Reducing the Footprint of Main Memory HTAP Systems: Removing, Compressing, Tiering, and Ignoring Data

Martin Boissier
Supervised by Prof. Hasso Plattner
Hasso Plattner Institute, University of Potsdam, Germany
martin.boissier@hpi.de

ABSTRACT

Gracefully reducing the main memory footprint (e.g., via compression and data tiering) is an unsolved challenge for HTAP database systems, where most traditional reduction methods are no longer applicable. Since advantages of a reduced footprint are manyfold, the issue is of high importance for main memory-resident databases. In this paper, we present our work on Hyrise and discuss how we break down this challenge into three aspects in order to reduce the main memory consumption without losing the performance advantage of in-memory databases. First, we reduce existing allocations by efficiently selecting indices and workload-driven compression configurations for table data. Second, we use hybrid table layouts to tier data with minimal impacts on the runtime performance. Third, we employ data structures that efficiently eliminate unnecessary accesses to secondary storage. As an outlook, we depict our vision to eventually unite these aspects in a holistic footprint optimization.

1. MEMORY FOOTPRINT REDUCTION

Database systems that store their entire working set in DRAM have a clear performance advantage over their disk-based counterparts. But DRAM can easily become the main cost driver for large server systems. Its capacity still remains limited, especially when compared to modern storage arrays with petabytes of storage. Moreover, DRAM is responsible for up to 40% of a server's energy consumption [15].

Besides cost considerations, growing data volumes limit the applicability of main memory-resident database systems. Basically, all in-memory database vendors have added capabilities to handle data volumes larger than DRAM capacities, which underlines the need of efficient means to lower the memory footprint (i.e., main memory consumption). A lower footprint provides obvious benefits such as decreased hardware costs but also enables faster recovery times. On top of that, gained free space can be used to execute faster algorithms with larger memory requirements or for adding auxiliary data structures such as secondary indexes.

One self-imposed condition that makes our project particularly interesting is the focus on hybrid transactional and analytical workloads (so-called HTAP, OLxP, or mixed workloads). Most existing approaches, e.g., page- or tuple-based eviction, incur disastrous performance penalties for workloads that are mostly dominated by sequential processing.

Proceedings of the VLDB 2018 PhD Workshop, August 27, 2018. Rio de Janeiro, Brazil.

We implement our research in the open source database Hyrise[1]. Hyrise is a columnar in-memory database optimized for HTAP workloads. The database is ACID-compliant and uses MVCC concurrency control. Data in Hyrise is horizontally partitioned into so-called chunks with a predefined size (comparable to [14]). Data is inserted in the most recent mutable chunk until this chunk reaches its limit. At this point, a new empty chunk is created and the recently filled chunk becomes immutable (using append-only for updates and MVCC invalidations for deletions). We approach tiering in Hyrise by separating the problem into three aspects:

1. Before any data is evicted (to presumably significantly slower storage tiers), existing data structures are compressed or even removed (cf. Section 3).
2. We use hybrid table layouts to vertically partition and evict data to secondary storage (cf. Section 4).
3. We employ auxiliary data structures to ignore irrelevant data (i.e., data skipping or pruning, cf. Section 5).

In this paper, we will discuss these three aspects which are already published or currently work in progress. The final objective of the pursued thesis is to create a holistic view onto these three aspects and properly balance them.

2. EXISTING WORK

In recent years, a variaty of main memory-resident database systems has been released (e.g., Hekaton [8], SAP HANA [10], H-Store [12], HyPer [13]). Even though these databases aim to fully exploit DRAM's performance characteristics, they all have in common that means to handle larger-than-DRAM data sets are incorporated (cf. Hekaton's project Siberia [9], HANA's paged attributes [18, 21], anti-caching in H-Store [7], HyPer's chunk compaction [11]).

To distinguish our work from these systems, we categorize the related work in three groups: (i) means to classify cold data, (ii) tiering granularity, and (iii) access avoidance.

