attribute Separation

To get an idea of the potential for data tiering in main memory-resident databases, we have analyzed several production systems. Amongst them has been, e.g., a live production

SAP ERP system of a Global 2000 company and a large German blogging platform. We focused on voluminous tables as they have a larger potential for footprint reductions. In the context of tiered HTAP databases, important findings were:

- Tuple reconstructions (comparatively expensive operations in column stores) are highly skewed towards few entities. For most tables that skew is also strongly time-correlated.
- Attributes can often be separated into two distinguished groups: **(i)** columns that are accessed for table searches, aggregations, joins, and other sequential operations that favor columnar layouts; and **(ii)** remaining columns that are either never accessed or point-accessed, e.g., for tuple reconstruction or probing.

These findings are not particularly new but have – to our best knowledge – not been exploited for data tiering of HTAP systems yet. We analyzed the five largest tables of the financial module in a production SAP ERP enterprise system. We concentrated on the financial module, as it covers the largest analytical load. We found that these five tables have in average 256 attributes of which less than 41 are accessed for filtering (i.e., selection). Particularly interesting for our tiering approach is the fact that many of the filtered columns are either **(i)** filtered very seldom or **(ii)** usually filtered in combination with other highly restrictive attributes. As shown in Table I, when looking for attributes that are filtered at least once every 1000th query execution, the skew is even higher. The negative impact of these columns not being DRAM-resident can be negligible with an execution model that is aware of the interaction between attribute selectivity and placement (see Section II-B).

TABLE I
ACCESS STATISTICS OF A PRODUCTION SAP ERP SYSTEM

Table	Attribute count	Attributes filtered	Attributes filtered in $\geq 1‰$ of queries
BSEG	345	50	18
ACDOCA	338	51	19
VBAP	340	38	9
BKPF	128	42	16
COEP	131	22	6

These characteristics are not only true for most enterprise systems, but also for many data mining applications. The bing.com data analysis platform contains tables with thousands of attributes of which only few are regularly filtered [13].

B. Our Approach

Our tiering approach combines our work on hybrid data layouts in Hyrise [14], workload analyses, and data tiering. The idea is straightforward:

1) We separate attributes into point access-dominated attributes and attributes that are mainly processed sequentially by analyzing the database's plan cache.
2) Given a main memory budget, our column selection solution efficiently determines which attributes to evict.
3) The memory-resident columns remain dictionary-encoded while the remaining attributes are stored in a row-oriented and latency-optimized format on secondary storage.

The model is deliberately simple. Each column of a tiered table is completely and solely stored in one single format without any replication or additional data structures. We are aware that – from a storage engine viewpoint – it is appealing to add further formats such as disk-optimized column formats for secondary storage that still allow scans for rarely filtered attributes. But we think a simple model is superior in an end-to-end consideration as it is better maintainable and does not add complexity to higher-level functionalities such as query optimization (see Section II-B).

In our case of tiered HTAP databases, the most important optimization challenge is to decide which data to keep in DRAM and which data to evict. In Section III, we present a novel linear programming solution to the column selection problem that is able to efficiently answer this question. Most notably, the presented model incorporates *selection interactions*.

As shown in Section III, the integration of selection interaction significantly improves the solution quality over various heuristics. For tuple-based query evaluation in row stores (cf. Volcano model [15]), vertical partitioning is usually approached by heuristics that count filtering frequencies and determine partitions accordingly (cf. [7], [16]–[18]). This is a feasible approach for tuple-based query evaluation in row-oriented databases as the execution of the first predicate automatically loads the tuple's remaining attributes.

However, most HTAP databases are column stores, which execute predicates in a fundamentally different way. Here, predicates are (usually) ordered by their selectivity (with the most restrictive predicate executed first) and executed successively where each operator passes a list of qualifying positions, cf. [19]. Due to the multiplicative selectivities of conjunctive predicates, the expected number of accesses decreases with each executed predicate. Hence, the effect of having frequently accessed columns in DRAM is reduced. It can be advantageous to store other columns in DRAM although they are less often filtered. For compiled query engines, operators are chained and process in a tuple-based manner and only load the next predicated attribute in case the previous (conjunctive) predicate evaluated as true. Both ways, with each successive filter, a smaller part of the following attribute is accessed.

As a consequence, counting filter frequencies is not a fitting model to vertically partition relations in columnar execution engines. The problem to solve rather resembles the problem of index selection with index interaction, cf. [20].

II. ARCHITECTURE

We have implemented our concept in Hyrise[1]. Hyrise is a hybrid main memory-optimized database for HTAP workloads. Each table in Hyrise consists of two partitions, a write-optimized delta partition (cf. C-Store's *writable store* [21]) and a read-optimized main partition. Using an insert-only approach, data modifications are written to the delta partition, which is periodically merged into the main partition [22]. Attributes in the main partition are dictionary-encoded with a bit-packed

[1]Hyrise repository: https://github.com/hyrise/hyrise

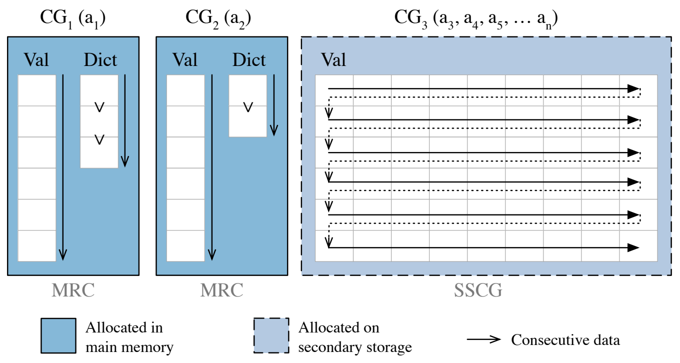


Fig. 1. Exemplary hybrid table layout with three column groups (CG). The first two groups are both dictionary-encoded memory-resident columns (MRC). The remaining attributes are stored without any compression in a secondary storage column group (SSCG).

order-preserving dictionary. Attributes in the delta partition use an unsorted dictionary with an additional B+-tree for fast value retrievals. ACID compliance in Hyrise is implemented using multi-version concurrency control [23].

Hyrise is able to combine row- and column-oriented data layouts as well as horizontal and vertical partitioning in a free manner, a concept which has recently been adopted by Peloton [8]. While Hyrise's initial objective was to improve cache hit rates in the case of full DRAM residence, our approach's objective is to mitigate the negative performance impact of secondary storage. Our implementation uses a simplified hybrid format consisting of column groups of variable lengths, comparable to H_2O's "group of columns" [7].

A. Storage Layout

To keep complexity manageable, we deliberately limited ourselves to the most common data structures for databases: **(i)** singular column groups storing exactly one attribute and being completely DRAM-resident and **(ii)** row-oriented column groups that store attributes adjacent together and reside on secondary storage. The data layout is depicted in Figure 1.

Singular Columns: Attributes dominated by sequential reads are stored column-oriented (hereafter referred to as a ***M**emory-**R**esident **C**olumn, **MRC***) using order-preserving dictionary encoding, the de-facto standard for HTAP databases [24], [25]. The goal is to be able to execute all sequential operations, e.g., filtering and joining, on MRCs. We apply well-known optimization techniques on MRCs such as vectorization, SIMD, and processing compressed data with late materialization.

Uncompressed Column Groups: One of the most expensive operations for disk-resident column stores are wide tuple reconstructions (cf. [26]). For a table with 100 attributes, e.g., a full tuple reconstruction from a disk-resident and dictionary-encoded column store reads at least 800 KB from disk (i.e., 100 accesses to both value vector and dictionary with 4 KB reads each). In contrast, row-oriented column groups (hereafter referred to as a ***S**econdary **S**torage **C**olumn **G**roup, **SSCG***) are optimized for tuple-centered accesses, e.g., tuple reconstruction or probing. For performance reasons, SSCGs are stored uncompressed. This way, we trade off space consumption (assuming secondary storage layers are cheap) with performance due to improved data locality for tuple accesses. Further page-wise compression is possible but not yet implemented. Using the proposed SSCGs, full-width tuple reconstructions require only single 4 KB page accesses to secondary storage. Tuple inserts, deletions, and updates are handled via the DRAM-resident delta partition.

Consequently, attributes stored in an SSCG resemble disk-resident row stores. This way, we exploit both major advantages of row-oriented data structures. First, the comparatively easy eviction since a tuple's attributes are stored consecutively in one place. Second, advantages for tuple reconstruction with perfect cache locality for point accesses. In contrast, MRC-attributes resemble in-memory databases such as SAP HANA [24] or HyPer [25], which execute analytical/sequential operations on columnar dictionary-encoded attributes.

