A Demonstration of HYRISE— A Main Memory Hybrid Storage Engine

Martin Grund
Hasso-Plattner-Institute
Germany
martin.grund@hpi.uni-potsdam.de

Philippe Cudre-Mauroux
MIT CSAIL, USA &
U. of Fribourg, Switzerland
pcm@unifr.ch

Samuel Madden
MIT CSAIL
USA
madden@csail.mit.edu

ABSTRACT

We propose to demonstrate HYRISE, a main memory hybrid database system, which automatically partitions tables into vertical partitions consisting of variable numbers of columns based on access patterns to each table. Using an accurate model of cache misses, HYRISE is able to predict the performance of different partitionings, and to automatically select the best partitions using an automated database partitioning algorithm. Our demonstration will show the results of the physical partitioning based on different query workloads, allowing demo attendees to visualize, fine-tune, and modify the partitioning using a GUI. It will then show how the various physical designs affect the query plans and the performance of the database as a whole. Attendees can thus experiment with various physical models, and can grasp the potential of hybrid partitionings, which achieve a 20% to 400% performance improvement over pure all-column or all-row designs on our realistic hybrid workload derived from customer applications.

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 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). 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 [3, 4]. In addition, HYRISE stores columns that are accessed in OLTP-style queries in wider partitions, to reduce cache misses when performing single row retrievals.

A paper about HYRISE will appear in this VLDB 2011 [5], along with this demo. In this paper, we describe the architecture of HYRISE and some of its key features, and provide a description of the demonstration we plan to show.

2. HYRISE ARCHITECTURE

The main architectural components of HYRISE are depicted 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 decides on how to partition the data. The layout manager can make decisions based on two methods: i) it can analyze a sample query workload and automatically suggest a partitioning minimizing the expected cost of the workload (see below Section 2.3) or ii) it can suggest a partitioning and let the database administrator manipulate and finalize the partitioning through a visual interface.

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 allow to directly write back to main memory without loading

Proceedings of the VLDB Endowment, Vol. 4, No. 12

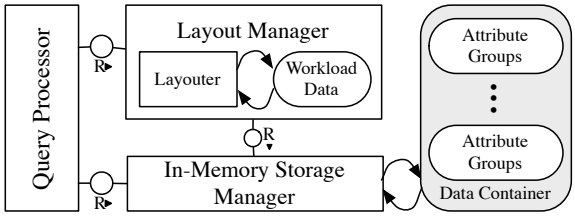

Figure 1: The HYRISE architecture.

the written content into the CPU cache. 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 the different HYRISE components below, before describing the contents of our demonstration.

2.1 Storage Manager

HYRISE 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*. Containers are physically stored as a list of large, contiguous and compressed blocks of data. Data types are dictionary-compressed into fixed-length fields to allow direct access (offsetting) to any given position.

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.

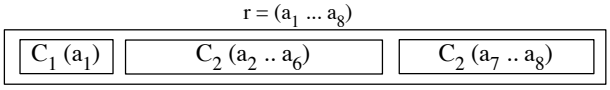

Figure 2: An example of physical partitioning where a table containing eight attributes is partitioned into three containers, containing one, five, and two attributes respectively.

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 the following operators: projection, selection, sort, and group by. 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 to create the output tuples. For join plans, predicates are first applied on the dimension tables. Then,

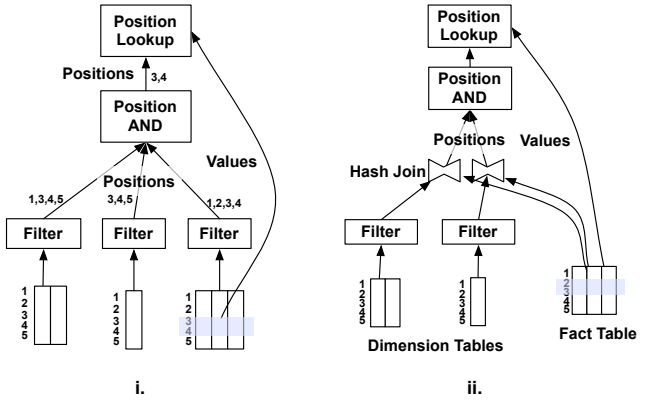

Figure 3: Two examples of query plans illustrating i) non-join queries and ii) join queries.

foreign-key hash-joins are used to build position lists from the fact tables. Additional predicates can 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. Figure 3 shows two sample query plans.