2.1 Identifying Cold Data

The primary way to identify cold data (cold data usually refers to the most infrequently accessed data) depends on the workload targeted. OLTP-optimized databases such as H-Store and SQL Server's Hekaton evict single tuples with a strong focus on transaction throughput and latency.

The HTAP-optimized database SAP HANA does not employ a workload-driven system but rather is the application expected to provide business rules that define data being

[1]Hyrise on GitHub: `https://git.io/hyrise`

(i) no longer relevant for the daily ongoing business and (ii) which cannot be modified anymore (e.g., due to regulatory agreements). The reason for this approach is SAP's strong devotion to enterprise systems [21]. Another HTAP-optimized database, HyPer, assumes horizontal partitions (chunks) become colder with their age increasing [14].

2.2 Tiering Granularity

OLTP-optimized databases usually evict fine-granular horizontal entities, usually pages or tuples. Moreover, H-Store also tiers indexes [22].

SAP HANA allows defining columns as *page loadable*, making columns loadable and evictable on a page basis. Also, indexes and column dictionaries can be page loadable. HyPer varies the compression and page size with the age of a chunk. Chunks start being completely uncompressed (*hot*), then become *cold*, and eventually are stored as immutable *frozen* chunks using huge pages and heavier compression [11]. Recently, HyPer gained the capability to also evict data to secondary storage [16].

2.3 Access Avoidance

To avoid unnecessary accesses to evicted data, most approaches create additional data structures or aggregates that allow pruning accesses for the majority of queries.

Hekaton uses *adaptive range filters* which allow pruning both point queries (even though less effective than Bloom filters) and range queries [2]. SAP HANA stores *synopses* per attribute, which include, e.g., minimum and maximum values of a column [18]. HyPer uses *positional small materialized aggregates* storing clustered min/max aggregates to prune or limit sequential scans to a smaller subset [14].

To our best knowledge, the discussed approaches lack the means for a workload-driven data tiering that can be dynamically configured by changing the memory budget. Further, we see unused potential of access avoidance data structures to be further used for more accurate cardinality estimations.

3. COMPRESSION & REMOVAL

The first step to reduce the memory footprint is compression and removal of existing data structures.

3.1 Compression

Hyrise supports different column compression schemes (e.g., dictionary encoding, run-length encoding, frame-of-reference, SIMD-BP128). While this topic is well researched, there has only been little work on how to integrate various compression schemes efficiently. The capability of compressing columns individually usually either introduces (i) a high maintenance overhead as operators have to be adapted for each compression schema or (ii) runtime degradations due to added indirections via virtual method calls. We implement an approach similar to QuickStep [20] and use C++ meta-programming to avoid both drawbacks at the expense of compile time (cf. Fig. 2(a)). Similar to [5], we divide compression schemes into logical-level techniques and physical-level techniques, which allows us to use the implemented compression techniques also for other data structures (e.g., indexes or position lists that are passed between operators).

When it comes to deciding which compression scheme to use, there is a clear lack in database research. Most commercial systems use simple data statistics and heuristics similar to Abadi et al.'s decision tree for compression selection [1].

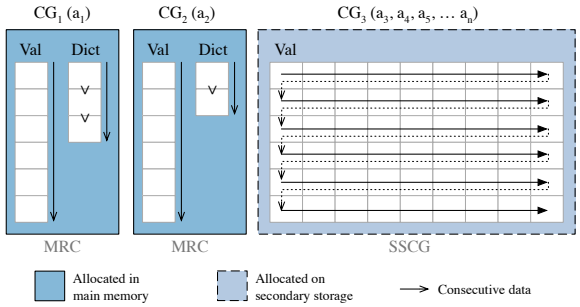


Figure 1: Depiction of a tiered hybrid layout (cf. [3]).

To our best knowledge, no existing approach incorporates workload knowledge and allows selecting compression configurations for a given memory budget. As a consequence, we developed a novel heuristic to determine optimized compression configurations. Given a workload and a memory budget, this selection approach determines for each column in each chunk a suitable compression scheme using analytical size estimations and regression-based runtime predictions.