B. Query Execution & Optimization

In Hyrise, filters are executed using indices if existing. Afterwards, the remaining filters are sorted by increasing selectivity (we define attribute selectivity as $1/n$ for an attribute with n distinct values[2]). With the introduction of SSCG-placed attributes, the only change is that filters on non-indexed columns are sorted first by the location (DRAM-resident or not) and second by selectivity. The goal is to ensure fast query filtering via index accesses or scans on DRAM-resident columns.

The goal is to keep columns in DRAM that are regularly used in sequential operations. It turns out that it is not necessary to keep all these attributes DRAM-resident. As discussed in Section I-A, many attributes are filtered in combination with highly restrictive attributes. Hyrise's query executor switches from scanning to probing as soon as the fraction of remaining qualifying tuples falls below a certain threshold (usually set to $0{,}01\%$ of the table's tuple count). Probing a DRAM-resident cell is still faster than accessing a 4 KB page from secondary storage, but the further the tuple probing is delayed, the higher the probability that the currently evaluated tuple is part of the result set. This way, we piggyback probing during filtering to load the remaining attributes in DRAM in case several projected attributes of the tuple are part of the result.

C. Data Eviction

For the process of data eviction and caching, we use EMC's memory management library AMM (*Advanced Memory Manager*, previously known as EMT [28]) that provides a facility to pre-allocate fixed-size page caches. In [28], we have compared AMM's performance against Linux's default `mmap` and `malloc` implementation under memory pressure.

III. Column Selection

The question of what data to keep in DRAM or evict is challenging. Classical page-based eviction mechanisms worked well for OLTP systems, but less so for HTAP systems with analytical queries often accessing attributes in their full length.

[2]For inequality predicates, we use heuristics similar to [27].

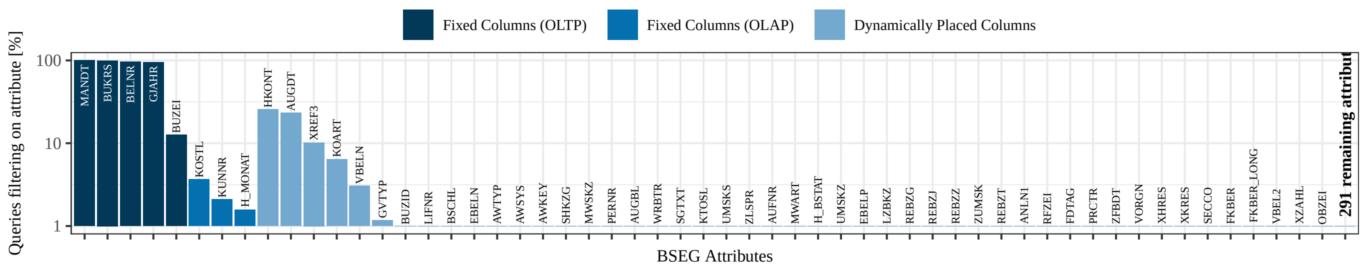


Fig. 2. This graphs shows the attributes of table *BSEG* of a production SAP ERP system and their filter frequencies. 291 attributes have not been filtered at all and will thus be evicted. The first two groups show pinned attributes (e.g., primary key attributes for OLTP) while the allocation of the remaining attributes will be decided by the column selection solution.

The goal of column selection is to determine which attributes/columns to evict given DRAM budget constraints. Therefore, we analyze the workload (in form of the database's plan cache) in order to determine sequential accesses. As a trivial preprocessing step, we add all attributes to the list of columns to evict which have not been filtered at all. Following, the actual column selection is done to further lower the memory footprint. Here, we determine which additional attributes have the smallest negative effect on performance when added to the list of columns to evict (i.e., being added to an SSCG).

We determine the utility (i.e., the expected performance) by calculating the required data that needs to be processed. This bandwidth-centric model is driven by the observation that mixed workload systems are increasingly limited by the bandwidths of modern NUMA servers.

At first sight, the problem is related to the binary Knapsack problem [29]. We maximize the utility (i.e., the expected performance) for a given space constraint (i.e., the DRAM budget). Unfortunately, Knapsack approaches cannot be used to solve column selection problems as the utility of having a column in DRAM typically depends on other column decisions (i.e., selection interaction). The problem belongs to the class of resource allocation problems, which in general, have to be solved using suitable solvers (cf. [30]).

The column selection is an autonomous process. In case the database administrator has additional performance requirements, attributes can be pinned in DRAM. This can be done, e.g., for primary key attributes in order to ensure sufficient transactional performance or to fulfill service-level agreements (SLAs) for particularly important processes (see Figure 2).

The remainder of this section is organized as follows. In part A, we present an integer linear programming model to solve column selections problems. Further, we prove that a continuous relaxation of the problem can be used to identify Pareto-efficient solutions. In part B, we demonstrate the applicability of our model on a real-life production system. In part C, we verify the high performance of our solution compared to benchmark heuristics using more general reproducible examples. In part D, we include reallocation costs to be able to deal with dynamic workload scenarios. In part E, we analyze the structure of solutions of the continuous model and the integer model. In part F, we finally show that optimal solutions can be computed explicitly in milliseconds without using a solver. In part G, we illustrate the scalability of our explicit solution (called *Schlosser Heuristic*). In part H, we propose a general solution principle to approximate efficient solutions.

A. Model Description

We want to select columns to be in DRAM such that (i) the overall performance is maximized, i.e., a workload's execution time is minimized, and (ii) the total DRAM used does not exceed a given budget. We consider a system's workload characterized by N columns and Q queries.

In our bandwidth-centric workload model, each query j is characterized by the set of columns $q_j \subseteq \{1,...,N\}$, $j=1,...,Q$ that are accessed during query evaluation. All accesses are modeled as scans with a particular selectivity (e.g., OLAP joins and aggregations are large sequential accesses). The access costs of a query depend on whether or not occurring columns are in DRAM. To indicate whether a column i is either stored in DRAM (1) or not (0), we use binary decision variables x_i, $x_i \in \{0,1\}$, $i=1,...,N$. The size of column i in bytes is denoted by a_i. The total scan costs F depend on the column allocation $\vec{x} := (x_1,...,x_N)$ and are defined by the sum of scan costs f_j of all queries j multiplied by their number of occurrences b_j:

$$F(\vec{x}) := \sum\nolimits_{j=1,...,Q} b_j \cdot f_j(\vec{x})$$

Hyrise executes attribute filters sequentially in ascending order of the attribute's selectivity. Consequently, the expected selectivity of each attribute depends on the other attributes in the query. In our model, we assume that columns with lower selectivity are scanned first. We also account for the selectivity of columns that have been already scanned.

The selectivity s_i of column i, $i=1,...,N$, is the average share of rows with the same attribute. Note, for simplicity we only define selectivity for equi-predicates with uniform value distributions. Hyrise estimates selectivities similar to [27] (using distinct counts and histograms when available) which are straightforward to implement in our cost model. We describe scan costs f_j of query j, $j=1,...,Q$, by

$$f_j(\vec{x}) := \sum_{i \in q_j} (x_i \cdot c_{mm} + (1-x_i) \cdot c_{ss}) \quad \cdot a_i \cdot \prod_{k \in q_j : s_k < s_i} s_k \qquad (1)$$

where $c_{mm} > 0$ is a scan cost parameter for main memory; $c_{ss} > 0$ denotes the cost parameter for secondary storage. Both parameters can be calibrated; they describe the time it takes to read data (e.g., seconds to read a gigabyte of data) and is used to calculate estimated runtimes. Typically, we have $c_{mm} < c_{ss}$.

Given a column selection $\vec{x}$, the allocated space in DRAM required to store the main memory-resident data amounts to:

$$M(\vec{x}) := \sum_{i=1,\ldots,N} a_i \cdot x_i$$

1) Step 1: Initial Optimization Problem: The problem is to minimize total scan costs such that the DRAM used does not exceed a given budget A, i.e., we consider the objective:

$$\underset{x_i \in \{0,1\}, i=1,\ldots,N}{\text{minimize}} \quad F(\vec{x}) \tag{2}$$

$$\text{subject to} \quad M(\vec{x}) \leq A \tag{3}$$

As we avoided conditional expressions in the definitions of scan costs and the DRAM used the integer problem (2) – (3) is linear, and thus can be solved using standard integer solvers.

