HYRISE—A Main Memory Hybrid Storage Engine

Martin Grund
Hasso-Plattner-Institute

Jens Krüger
Hasso-Plattner-Institute

Hasso Plattner
Hasso-Plattner-Institute

Alexander Zeier
Hasso-Plattner-Institute

Philippe Cudre-Mauroux
MIT CSAIL

Samuel Madden
MIT CSAIL

ABSTRACT

In this paper, we describe a main memory hybrid database system called HYRISE, which automatically partitions tables into vertical partitions of varying widths depending on how the columns of the table are accessed. For columns accessed as a part of analytical queries (e.g., via sequential scans), narrow partitions perform better, because, when scanning a single column, cache locality is improved if the values of that column are stored contiguously. In contrast, for columns accessed as a part of OLTP-style queries, wider partitions perform better, because such transactions frequently insert, delete, update, or access many of the fields of a row, and co-locating those fields leads to better cache locality. Using a highly accurate model of cache misses, HYRISE is able to predict the performance of different partitionings, and to automatically select the best partitioning using an automated database design algorithm. We show that, on a realistic workload derived from customer applications, HYRISE can achieve a 20% to 400% performance improvement over pure all-column or all-row designs, and that it is both more scalable and produces better designs than previous vertical partitioning approaches for main memory systems.

1. INTRODUCTION

Traditionally, the database market divides 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.

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 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 [22, 7], 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. Commercially available systems already offer up to 1 TB of main memory (e.g., the Fujitsu RX600 S5).

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. In this paper, we carefully profile the cache performance of a modern multi-core machine and develop a cost model that allows us to predict the layout-dependent performance of a mixed OLTP/OLAP query workload on a fine-grained hybrid row/column database.

Our model 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 [5, 6]. 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 [6, 3, 12, 25], 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. Our work is closest in spirit to Data Morphing [12], which also proposes a hybrid storage model, but we have extended their approach with a more accurate model of cache and prefetching performance for modern processors that yields up to 60% fewer cache misses compared to the layouts proposed by Data Morphing. Furthermore, the layout algorithms described in the Data Morphing paper are exponential (2^n) in the number of attributes in the input relations, and as such do not scale to large relations. Our algorithms scale to relations with hundreds of columns, which occur frequently in real workloads.

We note that several analytics database vendors have announced support for hybrid storage layouts to optimize performance of par-

Proceedings of the VLDB Endowment, Vol. 4, No. 2

ticular workloads. For example, Vertica introduced FlexStore, which allows columns that are accessed together to be physically stored together on disk. VectorWise, Oracle, and GreenPlum have made similar announcements. None of these vendors have released detailed information about how their hybrid schemes work, and they do not appear to have database designers such as ours that can automate hybrid partitioning, but these products acknowledge the importance of hybrid designs such as those we explore in this paper.

In summary, we make several contributions in this paper:

1. We develop a detailed cache performance model for layout-dependent costs in hybrid main memory databases.
2. We develop an automated database design tool that, given a schema, a query workload, and using our analytical model, recommends an optimal hybrid partitioning.
3. We show that our system, running on a customer-derived benchmark (which we describe in detail), is 20% to 400% faster than either a pure-row or a pure-column store running on the same data. We also show that our designs are better than previous hybrid storage schemes.

Before describing the details of our model and design algorithms, we provide a brief overview of the architecture of HYRISE.

2. HYRISE ARCHITECTURE

The following section describes the architecture of HYRISE. The main architectural components are shown in Figure 1. The *storage manager* is responsible for creating and maintaining the hybrid containers storing the data. The *query processor* receives user queries, creates a physical query plan for each query, and executes the query plan by calling the storage manager. The *layout manager* analyzes a given query workload and suggests the best possible layout (partitioning) for this workload to the storage manager.

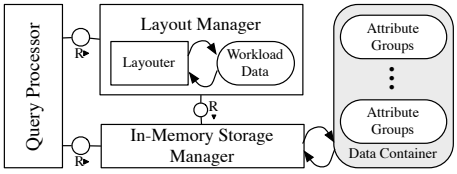


Figure 1: HYRISE architecture

We have built a prototype of this architecture. Our prototype executes hand-coded queries based on the query processor API and currently lacks support for transactions and recovery. We omit these features because we believe they are orthogonal to the question of which physical design will perform best for a given workload. However, to minimize the impact of transactions in HYRISE, in addition to normal write operations, we use non-temporal writes, which make it possible to write directly back to main memory without loading the written content into the CPU cache (see Appendix E.) Even though our prototype currently executes one query at a time only, we use thread-safe data structures that include latch acquisition costs to support later query parallelization.

We give an overview of both the storage manager and the query processor below. The approach used by the layout manager to select good layouts is described in detail in Section 4.

2.1 Storage Manager

Our HYRISE prototype supports a fine-grained hybrid storage model, which stores a single relation as a collection of disjoint vertical partitions of different widths. Each partition is represented by a data structure we call *container*. Each attribute is mapped to one and only one container. A container provides methods to access the various values it holds. Containers are physically stored as a list of large contiguous blocks of memory. Data types are dictionary-compressed into fixed-length fields to allow direct access (offsetting) to any given position (exploring further compression schemes is an area of future work.) Position offsets typically come from another container or from a value-index lookup.

Figure 2 shows an example of a relation r with eight attributes partitioned into three containers. In this example, the first container contains one attribute only. The second and third containers contain five and two attributes respectively.

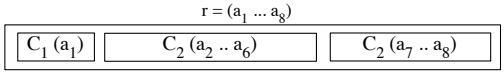


Figure 2: Partitioning example

Since our findings in enterprise software show that historic data must be kept for legal reasons [18] our system currently focuses on selections and insertions. In order to keep track of all data changes, we handle updates and deletions using validity timestamps as described in [23].

2.2 Query Processor

The HYRISE query processor creates a query plan, consisting of a tree of operators, for every query it receives. HYRISE currently implements projection, selection, join, sorting, and group by operators. For joins HYRISE includes hash and nested loops join algorithms. Most of our operators support both early and late materialization, meaning that HYRISE provides both position or value-based operators [1]. In late materialization, filters are evaluated by determining the row indexes ("positions") that satisfy predicates, and then those positions are looked up in the columns in the SELECT list to determine values that satisfy the query (as opposed to early materialization, which collects value lists as predicates are evaluated.)

Non-join queries are executed as follows: index-lookups and predicates are applied first in order to create position lists. Position lists are combined (e.g., *ANDed*) to create result lists. Finally, results are created by looking-up values from the containers using the result lists and are merged together to create the output tuples. For join plans, predicates are first applied on the dimension tables. Then, foreign-key hash-joins are used to build position lists from the fact tables. Additional predicates can then be applied on the fact tables to produce additional position lists. All position lists are combined with the output of the joins, and the final list of positions is used to create the final results. Query execution is currently single-threaded and handles one operator at a time only; we are extending HYRISE to support efficient parallel execution for multi-core processors.

3. HYBRID MODEL

In this section, we derive a cost model for the most important operations performed in HYRISE. This cost model will be used in Section 4 to compare the performance of various hybrid layouts given a query workload.

We distinguish between *layout-dependent* and *layout-independent* costs. Layout-dependent operations access the data from its primary physical representation—the costs of these operators vary depending on the physical storage structures used. Layout-independent operations occur when accessing intermediate results that are created as a result of query processing. The cost of such operators does not vary when the physical storage layout changes. The magnitude of layout-independent costs depends on the materialization strategy, since early materialization will result in more intermediate results. We focus on layout-dependent operations in the following since these are the only operations that benefit from changes of the physical layout.

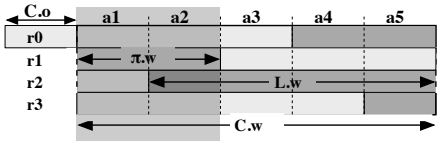


Figure 4: A projection projecting the first two attributes of a 5-attribute container

Most of the layout-dependent costs incurred in a main-memory system like HYRISE originate from CPU stalls—typically caused by cache misses when moving data from main memory; those CPU stalls are known to account for a significant fraction of the total cost of a query (see [4]). Our experiments in Section 5 show that cache misses are a good predictor of query runtime.

Our model is based on a detailed analysis of such costs, taking into account the cache misses for different cache levels (e.g., L1 and L2 cache). Unlike previous cache-aware cost models (Manegold *et al.* [16] and Boncz *et al.* [6]), which focused on column-oriented designs only, we analyze sets of hybrid data containers that can store an arbitrary number of columns or rows. We choose this level of detail so that our cost model can be reused in a cost-based optimizer.

