Optimal Query Operator Materialization Strategy for Hybrid Databases

Martin Grund, Jens Krueger, Matthias Kleine, Alexander Zeier, Hasso Plattner
Hasso-Plattner-Institut
August-Bebel-Str. 88
14482 Potsdam, German
{*martin.grund, jens.krueger, matthias.kleine, alexander.zeier, hasso.plattner*}*@hpi.uni-potsdam.de*

***Abstract*—Recent research shows that main memory database system provide many different advantages over traditional disk based systems. Furthermore it is shown that the way how data is persisted in such a system is very important. Modern systems provide a hybrid row- and column-oriented storage layer that proves to be optimal for certain workloads. To further optimize the query execution it becomes to crucial to select the best possible query operators. However, not only the implementation of the operator is very important but as well the way how intermediate results are handled. In *HYRISE*, we implemented different possibilities of query operator materialization and show in this paper when to chose which kind of output. The results of our experiments can be directly used during plan creation by a cost-based query executor.**

***Keywords*-Hybrid Main Memory Database; Query Execution; Column Store; Materialization.**

I. Introduction

Main memory database systems have proven to be advantageous for various scenarios ranging from high-performance analytical data warehouse accelerators to classical Online Transactional Processing (OLTP) databases. Due to the available size of main memory on a single rack server of currently 1TB almost all enterprise like applications with a mixed transactional / analytical focus can be run on such systems. Another great advantage of in-memory data processing is that data access operations are more predictable compared to disk access based operations.

Since data is no longer stored in secondary structures like the buffer pool of traditional disk based databases but operations are directly executed on the primary data it becomes important to deal with the question on how to efficiently handle intermediate results and execute any given query in the best possible way.

For our research the main focus is query execution in a hybrid main memory database system such as *HYRISE* [1]. In *HYRISE*, relational tables are partitioned into disjoint vertical partitions. Data is stored dictionary compressed and single columns can be furthermore bit-compressed to achieve higher compression ratios. In such an environment it becomes crucial to choose the right materialization strategy to lower the amount of copied data but on the other hand improve the cache miss patterns of different queries during the query plan execution.

The authors of [2] identify different materialization strategies for column-oriented DBMS and explore the trade-offs that exist between them. This paper builds on these ideas and enhances their model for hybrid main memory databases and furthermore empirically evaluates variations of them using *HYRISE*, a hybrid main memory based DBMS research prototype. Section V gives a brief summary of related work, Section II summarizes the materialization strategies presented in [2], Section III describes their adaption within *HYRISE*, and Section IV evaluates the performance of the implementation, followed by a conclusion in Section VI.

II. Existing Materialization Strategies

This section gives a brief summary of the materialization strategies for column-based DBMS that are identified in [2]. Compared to our extensions, their work primarily covers column-stores, while it is important for *HYRISE* to support different kinds of materialization strategies for hybrid databases. The authors recognize two different aspects of materialization strategies, time of materialization, i. e., late vs. early materialization, and parallel vs. pipelined materialization, whose influence on execution plans in explained using the following example SQL query:

```
SELECT col1 , col2 FROM table
WHERE col1 < CONST1 AND col2 < CONST2
```

Abadi et al. [2] make use of specialized plan operators that are explained briefly in the following plan descriptions. The query plans are illustrated using diagrams, as the one shown in Figure 1, where *type of output* is either *mat*, *pos*, or *mat+pos*, indicating output of materialized data, positions, or materialized data and positions. Inputs that are filtered by a predicate are <u>underlined</u>.

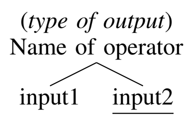


Figure 1. Example Query Plan Diagram

Table I gives a detailed list of the basic operators used during the query plans. In the following paragraphs we will present the different materialization strategies that are implemented in *HYRISE*.