2) Step 2: Relaxation of Variables: We model problem (2) – (3) as a continuous linear problem, i.e., We allow the variables x_i, $i = 1,\ldots,N$, to take continuous values between 0 and 1:

$$\underset{x_i \in [0,1], i=1,\ldots,N}{\text{minimize}} \quad F(\vec{x}) \tag{4}$$

The relaxed problem (4) with (3) can be solved using standard solvers. However, the solution is not necessarily of integer type. In Step 3, we use a reformulation of (4) and (3) to guarantee admissible integer solutions in a continuous framework.

3) Step 3: Penalty Formulation of Size Constraint: We omit constraint (3) and include a penalty term in the objective (4) for the DRAM space used:

$$\underset{x_i \in [0,1], i=1,\ldots,N}{\text{minimize}} \quad F(\vec{x}) + \alpha \cdot M(\vec{x}) \tag{5}$$

The penalty parameter α is assumed to be non-negative. The new problem (5) has the following fundamental property.

Lemma 1. *For all α the solution of the continuous linear problem formulation* (5) *is guaranteed to be integer.*

Proof. The isoquants of objective (5) form a hyperplane. Minimizing the linear objective (5) corresponds to the point in which the best hyperplane touches the feasible region (for $\vec{x}$). Hence, a corner of the feasible region (an N-dimensional cube) is always part of an optimal solution. Since all corners have integer coordinates (total unimodularity of the constraint matrix) an optimal solution of integer type is guaranteed.

Note, the optimal (integer) solution of (5) depends on the penalty parameter α. The higher α, the lower is the DRAM space used by an optimal solution. While for $\alpha = 0$ (no penalty) all columns are in DRAM, for $\alpha \to \infty$, no column is selected at all. Hence, α can be chosen such that the associated column selection $\vec{x} = \vec{x}(\alpha)$ just satisfies the budget constraint (3).

B. Application to Enterprise System

We applied the integer and continuous solution approaches to the workload and data of the `BSEG` table of a production SAP ERP system (overall ~20,000 plans, 60 for `BSEG`).

Without loss of generality, we used $A = A(w) := w \cdot \sum_{i=1,\ldots,N} a_i$, where w, $w \in [0,1]$, is the relative memory budget. By "relative performance" we denote the minimal scan costs (where all columns are DRAM-resident) divided by the scan costs $F(\vec{x})$ of a specific solution $\vec{x}$ as defined in (1)-(2).

Figure 3 illustrates solutions of the integer and the continuous model for different memory budgets w. We observe that the relative performance is higher the more DRAM is allowed.

The first interesting aspect is the initial eviction rate of over 78% that is achieved solely by evicting attributes that have not been selected. The remaining 22% are chosen by the continuous and the integer solution. The workload of the `BSEG` heavily relies on one of the largest columns called `BELNR`. The sudden drop of performance for eviction rates over 95% is caused by the fact that `BELNR` no longer fits into the memory budget. According to our model, sequential accesses are slowed down by less than 25% for eviction rates of up 95%.

Note, the integer formulation, cf. problem (2) – (3), allows identifying optimal combinations of performance and DRAM budgets. These combinations cannot be dominated by others and hence, form a "Pareto-efficient frontier" (cf. Figure 3).

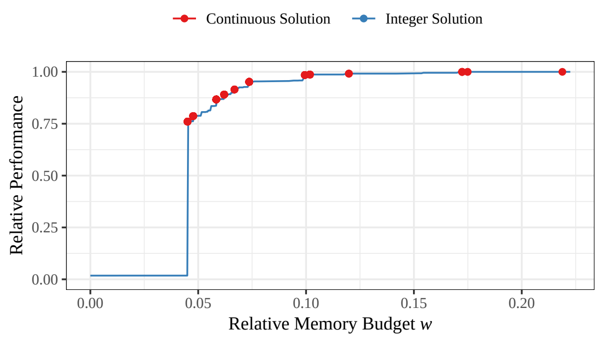


Fig. 3. Comparison of optimal integer and continuous solutions for `BSEG` table: Different combinations of relative performance and data loaded in DRAM (cf. efficient frontier).

Using different penalty parameter α, the continuous problem formulation (5) allows identifying feasible combinations of performance and DRAM budgets used, which are also efficient.

Theorem 1. *For all $\alpha > 0$ the solutions of the continuous problem* (5) *are part of the efficient frontier, which is characterized by optimal solutions of the integer problem* (2)-(3) *for different DRAM budgets A. Hence, they are Pareto-efficient.*

Proof. For an arbitrary but fixed penalty $\alpha > 0$ let $\vec{x}^* = \vec{x}^*(\alpha)$ be an optimal solution of the continuous problem (5). It follows that $\vec{x}^*$ is also an optimal solution of the continuous problem (4) subject to (3) with budget $A := A(\alpha) := M(\vec{x}^*(\alpha))$, since a better solution $\vec{x}$ of (4) st. (3) with $F(\vec{x}) < F(\vec{x}^*)$ and $M(\vec{x}) \leq M(\vec{x}^*)$ would imply that $\vec{x}^*$ is not optimal for (5). Further, let $\vec{x}^*_{int} = \vec{x}^*_{int}(\alpha)$ be an optimal solution of the integer problem (2) subject to (3) with the same budget $A := A(\alpha) := M(\vec{x}^*(\alpha))$. It follows that $F(\vec{x}^*_{int}) \geq F(\vec{x}^*)$ since the feasible space in problem (5) dominates the feasible space in problem (2), i.e., $\{0,1\}^N \subseteq [0,1]^N$. Further, Lemma 1 implies that $\vec{x}^*$ is of integer type and *thus, an admissible candidate for problem (2) with budget $A := M(\vec{x}^*)$. It follows $F(\vec{x}^*_{int}) = F(\vec{x}^*)$. Finally, $M(\vec{x}^*_{int}) < M(\vec{x}^*)$ is not possible as $F(\vec{x}^*_{int}) + \alpha \cdot M(\vec{x}^*_{int}) < F(\vec{x}^*) + \alpha \cdot M(\vec{x}^*)$ would imply that $\vec{x}^*_{int}$ is a better solution to (5) than $\vec{x}^*$, which is a contradiction. Consequently, we also have $M(\vec{x}^*_{int}) = M(\vec{x}^*)$.*

C. Performance Comparisons

We demonstrate the performance of our approaches compared to benchmark heuristics. We consider a general scalable class of reproducible column selection problems. The considered heuristics are greedy approaches, which resemble the status quo for vertical partitioning models (see Section V). We consider three heuristics which assess attributes by (i) the selection frequency (cf. [16]), (ii) by selectivity (cf. [19]), and by weighing selectivity and size of each attribute (cf. *reactive unload* in [31]). The assessment of attributes is motivated by LRU approaches and the used metric to build the eviction order.

Example 1. *We consider N columns, Q queries, and randomized parameter values*[3]. *We compare optimal integer solutions (cf. (2)-(3)) solutions of the continuous model (cf. (5)) as well as allocations of the following three benchmark heuristics:*

(H1) *Include columns in DRAM that are most used (in descending order), measured by the number of occurrences g_i, where $g_i := \sum_{j=1,\dots,Q, i \in q_j} b_j$, $i = 1,\dots,N$.*

(H2) *Include columns in DRAM that have the smallest selectivity s_i, $i = 1,\dots,N$, (in ascending order).*

(H3) *Include columns in DRAM that have the smallest ratio of selectivity and number of occurrences, i.e., s_i/g_i, $i = 1,\dots,N$, (in ascending order).*

If a column does not fit into the DRAM budget anymore, it is checked if columns of higher order do so. In all heuristics, columns which are not used at all ($g_i = 0$) are not considered.

We solve Example 1 for different DRAM budgets $A(w)$. We consider $N = 50$ columns and $Q = 500$ queries. We apply the integer and the continuous solution approach as well as the heuristics (H1)-(H3). Figure 4 illustrates different admissible combinations of estimated runtime and associated relative DRAM budget w for the different column selection strategies.

The solutions of the integer problem form the efficient frontier. The solutions of the continuous problem are again part of the efficient frontier. We observe that both our approaches outperform all three heuristics (H1)-(H3). Depending on the DRAM budget, performance is up to $3\times$ better.