To illustrate our model, we provide the cache misses and CPU cycles for different operations as measured in our system. All measurements were executed using an Intel E5450 quad-core CPU with 32KB per core L1 data and instruction cache (8-way associative, 64 byte cache lines), a shared 6MB L2 cache (24-way associative, 64 byte cache lines), and 64 GB of PC2 5300 CL2 RAM.

3.1 Notation

We consider a database DB, consisting of a list of relations $r \in \mathcal{R}$. Each relation r is defined by a list of attributes $(a_{1_r}, \ldots, a_{m_r})$ and contains a certain number of tuples $r.n$. Each relation is decomposed into a set of containers $C_{1_r}, \ldots, C_{n_r}$. Each container stores a subset of the attributes of r: $C_{i_r} = (a_{k_r}, \ldots, a_{l_r})$ (in the remainder of this section, we omit the subscript r for readability). We say that each container stores a contiguous list of tuple *fragments*. We write $C_i.w$ to denote the width of the container in bytes, and $C_i.n$ for the number of rows in the container.

In the following, we evaluate the number of cache misses for the main operations supported by our system. We write $L_i.w$ to express the length (in bytes) of a cache line for cache level i, and $L_i.n$ to indicate the number of cache lines available for cache level i. The total size of the cache for level i is thus $L_i.n \times L_i.w$. Loading a cache line through the cache hierarchy causes the CPU to stall. We write $L_i.cpu$ to express the number of CPU cycles spent to load a line from cache level i to cache level $i-1$.

3.2 Partial Projections

We start by evaluating the number of cache misses that occur when performing a projection π on a container C. Initially, we restrict the projections to a series of contiguous attributes in C, starting at an offset $\pi.o$ from the beginning of the container and retrieving $\pi.w$ bytes of attributes. A simple example is depicted in Figure 4 for a 5-attribute container and a projection retrieving the first two attributes of the container ($\pi.o = 0$ and $\pi.w = 8$ *bytes* considering 4-byte attributes).

Projections are executed by reading the relevant portions of the containers. If the data is not already cached, the system reads it from RAM and loads it into the cache hierarchy—assuming an inclusive cache hierarchy (as in Intel Processors)—one cache line and level at a time. Two cases can occur depending on the width of the container, the projection, and the cache line. In the first case, when

$$C.w - \pi.w < L_i.w \tag{1}$$

the entire container must be read, resulting in a *full scan*. This happens whenever the non-projected segments of the container ($C.w - \pi.w$ for each container row) are strictly smaller than a cache line and can *never* be skipped when retrieving the projected pieces. The number of cache misses incurred by a full scan is:

$$\mathit{Miss}_i(C, \pi) = \left\lceil \frac{C.w \times C.n + C.o}{L_i.w} \right\rceil \tag{2}$$

Here, $C.o$ denotes the offset (in bytes) between the beginning of the container and the first preceding address that can be mapped to the beginning of a cache line. In this case, the number of cache misses corresponds to the number of cache lines needed to read the entire container ($C.w \times C.n$), plus any additional accesses if the address of the beginning of the container is not aligned to the beginning of a cache line (i.e., $C.o \neq 0$).

If the condition in equation 1 does not hold, parts of the container can be skipped when executing the projection. The number of cache misses incurred by such a *partial* projection depends on the alignment of each row r with respect to the cache lines. We first determine the offset $r.o$ from the start of the container to the start of the r-th row of the container:

$$r.o = C.w \times r. \tag{3}$$

The offset between the beginning of the projection of the r-th row and the beginning of the nearest previous cache line is:

$$\mathit{lineoffset}_i(r, \pi) = (C.o + r.o + \pi.o) \mod L_i.w. \tag{4}$$

To retrieve the projected attributes for the r-th row, the system has to read $\pi.w$ bytes, in addition to the $\mathit{lineoffset}_i(r, \pi)$ bytes implicitly read by the cache because of the misalignment between the cache line and the projected segment. The number of cache lines required to read the r-th row is thus:

$$\mathit{Miss}_i(r, \pi) = \left\lceil \frac{\mathit{lineoffset}_i(r, \pi) + \pi.w}{L_i.w} \right\rceil \tag{5}$$

Finally, the total number of cache misses incurred by the partial projection is:

$$\mathit{Miss}_i(C, \pi) = \sum_{r=0}^{C.n-1} \mathit{Miss}_i(r, \pi). \tag{6}$$

Due to the high number of iterations to calculate the total cache misses, we would like to replace equation 6 with an exact calculation (compared to the average calculation in [16]). The key observation is that the value of $\mathit{lineoffset}_i(r, \pi)$ follows a repeating pattern, depending on the value of $Li.w$ and $C.w$. In general, the number of distinct values of $\mathit{lineoffset}_i(r, \pi)$ is known as the *additive order* of $C.w \mod Li.w$ [14], and has v distinct values:

$$v = Li.w / gcd(C.w, Li.w) \tag{7}$$

where $gcd(C.w, Li.w)$ is the greatest common divisor of $C.w$ and $Li.w$. Hence, it is enough to evaluate equation 6 for the first v rows, and then multiply the result by $C.n/v$; that is:

$$\mathit{Miss}_i(\pi, C) = \frac{C.n}{v} \sum_{r=0}^{v} \mathit{Miss}_i(r, \pi). \tag{8}$$

To illustrate this model, we compare the number of cache misses for different layouts. Figure 3(a) shows the results of an experiment on two different layouts, one with 100 narrow one-attribute containers, and the other one with only one wide 100-attribute container (all attributes are 4 byte long). Both layouts have the same total width. The figure reports the total number of L2 cache misses for partial projections ranging from one to 100 attributes, as well as the number of misses predicted by our model (note that the lines completely overlap.) Figure 3(b) further shows that there is a relation between the number of cache misses and the number of CPU cycles for these operations. Cache misses are highly correlated with—and a good predictor of—total CPU cycles in database access methods because the primary action of these operators is to retrieve values from memory, and cache misses tend to dominate these memory access costs for memory-bound operations.

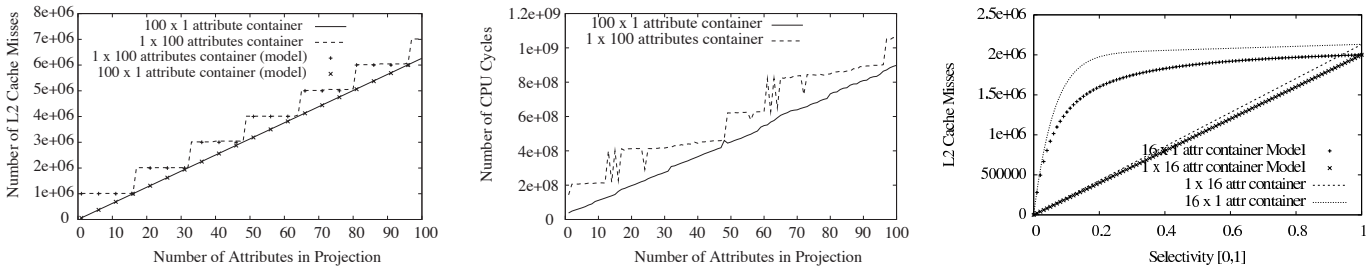


Figure 3: Modeled vs Measured L2 misses (a); CPU cycles (prefetcher off) (b); L2 Misses Row/Column Containers and Varying Selectivity (c)

3.3 Combining Partial Projections

The previous section discussed the projection of contiguous attributes from a container. However, query plans often need to project non-contiguous sets of attributes. Non-contiguous projections can be rewritten as a set of partial projections $\{\pi_1, \ldots, \pi_k\}$, each of which retrieves a list of contiguous attributes in C. Such projections define a set of *gaps* $\{\gamma_1, \ldots, \gamma_l\}$, i.e., contiguous attributes groups that are not projected. For instance, a projection on the first and third attribute of a five-attribute container is equivalent to two partial projections—one on the first and one on the third attribute. The projection defines two gaps, one on the second attribute, and a second one on the fourth and fifth attributes. Two cases can occur depending on the width $\gamma.w$ of the gaps:

Full-scan: if $\forall \gamma_i \in \{\gamma_1, \ldots, \gamma_l\}, \gamma_i.w < L_i.w$, all gaps are strictly smaller than a cache line and cannot be skipped. Thus, the projection results in a full scan of the container.