Table I
DIFFERENT MATERIALIZING QUERY PLAN OPERATORS

DS1	Reads all data for a given column, applying a given selectivity. The output is a list of positions.
DS2	Reads all data for a given column, applying a given selectivity. The output is a list of position value pairs.
DS3	The data of a column is read and filtered with a list of positions. The output is a column of values corresponding to those positions.
DS4	A column is read and a list of positions is applied as a filter, tuples satisfying a predicate are selected, producing a list of positions.

Early Materialization / Pipelined: As illustrated in Figure 2, DS2 scans col1, filtering by predicate, outputting positions and data. DS4 index-scans col2 using the positions of the first scan, filtering by predicate, outputting the merged data.

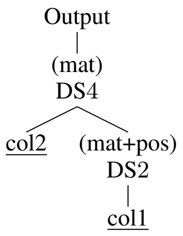


Figure 2. Plan for Early Materialization / Pipelined

Early Materialization / Parallel: As illustrated in Figure 3, SPC scans both columns in parallel, filtering by both predicates simultaneously.

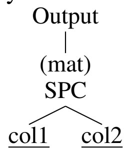


Figure 3. Plan for Early Materialization / Parallel

Late Materialization / Pipelined: As illustrated in Figure 4, DS1 scans col1, filtering by predicate, outputting positions only. DS3 and DS1 index-scan col2 using these positions, filtering by predicate, outputting positions only. Both columns are index-scanned using these positions and the results are merged.

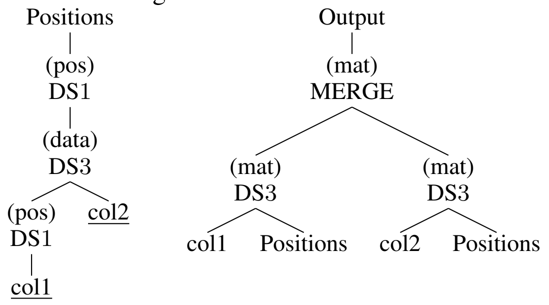


Figure 4. Plan for Late Materialization / Pipelined

Late Materialization / Parallel: As illustrated in Figure 5, each column is scanned with DS1 outputting positions. The positions are combined with an AND. These merged positions are processed as in the previous plan, i. e., both columns are index-scanned and the results are merged.

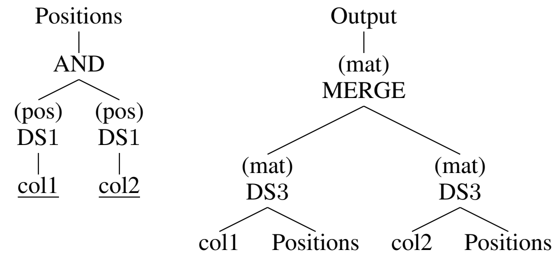


Figure 5. Plan for Late Materialization / Parallel

III. ADOPTION IN *HYRISE*

This section describes how the example query plans were adapted in *HYRISE*.

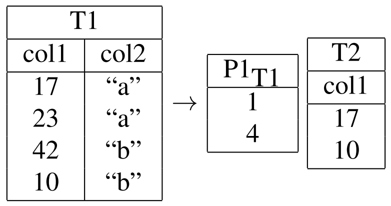
A. Plan Operators

The implementation makes use of new plan operators that are extensions of existing *HYRISE* plan operators. They do not directly correspond to the plan operators introduced in Section II but are an adaption of them to *HYRISE*.

TableScan: The `TableScan` provided by *HYRISE* scans all columns of a table in parallel, applies predicates, and outputs materialized data. Input can be either a list of positions or materialized data.

PositionTableScan: The `PositionTableScan` scans all columns of a table in parallel, applies predicates, and outputs positions. It corresponds to the DS1 operator.

MaterializingScan: This new plan operator is based on the DS2-operator of [2]. It accepts a raw table or a table with associated positions as input. In addition, it accepts predicates. As DS2, it produces materialized data and positions. Being a projection scan, it produces a new table that contains a configurable subset of the columns of the original table. It additionally outputs a list of positions that indexes the original table.