In general, the heuristics (H1)-(H3) are reasonable heuristics. In Example 1, some columns are more often included in queries. Hence, on average, the corresponding g_i values are higher. Moreover, in our example, g_i and selectivity s_i are slightly negatively correlated. Hence, columns with a small selectivity are more likely to be less often used. This explains why pure heuristics like (H1) and (H2) are suboptimal. Heuristic (H3)

[3]Repository containing definition and reproducible example code: https://github.com/hpi-epic/column_selection_example

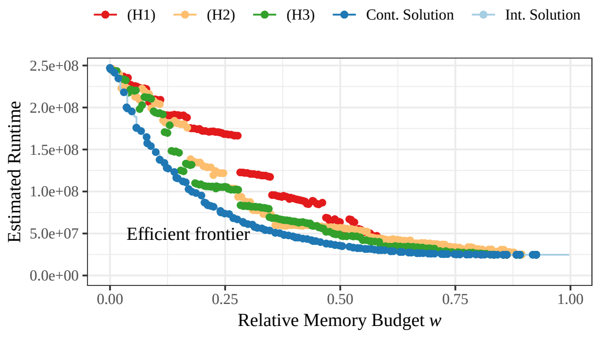


Fig. 4. Estimated runtime comparison of integer and continuous approach vs. heuristics (H1)-(H3) for Example 1.

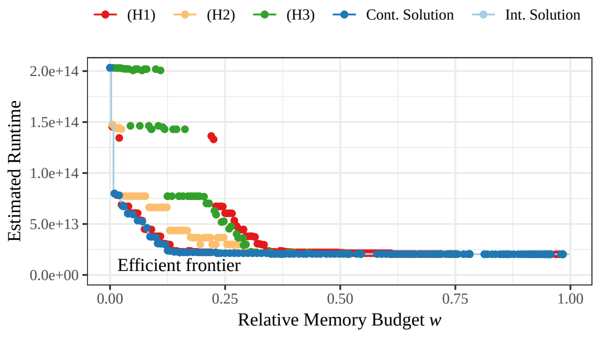


Fig. 5. Estimated runtime comparison of integer/continuous approach vs. heuristics (H1)-(H3) for the `BSEG` table of the traced enterprise system.

achieves better performance results as both effects are taken into account. However, those results are still far from optimal, because more complex effects, such as selection interactions are not taken care of. As in real-life workloads, in our example, some columns often occur in queries simultaneously. Hence, it is advisable just to select *some* of them to be in DRAM. Our model-based solution yields better results because all complex effects are taken into account.

Example 1 can be used to study the performance of heuristics for various workload characteristics. For special cases, rule-based heuristics may perform well. As soon as selection interaction becomes a factor, however, only advanced approaches can lead to good results. Note, that in real-life settings, workloads are typically of that complex type.

In this context, Figure 5 shows the performance results for the `BSEG` table (see Section III-B). We observe, that the interaction between selectivity and the number of accesses even leads to an (up to $10\times$) worse performance of the heuristics. It is unlikely that simple heuristics exist that fit all scenarios. This is due to the unforeseeable character of a workload which is characterized by the complex interplay of the quantities b_j, g_i, s_i, and a_i as well as further important effects such as selection interaction or the structure of queries q_i. Hence, an effective allocation strategy will have to find complex customized solutions (in a reasonable amount of time) that take all these effects *simultaneously* into account. Note, solutions of the continuous model are of that particular type as they are guaranteed to be efficient in *any* setting.

D. Dynamic Workloads and Reallocation Costs

Another import aspect for real-life settings are reallocation costs. As workloads typically change over time, current data placements have to be recalculated regularly. However, reorganizations of data allocations are costly and time-consuming. The challenge is to identify reallocations that have a significant impact on performance compared to their costs.

We consider a current column allocation $y_i \in \{0,1\}$, $i = 1,\dots,N$, and a cost parameter β for changing the memory location of a column (from secondary storage to DRAM or vice versa). We extend model (5) to minimize total scan costs, DRAM space used, and memory changes made:

$$\underset{x_i \in [0,1], i=1,\dots,N}{\text{minimize}} \quad F(\vec{x}) + \alpha \cdot M(\vec{x}) + \beta \cdot \sum_{i=1,\dots,N} a_i \cdot |x_i - y_i| \tag{6}$$

Due to reallocation costs, the objective of the problem becomes nonlinear. To linearize (6), we introduce additional variables z_i, $i = 1,\dots,N$, which are continuous on $[0,1]$. Further, we add two families of linear constraints such that problem (6) can be equivalently rewritten as:

$$\begin{aligned} \underset{x_i, z_i \in [0,1], i=1,\dots,N}{\text{minimize}} \quad & F(\vec{x}) + \alpha \cdot M(\vec{x}) + \beta \cdot \sum_{i=1,\dots,N} a_i \cdot z_i & (7) \\ \text{subject to} \quad & x_i - y_i \le z_i, \quad i = 1,\dots,N \\ & y_i - x_i \le z_i, \quad i = 1,\dots,N \end{aligned}$$

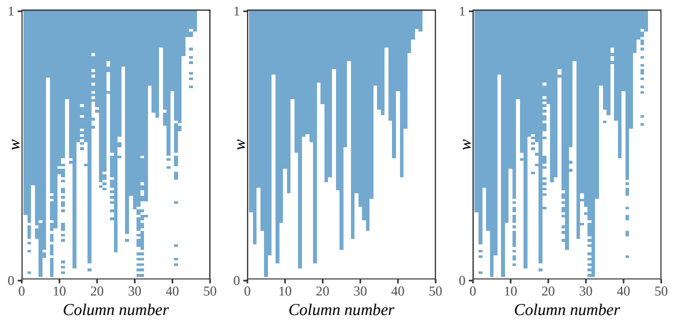
The new constraints guarantee that $z_i = |x_i - y_i|$ for all i. Further, as the total unimodularity of the constraint matrix is still satisfied integer solutions of (7) and thus of (6) are guaranteed.

In practice, β is selected depending on the system's requirements. In most cases, physical data maintenance is executed during night times. In this context, the numbers b_j should be normalized on a daily basis and $\beta = 1$ may serve as a reference case. We can obtain the expected maintenance duration (usually bound by the secondary storage bandwidth) and adjust β accordingly so that we only initiate reallocations that will finish within the allowed maintenance time frame.

E. Solution Structures

Looking at optimal allocations of the continuous problem for different parameters, α reveals an interesting structure, which is illustrated in Figure 6. Figure 6(a) shows the optimal column selection of the integer problem for different w values. Figure 6(b) illustrates the solution of the corresponding continuous problem. While the optimal column allocations of the integer problem are quite complex, the solutions of the continuous problem have a recursive structure.

Remark 1. *Consider problem* (6)*. If a column $i, i = 1,\dots,N$, is assigned to an optimal DRAM allocation for a budget $A = A(\alpha)$, $A \ge 0$, then column i is also part of optimal DRAM allocations for all larger budgets $\widetilde{A} > A$. Consequently, solutions of problem* (6) *have a recursive structure and columns are assigned to an optimal allocation in a fixed order.*



(a) Integer solution (b) Continuous solution (c) Continuous solution with remainder filling

Fig. 6. Optimal column selections (integer, continuous, and continuous with filling) for varying DRAM budgets, $w \in [0,1]$. The graph indicates whether a column is in DRAM (shown blue, $x_i = 1$, $i = 1,\dots,50$) for Example 1.

The order of columns described in Remark 1, which we call "performance order", allows ordering the set of columns according to their impact on reducing runtime.

Finally, Figure 6(c) depicts the case, in which for a given budget $A(w)$ the unused space of the next smaller Pareto-optimal solution of the continuous model is filled with further (smaller) columns according to the order described in Remark 1. Note, this recursive allocation strategy (with filling) closely resembles the optimal integer solution (cf. Figure 6(a)).

A huge *disadvantage* of the continuous approach, however, is that the model has to be solved multiple times in order to identify the optimal penalty parameter α that is associated with the intended DRAM budget A. The goal of the next subsection is to overcome this problem.

F. Explicit Solution

The results obtained in the previous sections (cf. Remark 1 and Theorem 1) can be exploited to compute optimal solutions of (6) without solving any optimization program. The key idea is to *explicitly* derive the performance order o_i.