3.2 Selection & Removal

For the removal of data structures, we concentrate on auxiliary data structures such as secondary indices. This problem is a well-studied topic and known as the *index selection problem*. We employ an adapted version of the heuristic presented in [4] for the index selection problem. First evaluations show a comparable solution quality with state-of-the-art approaches such as CoPhy [6] with a significantly shorter runtime what allows us to process large systems with larger sets of candidates (see short comparison in Table 1).

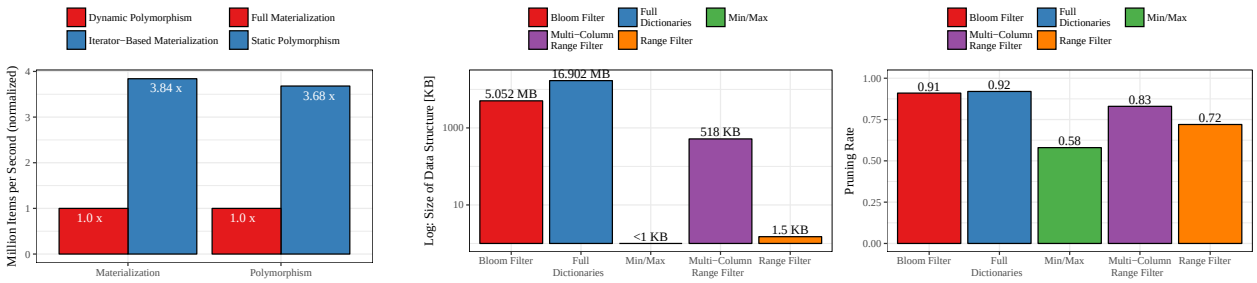
	CoPhy (100)	CoPhy (200)	CoPhy (500)	CoPhy (9 912)	Our Approach
Mem. traffic (TB)	5 371	4 730	2 489	403	425
Runtime in seconds	0.62	7.23	121.00	DNF	0.65

Table 1: Comparing our index selection heuristic against CoPhy for a large production ERP system (numbers in parentheses denote the size of the candidate set; unlimited set for our approach).

4. DATA TIERING

Chunks in Hyrise can use hybrid layouts that vertically separate the attributes into two groups. The first group contains all attributes that have been accessed apart from projections and is stored in a columnar fashion remaining in DRAM. The second group contains all attributes that have only been projected or not accessed at all. Attributes of this group are stored in a row-major format and evicted to secondary storage. For memory budgets lower than the size of the first group in DRAM, a workload-driven Pareto-optimal heuristic selects which columns to keep in DRAM. Evaluations using an Intel Optane P4800X showed eviction rates of up to 80% with neglectable performance hits for a real-world enterprise SAP system. This hybrid table layout has recently been presented at the ICDE 2018 [4].

As of now, we do not consider index tiering. In the long run, we plan to adapt a flexible storage manager comparable to [16], which is applicable to any data structure.



(a) Performance improvements due to iterator-based materialization and static polymorphism.

(b) Comparison of five data structures for access avoidance. Bloom filters and full dictionaries show the highest pruning rates but also largest space consumptions.

Figure 2: Selected comparisons: (a) the effects of zero-cost abstractions for compression schemes and (b) a comparison of access avoidance data structures for workload and data of an production ERP system.

5. ACCESS AVOIDANCE

When we talk about access avoidance in the context of Hyrise, we mean auxiliary data structures and aggregates that (i) enable pruning (or elimination) of chunks which are irrelevant for processing the query result and (ii) improve cardinality estimations during query optimization. As Hyrise's chunk concept guarantees that chunks cannot be modified as soon as they reach their size limit, we create several aggregates and auxiliary data structures on each chunk without the need of keeping them updated with every transaction. The creation is done during chunk compaction to piggy-back the full chunk iteration during compression.