Independent projections: if $\exists \gamma_i \in \{\gamma_1, \ldots, \gamma_l\} \mid \gamma_i.w \geq L_i.w$, there exist portions of the container that might potentially be skipped when executing the projection. The projection is then equivalent to a set of $1 + \sum_{i=1}^{l} \mathbb{1}_{\gamma_i.w \geq L_i.w}$ partial projections defined by the gap boundaries (where $\mathbb{1}$ is an indicative function used to express the gaps that are greater than the cache line). Writing Γ_i to express the i-th largest gap whose width $\Gamma_i.w \geq L_i.w$, the equivalent partial projections can be defined as

$$\pi_i^{eq}.o = \Gamma_i.o. \tag{9}$$

and

$$\pi_i^{eq}.w = \Gamma_{i+1}.o - (\Gamma_i.o + \Gamma_i.w) \tag{10}$$

Similarly, we can merge the first and last projections by taking into account the fact that the last bytes of a row are stored contiguously with the first bytes of the following row. Hence, we merge the first and last projections when the gap Γ_{row} between them is smaller than a cache line, i.e., when

$$\Gamma_{row} = (\pi_{first}^{eq}.o + C.w) - (\pi_{last}^{eq}.o + \pi_{last}^{eq}.w) < L_i.w. \tag{11}$$

The final projection π^{eq} is in that case defined as follows: $\pi^{eq}.o = \pi_{last}^{eq}.o$ and $\pi^{eq}.w = \pi_{first}^{eq}.w + \pi_{last}^{eq}.w + \Gamma_{row}$.

Using this framework, we can model the impact of complex queries involving the projection of an arbitrary set of attributes from a container without erroneously counting misses twice.

3.4 Selections

In this subsection, we consider projections that only retrieve a specific subset $\mathcal{S}$ of the rows in a container. We assume that we know the list of rows $r_i \in \mathcal{S}$ that should be considered for the projection (e.g., from a position lookup operator). The selectivity of the projection $\pi.s$ represents the fraction of rows returned by the projection. Our equations in this section must capture the fact that highly selective projections touching a few isolated rows can generate more cache misses per result than what full-scans would do.

Two cases can also occur here, depending on the relative sizes of the container, the projection, and the cache line:

Independent selections: whenever $C.w - \pi.w - 1 \geq L_i.w$, the gaps between the projected segments cause each row to be retrieved independently of the others. In that case, the cache misses incurred by each row retrieval are independent of the other row retrievals. The total number of cache misses incurred by the selection is the sum of all the misses incurred when independently projecting each row $r \in \mathcal{S}$ from the set of selected rows:

$$\mathit{Miss}_i(C, \pi)_{sel} = \sum_{r=0}^{C.n-1} \mathit{Miss}_i(r, \pi)\ \pi.s. \tag{12}$$

This number of misses can be efficiently computed using the *additive order* approach described above.

Overlapping selections: when $C.w - \pi.w - 1 < L_i.w$, retrieving the projection for a given row may retrieve parts of the projection for a different row. This effect is particularly apparent for low-selectivity projections on narrow containers, for which several rows can fit on a single cache line. The average number of rows that can fit in one cache line is equal to $L_i.w/C.w$. For each cache line fetched to retrieve a selected row, there are on average

$$\mathit{totalCachedRows} = 1 + \pi.s\ \left(\frac{L_i.w}{C.w} - 1\right) \tag{13}$$

selected rows cached, assuming that the rows are selected independently of each other. The average number or misses is thus

$$\mathit{Miss}_i(C, \pi)_{sel} \cong \frac{\pi.s}{\mathit{totalCachedRows}}\ \mathit{Miss}_i(C, \pi). \tag{14}$$

Figure 3(c) compares the measured and modeled number of cache misses for selections on two layouts: one consisting of 16 one-attribute containers and a second one consisting of one 16-attribute wide container. Both layouts have the same total width and both have 2M tuples. For low selectivities, using wider containers results in fewer cache misses, since each (whether narrow or wide) container generates at least one cache miss per tuple fragment retrieved for very selective projections.

3.5 Joins and Aggregates

Our system currently uses late-materialization based hash joins and hash-based GROUP BYs. These operations can both be modeled as partial projections and selections. For a join of tables R and S, where R is the table that is to be hashed, R is filtered via a partial projection and a position lookup (using positions from early operators in the plan). The resulting hash table is then probed with a similarly filtered version of S. For GROUP BYs, the grouping column is also filtered via a partial projection and position lookup.

3.6 Padded Containers and Reconstruction

Padding and Alignment: For partial projections it is possible to reduce the total number of cache misses by performing narrow row-padding such that the beginning of each row coincides with the beginning of a cache line. For a padded container C, the padding (empty space) $\rho.w$ to insert at the end of each row to achieve such an effect is $C.w \mod L_i.w$ bytes wide for a given cache level. The expressions given above to compute the number of cache misses can be used on padded containers by replacing the width of the container

$C.w$ and the width of the row $r.w$ by their corresponding expressions taking into account padding, namely $C.w + \rho.w$ and $r.w + \rho.w$ respectively. As an example: For a 90-attribute container, where each attribute is 4 bytes, an additional 24 bytes of padding is added per tuple. This increases access performance in 87% of all simple projections with an average speedup of $\approx 7\%$.

Depending on the associativity of the L1 cache and the implemented replacement policy for cache lines, HYRISE applies a special alignment policy to avoid early evictions due to cache set collisions (for details, see Appendix A).

Tuple Reconstruction: If the final result is materialized into one result set, output tuples must be built from multiple containers. In the rare case that the width of an output tuple is wider than the available cache size at a given level (e.g., 32KB for L1, 6MB for L2), evictions will occur before the last attribute of the tuple is written, triggering additional cache misses. To avoid these misses, the output must be written one container at a time instead of one tuple at a time.

4. LOGICAL DATABASE DESIGN

There are a very large number of possible hybrid physical designs (combinations of non-overlapping containers containing all of the columns) for a particular table. For a table of n attributes, there exist $a(n)$ possible hybrid designs, where

$$a(n) = (2n-1)a(n-1) - (n-1)(n-2)a(n-2) \quad (15)$$

where $a(0) = a(1) = 1$. This corresponds to the number of partitions of $\{1, \ldots, n\}$ into any number of ordered subsets.

There are for instance 3,535,017,524,403 possible layouts for a table of 15 attributes. Most previous hybrid database systems do not automatically suggest designs to the database administrator (see Section 6) — the only automated hybrid designer we are aware of, the HillClimb Data Morphing algorithm [12], 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 two new 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. Our first algorithm has a worst-case running time that is exponential in the problem size, but incorporates several pruning steps that allows it to scale to wide tables in practice. Our second algorithm includes a partitioning step and can scale to larger problems, while introducing bounded sub-optimality.

4.1 Layouts

We start by extending the model described in Section 3.1. We consider a query workload W, consisting of a set of queries q_i: $W = \{q_1, \ldots, q_m\}$ that are regularly posed against the database. Each query has a weight $w_1, \ldots, w_m$ that captures the relative frequency of the query. Furthermore, each query has a cost, representing the time required by the database to answer the query. The time needed to answer all queries of a given workload W is thus proportional to:

$$Cost_{DB}(W) \sim \sum_{i=1}^{m} w_i \, Cost_{DB}(q_i) \quad (16)$$

where $Cost_{DB}(q_i)$ is the cost (i.e., total number of cache misses weighted by the correct value of $L_i.cpu$ for each cache level) associated with the operations performed as part of q_i as described in the preceding section. The containers implicitly split the relations into sets of *partitions* $P_1, \ldots, P_n$ with:

$$\forall a_i \in R \, \exists P_i \mid a_i \in P_i \wedge a_i \notin P_j \forall P_j \neq P_i. \quad (17)$$

We call the sets of partitions following the preceding condition *layouts* $\lambda \in \Lambda$. Note that layouts consist of unordered sets of partitions, such that 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.

We write $\boldsymbol{\lambda} = (\lambda_{R_1}, \ldots, \lambda_{R_r})$ to express the list of layouts by which relations $R_1, \ldots, R_r$ are stored in the system. The rest of this section is devoted to the determination of good layouts that will minimize the query response time. 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:

$$\boldsymbol{\lambda_{opt}} = \underset{\boldsymbol{\lambda}}{\operatorname{argmin}} \left(Cost_{\boldsymbol{DB}}(W) \right). \quad (18)$$

4.2 Layout Selection

We use the model defined in the previous section to automatically determine good hybrid layouts given a database DB and a workload W. Based on the model, we make two observations: First, projections retrieving $\pi.w$ bytes out of a $C.w$-wide container often incur an overhead. This overhead is caused by loading attributes into cache that are not used by the projection. This overhead is proportional to $C.w - \pi.w$ for full scans, and can vary for partial projections and selections depending on the exact alignment of the projection and the cache lines. We call this overhead *container overhead* cost.