The decision whether a column is in DRAM boils down to the question whether x_i has a positive or negative effect on (6). Due to its structure, objective (6) can be written as $\sum_{i=1,\dots,N} c_i(x_i) + C$, where C is a constant. Collecting terms that depend on x_i, we obtain that c_i, $i = 1,\dots,N$, amounts to

$$c_i(x_i) := a_i \cdot ((S_i + \alpha) \cdot x_i + \beta \cdot |x_i - y_i|) \tag{8}$$

where, $i = 1,\dots,N$,

$$S_i := \sum_{j=1,\dots,Q: i \in q_j} b_j \cdot (c_{mm} - c_{ss}) \cdot \prod_{k \in q_j : s_k < s_i} s_k$$

Hence, whether a selection of column i has a positive or negative effect on (6) depends on its effect on (8).

Theorem 2. *(i) Pareto-optimal solutions of problem* (6) *that satisfy a given DRAM budget A can be calculated as follows: Include as many columns as possible in DRAM in the order o_i, $i = 1,\dots,N$, with $S_i + \beta \cdot (1 - 2 \cdot y_i) < 0$, defined by*

$$o_i := |\{k = 1,\dots,N : S_k - 2 \cdot \beta \cdot y_k \le S_i - 2 \cdot \beta \cdot y_i\}| .$$

(ii) The structure described in Remark 1 generally holds.
(iii) In (i) columns are recursively chosen such that the additional runtime improvement per additional DRAM used is maximized.

Proof of (i) Considering (8)*, we distinguish the following four cases,* $i = 1,...,N$*:*

- *If* $y_i = 0$ *and* $S_i + \alpha + \beta < 0$ *then* (8) *decreases in* x_i.
- *If* $y_i = 0$ *and* $S_i + \alpha + \beta \geq 0$ *then* (8) *increases in* x_i.
- *If* $y_i = 1$ *and* $S_i + \alpha - \beta < 0$ *then* (8) *decreases in* x_i.
- *If* $y_i = 1$ *and* $S_i + \alpha - \beta \geq 0$ *then* (8) *increases in* x_i.

Summarizing the four cases, we obtain that if, $i = 1,...,N$,

$$S_i + \alpha + \beta \cdot (1 - 2 \cdot y_i) < 0 \tag{9}$$

then (8) *decreases in* x_i *and* $x_i^* = 1$ *is optimal else we obtain* $x_i^* = 0$. *Hence, if* α *decreases then the left-hand side of* (9) *decreases as well and, in turn, one column i after another is included in DRAM. The order in which columns are included in DRAM coincides with the performance order* o_i *(cf. Remark 1). Now,* o_i *can be easily determined by comparing each column i's critical* α *value that puts the left-hand side of* (9) *equal to zero. The column with the smallest value* $S_i - 2 \cdot \beta \cdot y_i$ *is the first column to be put in DRAM. Finally, the order* o_i *allows computing Pareto-optimal solutions of* (6) *that are admissible for a given budget A.*

Proof of (iii). Assume a column allocation $\vec{x}$ *corresponds to a runtime of* $F(\vec{x})$ *(including reallocation costs). Selecting a new column i in DRAM (*$x_i := 1$*) reduces the value F by* $c_i(1) - c_i(0) = a_i \cdot (S_i + \beta \cdot (1 - 2 \cdot y_i))$ *(cf.* (8)*) while the DRAM budget used increases by* a_i.

Note, the strategy defined in Theorem 2 (i) combines two advantages: Allocations are necessarily Pareto-optimal *and* can be computed as fast as simple heuristics (cf. (H1)-(H3)) since no penalty values α are needed anymore.

Remark 2. *The result of Theorem 2 (i) can be combined with a filling heuristic: Include columns in DRAM that are of highest importance, cf.* o_i. *If a column does not fit into the DRAM budget A anymore, it is checked if columns of higher order do so, cf. Figure 6(c).*

G. Scalability

Enterprise systems often have thousands of tables. For those systems, it is unrealistic to expect that the database administrator will set memory budgets for each table manually. Our presented solution is able to determine the optimal data placement for thousands of attributes. We measured the solution runtime for a large synthetic data set with a varying number of queries and attributes using the MOSEK solver[4].

Table II compares the computation time of the integer model and the explicit solution in the setting of Example 1 for different numbers of columns N and queries Q.

Table II shows that optimized data placements can be efficiently calculated, even for large systems. The linear problem is manageable for state-of-art integer solvers. However, runtimes can become large when the size of the system is large. The explicit solutions (cf. Theorem 2) have been computed locally using a simple single-threaded C++ implementation. As expected, the computation of the explicit solution is orders of magnitudes faster and allows for immediate response.

[4]MOSEK Solver: https://www.mosek.com

TABLE II
RUNTIME COMPARISON OF COLUMN SELECTION

Column Count	Query Count	Mean Runtime Integer Solution	Mean Runtime Explicit Solution
100	1 000	0.01 s	0.001 s
500	5 000	0.13 s	0.01 s
1 000	10 000	0.32 s	0.01 s
5 000	50 000	6.74 s	0.03 s
10 000	100 000	27.4 s	0.07 s
20 000	200 000	113.6 s	0.15 s
50 000	500 000	2210.3 s	0.48 s

Hence, it becomes easy for a database administrator to (i) dynamically update optimized allocations and (ii) decide whether it is worth to allow for a slightly larger budget compared to the expected additional performance.

H. Solution Principle for Generalized Selection Problems

The allocation strategy described in Theorem 2 (iii) reveals a *general* solution principle to approach the challenging problem of identifying the key columns that have the most impact.

Remark 3. *We propose the following recursive heuristic: Columns are subsequently selected such that the "additional performance" per "additional DRAM used" is maximized.*

Remark 3's heuristic allows to approximate Pareto-optimal combinations of performance and memory budget. This approach is effective as the efficient frontier is naturally characterized by a convex shape, see Figures 4 and 5, since the value of an additional unit of DRAM is decreasing with the memory budget (diminishing marginal utility).

This principle can be easily adapted to compute optimized allocations for more general column selection problems with highly complex scan cost functions. Note, for such problems even state-of-the-art nonlinear solvers might fail. Moreover, the approach can also be applied if query optimizers are used as they similarly allow to estimate and to compare the relative performance improvements of specific column selections.

I. The Bottom Line

Our solution approach is applicable to general column selection problems. Performance is close to optimal and outperforms other heuristics since coupled aspects as selectivity, size, and frequency of queries are taken into account. Computation is fast and scalable. We have shown how to determine an optimized order of columns which can be exploited as a near-optimal heuristic for any DRAM budget that is performance-wise on par with the fastest heuristics. Our penalty formulation for allocated main memory mirrors opportunity costs for DRAM and might be even more realistic than hard DRAM budget constraints. Moreover, our model is able to take reallocation costs for DRAM into account. This way, we can determine whether it is advantageous to evict or load data over time.

IV. Tiering Performance

In this section, we evaluate the performance of our implementation. All benchmarks have been executed on a four-socket Fujitsu Primergy RX4770 M3 with Intel Xeon E7-4850 v4 CPUs (16 cores per socket, 40M L3-Cache, 2.1 GHz), 2 TB of DRAM, running 64-bit Ubuntu 14.04 LTS with kernel 4.2. We evaluated the following devices:

- **CSSD:** consumer-grade solid-state drive (Samsung SSD 850 Pro) with 256 GB storage capacity.
- **ESSD:** enterprise-grade SANDISK Fusion ioMemory PX600 SSD with 1 TB storage capacity.
- **HDD:** SATA-connected Western Digital WD40EZRX HDD with 4 TB storage capacity and 64MB cache.
- **3D XPoint:** 3D XPoint-based Intel Optane P4800X.

Both solid-state drives are NAND devices which are widely used in modern data centers, whereas the ESSD is a bandwidth-optimized device that reaches its top performance with large IO queues. The 3D XPoint device is the first generation of solid-state drives that does not use a NAND architecture. This device is particularly interesting for us as it has ~10× lower random access latencies than NAND devices even for short IO queues. The HDD device serves as a reference device. Due to its poor random access performance, we do not include the device in the materialization measurements.

Benchmark Data Sets: We evaluated the performance on three different data sets: *(i)* The **SAP ERP** data set which reflects characteristics (distinct counts, data types, etc.) of the `BSEG` table of the analyzed production SAP ERP system [32]. The `BSEG` table is the central table of the financial module and has the highest analytical load in an SAP ERP system (20 M tuples with 345 attributes). *(ii)* The **TPC-C** data consists of the `ORDERLINE` table of the TPC-C benchmark with a scale factor of 3 000 (300 M tuples). *(iii)* The **Synthetic** data set is a table with 10 M tuples and 200 attributes which are filled with random integer values. Both `BSEG` and `ORDERLINE` tables belong to the largest tables of each system and are thus of special interest for our focus on cold data eviction. `ORDERLINE` and `BSEG` have vastly differing widths (10 vs. 345 attributes) and depict both extremes for the effect on tuple reconstruction in our implementation.

