An overview of HYRISE - a Main Memory Hybrid Storage Engine

Martin Grund[1], Philippe Cudre-Mauroux[2], Jens Krueger[1], Samuel Madden[3], Hasso Plattner[1]
[1]Hasso Plattner Institute, 14482 Potsdam, Germany
[2]eXascale Infolab, University of Fribourg, Switzerland,
[3]Database Group, MIT CSAIL, USA

Abstract

HYRISE is a new relational storage engine for main memory database systems. It is built on the premise that enterprise application workloads can benefit from a dedicated main-memory storage engine. The key idea behind HYRISE is that it provides dynamic vertical partitioning of the tables it stores. Since enterprise applications typically use a large number of very wide tables, we designed a novel layout algorithm specifically tailored for enterprise data. Our algorithm uses a main memory cost model that is able to precisely estimate the physical access cost of database operators and to determine a vertical partitioning that yields the best execution performance.

1 Introduction

Traditionally, the database market is divided into transaction processing (OLTP) and analytical processing (OLAP) workloads. OLTP workloads are characterized by a mix of reads and writes to a few rows at a time, typically through a B+Tree or other index structures. Conversely, OLAP applications are characterized by bulk updates and large sequential scans spanning few columns but many rows of the database, for example to compute aggregate values. Typically, those two workloads are supported by two different types of database systems – transaction processing systems and warehousing systems.

This simple categorization of workloads, however, does not entirely reflect modern enterprise computing. First, there is an increasing need for "real-time analytics" – that is, up-to-the-minute reporting on business processes that have traditionally been handled by warehousing systems. Although warehouse vendors are doing as much as possible to improve response times (e.g., by reducing load times), the explicit separation between transaction processing and analytics systems introduces a fundamental bottleneck in analytics response times. For some applications, directly answering analytics queries from the transactional system is preferable. For example "available-to-promise" (ATP) applications process OLTP-style queries while aggregating stock levels in real-time using OLAP-style queries to determine if an order can be fulfilled. Recently two new benchmarks TPC-CH[6] and a mixed workload benchmark presented in[3] emerged underlining the current trend in modern database systems towards mixed workloads scenarios.

Unfortunately, existing databases are not optimized for such mixed query workloads because their storage structures are usually optimized for one workload or the other. To address such workloads, we have built a

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

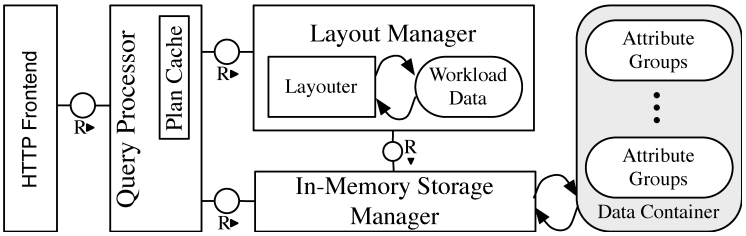

Figure 1: Architecture of HYRISE

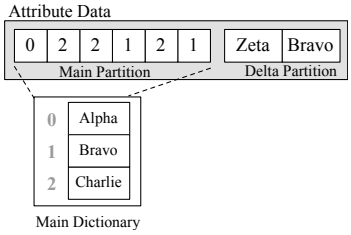

Figure 2: Logical structure of an attribute

main memory hybrid database system, called HYRISE, which partitions tables into vertical partitions of varying widths depending on how the columns of the tables are accessed (e.g., transactionally or analytically).

We focus on main memory systems because, like other researchers [11], we believe that many future databases – particularly those that involve enterprise entities like customers, outstanding orders, products, stock levels, and employees – will fit into the memory of a small number of machines.

Main memory systems present a unique set of challenges and opportunities. Due to the architecture of modern CPUs and their complex cache hierarchy, comparing the performance of different main memory layouts can be challenging.

Our system captures the idea that it is preferable to use narrow partitions for columns that are accessed as a part of analytical queries, as is done in pure columnar systems [4, 5]. In addition, HYRISE stores columns that are accessed in OLTP-style queries in wider partitions, to reduce cache misses when performing single row retrievals. Though others have noted the importance of cache locality in main memory systems [5, 2, 8, 12], we believe we are the first to build a dedicated hybrid database system based on a detailed model of cache performance in mixed OLAP/OLTP settings.