T1	
col1	col2
17	"a"
23	"a"
42	"b"
10	"b"

$\rightarrow$

$P1_{T1}$
1
4

T2
col1
17
10

Figure 6. MaterializingScan on an example table T1 with predicate col1 $<$ 20 produces positions $P1_{T1}$ and materialized result T2

Figure 6 shows an example application of this plan operator. A `MaterializingScan` is applied to table T1. No extra positions are given as input. The predicate supplied is col1 $<$ 20 and col1 is the only column to be projected. The result is a new position list as well as a new table with the filtered column col1.

TableScanUsingExistingData: This new plan operator is based on the DS4-operator of [2]. It is designed to work on the output of a `MaterializingScan`. As input it takes a table, a list of positions into this table as well as a materialized version of some columns of the table at these positions. It also accepts predicates. It index-scans the input table and outputs a materialized table. The materialized columns that are input into this operator are not directly used as output. Instead only the rows where the predicates match are copied into the output. Thus, the operator produces a table with the same layout as the first input table.

T1	
col1	col2
17	"a"
23	"a"
42	"b"
10	"b"

$P1_{T1}$
1
4

T2
col1
17
10

$\rightarrow$

T3	
col1	col2
10	"b"

Figure 7. `TableScanUsingExistingData` on the results of the last example, with the predicate col2 = "b"

Figure 7 shows the application of an `TableScanUsingExistingData` to the results of the example shown in Figure 6, using the predicate col2 = "b". T1 is scanned using the position of $P1_{T1}$. For all rows where col2 matches the predicate, the data from col2 of T1 and col1 of T2 is added to the result.

B. Query Plans

The query plans introduced in [2] have been adapted for *HYRISE*. As they are not exactly the same plans, they are given different names to avoid confusion.

Plan 1 - One Scan: This plan corresponds to the early materialization / parallel scan. As illustrated in Figure 8, it consists of only one plan operator that accesses both input columns simultaneously, i. e., for each row both columns are read and written to the output if both predicates match.

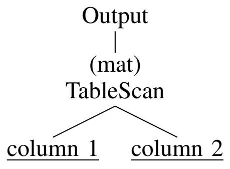


Figure 8. Plan 1 - One Scan

Given a column layout, this plan is expected to be efficient if the number of values that have to be accessed is high for both columns. It is expected to be relatively inefficient if the selectivity is low on column 1 as in that case both columns will still always be read. That is not the case for the other plans.

Given a row layout, this plan is expected to be efficient for low to high selectivities as long as the columns are adjacent or not too wide, as then reading both columns simultaneously causes less cache misses than reading them sequentially. For very low selectivities, the position based scans might still be more efficient, as then the number of total cache accesses is expected to be much lower for them than for the *One Scan* plan, even if the number of cache misses is expected to be slightly higher.

Plan 2 - No Data: This plan has no direct correspondence to the original execution plans, but it is an optimized version of the late materialization / parallel plan. As illustrated in Figure 9, it consists of two plan operators. A `PositionTableScan` on column 1, which produces only positions for the rows where the predicate on column 1 matches. This operator is followed by a `TableScan`, which filters column 2 by its predicate at these positions and materializes both columns.

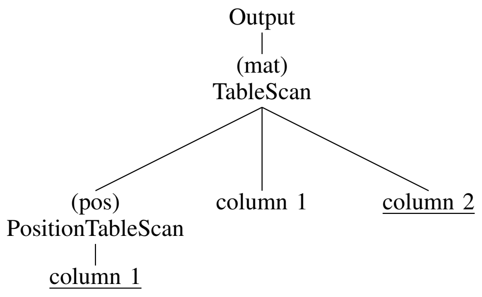


Figure 9. Plan 2 - No Data

Given a column layout, this plan is expected to be efficient when selectivity is low on column 1. Then, unlike in the *One Scan* plan, not many of the second column's rows have to be accessed. The plan is expected to be less efficient than the *One Scan* plan if the selectivity is high on column 1 and low on column 2, as then both columns have to be read nearly completely in both plans, but this plan generates a large amount of unused temporary position data.