Before discussing end-to-end performance, we briefly discuss the modified components compared to vanilla Hyrise. Our data allocation model aims to keep all sequentially accessed columns in DRAM. Hence, analytical performance remains the same except from very tight DRAM budgets. But several components that potentially negatively impact the transactional performance of Hyrise have been modified.

Transaction Handling: Hyrise uses MVCC to ensure concurrent and transactionally safe accesses the data [33]. MVCC-related columns (cf. [23]) are kept unchanged and DRAM-allocated. Thus, transaction performance is not impacted.

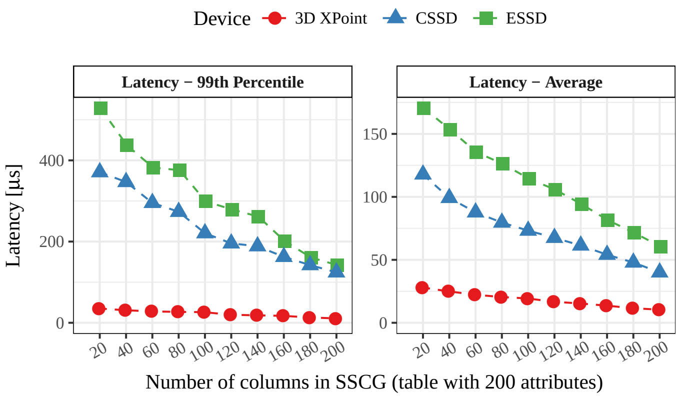


Fig. 7. Latencies for full-width tuple reconstructions on synthetic data set (uniformly distributed accesses).

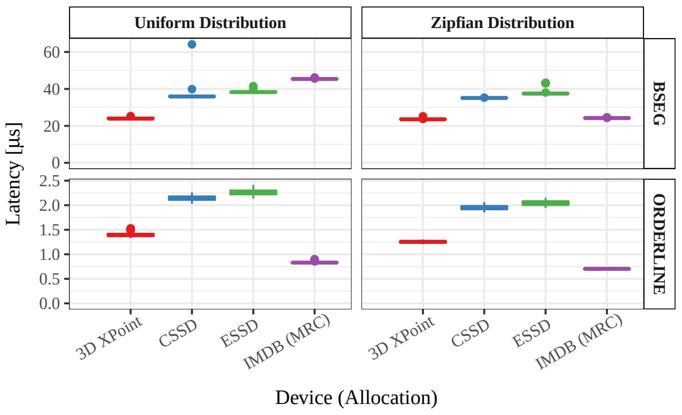


Fig. 8. Latency box plot for full-width tuple reconstructions on tables `ORDERLINE` and `BSEG` (uniform- and zipfian-distributed accesses).

Indices: To ensure high throughput for point accesses, Hyrise has several index structures such as single column B+-trees and multi-column composite keys [34]. As of now, we do not evict indices and keep them completely DRAM-allocated.

Data Modifications: Throughput and latency of modifications are not impacted, because by using an insert-only approach, modifications are handled by the delta partition which remains fully DRAM-resident (cf. Section II). However, the periodic process of merging the delta partition with the main partition is negatively affected. But since the merge process is asynchronous and non-blocking, ongoing transactions are neither blocked nor slowed down during the merge phase [22].

A. End-to-End Benchmark Performance

We evaluated the performance of our approach for TPC-C and the CH-benCHmark [35]. Due to the unchanged transactional components, only reading queries are impacted. Hence, excluding the asynchronous merge phase, runtime performance depends largely on the given memory budget. For TPC-C's largest table `ORDERLINE`, 4 out of 10 columns are selected/aggregated leaving at least 6 columns to be evicted to disk. Probably not surprisingly given TPC-C's complexity, our data allocation model separates `ORDERLINE` into the four primary key attributes as memory-resident columns (MRCs) and the remaining attributes into a secondary storage column group (SSCG). Table III shows the relative performance impact of TPC-C's *delivery* transaction and CH-query #19.

TABLE III
PERFORMANCE IMPACT ON TPC-C'S DELIVERY TRANSACTION AND Q19 OF CH-BENCHMARK (3D XPOINT)

	Data Evicted	**Slowdown**
TPC-C Delivery	80 %	1.02x
CH-query #19	80 %	6.70x
CH-query #19	63 %	1.12x

While the performance results for TPC-C are promising, please note that TPC-C's workload and data model are simple and no performance-critical path accesses tiered data. The same is partially true for the CH-benCHmark that accesses ORDERLINE mainly for grouping, filtering, and joining on (non-tiered) primary key columns. More interesting is CH-query #19 which filters on a non-primary-key column. Given a DRAM budget of $w = 0.2$, only the primary key columns of ORDERLINE remain DRAM-resident and even analytically accessed columns would be evicted as part of an SSCG. For query #19, the join predicate on ol_i_id and the predicate on ol_w_id are not impacted, but the range predicate on ol_quantity is executed on a tiered column. For a warehouse count of 100, Hyrise probes ol_quantity with a selectivity of ~0.05. In the configuration shown in Table III, the probing alone is slowed down by a factor of 40× which leads to an overall query slowdown of 6.7×. If we allow for a larger DRAM budget of $w = 0.4$, columns ol_delivery_d and ol_quantity become DRAM-resident lowering the slowdown to 1.12×, which we attribute to the narrow materialization of ol_amount (SSCG-placed).

The remainder of this section studies the three access patterns that have been altered in our prototypical implementation: (i) random accesses for wide tuple reconstructions, (ii) sequentially scanning a tiered column, and (iii) probing a tiered column.

B. Tuple Reconstruction

Our main focus with respect to tiering performance is the latency of wide tuple reconstructions. Especially for column stores, wide tuples reconstructions are expensive as each attribute materialization potentially leads to two L3 cache misses (access to value vector and dictionary). We measure reconstruction performance by accessing randomly chosen tuples using a uniform random distribution. We set AMM's page cache to 2% of the evicted data size and disable page pinning. The uniform distribution of accesses reflects the worst case scenario for our implementation with almost no cache hits during tuple reconstruction.

We measure the mean latency as well as the 99th percentile latency of full-width tuple reconstructions on the synthetic data set. We vary the number of attributes stored in the SSCG from 20 to 200. For each benchmark, we execute 10 M tuple reconstructions. The results are shown in Figure 7. For the uniformly distributed accesses, we observe that the latency-optimized 3D XPoint device outperforms the NAND devices. This trend is even more nuanced when comparing the 99th percentile latencies. Most notably, SSCG-placed tuples on 3D XPoint outperform fully DRAM-resident dictionary-encoded tuples when $\geq 50\%$ of the attributes are stored in the SSCG.

The second benchmark evaluates tuple reconstructions for BSEG and ORDERLINE tables with zipfian ($\alpha = 1$) and uniformly distributed accesses, shown in Figure 8. *IMDB (MRC)* denotes a columnar, dictionary-encoded, and fully DRAM-resident data schema. The BSEG table consists of 20 MRC-attributes and 325 attributes in an SSCG (ORDERLINE: 4 MRC and 6 attributes in SSCG). The results show that runtimes are dominated by the width of the SSCG with an increasingly lower latency the higher the share of SSCG-placed attributes gets. The tiered uncompressed column group (SSCG) is able to compensate the negative performance impact of accessing secondary storage. It can even outperform full DRAM-resident cases since non-local DRAM accesses (twice per attribute due to dictionary-encoding) quickly sum up for wide tables. For wide tables such as the BSEG table, the performance improves up to ~2× for uniform accesses and ~1.1× for zipfian accesses. For the narrow ORDERLINE table, performance degrades by ~70% for uniform accesses.

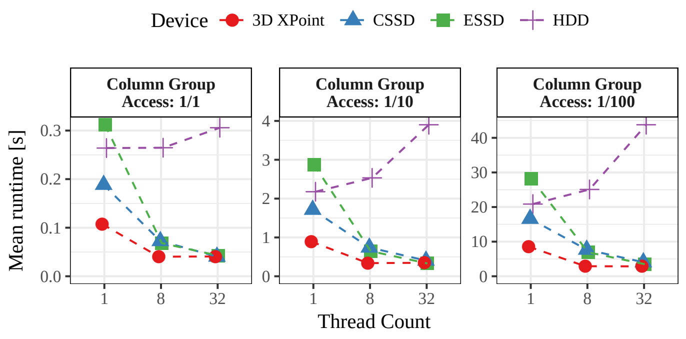
C. Sequential Access Patterns

To evaluate the impact on analytical workloads, we measure the performance of sequential scans and probing.