2 HYRISE Overview

In this section we provide an overview of HYRISE and its main components. The general architecture of HYRISE is depicted in Figure 1 and consists of the following main components: a unique hybrid storage engine, a hybrid layout manager and a query engine.

Hybrid Storage Engine The key concept of HYRISE is to natively support vertical partitioning of relational tables into disjoint sets of attributes. Each partition is internally represented as a *container*. Containers can in turn be freely composed to build the relational table. We built HYRISE to optimize data access to these containers. Traditionally, databases uses page structures to store data. Inside a page, tuples are stored one after the other. In contrast, tuples in a HYRISE containers are stored inside a single compressed block of main memory. Additionally, HYRISE uses dictionary compression to replace actual values with an encoded value id. The dictionaries are stored as sorted lists allowing offset-based mapping from encoded value to actual values. This solution has two advantages: First, the effect of each access to a container can be calculated independently of the actual value or value distribution. Second, expensive value-based comparisons, e.g. on string values, can be executed in the dictionary directly using efficient binary operations on integers. Due to the explicit ordering of value ids, HYRISE only materializes intermediate results as values when it is explicitly required to do so during query execution.

Modifications to a container are stored in an uncompressed delta partition to avoid having to re-compress or re-encode partitions when data is modified. To optimize memory consumption and query execution performance, data is cyclically re-compressed[9]. The logical structure of an attribute with a sorted main dictionary and the delta partition is shown in Figure 2. All tuples in HYRISE are stored following an insert-only approach. Deletes

and updates are transformed into operations setting an invalid attribute on the delta container. Depending on the chosen strategy, invalid tuples are either kept to allow time-travel operations[10] or they are simply removed during the re-compression phase.

Query Execution Query execution in HYRISE is flexible and allows choosing from different materialization and execution strategies. Initially, query plans in HYRISE are hand-coded operator graphs represented in JSON.

Parallelization of analytical queries is achieved by parallellizing the data flow of the query plan. Relational plan operators are split into smaller independent tasks that are then executed in parallel. Update operations are serialized per container, which allows us to reduce latch and lock costs for read-only queries. Intermediate results in HYRISE are modeled as temporary tables, which are either directly materialized or stored as position lists. The advantage of position lists is that expensive data copying can be avoided as long as possible[1].

Layout Manager The layout manager in HYRISE takes as input a sample workload, and returns as output a specification of the physical layout of the database. This specification is used to generate the containers that are stored in HYRISE, or to transform the layout of a previously loaded containers into a different representation.

Our HYRISE layout engine currently supports three different algorithms: a candidate based layout algorithm, a divide and conquer algorithm and an incremental layout algorithm. The first two algorithms are optimal, in the sense that they always find the best layout, while the divide and conquer algorithm is a scalable but approximate algorithm. We give more detail on our layout algorithms below in Section 3.2.

Workload Evolution Taking advantage of the layout manager described above, we were able to build a system that can easily adapt in case the workload changes. Our incremental layout algorithm allows us to compute a new optimal layout based on the delta between two separate workloads. With this incremental algorithm, our system can frequently recompute the layout and detect if a change of the layout is required, even for very large applications and frequently changing workloads.

3 Automated Logical Database Design

The design goal for our layout algorithm is to scale both with the number of attributes in tables of large enterprise applications and with the huge number of tables in these applications.

The number of possible hybrid physical designs (combinations of non- overlapping containers containing all of the columns) for a particular table is huge. For a table of n attributes, there exist $a(n)$ possible hybrid designs, where $a(n) = (2n-1)a(n-1) - (n-1)(n-2)a(n-2)$, where $a(0) = a(1) = 1$. There are for instance $3,535,017,524,403$ possible layouts for a table of 15 attributes. The only automated hybrid designer we are aware of, the HillClimb Data Morphing algorithm [8], does not work for wide tables in practice since it scales exponentially (2^n) with the number of attributes in both time and space. We propose a new set of algorithms that can efficiently determine the most appropriate physical design for tables of many tens or hundreds of attributes given a database and a query workload.