Second, when the output tuples can be reconstructed without any cache eviction (Section 3.6), the cost expression *distributes* over the set of queries, in the sense that each cost can be decomposed and computed separately for each partition and the corresponding subsets of the queries $\{q_i, \ldots, q_j\}$ accessing the partition:

$$Cost_{\boldsymbol{DB}}(\{q_1, \ldots, q_m\}) = \sum_{P \in \boldsymbol{\lambda}} Cost_P(\{q_i, \ldots, q_j\}) \quad (19)$$

Based on our model and the above observations our layout algorithm works in three phases called *candidate generation*, *candidate merging*, and *layout generation* phases. An example is described in detail in Appendix B.

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 $\{P_1^1, \ldots, P_{|P|}^1\}$, 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_i^1 and P_j^1 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. To do this, we compute the cost of the workload W, $Cost_{P_i^n}(W)$ on every candidate partition P_i^n obtained by merging n primary partitions $(P_1^1, \ldots, P_n^1)$, for n varying from 2 to $|P|$. If this cost 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: In that case, the candidate partition can systematically be replaced by an equivalent or more optimal set of partitions consisting of the n primary partitions $P_1^1, \ldots, P_n^1$ since

$$Cost_{P_i^n}(\{q_i, \ldots, q_j\}) \geq \sum_{m=1}^{n} Cost_{P_m^1}(\{q_i, \ldots, q_j\}) \quad (20)$$

and since the other terms of the total layout cost (Equation 19) are not affected by this substitution. 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 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 4.2, 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).

4.3 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. In this section, we propose an approximate algorithm that clusters the primary partitions that are often co-accessed together, generating optimal sub-layouts for each cluster of primary partitions, and finally combining the optimal sub-layouts.

We start by generating a $|P| \times |P|$ matrix M, storing in each entry $M(i, j)$ the number of times the two primary partitions $\{P_i^1, P_j^1\}$ are accessed together. Computing this number is done using our cost model to estimate how many rows are accessed by each query in each primary partition. This matrix is symmetric and can be seen as a graph where each vertex is a primary partition, and the weight between two vertices i and j ($M(i, j)$) represents the co-access affinity between the two primary partitions P_i^1, P_j^1.

We partition this graph in order to obtain a series of *min-cut* subgraphs each containing at most K primary partitions, where K is a constant. These cuts seek to minimize the total cost (weight) of all edges that must be removed. This is a very well studied problem in the theory community, and there exist practical implementations of these algorithms; in our system, we use `metis`, an efficient and approximate multilevel k-way partitioner [15].

At this point, each subgraph contains a set of primary partitions that are accessed together, and which thus represent excellent candidates for our merging phase (Section 4.2). We determine the optimal layout of each subgraph separately using the algorithm described above (Section 4.1), which is in the worst-case exponential with the maximum number of primary partitions in a subgraph (K).

Finally, we combine the sub-layouts obtained in the previous step: we incrementally combine pairs of partitions P_i and P_j belonging to two different sub-layouts and yielding the most savings according to our cost model, until no further cost-reduction is possible. This final step requires $O(|P| * \binom{|P|}{2})$ partition evaluations in the worst-case (here $|P|$ is the total number of partitions in all sub-layouts), but is much faster in practice since the most similar primary partitions are already clustered together in the same subgraph, and since narrow partitions yielding many different combinations are trivial to evaluate as they contain few attributes.

This approximate algorithm is very effective in practice (it always finds the optimal layouts for our workload and for $K > 3$, for instance). The only times it generates suboptimal layouts is when a complex combination of partitions belonging to different sub-layouts yields a smaller cost than the one found using our greedy algorithm. The penalty incurred is in any case never greater than $2 * \sum_{cut} M(i, j)$ (where $\sum_{cut} M(i, j)$ represents the set of edges removed during the graph-partitioning phase), which is the maximal penalty incurred by the partitioning phase. It is also very efficient for relatively small values of K (see Appendix F for details).

5. PERFORMANCE EVALUATION

In this section we evaluate the performance of HYRISE on a workload derived from a customer application. Our goal is to compare the performance of an all-row or all-column database design against our hybrid approach, and to validate the cost model and database designer described above. To increase the execution performance we performed several additional optimizations, including memory alignment, narrow-row padding and cache-set collision avoidance, as described in Appendix A.

To evaluate our model we choose a set of queries derived from an SAP enterprise resource planning (ERP) application that includes several analytical queries that model reporting over the recent history of these transactions. To show the robustness of our approach we execute one analytical query in two different versions (Q11 and Q12), with a layout-dependent selectivity of 2% (Q11) and 50% (Q12).

We built our own application-derived workload because real enterprise applications (such as those we have encountered at SAP) exhibit significant differences in terms of number of attributes per table from benchmarks like TPC-C, TPC-E, and TPC-H. For example, in TPC-E (the most complex of these three benchmarks) the maximum number of attributes per relation is about 25; in SAP's enterprise applications it is not uncommon to see tables with 200 attributes or more. A second reason for creating our own benchmark is that we wanted to execute both OLTP-style and analytical-style queries on the same data, and is not easy to retrofit an existing benchmark like TPC-C or TPC-H to support both analytical and transactional queries without substantailly changing benchmark. A detailed description of the benchmark and its queries are given in Appendix C.

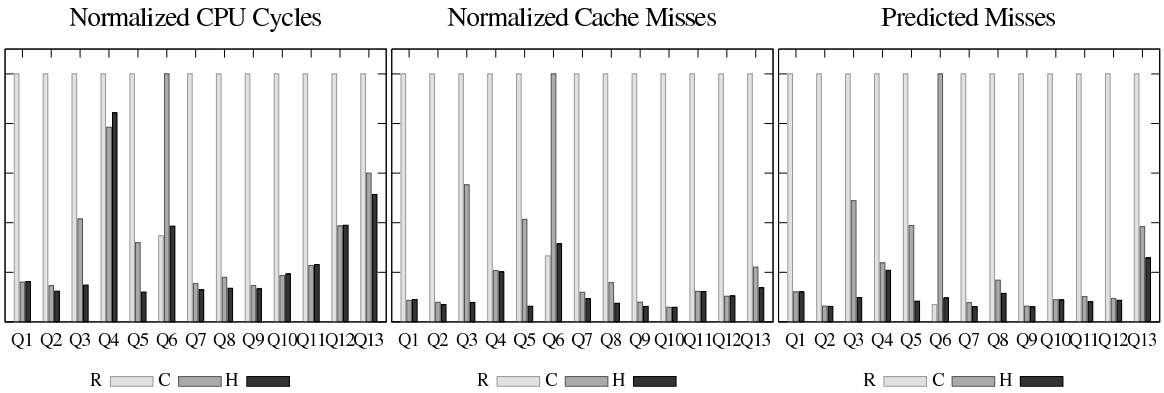
5.1 Schema and Query Selection

The schema for our workload is based on a CRM application. The main tables represents sales orders (`VBAK`) and sales order line items (`VBAP`). The schema also contains tables for materials (`MARA`), material texts (`MAKT`), business partners (`KNA1`), business partner addresses (`ADRC`), and the material hierarchy table (`MATH`).

When running queries, we populated tables with sizes obtained from interviews with consultants familiar with the application. Specifically, the sales order header table (VBAK) contains 3.6M entries and the sales order line item table (VBAP) 144M items. Each sales order has between 3 and 5 items. The sizes of the additional tables are $600,000$ materials (MARA), $600,000$ material texts (MAKT), $180,000$ addresses, $144,000$ business partners (ADRC) and 1M elements in the material hierarchy. The selectivity of the queries is matched to the results from the actual application deployment. The total system size in memory is about 28 GB.

As an example, we show the result that our layout generator produced for the `VBAP` sales order line-item table. In the following notation `4_block` represents an attribute which is 4 blocks wide, normalized to the width of 4 bytes per block.