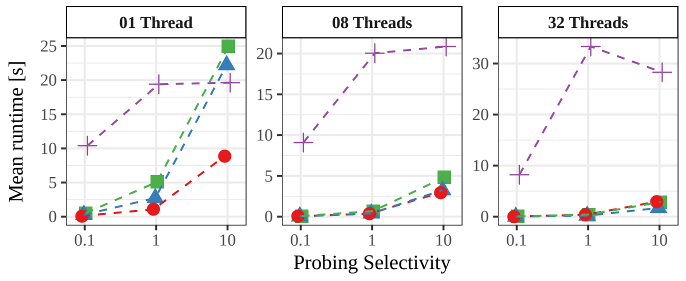
Scanning: An important assumption of our approach is that the vast majority of sequential processing remains on DRAM-allocated attributes (see Section III for column selection and pinning). In case the workload is well known and does not change significantly, sequential processing is expected to never access secondary storage. As columns that are not tiered remain unmodified, the performance remains the same. Nonetheless, unexpected workloads patterns or very low DRAM budgets pose performance problems. They might still occur due to (i) exceptional or seasonal queries or (ii) changing workload patterns that have not yet resulted in an adapted data placement.

Figure 9(a) shows the performance of column scans with varying thread counts and widths of the SSCG. A column group access of $1/1$ means that we scan one attribute in an SSCG that contains a single attribute. $1/100$ means that one attribute out of 100 in an SSCG was scanned. As expected, the costs scale linearly with the width of SSCG. The reason is the effective data that is read with each single 4 KB page access. With 100 integer columns, each 4 KB page contains 10 values to scan while each page for an SSCG of 10 attributes contains 100 values to scan. HDDs perform well for pure sequential requests but significantly slow down with concurrent requests by multiple threads while modern SSDs require concurrent access/larger IO queues for full performance.

Probing: Figure 9(b) shows the probing performance. Due to our data placement model, we expect probing to happen infrequently on tiered attributes, but more frequently than



(a) Sequential scanning



(b) Probing

Fig. 9. Runtime performance of sequential access patterns.

scanning. Again, thread count has a significant impact on the performance of NAND devices as does the selectivity.

Table IV lists the relative slowdown comparing the measurements discussed before (cf. Figures 9(a) and 9(b)) with a full DRAM-resident and dictionary-encoded columnar system. As expected, tuple reconstructions can be sped up, depending on the number of columns accessed and their storage tier. Sequential accesses slow down linearly with the number of attributes stored in the SSCG. Due to non-sequential access pattern, HDDs perform clearly worse probing than scanning.

TABLE IV
PERFORMANCE OF ANALYTICAL ACCESS PATTERNS: COMPARING SSCG ON 3D XPOINT VS. DRAM-RESIDENT MRC (32 THREADS). SHOWING RELATIVE SLOWDOWN (*latency SSCG*/*latency MRC*)

	1 Thread	**8 Threads**	**32 Threads**
Uni. Tuple Rec. (100/200)	1.02	0.92	0.86
Uni. Tuple Rec. (180/200)	0.81	0.72	0.64
Zipf. Tuple Rec. (100/200)	0.92	0.83	0.77
Zipf. Tuple Rec. (180/200)	0.75	0.67	0.60
Scanning (1/100)	335.69	644.44	548.85
Probing (1/100, 0.1%)	5 447.11	301.89	78.95
Probing (1/100, 10%)	4 446.25	1 195.00	987.50

D. The Bottom Line

The evaluation of an SAP ERP system shows that usually between 5-10% of the attributes are accessed for sequential operations, while our prototype with SSCG-placed attributes outperforms the fully DRAM-resident counterpart as soon as more than 50% of the attributes are stored in the SSCG (cf. Figure 2). We see that SSCGs can compensate the performance impact of secondary storage access. We conclude that tiering SSCGs is worth the added complexity in Hyrise. As long as the workload is known and not frequently changing, performance can be improved while reducing memory footprints. But in case of recurring analytical queries on SSCG-placed attributes, the only feasible approach from a performance perspective is to load the columns back into DRAM as MRCs.

V. RELATED WORK

In this section, we briefly discuss related areas of research: physical design advisors, hybrid data layouts, and data tiering.

Several prior publications have used solver-based approaches for allocation problems. Rotem et al. [18] optimize concurrent accesses to parallel disks for two-ways joins via join graphs using integer programming. Ozmen et al. [17] also focus multi-disk setups and distribute relations to balance load and minimize interference using non-linear programming. Microsoft's AutoAdmin recommends column groups using strongly pruning greedy heuristics that estimate costs by counting co-occurrences of attributes [16]. In its essence, AutoAdmin is comparable to our heuristics as it uses occurrence counting. Rösch et al. [36] built a storage advisor for the HTAP database SAP HANA, which recommends both horizontal as well as vertical partitioning. Their vertical partitioning approach splits tables into row- and column-oriented tables, comparable to our hybrid data format. Halim et al. [37] presented an approach called database cracking. Both [38] and [37] focus on increasing performance for disk-optimized databases while our approach rather focuses on preserving performance of an in-memory database when a substantial share of data is evicted to secondary storage. In contrast to our proposal, all the mentioned approaches for physical design optimization disregard the effects of successive columnar filtering.

In the field of hybrid data layouts for disk-resident databases, PAX [39] stores a page's tuples in a columnar format to improve cache locality. There are several comparable approaches for hybrid disk-based layouts without multiple data copies. They have in common that point accesses (e.g., via tuple reconstructions) are fast due to reading only single pages while (analytical) sequential accesses are decelerated compared to columnar approaches (DSM). We think using formats such as PAX for SSCGs is useful and beneficial to restrict the negative impact of sequential accesses (cf. Section IV-C) and see it as future work. Idreos et al. [26] found columnar tuple reconstruction from secondary storage to be problematic and use sideway cracking at the expense of additional mapping structures and multiple data copies. In the field of DRAM-resident databases, Peloton [8] is a recent hybrid approach with a flexible tile concept that can resemble both row- and column-oriented schemes, similar to Hyrise. H_2O [7] uses adaptable column groups to adjust to the given workload on the fly.

Recent DRAM-optimized approaches such as Anti-Caching [40] or Siberia [41] horizontally separate and evict data that has not been accessed for a certain period using LRU-like approaches. Both approaches focus on tuple-/block-wise eviction optimized for transactional workloads. In the context of HTAP, HyPer compresses cold columnar data but does not (yet) evict data [10], [25]. Hoeppner et al. proposed a solution for SAP HANA [42]. Infrequently accessed attribute vectors are evicted while dictionaries are left DRAM-resident.

VI. Conclusion & Future Work

We have presented a tiered main memory-optimized database for mixed workloads using hybrid table layouts. The hybrid layout separates attributes into two groups in order to retain the analytical performance of in-memory column stores and improve access performance to cold data on secondary storage. We evict a significant share of the data to secondary storage while simultaneously showing comparable (and for several real-world workloads *improved*) performance.

To determine which columns to keep in DRAM, we have introduced a novel optimization model using integer linear programming (ILP). The model's performance is optimal and outperforms heuristics since coupled aspects such as selectivity, size, and frequency of queries are incorporated. Our model is also able to take reallocation costs for secondary storage into account to determine whether a reconfiguration is advantageous.

Further, we have shown two interesting results. First, we prove that solutions of a transformed continuous LP coincide with optimal ILP solutions for specific memory budgets. Second, we show how to compute these optimal solutions of the transformed LP explicitly. Hence, our explicit solution combines both advantages: optimized performance *and* rapid computation times. In addition, our results reveal a recursive solution principle which can be used as an effective heuristic in general column selection problems.

In future works, we will analyze scenarios in which workload changes over time in more detail. To react to changing workloads – which are characterized by query frequencies – varying time frames (moving windows) of historic workload data can be used to feed the model and to adapt the data layout successively. Further, our model can also be directly combined with approaches to predict future workloads (e.g., via time series analyses). Estimations of expected query frequencies can be used in our model framework to compute optimized column selections for anticipated workloads.