Given a row layout, this plan is expected to be worse than the *One Scan* plan for most of the selectivities, especially if the columns are adjacent or are not too wide, so that one row of both columns fits into one cache line. For very low column 1 selectivities though, this plan is expected to require roughly half of *One Scan*'s cache accesses with only slightly higher cache misses, and might thus be faster.

Plan 3 - Data: This plan is closest to the Early materialization / pipelined plan. As illustrated in Figure 10, the `MaterializingScan` produces positions as well as values for column 1 where the predicate matches. Unlike the DS2 operator it produces these as two separate columns, not one column containing (position, data) pairs, the reason being a *HYRISE* implementation detail. The `TableScanUsingExistingData` scans column 2 at these positions. The rows where the second predicate matches are output materialized.

Given a column layout, this plan is expected to be more efficient than the *No Data* plan for very low selectivities on column 1. Then, the materialization of the final result, i. e., the SimpleTableScan in the *No Data* or the SimpleTableScanUsingExistingData in this plan, has to access very few rows of column 1. Doing so by accessing the original column by index is more expensive than sequentially accessing prematerialized data. It is expected to be inefficient if selectivity

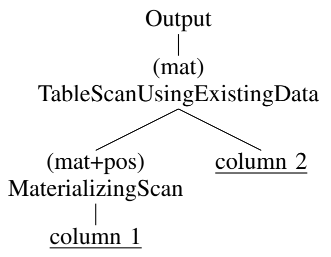


Figure 10. Plan 3 - Data

is high on column 1 and low on column 2, as a large part of column 1 will be materialized that will later on not be used. Given a row layout, the expectations are similar to those of the *No Data* plan, with the *Data* plan again performing better for lower selectivities than the *No Data* plan.

IV. Evaluation

As already pointed out in Section III-B, each of the plans is expected to work best at a certain configuration of selectivities. In order to empirically evaluate these assumptions, the algorithms were run on an IBM Series Blade, Xeon 5450, 64 GB RAM using different table layouts. First, a 2-column table stored in column-layout. Second, a 2-column table stored in row-layout. Third, a 60-column table stored in row-layout. Each table contains 1.000.000 rows; each column is 4 byte wide. The data was generated using the *HYRISE* data generation tool.

A. Column Layout

In order to measure the general performance, each plan was run for selectivities varying for both columns, each selectivity ranging from 0 to 1 in steps of 0.01. For each pair of selectivities, each plan was run and the total CPU cycle count was measured and averaged over three runs.

Figure 11a shows the algorithm with the lowest total CPU cycle count for each combination of selectivities. As can be clearly seen, the influence of the first column's selectivity is considerably larger than that of the second. As expected, the *One Scan* plan outperforms the other plans for high selectivities on both columns whereas the position based scans are better at lower selectivities on column 1.

Figure 11b shows the ratio of CPU cycles of the algorithm with the highest count to that with the lowest count at the given selectivities. This ratio ranges from values between approximately 1 and 2.4. Two areas can be identified where high ratios appear. First, for a low selectivity on column 1 independent of the second column's selectivity. Second, for a high selectivity on column 1 and a low selectivity on column 2.

The relative performance at these extremes can be seen in more detail in Figure 12a, which shows the CPU cycles of all three plans for two fixed selectivities for column 2 of 1.25e-3 = (25 rows / 2.000.000 rows) and 1.0 in dependence of the first column's selectivity. Figure 12b depicts the same using a logarithmic scale on the x-axis, allowing differences for low selectivities on column 1 to be discerned more easily.

For low selectivities on column 1 the ratio from *One Scan* to *Data*, i. e., the highest to the lowest CPU count, is close to 2. This is expected behavior, as the position based algorithms do not have to read much of column 2 whereas the *One Scan* algorithm always must read both columns.

As can be seen in Figure 12a, the ratio of *One Scan* to *Data* is largest for a high selectivity on column 1 and a low selectivity on column 2. As can be seen in Figure 12c, *Data* also accesses twice as much memory as *One Scan*. This is expected. For this configuration, *One Scan* reads both input columns once and writes nothing, whereas *Data* reads the first column, writes positions and materialized data and reads the second column, thus performing roughly twice as many memory accesses as the *One Scan* algorithm.