3.1 Layouts

A layout in HYRISE is defined by a set of containers that implicitly splits a relational table into a set of disjoint partitions $P_1, \ldots, P_n$. A single attribute can only occur in one and only one partition. While the attributes inside a single partition are ordered, a valid layout consists of an unordered set of partitions, such that such the layouts $\lambda_1 = \{(a_1, a_2), (a_3, a_4)\}$ and $\lambda_2 = \{(a_3, a_4), (a_1, a_2)\}$ are considered identical.

Formally, given a database DB and a workload W, our goal is to determine the list of layouts $\boldsymbol{\lambda_{opt}}$ minimizing the workload cost. The cost model we use in this case is based on cache misses and presented in more detail in [7].

3.2 Layout Selection

Based on our cost model, which allows us to estimate the number of cache misses occurred by an operation, we make two observations: First, a relational projection that retrieves $\pi.w$ bytes out of a $C.w$-byte container generally incurs an overhead that is caused by loading bytes into cache that are not read by the projection. This overhead is proportional to $C.w - \pi.w$ for full scans, and can vary for selections depending on the exact alignment of the projection and the cache lines. We call this overhead *container overhead* cost.

Second, when output tuples can be reconstructed without any cache eviction, the cost expression *distributes* over the set of queries, in the sense that the total access cost of a workload can be computed by summing the cost of each query individually on the partitions it accesses.

3.3 Candidate Based Layout Selection

Based on our model and the above observations, our layout algorithm works in three phases called *candidate generation*, *candidate merging*, and *layout generation* phases.

Candidate Generation The first phase of our layout algorithm determines all primary partitions for all participating tables. A primary partition is defined as the largest partition that does not incur any container overhead cost. For each relation $\mathcal{R}$, we start with the complete set of attributes $\{a_1, \ldots, a_m\}$ in $\mathcal{R}$. Each operation op_j implicitly splits this set of attributes into two subsets: the attributes that are accessed by the operation, and those that are ignored. The order in which we consider the operations does not matter in this context. By recursively splitting each set of attributes into subsets for each operation op_j, we end up with a set of $|P|$ *primary* partitions $\{P^1_1, \ldots, P^1_{|P|}\}$, each containing a set of attributes that are always accessed together. The cost of accessing a primary partition is independent of the order in which the attributes are laid out, since all attributes are always queried together in a primary partition.

Candidate Merging: The second phase of the algorithm inspects permutations of primary partitions to generate additional candidate partitions that may ultimately reduce the overall cost of the workload. Our cost model shows us that merging two primary partitions P^1_i and P^1_j is advantageous for wide, random access to attributes since corresponding tuple fragments are co-located inside the same partition; for projections, the merging process is usually detrimental due to the additional access overhead (which occurs unless both primary partitions are perfectly aligned to cache lines.)

This tension between reduced cost of random accesses and penalties for large scans of a few columns allows us to prune many of the potential candidate partitions by comparing the cost of the workload for the set of primary partitions with their merged counterpart. If the cost for the merged partitions is equal to or greater than the sum of the individual costs of the partitions (due to the container overhead), then this candidate partition can be discarded. If a candidate partition is not discarded by this pruning step, it is added to the current set of partitions and will be used to generate valid layouts in the following phase.

Layout Generation The third and last part of our algorithm generates the set of all valid layouts by exhaustively exploring all possible combinations of the partitions returned by the second phase. The algorithm evaluates the cost of each valid layout consisting of a covering but non-overlapping set of partitions, discarding all but the physical layout yielding the lowest cost. This last layout is the optimal layout according to our cost

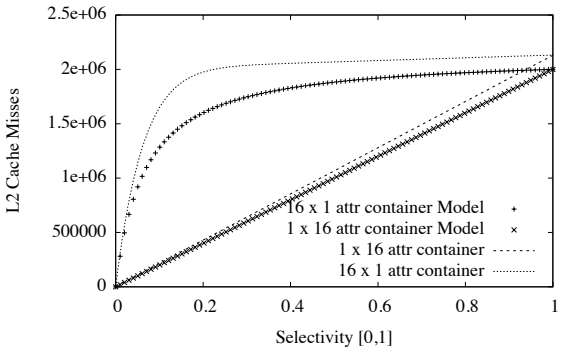

Figure 3: Comparing predicted and measured Level 2 cache misses for the selection operator with varying selectivity.