Data gathered per attribute during compaction include the share of NULL values, distinct values count, the number of runs (used to estimate compression rates for run-length encoding), and minimum/maximum values (cf. [17]). On top of these aggregates, we use (as of now very simple) heuristics to select which of the following – more sophisticated – auxiliary data structures are potentially beneficial:

Pruning Histograms are extended histograms with pruning capabilities, combining equi-width histograms and range filters. These histograms store value ranges with the objective to create as large as possible gaps between the value ranges. These gaps allow the pruning of queries whose predicate ranges are not intersecting with any value range in the filter. Each value range further stores the number of elements within the range and the number of distinct elements.

Multi-Column Pruning Histograms are each created between two selected attributes. In addition to the pruning histograms of both attributes, a bitmap is stored which signals whether tuples exist for value range combinations (see depiction in Figure 3). This way, queries with conjunctive predicates might be pruned where each predicate alone might not indicate a pruning possibility. Moreover, this data structure can be used (to some extend) to gather information about dependencies between attributes.

Approximate Set Membership Filters – similar to Bloom filters – allow approximate pruning of accesses.We use *counting quotient filters* (CQF) by Pandey et al. [19]. CQFs allow checking for set memberships as well as estimating a queried item's frequency of occurrence in the set. Hence, besides access pruning, CQFs can improve cardinality estimations for skewed data.

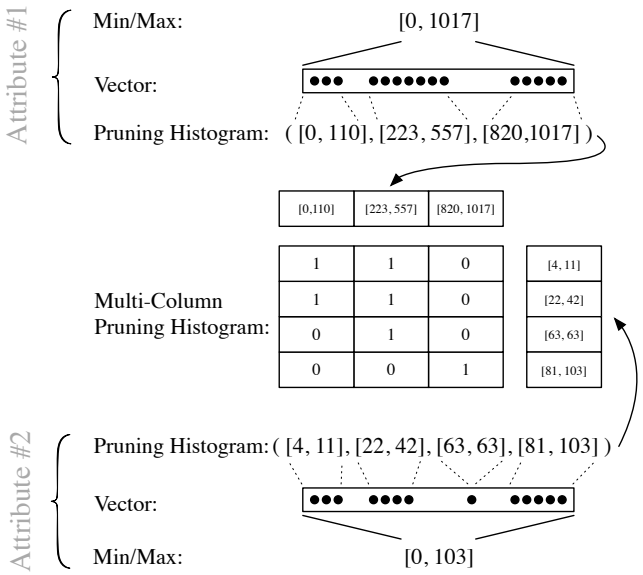


Figure 3: Depiction of pruning histograms in Hyrise.

Figure 2(b) shows pruning rates and memory consumption of various data structures for workload and data of a production ERP system. Particularly pruning histograms and multi-column pruning histograms offer a promising trade-off between pruning rate and size. For highly skewed attributes with many point accesses, CQFs are a viable alternative.

An interesting challenge is the balancing of data eviction and creation of data structures that avoid accesses to the evicted data. As a consequence, besides implementing and integrating the mentioned data structures in Hyrise, the main objective will be a selection model. For a given memory budget, this model decides (i) which data structures to create and (ii) how much space to allocate for each structure (all proposed structures are dynamically sizable).

5.1 Query Planning & Optimization

Recognizing that a chunk can be pruned for a given predicate (e.g., during a linear table scan) can improve efficiency

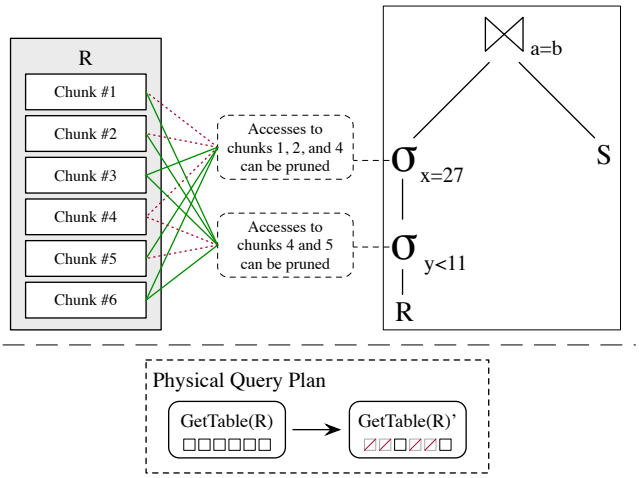


Figure 4: Example of a query plan with two predicates. For conjunctive predicates, lists of excluded chunks per predicate can be unioned and pushed down. For the physical query plan, the `GetTable` operator is modified to discard non-skippable chunks.

and performance but leaves large potentials unused: (i) improved cardinality estimations and (ii) combined pruning of conjunctive filter predicates.