2.3 Layouter

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. There exists for example 73 different hybrid layouts for a table of 5 attributes, 4,596,553 layouts for a table of 10 attributes, or 3,535,017,524,403 possible layouts for a table of 15 attributes. HYRISE is to the best of our knowledge the first hybrid database system offering a fully automated designer capable of determining the most appropriate physical design for tables of many tens or hundreds of attributes given a database and a query workload.

Our layouter is based on a very accurate cost-model, used to compare the performance of various hybrid layouts given a query workload. Most of the query execution time spent in a main-memory system like HYRISE originates from CPU stalls—typically caused by cache misses when moving data from main memory into CPU registers; those CPU stalls are known to account for a significant fraction of the total cost of a query (see [2]).

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), the alignment of the data with regards to the cache lines, the various prefetching policies available, and the cache contention. Our model captures all important operations available in HYRISE including:

1. projections
2. selections
3. joins
4. aggregations
5. combinations of intermediate results and
6. result reconstructions.

The details of our cost-model are available in [5].

The role of the layouter is to determine good layouts that will minimize query response time. More formally, given a database DB and a workload W, the layouter determines one (or several)

physical layout $\boldsymbol{\lambda_{opt}}$ minimizing the workload cost according to our cost model:

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

The layouter generates the best possible layouts in three phases. The first phase, called *candidate generation*, generates the largest partitions presenting no overhead according to the given workload (that is, the partitions whose attributes are always co-accessed together). The second phase, *candidate merging*, generates new partitions by iteratively merging and pruning the candidates returned by the first phase. The third and final phase, called *layout construction*, creates all valid layouts by combining the candidates of the second phase, and returns the best layout according to our cost-model.

Scalable Partitioning. The worst-case space complexity of our layout generation algorithm is exponential with the number of candidate partitions. 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* during the first phase described above) and since operations across those partitions are often relatively infrequent (thus, drastically limiting the number of new partitions generated by the second phase above).

For large relations and complex workloads involving hundreds of different frequently-posed queries, the running time of the above algorithm may still be high. Thus, HYRISE supports a second, approximate but very scalable partitioning algorithm. This second algorithms starts like the first one above by generating candidate partitions. In addition, it computes the *affinity* between the candidate partitions, namely the frequency with which pairs of partitions are accessed together.

Our scalable layouter partitions this graph in order to obtain a series of *min-cut* subgraphs each containing at most K primary partitions (where K is a system parameter). HYRISE uses an existing approximate multilevel k-way graph partitioner [6] in this context. At this point, each subgraph contains a set of candidate partitions that are often accessed together, and which thus represent excellent candidates for our merging phase (see above). We determine the optimal layout of each subgraph separately, 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 in order to yield the most savings according to our cost model, until no further cost-reduction is possible. Figure 4 gives a graphical illustration of the process. This approximate algorithm is very effective in practice, as will be shown in the demonstration and as explained in [5].

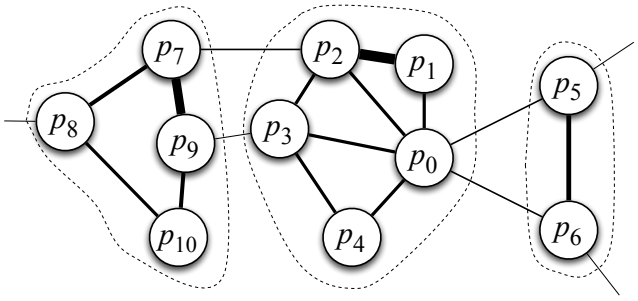

Figure 4: An example of scalable partitioning: a graph of eleven candidate partitions (nodes) with their respective affinities (edges); the graph is partitioned to obtain a series of *min-cut* subgraphs, each containing at most K partitions; our algorithm then determines the optimal layout for each subgraph, and finally combines all subgraphs.