1. `(('VBELN')`
2. `('MATNR')`
3. `('KWMENG','AEDAT')`
4. `('94_block')`
5. `('1_block', '1_block', '4_block',`
   `'70_block', '39_block','NETWR'))`



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Total
R	56770	24030	27050	15250	96780	90	13890.3	52301.5	5431.5	32297.6	29687.8	117471.1	4899.2	475949
C	9050	3510	11220	11930	30940	260	2154.8	9416.0	795.5	6032.4	6744.1	45468.6	2939.8	140461.2
H	9290	2910	4010	12810	11660	100	1795.3	7114.8	723.2	6243.5	6852.6	45751.1	2517.7	111778.2

Figure 5: Benchmark Results; graphs at the top show normalized (slowest system=1.0) CPU cycles (left) and normalized L2 cache misses (right); table shows absolute CPU cycles / 100k

Each item represents one vertical partition in the table. While there is no particular order for the partitions, all attributes inside the partition are stored in order. We chose this table as an example since it is accessed by most of the queries and is partitioned to optimize for the competing needs of several queries. Since the sales order number (`VBELN`) and the related material (`MATNR`) columns are often scanned in their entirety (e.g., by Q6 and Q8) the layouter chooses to store them as single columns. The amount `KWENG` and the delivery data `AEDAT` columns are always accessed together (Q11,Q12) and thus are stored as a group of two attributes. The rest of the attributes are merged by the layout algorithm to achieve best possible performance on `SELECT*` queries.

5.2 Performance

For each of the all-rows, all-columns, and optimal HYRISE designs, we used our cost model to estimate the total cost and also executed them in the HYRISE query executor. For all queries we captured the complete query execution time both in CPU cycles and last level cache misses (in our case L2 cache misses). For this benchmark we choose late materializing plan operators (e.g., joins and aggregates that compute final results by going back to underlying physical representation) so that the performance of all plan operators is directly affected by the physical representation. We tried other (early materialization) plans and found them to be slower for these queries. Of course, in some settings early materialization-based operators may perform better than late materialization, but in these cases the performance of the operators will be unaffected by our choice of storage layouts.

The results for all of the queries are shown in Figure 5. The table shows the absolute number of CPU cycles for each of the three designs. The graphs compare the normalized performance of each system in terms of the number of CPU cycles (left), actual number of L2 cache misses (middle), and number of L2 cache misses predicted by our model (right). Here "normalized" means that for a given query, the worst-performing system (in terms of CPU cycles or cache misses) was assigned a score of 1.0, and the other systems were assigned a score representing the fraction of cycles/misses relative to the worst-performing system. For example, on Q1, all-columns and HYRISE used about 16% of the cycles as all-rows.

There are several things to note from these results. First, in terms of actual cache misses and CPU cycles, HYRISE almost always does as well as or outperforms the best of all-rows and all-columns. For those queries where HYRISE does not outperform the other layouts, our designer determines it is preferable to sacrifice the performance of a few queries to improve overall workload performance.

The second observation is that cache misses are a good predictor of performance. In general, the differences in cache misses tend to be more pronounced than the differences in CPU times, but in all cases, the best performing query is the one with the fewest cache misses. Third, the model is a good predictor of the actual cache misses. Though there are absolute differences between the normalized and predicted cache misses, the relative orderings of the schemes are always the same. In general, the differences are caused by very hard to model differences, such as the gcc optimizer (which we ran at -O3), which can affect the number of cache misses.

In summary, HYRISE uses 4x less cycles than the all-row layout. HYRISE is about 1.6x faster than the all-column layout on OLTP queries (1–9), with essentially identical performance on the analytical queries (10–13). For some OLTP queries it can be up to 2.5x faster than the all-column layout. Of course, in a hybrid database system, the actual speedup depends on the mix of these queries – in practice that many OLTP queries will be run for every OLAP query, suggesting that our hybrid designs are highly preferable.

5.3 Data Morphing Layouts

In this section, we describe the differences between the behavior and performance of our layout algorithm and the Hill-Climb Data Morphing algorithm proposed by Hankins and Patel [12] (the paper proposes two algorithms; Hill-Climb is the optimized version.) We could not run Hill-Climb on our entire benchmark because (as noted in the Introduction) the algorithm scales exponentially in both time and space with the number of attributes (see Appendix D), and thus can only be used on relatively simple databases.

Instead, we ran a simplified version of our benchmark, focusing on the smallest relation (MATH) — the only one Hill-Climb could handle — and query 13 which runs over it. Here, Data Morphing suggests a complete vertical partitioning, which performs 60% worse in terms of cache misses and 16% worse in terms of CPU cycles compared to the layout used by HYRISE. The reason for this difference is mainly due to the lack of partial projections in the Data Morphing cost model. We would expect to see similar performance differences for other queries if Data Morphing could scale to them, since the Data Morphing model is missing several key concepts (e.g. partial projections, data alignment, and query plans—see Appendix D).

6. RELATED WORK

As mentioned in Section 5.3, the approach most related to

HYRISE is the Data Morphing approach of Hankins and Patel [12]. Data Morphing partitions relations into both row and column-oriented storage. The main differences between our approaches and Data Morphing are in the fidelity of our cache-miss model (we model many cases that Data Morphing is unable to capture), and in our physical database design algorithm. Taken together, these make HYRISE significantly faster than Data Morphing, and also allow it to scale to tables with tens or hundreds of attributes, whereas Data Morphing cannot scale to tables with large numbers of attributes.

Vertical partitioning is a widely used technique that has been explored since the early days of the database community [24, 13, 11, 17, 2]. Some of this work [9, 10, 17, 2] attempts to automatically derive good partitions, but does so with an eye towards minimizing disk seeks and I/O performance rather than main memory costs as we do in HYRISE. As such, these systems do not include careful models of cache misses. The work of Agrawal et al [2] is most similar to our approach in that it uses as cost-based mechanism to identify partitions that are likely to work well for a given workload.

Recently there has been a renewed interest in pure vertical partitioning into a "column-store", e.g., DSM [8], Monet and MonetDB/X100 [5, 6], C-Store [21]. As "pure" column systems, these approaches are quite different than HYRISE. The Monet system is perhaps most related because its authors develop complete models of cache performance in column stores.

There have been several attempts to build systems in the spirit of HYRISE that are row/column hybrids. PAX [3] is an early example; it stores data from multiple columns in a disk block, but uses a column-wise data representation for those columns. In comparison to the cache miss performance of HYRISE when scanning a narrow projection, PAX will incur somewhat more cache misses when scanning just a few columns from a table (since it will have to jump from one page to the next in memory). Similarly, in comparison to HYRISE scanning a wide projection, PAX will incur more cache misses when scanning many columns from a table (since it will have to jump from one column to the next in memory.)

We chose not to compare our work against PAX directly because the Data Morphing paper [12] showed that a hybrid system like HYRISE can be up to a factor of 2 faster for workloads that read just a few columns, and as we show in Section 5.3, HYRISE generally performs better than Data Morphing. Fractured Mirrors [19] and Schaffner et al. [20] are hybrid approaches that consider both row and column representations and answers queries from the representation that is best for a given query; this leads to good query performance but has substantial synchronization overhead. Unlike HYRISE, neither of these systems nor PAX vary their physical design based on the workload, and so do not focus on the automated design problem we address.

7. CONCLUSIONS

In this paper, we presented HYRISE, a main memory hybrid database system designed to maximize the cache performance of queries. HYRISE creates vertical partitions of tables of different widths, depending on the access patterns of queries over tables. To determine the best partitioning, we developed an accurate model of cache misses that is able to estimate the number of misses that a particular partitioning and access pattern will incur. We presented a database design algorithm based on this model that finds partitionings that minimize the number of cache misses for a given workload, and that is able to scale to tables with a large number of columns.

Our results show that HYRISE is able to produce designs that are 20% to 400% faster than either a pure-column or pure-row approach on a realistic benchmark derived from a widely used enterprise application. We also show that our approach leads to better physical designs and can scale to larger tables than Data Morphing [12], the previous state of the art workload-aware approach for partitioning main memory databases. As future work, we plan to examine horizontal partitioning as well as additional hybrid-based query optimizations, and to optimize HYRISE for future many-core processors with multiple memory channels and an increasing parallelism.

8. REFERENCES

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

[2] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD Conference*, pages 359–370, 2004.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65, 1999.

[6] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[7] S. K. Cha and C. Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In *VLDB*, pages 1033–1044, 2004.

[8] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD Conference*, pages 268–279, 1985.

[9] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Transactions on Software Engineering*, 16(2):248–258, 1990.

[10] P. De, J. S. Park, and H. Pirkul. An integrated model of record segmentation and access path selection for databases. *Information Systems*, 13(1):13–30, 1988.