Given a high selectivity on both columns, *One Scan* is only 1.25 times faster than *Data*, as can be seen in Figure 12b, and requires 1.4 times the number of L1 cache accesses. We measured, that for this configuration of selectivities about a third of *Data*'s CPU cycles is used by the `MaterializingScan` while the remaining cycles are used by the `TableScanUsingExistingData`, making the `TableScanUsingExistingData` 1.2 times faster than the complete *One Scan*. This is interesting, as the scan using existing data has to perform more work than the *One Scan* plan. While the *One Scan* plan reads and writes both columns, the `TableScanUsingExistingData` does the same but additionally reads positions.

For this configuration of selectivities, further investigation is required to identify the reasons for the unexpectedly good performance of the *Data* plan. Nevertheless, for the most configurations of selectivities the plans perform as expected.

B. Row Layout

The queries were executed on a 2-column and a 60-column table. Figue 13 shows the CPU cycles for fixed column 2 selectivities for the 2-column layout. As can be seen, the performance is very close to the column-layout performance. The *One Scan* plan is, as expected, faster than the others for high selectivities, as it only has to read one continuous block of data once whereas the position based scans have to process this block twice, thus causing more cache misses. As can be seen in Figure 13a, *One Scan* is still slower than the position based scans for low column 1 selectivities. There, the number of cache accesses that were measured for the *One Scan* plan, which evaluates predicates on both columns in parallel, were, as expected, higher than for the other plans, with the cache misses measured only slightly higher.

In order to analyze the algorithms' performance for larger containers, they were executed on a 60-column / 1 container table. All queries still output only the first two columns. The `MaterializingScan` of the *Data* plan still only

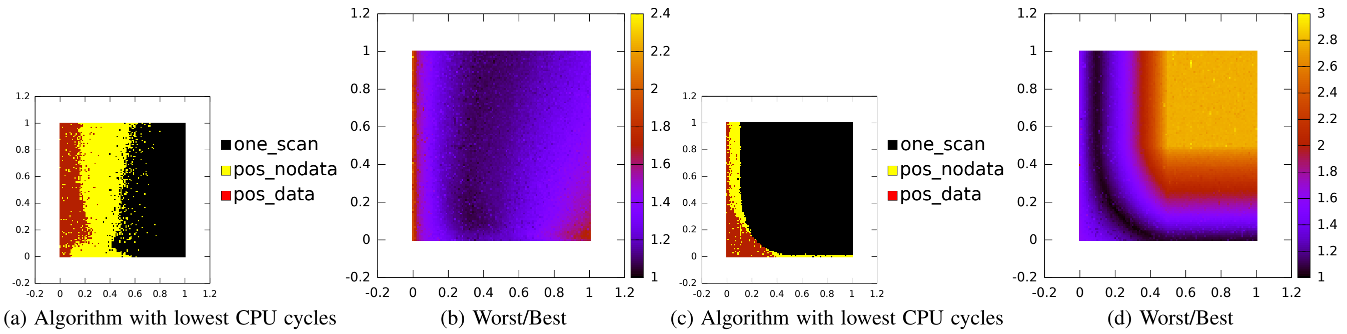


Figure 11. Figure (a) and (b) for 2 columns / 2 containers and (c) and (d) for 60 columns / 1 container. Comparing CPU cycles of all three algorithm across selectivities; x-axis = Column 1; y-axis = Column 2