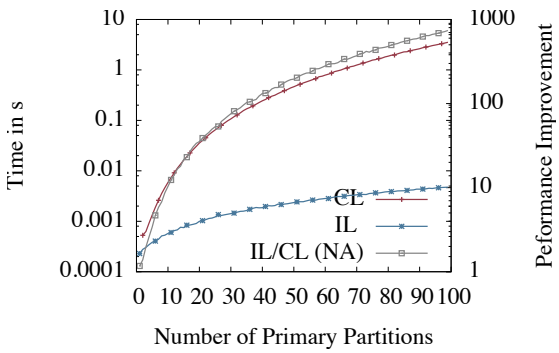

Figure 4: Comparing layout generation performance for Candidate Layouter (CL) and Incremental Layouter (IL)

model, since all potentially interesting permutations of attributes are examined by our algorithm (only irrelevant permutations, such as subsets of primary partitions or suboptimal merges from Section 3.3, are discarded).

The worst-case space complexity of our layout generation algorithm is exponential with the number of candidates partitions $|P|$. However, it performs very well in practice since very wide relations typically consist of a small number of sets of attributes that are frequently accessed together (thus, creating a small number of primary partitions) and since operations across those partitions are often relatively infrequent (thus drastically limiting the number of new partitions generated by the second phase above.)

3.4 Divide and Conquer Partitioning

For large relations and complex workloads involving hundreds of different frequently-posed queries, the running time of the above algorithm may still be high. To address this situation, HYRISE provides an approximate layout algorithm that clusters the primary partitions that are frequently co-accessed together. To retrieve the partitions that are accessed together, we build a symmetric affinity matrix that is transformed into a weighted graph. Based on the weights, we partition the graph in order to obtain a series of *min- cut* subgraphs to minimize the total cost (weight) of all edges that must be removed. We then determine the optimal layout for each of the subgraphs, which is computationally much lighter than considering the whole graph. The intermediate layouts of the subgraphs are used as candidates for the merging phase. Finally, we combine the partitions of the sub-layouts from the previous step to find a near-optimal overall layout.

4 Comparison and Performance Characteristics

To evaluate our system we used a mixed workload benchmark consisting out of 13 queries, including 9 transactional queries, and 4 analytical queries. Using our layout algorithm, we were quickly able to determine the optimal layout based on our accurate cache-miss cost model (Figure 3.) In total for the above scenario our hybrid partitioning is 4 times better than storing the data using rows and about 60% faster compared to storing the data fully decomposed as columns. The results of the individual queries as shown in [7] show a strong tension between the different layouts. It is important to mention that our algorithm optimizes for the best result of the overall observed workload and might sacrifice the performance of an individual query for this goal.

In addition, we evaluated the performance of our layout algorithm to validate its applicability in the area of enterprise applications with very wide schemas and many thousand tables. A table with 100 primary partitions, with hundreds of attributes, can be handled in under 10s, when we apply our incremental layout algorithm (Figure 4.)

5 Conclusions

In this paper, we gave an overview of HYRISE, a new hybrid in-memory storage engine we have built that can outperform both row and column-wise storage systems. To do this, HYRISE employs workload-aware layout algorithm that determines the optimal physical layout of a database and scales to large-scale of enterprise data. The overall goal of our work is to provide an application-specific storage layer for enterprise data. Based on further analyses of enterprise applications, we see are beginning to explore different storage containers that may not be strictly relational, but that also integrate semi-structured and unstructured data into one main-memory system.

References

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[3] Anja Bog, Kai Sachs, Alexander Zeier, and Hasso Plattner. Normalization in a Mixed OLTP and OLAP Workload Scenario. In *Third TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, 2011.

[4] Peter Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65. Morgan Kaufmann, 1999.

[5] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[6] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload CH-benCHmark. In *DBTest*, page 8. ACM, 2011.

[7] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[8] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[9] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

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

[11] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160. ACM, 2007.

[12] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Workshop on Data Management on New Hardware*, pages 47–54. ACM, 2008.