[11] M. Hammer and B. Niamir. A Heuristic Approach to Attribute Partitioning. In *SIGMOD Conference*, pages 93–101, 1979.

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

[13] J. A. Hoffer and D. G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *VLDB*, pages 69–86, 1975.

[14] B. L. Johnston and F. Richman. *Numbers and Symmetry: An Introduction to Algebra*. CRC-Press, 1997.

[15] G. Karypis and V. Kumar. Multielvel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[16] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, pages 191–202, 2002.

[17] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

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

[19] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, pages 430–441, 2002.

[20] J. Schaffner, A. Bog, J. Krueger, and A. Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *BIRTE*, 2008.

[21] M. Stonebraker, D. J. Abadi, and A. B. et al. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[22] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[23] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[24] P. J. Titman. An Experimental Data Base System Using Binary Relations. In *IFIP Working Conference Data Base Management*, 1974.

[25] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.

APPENDIX

In these appendices, we provide several examples of the behavior of HYRISE's physical container design and describe several extensions that further improve HYRISE's performance on modern machines (Appendix A.) In Appendix B we give an example of HYRISE's layout selection. We also describe the details of the new benchmark we have developed for this paper (Appendix C.) In addition, we give a compact comparison to the Data Morphing cost model (Appendix D), detailed information on Write Operations (Appendix E) and Layout Generation (Appendix F.)

A. PHYSICAL DESIGN AND EXECUTION

Container Alignment Example: As described in Section 3, container alignment on cache boundaries can have a dramatic effect on the number of cache misses. For example, Figure 6 gives the number of cache misses for two containers and for partial projections retrieving 0 to 80 attributes from the containers. The first container is a 80-attribute wide container while the second container is a 86-attribute wide container (all attributes are 4 bytes wide). The first container has a width that is a multiple of the cache line size. The 86-attribute container is not aligned to the cache lines and suffers more cache misses for the partial projections, although the amount of data retrieved is the same in both cases. If this container were to be padded to 384 bytes (instead of using 344 bytes corresponding to the width of its 86 attributes) then both the 80 and the 86 wide containers would behave similarly in terms of cache misses. For this reason, properly aligning containers as done in HYRISE is essential.

Cache Set Collision: Cache collisions due to associativity conflicts can be problematic in cache-aware systems. For this reason, HYRISE automatically adjusts its container alignment policy in order to minimize these cache set collisions.

When the OS allocates a large memory region (for example when creating a container), it usually automatically aligns the beginning of the region with the beginning of a virtual memory page. Virtual memory pages have a fixed size—the address of their first byte always is a multiple of the system-level memory page size $PAGESIZE$ (which is a system constant that can be determined by calling `getconf PAGESIZE`).

The total number of cache sets $\#sets$ is equal to $L_i.n/assoc$, where $assoc$ is the associativity of the cache. Each memory address $address$ is mapped to a unique cache set set as follows:

$$\textit{set(address)} = \frac{address}{L_i.w} \mod \textit{\#sets}. \tag{21}$$

This mapping is cyclic and starts over every $\#sets * L_i.w$ bytes. When the memory page size is a multiple of this cycle length, i.e., when $PAGESIZE \mod (\#sets * L_i.w) = 0$, the addresses corresponding to the beginning of the containers are all systematically mapped to the same cache set, thus severely limiting the amount of cache available when processing several containers in parallel. This problem often occurs in practice (it occurs for our test system described in Section 3 for instance).

To alleviate this problem, we maintain a variable $\#containers$ counting the number of containers. When a new container is created, the system shifts the beginning of the container by $L_i.w * (\#containers \bmod \#sets)$ bytes, to maximize the utilization of the cache sets.

Figure 7 illustrates this point for our test system and 100 one-attribute wide containers. Each container is 4 bytes wide and the associativity of the L1 cache is 8 in this case. Without cache set collision optimization, the total number of cachable cache lines available when reading several containers in parallel is 8, since the containers are all aligned to the same cache set and share the same width. Cache evictions thus occur as soon as more than 8 attributes are read in parallel, significantly increasing the number of cache misses (see Figure 7). By offsetting the containers using the method described above, HYRISE is able to read all the containers in parallel without any early cache eviction (the system can read up to 512 containers in parallel in that case).

Cache set collisions often occur for the L1 cache. They occur less frequently for the L2 cache, which typically contains a much larger number of sets and has a higher associativity than the L1 cache.

Prefetcher Selection: In addition to allocating and aligning containers to minimize cache misses, HYRISE supports several cache prefetching policies that can be switched on a per-operator basis. Modern CPUs prefetch cache lines that the processor determines are likely to be accessed in the future. The advantage of this approach is that the data for a prefetched cache line starts to be loaded while the previous cache line is still being processed.

Most processors provide several prefetchers and allow applications to select which prefetcher they wish to use. For example, Intel processors based on the Intel Core architecture provide two different L2 hardware prefetchers. The first prefetcher is called Streamer and loads data or instructions from memory to the second-level cache in blocks of 128 bytes. The first access to one of the two cache lines in blocks of 128 bytes triggers the streamer to prefetch the pair of lines. The second is the Data Prefetch Logic (DPL) hardware prefetcher that prefetches data to the second level cache based on request patterns observed in L1.

DPL is able to detect more complicated access patterns, even when the program skips access to a certain number of cache lines; it is also able to disable prefetching in the presence of random accesses where prefetching may hurt performance.

To evaluate the performance impact of the different hardware prefetchers we created two layouts, λ_1, consisting of a single wide container of width w, and λ_2, consisting of a set of containers whose aggregate width was w. We accessed a list of random positions in each container, varying the selectivity from 0.0 to 1.0. For accesses to λ_1 there was no visible difference between the two prefetching implementations (Figure 8) but for accesses to λ_2, DPL used 24% fewer CPU cycles as it was able to predict skips between containers

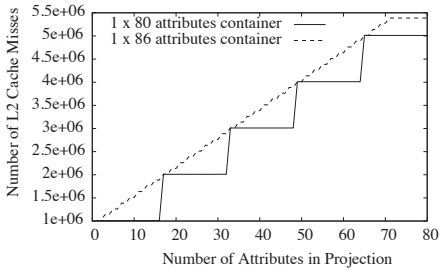


Figure 6: L2 Misses for Containers with Different Alignments

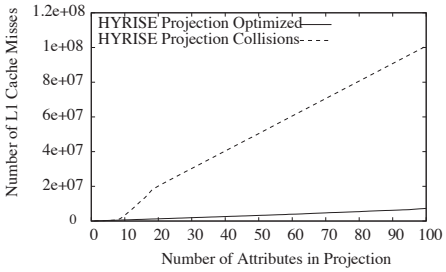


Figure 7: Experiment from Figure 3(a) with Cache Collision

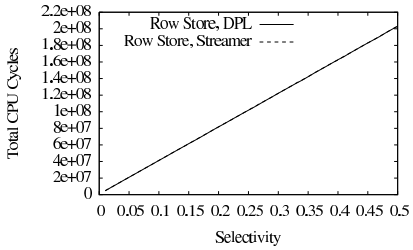


Figure 8: Comparison of DPL and Streamer for sequential access and row stores

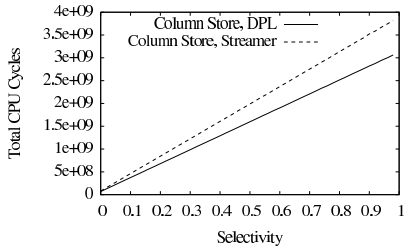


Figure 9: Comparison of DPL and Streamer for sequential access and column stores

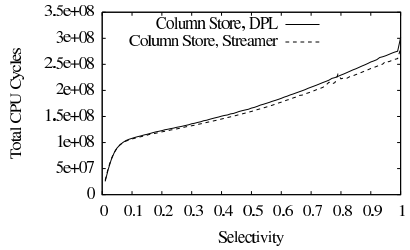


Figure 10: Comparison of DPL and Streamer for random access and column stores

(Figure 9). In a second experiment, we generated a list of sequential positions of varying selectivity and again accessed λ_1 and λ_2. In this case the streamer prefetcher was 7% faster than DPL when accessing λ_2 (Figure 10), with no difference when accessing λ_1. Based on these experiments, we use the DPL prefetcher by default in HYRISE. In cases where several small containers (like those in λ_2) are scanned with a high selectivity, we enable Streamer on a per-operator basis.