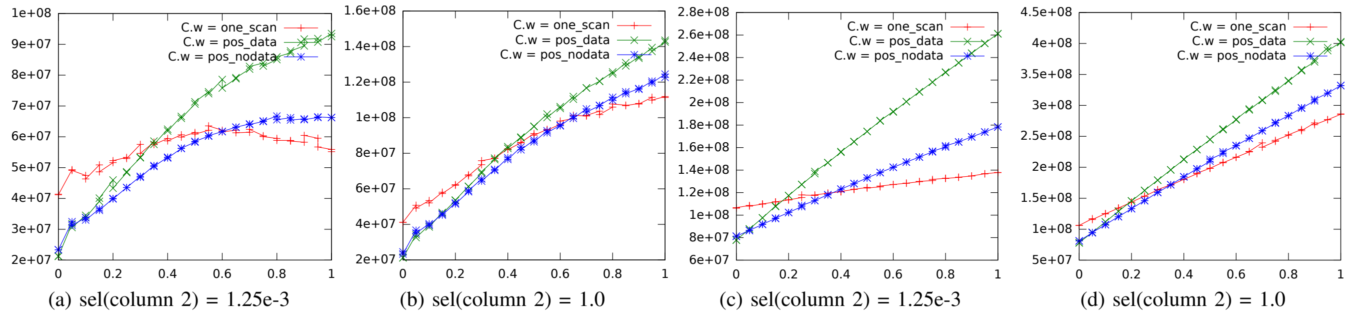


Figure 12. 2 columns / 2 containers. 12a and 12b Total CPU cycles. x-axis = selectivity on first column; Figure 12c and 12d are L1 data cache accesses for the same experiment. Selectivity on second column is fixed for all graphs.

materializes the first column. Figures 11c and Figure 11d give a general overview of the algorithms' performance. Unlike in previous benchmarks, the performance is strongly influenced by the selectivity of both columns. The *One Scan* plan performs best if the combined selectivity is not low and it is roughly 2.8 times faster than the *Data* scan for a selectivity of 1 on both columns. If selectivity is low on column 1 or very low on column 2, the position based plans are best, the *Data* plan performing roughly 1.6 times faster than the *One Scan* plan for very low selectivities on both columns.

Figure 13d provides a snapshot at fixed column 2 selectivities. Given a high column 1 selectivity, the *One Scan* plan surprisingly is slower for low column 2 selectivities than it is for high column 2 selectivities. This is the case even though the number of cache misses and cache accesses is lower for low column 2 selectivities than it is for high ones. Other counters, such as the number of branch mispredictions have not yet been measured and so a clear assessment can not be made, yet.

V. Related Work

The topic of materialization strategies has already been researched in the context of column-based as well as row-based DMBS. Abadi et al. [3] provide a general overview of column- and row stores and identifies the importance of late materialization in column stores. In [2], [4] Abadi et al. provide an evaluation and comparison of different materialization strategies in column-based DBMS. Different materialization strategies are identified and their performance for different kinds of queries is evaluated.

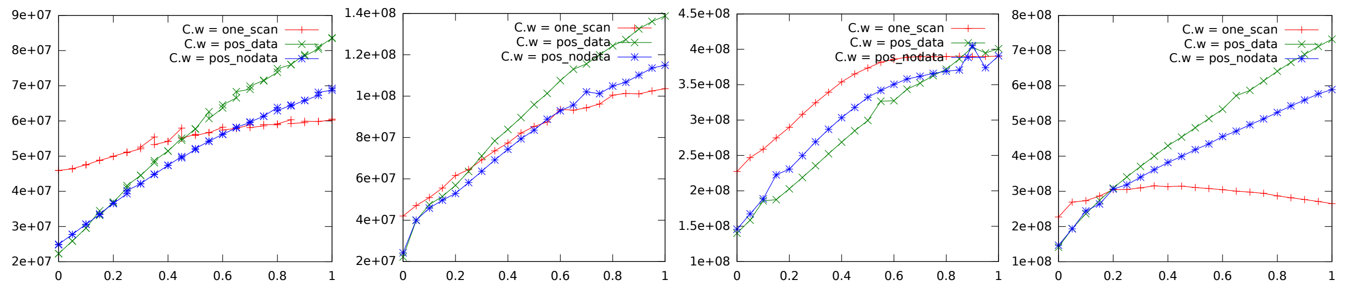
Ivanova et al. [5] analyze how materialized query plan results can be cached and reused for future queries to reduce execution times. This aspect of materialization strategies is complementary to the ones analyzed in this paper.