Cardinality Estimation: One very prominent remaining problem of query optimization is the accurate estimation of cardinalities. When pruning is executed as part of a physical plan operation, strongly pruning predicates cannot directly trigger a plan re-evaluation. In Hyrise, all auxiliary data structures are used within the query optimizer to improve selectivity estimations, which enables triggering re-evaluations and predicate re-orderings.

Conjunctive Predicate Chains: We have analyzed production enterprise systems and found that these systems conjunctively filter on several dimensions frequently [3]. Often, predicates in those chains exclude different chunks of the table. Hence, when the first filter operation scans chunks which are later found to be irrelevant for the result, this scan was superfluous. This is the case when pruning is solely used within the execution engine and not as part of query optimization. In Hyrise, we search conjunctive chains of predicates that are directly executed on physical tables. For conjunctive chains, we can union the lists of prunable chunks and directly prune those chunks during loading of the table before any operator is executed (see Figure 4).

We also plan to evaluate exploiting these data structures as part of query execution (e.g., for semi-join reductions).

6. CURRENT STATUS & FUTURE WORK

We have presented the current state of footprint reduction and data tiering in Hyrise. Considering our preliminary results, we are optimistic that we can make substantial contributions to the database community. Especially light-weight data structures that both improve efficiency and performance by avoiding unnecessary accesses as well as improved accuracy within query optimization appear promising.

Part of our upcoming work is the holistic optimization of Hyrise's memory footprint, incorporating all aspects from compression, over tiering, to access avoidance. Eventually, a holistic optimization is required as the presented aspects interact with each other. As soon as data is compressed or evicted, there will be an access penalty. Data tiering and accesses to secondary storage potentially cause devastating performance hits unless accesses can be pruned whenever possible. Hence, measures to avoid unnecessary accesses must be taken in order to preserve performance. The slower the data tier for eviction is, the more value pruning data structures provide.

7. REFERENCES

[1] D. J. Abadi et al. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD 2006*, pages 671–682, 2006.

[2] K. Alexiou et al. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.

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

[4] M. Boissier, R. Schlosser, and M. Uflacker. Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In *Proc. ICDE*, pages 209–220, 2018.

[5] P. Damme et al. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Proc. EDBT 2017*, pages 72–83, 2017.

[6] D. Dash, N. Polyzotis, and A. Ailamaki. CoPhy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[7] J. DeBrabant, A. Pavlo, S. Tu, et al. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[8] C. Diaconu et al. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. SIGMOD*, pages 1243–1254, 2013.

[9] A. Eldawy et al. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

[10] F. Färber et al. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[11] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP & OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.

[12] R. Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

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

[14] H. Lang et al. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proc. SIGMOD 2016*, pages 311–326, 2016.

[15] C. Lefurgy et al. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.

[16] V. Leis et al. LeanStore: In-memory data management beyond main memory. In *Proc. ICDE*, pages 185–196, 2018.

[17] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. VLDB 1998*, pages 476–487, 1998.

[18] A. Nica et al. Statisticum: Data statistics management in SAP HANA. *PVLDB*, 10(12):1658–1669, 2017.

[19] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proc. SIGMOD 2017*, pages 775–787, 2017.

[20] J. M. Patel et al. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[21] R. Sherkat et al. Page as you go: Piecewise columnar access in SAP HANA. In *Proc. SIGMOD*, pages 1295–1306, 2016.

[22] H. Zhang et al. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proc. SIGMOD 2016*, pages 1567–1581, 2016.