Database Loading: The output of the logical database designer is an annotated version of the original schema. When loading the data, we adopt several optimization techniques to improve the cache performance: First, all containers are allocated using a memory region that is aligned to the width of a cache line. Second, we perform *narrow row padding* to append padding to some of the containers in order to align their rows to the cache lines (see Section 3.6.) Finally, we shift the base pointers of the containers in order to minimize cache set collisions as described above in Appendix A.

Operator Implementation: The layouter component uses the *Partial Projection Model* to model materializing projections, predicate evaluation and the layout-dependent part of our hash join implementation. It uses the *Selection Model* to capture position-based selections. The layout-dependent costs of sorting and grouping operators are modeled by taking into account the data accesses (e.g., projections, selections of tuples for group-by and join columns) caused by those operations.

Layout-Independent Operations: Depending on the materialization strategy chosen for the given query plan not all costs will be layout-dependent. Although all queries of our benchmark only contain layout-dependent costs, for more complex scenarios with different materialization strategies layout independent operations may be needed. For example, there are layout-independent costs (e.g. index traversals) that would compete with the amount of cache used by the other operators; this behavior will need to be modeled, for example, using the notion of the repetitive random accesses presented in [16]. As future work, we are investigating building a cost based query optimizer on top of HYRISE that attempts to choose between layout dependent and layout independent operators.

B. LAYOUT SELECTION EXAMPLE

In this section, we give an example illustrating the candidate generation process described in Section 4.2. We consider a very simple workload on a relation containing N tuples with four attributes $a_1, \ldots, a_4$ and consisting of a projection $\pi_1 = \{a_1, a_2, a_4\}$ with weight w_1, a second projection $\pi_2 = \{a_2, a_3, a_4\}$ with weight w_2, and a selection σ_1 retrieving all attributes of a single tuple with weight w_3. The three resulting primary partitions are $P_1^1 = \{a_1\}$, $P_2^1 = \{a_2, a_4\}$, and $P_3^1 = \{a_3\}$, since π_1 creates partitions $\{a_1, a_2, a_4\}$ and $\{a_3\}$ and π_2 splits the first partition into $\{a_1\}$ and $\{a_2, a_4\}$ (σ_1 does not split the partitions any further). The costs associated with these partitions are:

$$Cost_{P_1^1}(W) = w_1 \times Cost(\pi(P_1^1)) + f_3 \times Cost(\pi(P_1^1)_{sel=1/N}) \quad (22)$$

$$Cost_{P_2^1}(W) = w_2 \times Cost(\pi(P_2^1)) + f_3 \times Cost(\pi(P_2^1)_{sel=1/N}) \quad (23)$$

$$Cost_{P_3^1}(W) = w_3 \times Cost(\pi(P_3^1)_{sel=1/N}) \quad (24)$$

Here $Cost(\pi(P_1^1)_{sel=1/N})$ reflects the cost of accessing one tuple for the selection.

In this example, merging P_1^1 and P_2^1 would be advantageous for the selection (since it touches all attributes), but introduces some overhead when performing projections (which never access a_1, a_2 and a_3 simultaneously). The exact overhead depends on the width of the attributes, the cache line size, and the frequency of the operations (as captured by our cost model in Section 3.)

C. BENCHMARK DESCRIPTION

Most database benchmarks are built to address a specific market (e.g., OLAP or OLTP). For example, TPC-C is an OLTP benchmark and TPC-H is an OLAP benchmark. Creating a hybrid of TPC-C and TPC-H is difficult since each benchmark uses different schemas and workloads. Consequently, we chose to create a new workload derived from a real database application. As a starting point we used the sales application of an Enterprise Resource Planning System, which covers a full range of operations in a sales scenario.

The business entities involved in sales are modeled as a large number of relations. This is both due to the application's use of highly normalized OLTP schemas and a result of so-called *header-items*. Header-items cause the sales order entity to be partitioned into a sales order header table and a sales line item table. The header contains data relevant to the entire sales order. For example, its description, order date, and sold-to-party are stored there. Attributes of the ordered material, number and price are kept in the line item table, with each row representing one item and belonging to one order. In general, a single sales order consists of several line items. Master data tables do not follow this pattern and store data in single tables for each type. For example, in the sales and distribution scenario, material and customer detail tables are both stored. The customer details table contains customer attributes, including name, account type, contact, and billing data. Specifics about a material, such as its description, volume, weight and sales-related data are kept in the material details table and the material hierarchy. In contrast to the tables used by TPC-E or TPC-C, the tables we consider are modeled after a real enterprise system and are much wider. The widest tables are the sales order line items table with 214 attributes and the material details table with 204 attributes. The other tables have between 26 and 165 attributes (e.g. `KNA1`).

Due to the complexity of these schemas, our benchmark queries

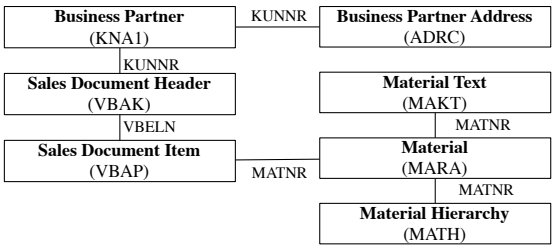


Figure 11: Simple Sales Schema

focus on sales order processing, leaving out operations for delivery of ordered items and billing of delivered items. Figure 11 illustrates the schema of our benchmark.

C.1 Benchmark Queries

Our benchmark includes queries that perform small transactional operations—including writes—as well as more complex, read-mostly aggregates on larger sets of data.

These queries composing our workload are as follows:

Q1 Search for a customer by first or last name (ADRC)
```
select ADDRNUMBER, NAME_CO, NAME1, NAME2, KUNNR
from ADRC where NAME1 like (..)
OR NAME2 like (..);
```

Q2 Read the details for this customer (KNA1)
```
select * from KNA1 where KUNNR = (...);
```

Q3 Read all addresses belonging to this customer (ADRC)
```
select * from ADRC where KUNNR = (...);
```

Q4 Search for a material by its text in the material text table (MAKT)
```
select MATNR, MAKTX from MAKT where
MAKTX like (..);
```

Q5 Read all details for the selected material from the material table (MARA)
```
select * from MARA where MATNR = (...);
```

Q6.a Insert a new row into the sales order header table (VBAK)
```
insert into VBAK (..)  values (..);
```

Q6.b Insert a new row into the sales order line item table based on the results of query Q5 (VBAP)
```
insert into VBAP (...)  values (...);
```

Q7 Display the created sales order header (VBAK)
```
select * from VBAK where VBELN = (..);
```

Q8 Display the created sales order line items (VBAP)
```
select * from VBAP where VBELN = (..);
```

Q9 Show the last 30 created sales order headers (VBAK)
```
select * from VBAK order by VBELN desc limit
30;
```

Q10 Show the turnover for customer KUNNR during the last 30 days
```
select sum(item.NETWR), header.KUNNR from
VBAK as header, VBAP as item where
header.VBELN = item.VBELN and
header.KUNNR = $1 and header.AEDAT >= $2;
```

Q11 Show the number of sold units of material MATNR for the next 10 days on a per day basis
```
select AEDAT, sum(KWMENG) from VBAP where
MATNR = $1 and AEDAT = (..)  group by AEDAT;
```

Q12 Show the number of sold units of material MATNR for the next 180 days on a per day basis
```
select AEDAT, sum(KWMENG) from VBAP where
MATNR = $1 and AEDAT = (..)  group by AEDAT;
```

Q13 Drill down through the material hierarchy starting on the highest level using an internal hierarchy on the table, each drill-down step reduces the selectivity, starting from 40% selectivity going down to 2.5% selectivity.

Queries Q1...Q9 can be categorized as typical OLTP queries, while queries Q10, Q11, Q12 and Q13 can be categorized as OLAP-style queries. Q1...Q6 are frequent, inexpensive OLTP queries and have a weight w_i of 100 in our benchmark. The other queries all have a weight of 1.

# Attributes	HYRISE	Data Morphing
2	2	1
3	1	1
5	1	7
7	1	38
9	1	43
11	1	872
13	1	15526

Table 1: Scaling Comparison: time (in [ms]) to derive layouts for both the optimal HYRISE algorithm and the Data Morphing algorithm for a table of varying width and a single query.

D. DATA MORPHING

Even on a simple workload consisting of only one query, the HillClimb algorithm suggested in [12] is not able to scale with the number of attributes. Table 1 shows an example where a sample relation is gradually extended one attribute at at time and where a simple query projecting on the first two attributes is run for each configuration. As can be seen on the figure, the cost of running HillClimb becomes rapidly prohibitive, even on this very simple setting.