The materialization strategies that are analyzed in this paper are implemented in a operator at a time query execution engine. Zukowski et al. [6] follow a different approach by materializing vertical data fragments at a time, trying to restrict the data to the CPU cache.

In addition, the implementation of compression for such main memory databases becomes more and more important as shown in [7], [8]. Krueger et al. show in [9] that it is possible to further optimize read optimized column databases with compression to possibly satisfy OLTP workloads.

VI. Conclusion

As has been seen in IV, the materialization strategies that were introduced in III mostly exhibit the expected relative performance for the simple selection query, especially if data is stored in column-layout. Table II summarizes the best and worst performance of each algorithm in dependence of the



(a) sel(column 2) = 1.25e-3 (b) sel(column 2) = 1.0 (c) sel(column 2) = 1.25e-3 (d) sel(column 2) = 1.0

Figure 13. 2 columns / 1 container for Figure 13a and 13b and 60 columns / 1 container for Figure 13c and 13d. Total CPU cycles. x-axis = selectivity on first column. Selectivity on second column is fixed

columns selectivity. As can be seen, the best plan depends mainly on the first column's selectivity.

Table II
SELECTIVITY CONFIGURATIONS THAT DELIVER THE BEST / WORST RESULTS FOR A COLUMN-LAYOUT

Plan	Best		Worst	
	Col 1	Col 2	Col 1	Col 2
One scan	high	high	low	-
No Data	low	-	high	low
Data	very low	-	high	low

The performance for multi-column containers did not completely match our expectations, with the *One Scan* plan performing unexpectedly slow for a high selectivity on column 1 and a low selectivity on column 1. As expected though, the *One Scan* plan performs best for most of the configurations, with the position based plans providing better performance when selectivity is low on column 1 or very low on column 1.

All together, the choice of materialization strategy greatly influences the query execution time. For the simple selection query analyzed, choosing the wrong materialization strategy resulted in up to 2.8-fold increased CPU cycles. For the simple example, the optimal materialization strategy can be predetermined if the table layout is known and the selectivities of the predicates can be estimated in advance. This can be achieved by applying established techniques that rely on collecting statistical data, such as histograms [10].

An actual implementation that chooses materialization strategies automatically for container-based DBMS, such as *HYRISE*, requires further research. The analysis performed on multi-column containers has shown that the layout of tables has a strong influence on materialization strategies. Yet, aspects such as the positions of columns within containers have not been analyzed, so that further research is required to turn these observations into knowledge that can be applied at query plan construction time.

Apart from that, many other areas have been left untouched, such as analyzing different kinds of queries or taking different types of data storage, e. g., compressed position lists, into account. Still, this paper and the accompanying implementation provide the required knowledge to leverage different materialization strategies in a hybrid main memory database system such as *HYRISE*.

REFERENCES

[1] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden, "Hyrise - a main memory hybrid storage engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[2] D. Abadi, D. Myers, D. DeWitt, and S. Madden, "Materialization Strategies in a Column-Oriented DBMS," in *ICDE 2007*, pp. 466–475.

[3] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD Conference*, 2008, pp. 967–980.

[4] D. Abadi, "Query execution in column-oriented database systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2008.

[5] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves, "An architecture for recycling intermediates in a column-store," in *SIGMOD Conference*, 2009, pp. 309–320.

[6] M. Zukowski, P. Boncz, N. Nes, and S. Heman, "MonetDB/X100-a DBMS in the CPU cache," *Data Engineering*, vol. 1001, p. 17, 2005.

[7] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD Record*, vol. 29, no. 3, pp. 55–67, 2000.

[8] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.

[9] J. Krüger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber, "Optimizing write performance for read optimized databases," in *DASFAA (2)*, 2010, pp. 291–305.

[10] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita, "Improved histograms for selectivity estimation of range predicates," *ACM SIGMOD Record*, vol. 25, no. 2, p. 305, 1996.