Acknowledgments

We want to thank our project partner Dell EMC^2. Besides supplying the AMM software, we want to thank Adrian Michaud, Kurt Roedszus, and Ken Taylor for their continuous support and feedback over the past years. Further, we want to thank Intel for providing the Intel Optane P4800X devices.

References

[1] D. J. Abadi, D. S. Myers *et al.*, "Materialization strategies in a column-oriented DBMS," in *Proc. ICDE 2007*. IEEE, 2007, pp. 466–475.
[2] D. J. Abadi *et al.*, "Column-stores vs. row-stores: how different are they really?" in *Proc. SIGMOD 2008*. ACM, 2008, pp. 967–980.
[3] H. Plattner, "The impact of columnar in-memory databases on enterprise systems," *PVLDB*, vol. 7, no. 13, pp. 1722–1729, 2014.
[4] P. Larson, C. Clinciu, C. Fraser *et al.*, "Enhancements to SQL Server column stores," in *Proc. SIGMOD 2013*. ACM, 2013, pp. 1159–1168.
[5] N. Mukherjee *et al.*, "Distributed architecture of oracle database in-memory," *PVLDB*, vol. 8, no. 12, pp. 1630–1641, 2015.
[6] V. Raman *et al.*, "DB2 with BLU acceleration: So much more than just a column store," *PVLDB*, vol. 6, no. 11, pp. 1080–1091, 2013.
[7] I. Alagiannis, S. Idreos, and A. Ailamaki, "H2O: a hands-free adaptive store," in *Proc. SIGMOD 2014*, 2014, pp. 1103–1114.
[8] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the archipelago between row-stores and column-stores for hybrid workloads," in *Proc. SIGMOD 2016*. ACM, 2016, pp. 583–598.
[9] T. Willhalm *et al.*, "SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units," *PVLDB*, vol. 2, pp. 385–394, 2009.
[10] H. Lang, T. Mühlbauer, F. Funke *et al.*, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *Proc. SIGMOD 2016*. ACM, 2016, pp. 311–326.
[11] T. Neumann, "Engineering high-performance database engines," *PVLDB*, vol. 7, no. 13, pp. 1734–1741, 2014.
[12] L. Ma, J. Arulraj, S. Zhao *et al.*, "Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems," in *Proc. DaMoN*. ACM, 2016, pp. 9:1–9:7.
[13] H. Bian *et al.*, "Wide table layout optimization based on column ordering and duplication," in *Proc. SIGMOD 2017*. ACM, 2017, pp. 299–314.
[14] M. Grund, J. Krüger, H. Plattner *et al.*, "HYRISE - A main memory hybrid storage engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.
[15] G. Graefe, "Volcan - an extensible and parallel query evaluation system," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 1, pp. 120–135, 1994.
[16] S. Agrawal *et al.*, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*, 2004, pp. 359–370.
[17] O. Ozmen *et al.*, "Workload-aware storage layout for database systems," in *Proc. SIGMOD*, 2010, pp. 939–950.
[18] D. Rotem *et al.*, "Data allocation for multi-disk databases," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 5, pp. 882–887, 1993.
[19] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in *CIDR 2005*, 2005, pp. 225–237.
[20] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index interactions in physical design tuning: Modeling, analysis, and applications," *PVLDB*, vol. 2, no. 1, pp. 1234–1245, 2009.
[21] M. Stonebraker, D. J. Abadi, A. Batkin *et al.*, "C-Store: A column-oriented DBMS," in *Proc. VLDB 2005*, 2005, pp. 553–564.
[22] J. Krüger *et al.*, "Fast updates on read-optimized databases using multi-core CPUs," *PVLDB*, vol. 5, no. 1, pp. 61–72, 2011.
[23] D. Schwalb *et al.*, "Efficient transaction processing for Hyrise in mixed workload environments," in *Proc. IMDM@VLDB 2014*. IMDM, 2014.
[24] F. Färber *et al.*, "The SAP HANA database – an architecture overview," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.
[25] F. Funke *et al.*, "Compacting transactional data in hybrid OLTP & OLAP databases." *PVLDB*, vol. 5, no. 11, pp. 1424–1435, 2012.
[26] S. Idreos *et al.*, "Self-organizing tuple reconstruction in column-stores," in *Proc. SIGMOD 2009*, 2009, pp. 297–308.
[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proc. SIGMOD 1979*. ACM, 1979, pp. 23–34.
[28] C. Meyer, M. Boissier, A. Michaud *et al.*, "Dynamic and transparent data tiering for in-memory databases in mixed workload environments," in *Proc. ADMS@VLDB 2015*, 2015, pp. 37–48.
[29] G. B. Mathews, "On the partition of numbers," *Proceedings of the London Mathematical Society*, vol. s1-28, no. 1, pp. 486–490, 1896.
[30] N. Katoh and T. Ibaraki, *Resource Allocation Problems*. Boston, MA: Springer US, 1998, pp. 905–1006.
[31] R. Sherkat *et al.*, "Page as you go: Piecewise columnar access in SAP HANA," in *Proc. SIGMOD*, 2016, pp. 1295–1306.
[32] M. Boissier *et al.*, "Analyzing data relevance and access patterns of live production database systems," in *Proc. CIKM 2016*, 2016, pp. 2473–2475.
[33] Y. Wu *et al.*, "An empirical evaluation of in-memory multi-version concurrency control," *PVLDB*, vol. 10, no. 7, pp. 781–792, 2017.
[34] M. Faust, D. Schwalb, and H. Plattner, "Composite group-keys: Space-efficient indexing of multiple columns for compressed in-memory column stores," in *Proc. IMDM@VLDB 2014*. IMDM, 2014, pp. 42–54.
[35] R. L. Cole *et al.*, "The mixed workload CH-benCHmark," in *Proc. DBTest 2011*. New York, NY, USA: ACM, 2011, pp. 8:1–8:6.
[36] P. Rösch *et al.*, "A storage advisor for hybrid-store databases," *PVLDB*, vol. 5, no. 12, pp. 1748–1758, 2012.
[37] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores," *PVLDB*, vol. 5, no. 6, pp. 502–513, 2012.
[38] A. Rasin *et al.*, "An automatic physical design tool for clustered column-stores," in *Proc. EDBT 2013*. ACM, 2013, pp. 203–214.

[39] A. Ailamaki *et al.*, "Weaving relations for cache performance," in *Proc. VLDB 2001*. VLDB Endowment, 2001, pp. 169–180.

[40] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-caching: A new approach to database management system architecture." *PVLDB*, vol. 6, no. 14, pp. 1942–1953, 2013.

[41] A. Eldawy *et al.*, "Trekking through siberia: Managing cold data in a memory-optimized database," *PVLDB*, vol. 7, no. 11, pp. 931–942, 2014.

[42] B. Höppner *et al.*, "An approach for hybrid-memory scaling columnar in-memory databases," in *Proc. ADMS@VLDB 2014*. ADMS, 2014.

VII. Notation

Workload	i	$1,...,N$, N the number of columns
	j	$1,...,Q$, Q the number of queries
	a_i	size of column i, $i = 1,...,N$
	s_i	selectivity of column i, $i = 1,...,N$ (i.e., average share of tuples with same value)
	q_j	columns used by query j, $j = 1,...,Q$, subset of $\{1,...,N\}$, e.g., $q_1 = \{8,6,13,14,87\}$
	b_j	frequency of query j, $j = 1,...,Q$
	g_i	number of queries j in a workload that include column i, i.e., $g_i := \sum_{j=1,...,Q, i \in q_j} b_j$, $i = 1,...,N$
	f_j	scan costs of query j, $j = 1,...,Q$
	c_i	coefficients of the transformed objective, $i = 1,...,N$
	S_i	auxiliary parameter, $j = 1,...,Q$
	o_i	optimized order of columns i, $i = 1,...,N$
Cost Parameters	c_{mm}	scan cost parameter **m**ain **m**emory
	c_{ss}	scan cost parameter **s**econdary **s**torage
	α	cost parameter for DRAM used
	β	cost parameter for reallocation
	A	DRAM budget
	w	share of total size of columns allowed in DRAM
Variables	x_i	decision variables: column in DRAM yes (1) / no (0), $i = 1,...,N$, and allocation vector $\vec{x} = (x_1,...,x_N)$
	$F(\vec{x})$	total scan costs of allocation $\vec{x}$
	$M(\vec{x})$	required DRAM budget of allocation $\vec{x}$
	y_i	given initial/current state for x_i, $i = 1,...,N$
	z_i	auxiliary variable for linearization, $i = 1,...,N$