Besides this scalability issue, the Data Morphing cost model is missing several key concepts that are useful in our context:

Partial Projections: Data Morphing only considers full-scan operations on the containers. While the Data Morphing model works fine for simple, non-overlapping queries and small, aligned containers, it can lead to arbitrarily bad results in more realistic cases. Consider, for example, performing a projection of the first 4–64 bytes of an aligned 256-byte wide container. Here, the Data Morphing cost model will over-estimate the number of misses by a factor of 3 versus the true value (correctly predicted by the HYRISE model) since only one cache line out of the four required by each row will be read by the scan. This causes Data Morphing to severely penalize wide containers in its designs. We confirmed that real designs produced by Data Morphing can show these types of slowdowns vs HYRISE.

Data Alignment: Data Morphing does not consider data alignment. This can lead to very bad cache estimations in many cases, e.g., for containers that are not aligned to the cache ($C.o \neq 0$, in terminology of Section 3), for partial projections (in addition to the case above), or for selections. For instance, the cost for full but relatively selective projections (e.g., $\pi.s < 10\%$) on a 60 byte container is approximated as 1 cache miss per result by Data Morphing, but is actually equal to 1.8 misses per result due to misalignment.

Query Plans: In addition to the points above, the data morphing approach does not include any notion of query plans. This is especially bad for complex queries which access the same container several times. Such repeated accesses are treated as only one full scan by Data Morphing.

The HYRISE cost model and layout algorithm take those important cases into account, leading to superior performance. Furthermore, the grouping and pruning algorithms in Section 4 allow us to scale to tables with hundreds of columns (such as those described in our full benchmark above), which Data Morphing is unable to do.

E. WRITING AND TRANSACTIONS

In this section we briefly describe the effect that transactions and updates have on cache performance in HYRISE. As noted in Section 2.1, in HYRISE, we use validity (begin and end) timestamps for all tuples, as a result we only append new values to existing tables. This design means that all updates are actually two writes, one to the end of the table and one to the validity timestamp of the previous version, although the previous version's timestamp is very unlikely to be read again. Furthermore, transactional operations like logging and updates read and write sets of data (when using multi-

Description	CPU Cycles	L2 Cache Misses
No Writes	1,105,317	5
Non-temporal Writes	29,902,648	11,683
Normal Writes	40,289,157	557,346

Table 2: Comparing temporal and non-temporal writes

version concurrency control) that are not re-accessed frequently and are private to a single transaction. These writes can result in cache pollution with data that is not likely to be read.

To avoid this pollution, we use non-temporal writes provided by the Intel SIMD Streaming Extensions in HYRISE. When using non-temporal writes the write-combining buffer of the CPU will capture all writes to a single cache line and then write back the cache line directly to memory without storing the modified data in the cache.

To measure the benefit of non-temporal writes, we allocated a single-column container with the size of the L2 cache (6 MB). After scanning the container and reading all values we then measured the number of cache misses performed by an additional scan over the data (which by itself should not produce any L2 cache misses since the data is already cached and the container fits into cache). Concurrently, we write data to a second container; we alternate between using the SSE extension for non-temporal writes and using plain `memcpy()`. We use the `_mm_stream_si128()` function to generate the required `MOVNTDQ` assembler instruction, which causes the write combined buffer of the CPU to collect all modifications for a cache-lines worth of data and write it back to main memory without storing the data in any cache. This operation models the process of accumulating a read set during the execution of a transaction.

Table 2 shows the results of the experiment. The results show that non-temporal writes use only 75% of the CPU cycles and 0.02% of the cache misses when compared to `memcpy()`, suggesting that this optimization significantly improves the performance of writes that are not frequently read.

F. HYRISE LAYOUT SELECTION

In this section we illustrate the influence of workload variations on the layouts produced by HYRISE.

Experiment 1: In this experiment, instead of simply running our layouter to find a layout for a given workload we use a workload for which one of two layouts is optimal, given a mix of queries run with different frequencies (or *weights*).

We use a table with 10 attributes ($a_1 \ldots a_{10}$) and a total width of 56 bytes. The workload consists of two queries: one OLTP-style query that scans attribute a_1 and, for the rows that match a highly selective predicate, returns all attributes; and two, an OLAP query that scans attribute a_1, applying a non-selective predicate, and then does a GROUP BY on the matching rows of attribute a_1 and aggregates over the values of attribute a_4.

From those two queries one of two layouts is optimal. The first, λ_1, separates attribute (a_1) and group ($a_2 \ldots a_{10}$) together; the second layout, λ_2, also separates (a_1), but in addition splits the remaining group into (a_4) and ($a_2 \ldots a_3; a_5 \ldots a_{10}$). Given both layouts we want to observe when the layout algorithm chooses to switch from one design to the other. Assuming that OLTP queries occur more frequently than OLAP queries, we wish to visualize when each layout is more appropriate. We vary OLTP selectivities from $0 < \sigma_{OLTP} < 0.5$ and OLAP selectivities from $0.5 < \sigma_{OLAP} < 1$. We then compute the cost of each query and determine the point at which the layouts have equal cost.

$$x \times Cost_{OLTP}(\lambda_1, \sigma_1) + Cost_{OLAP}(\lambda_1, \sigma_2) = x \times Cost_{OLTP}(\lambda_2, \sigma_1) + Cost_{OLAP}(\lambda_2, \sigma_2) \quad (25)$$

Equation 25 shows the formula used to calculate the cost for layout λ_1 and λ_2 based on the distinct selectivities σ_1 and σ_2. Figure 12 shows the result of this experiment. The contour lines define the region where λ_1's cost exceeds that of λ_2 for different values of σ_1 and σ_2. For example, when σ_1 is .005 and σ_2 is .55, if there are more than 100 OLTP queries per OLAP query, λ_1 is preferable (in fact, for any $\sigma_1 > .01$, λ_1 is preferable when there are at least 100 OTLP queries per OLAP query).

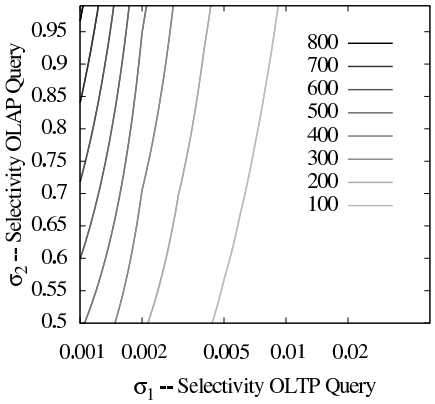


Figure 12: Contour plot showing the number of OLTP queries per OLAP query required before workload λ_1's cost exceeds that of λ_2 for different values of σ_1 and σ_2.

Experiment 2: A related question is how many partitions are typically generated by our algorithm. In the simplest case, for a table with n attributes and i independent queries that access the table on disjoint attributes with the same weight and selectivity, the table will be split into i partitions.

For more complex cases, our algorithm will only generate a partitioning that optimizes the *total workload cost*, and won't create a separate storage container for infrequently run queries. In most enterprise applications, there are a small number of query classes (and hence attribute groups) that are accessed frequently; ad-hoc queries may be run but they will likely represent a very small fraction of total access. Hence, these more frequent (highly weighted) queries will dominate the cost and be more heavily represented in the storage layout, resulting in a modest number of storage containers for most workloads (i.e., it is unlikely that a full column-oriented layout will be optimal for non-OLAP workloads.)

Figure 13 shows the result of an experiment where we used our approximate graph-partitioning algorithm to determine the best layout for a wide table of 500 4-byte attributes and an increasing number of frequently-posed queries. The experiment starts with a relatively expensive OLTP query (selecting all attributes with a selectivity of 40%). We then iteratively add three random OLAP queries that each project a random pair of attributes, until we have a total of 1000 random queries. As expected, the number of partitions in slowly converges to an "all-column" layout. The time taken to compute the layout using our approximate partitioning algorithm and for $K = 10$ varies from a few hundred milliseconds initially to a few minutes for the case with 1000 OLAP queries.

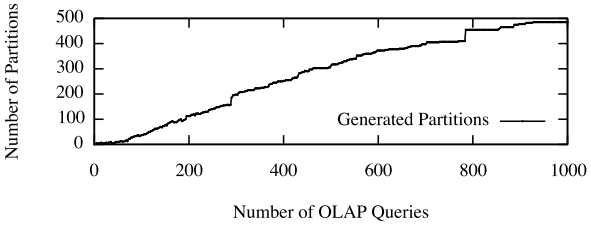


Figure 13: Increasing number of OLAP queries