3. DEMONSTRATION

Our demonstration will allow conference attendees to issue queries dynamically, view the query plans (Figure 3), results, and the time to execute the various queries. In addition, it will provide a visualization of the partitioning (Figure 2) and will allow attendees to experiment with various physical designs (e.g., columns, rows, various hybrid layouts) and observe the influence of the various physical designs on query execution and on the performance of HYRISE. We start below by detailing the data and queries we will use, before giving a few details about the user interface we will build for this demonstration.

3.1 Schema & Data

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 lightweight reporting over the recent history of these transactions. We chose to use our own application-derived workload in this demonstration 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.

Furthermore, we wanted to execute both OLTP-style and analytical-style queries on the same data. It is not easy to retrofit an existing analytical benchmark like TPC-H to support transactional queries, since the business model in TPC-C and TPC-H are quite different, and TPC-H uses a star-schema design whereas TPC-C uses a more conventional normalized schema. Even though a vertical partitioning of attributes is suitable for single-style workloads, we see the biggest performance gains for hybrid layouts for heterogeneous applications, due to the fact that neither row-wise nor column-wise storage would be optimal for such applications.

The business entities involved in our ERP scenario – following the sales and distribution processes – 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

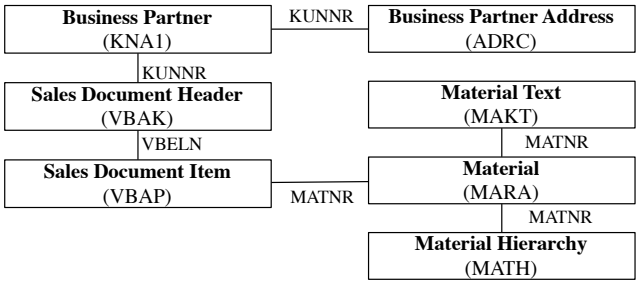

Figure 5: The sales schema used for the demo.

the material details table with 204 attributes. The other tables have between 26 and 165 attributes (e.g. KNA1). Figure 5 illustrates the schema of our benchmark.

3.2 Benchmark Queries

Our demonstration suggests a set of queries covering all standard operations of our ERP scenario, from small transactional operations—including writes—to more complex, read-mostly aggregates on larger sets of data. In addition, conference attendees will be able to write their own queries manually.

The default queries for our demonstration will be 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.

3.3 User Interaction

Users are given two main ways of interacting with our system during the demonstration:

1. The workload panel lets the user select one or several of our benchmark queries (see above) in order to create a workload. The user can add weights to the queries to simulate the fact that some queries are run much more often than others, and can also manually enter a query of his/her choosing if he/she prefers to. The user can also select one or several queries to be executed; the panel then displays the query execution plan the system picked to execute the query, the query results, and the time taken to execute the query. The system gives a number of additional details on the execution, including the number of containers touched, the number of tuples read, the number of cache lines loaded for each cache level, and the CPU time taken to execute each operator in the query execution plan.

2. The layouter panel shows the candidate partitions and the top-5 different partitionings that are automatically generated by our system based on the current data and the query workload selected by the user. In addition, the user is able to directly influence the physical layout of the database: he/she can select arbitrary attributes and move them from one partition to the other, or create entirely new partitions. The system dynamically computes the expected cost of running the current workload on the physical layout chosen by the user; It compares the user's layout to the top-5 partitionings determined automatically, and also to an all-row and an all-column layouts. Finally, the user can ask the system to rewrite the database according to the layout he/she picks (the rewriting takes a few seconds), and can then switch back to the workload panel to execute some queries on the new physical layout he/she created.

Using both the workload and the layouter panel, the user can thus get familiar with the concept of hybrid layouts and can experiment in real time with various query workloads and partitionings, and observe their impact on the overall performance of the system.

4. 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] 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.

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

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

[5] M. Grund, J. Krger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise - a main memory hybrid storage engine. PVLDB, pages 105–116, 2010.

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