Workload Representation across Different Storage Architectures for Relational DBMS

Andreas Lübcke
School of Computer Science
University of Magdeburg
Magdeburg, Germany
andreas.luebcke@ovgu.de

Veit Köppen
School of Computer Science
University of Magdeburg
Magdeburg, Germany
veit.koeppen@ovgu.de

Gunter Saake
School of Computer Science
University of Magdeburg
Magdeburg, Germany
gunter.saake@ovgu.de

ABSTRACT

Database systems differ from small-scale stripped database programs for embedded devices with minimal footprint to large-scale OLAP applications for server devices. For relational database management systems, two storage architectures have been introduced: the row-oriented and the column-oriented architecture. To select the optimal architecture for a certain application, we need workload information and statistics. In this paper, we present a workload representation approach that enables us to represent workloads across different DBMSs and architectures. Our approach also supports fine granular workload analyses based on database operations.

1. INTRODUCTION

New requirements for database applications [23, 26, 27] came up in recent years. Therefore, database management system (DBMS) vendors and researchers developed new technologies, e.g., column-oriented DBMSs (*column stores*) [1, 22, 30]. New approaches are developed to satisfy the new requirements for database applications, thus the number of candidates in the decision process has also increased. Moreover, new application fields imply a more complex decision process to find the suitable DBMS for a certain use case.

We need statistics to come to a suitable design decision. These statistics have to be represented system-independent for sound and comparable decision. That implies the independence of workload representation from different storage architectures. In this paper, we introduce a new approach of workload statistics aggregation and maintenance across different DBMSs and architectures. We showed in [16] that query-based workload analyses, as described in [7], are not suitable to select the optimal storage architecture. To overcome drawbacks of query-based workload analyses, we define workload patterns based on database operations. We introduce a workload decomposition algorithm that enables us to analyze query parts. Workload patterns represent the decomposed workloads to compare the performance of database operations for column and row stores. These workload patterns contain all statistics needed for cost estimations. We simulate the statistic gathering process with a exemplary workload.

23rd GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 31.05.2011 - 03.06.2011, Obergurgl, Austria.
Copyright is held by the author/owner(s).

2. STATISTICS REPRESENTATION

To select the optimal storage architecture, we have to analyze a given workload; thus, we need to decompose this workload. We have to map single operations of a workload (at least of one query) and their optimizer statistics to evaluable patterns. Therefore, we present our pattern framework which stores all necessary statistics for subsequent performance analyses. In [18], we illustrate the procedure of our decision process regarding the storage architecture selection. Below, we outline the design of our pattern framework.

2.1 Pattern Types

To analyze the influence of single operations, we propose three patterns for operations in workload queries. The three operation patterns are ***tuple operations***, ***aggregations and groupings***, and ***join operations***. We define a number of sub-patterns for each of those three to characterize particular operations more precisely within the patterns. This way, we support analyses based on the three patterns and additionally fine granular analyses based on sub-patterns, i.e., we can determine where the majority of costs emerge within a workload (at least one query).

First, the ***tuple operation pattern*** covers all operations that process or modify tuples, e.g., selection, sort operations. We propose this pattern for performance analyses because row stores process directly on tuples in contrast to column stores that costly reconstruct tuples. We identify the following sub-patterns:

Sort/order operation: Sort/order operation creates sequences of tuples and affects all attributes of a tuple. We consider duplicate elimination as a sort operation because an internal sort is necessary to find duplicates.

Data access and tuple reconstruction: Row stores always access tuples and column stores must reconstruct tuples to access more than one column.

Projection: Projection returns a subset of tuple attribute values and causes (normally) no additional costs for query execution.

Filtering: Filtering selects tuples from tables or intermediate results based on a selection predicate, e.g., selection in WHERE-clause and HAVING-clause.

Second, we cover all column processing operations in the ***aggregation and grouping pattern***, e.g., COUNT and MIN/MAX. We propose this pattern as counterpart to the tuple operation pattern. The operations of this pattern work only on single columns except for grouping operations which can also process several columns, e.g., GROUP BY CUBE. Due to column-wise partitioned data and single column processing, column stores perform well on aggregations (cf. [16]). We identify the following sub-patterns:

Min/Max operation: The min/max operation provides the minimum/maximum value of a single attribute (column).

Sum operation: This operation provides the sum of all values in one column.

Count operation: The count operation provides the number of attribute values in a column and COUNT(*) provides only the number of key values, thus it processes a single column.

Average operation: The average operation computes all values of a single column as well as the sum operation, but it can have different characteristics, e.g., mean (avg) or median.

Group by operation: This operation merges equal values according to a certain column and results in a subset of tuples. Grouping across a number of columns is also possible.

Cube operations: The cube operation computes all feasible combination of groupings for selected dimensions. This generation requires the power set of aggregating columns, i.e., n attributes are computed by 2^n GROUP BY clauses.

Standard deviation: The standard deviation (or variance) is a statistical measure for the variability of a data set and is computed by a two pass algorithm which means two cycles.

Third, the ***join pattern*** matches all join operations of a workload. Join operations are costly tasks for DBMSs. This pattern shows differences of join techniques between column and row stores, e.g., join processing on compressed columns or on bitmaps. Within this pattern, we evaluate the different processing techniques against each other. Consequently, we define the following sub-patterns:

Vector based: The column oriented architecture naturally supports vector based join techniques while row stores have to maintain and create structures, e.g., bitmap (join) indexes [15].

Non-vector based: This pattern matches "classic" join techniques (from row stores[1]) to differentiate the performance between vector and non-vector based join, thus we can estimate effects on the join behavior by architecture.

We only propose these two sub-patterns because the join concepts, e.g., merge or nested loop join, exist for both architectures. Hence, we assume that there is no necessity to map each join concept into its own sub-pattern. Figure 1 shows all introduced patterns and their relation to each other based on our exemplary workload.

2.2 Dependencies between Patterns

Database operations are not always independent from each other. We can identify dependencies between the following patterns: join, filtering, sort/order, group/cube, and data access pattern.

Join operations innately imply tuple selections (filtering pattern). However, the tuple selection itself is part of the join operation by definition, thus we assume that an additional decomposition of join operations is not necessary. Moreover, new techniques would have to be implemented to further decompose join operations and gather the necessary statistics. Hence, the administrative cost for tuning will be noticeably increased. To a side-effect, the comparison of join techniques belonging to different architectures will be no longer possible because of system-specific decomposition.

We state that two different types of sort/order operation can occur, i.e., implicit and explicit sort. The explicit sort is caused by workload or user, thus we consider this operation in the sort/order pattern. In contrast, we do not consider the implicit sort operation in the sort/order pattern because this sort operation is caused by the optimizer, e.g., for sort-merge join. Therefore, we assign all costs of grouping to the GROUP BY (or CUBE) pattern including the sort costs to sustain comparability.

Third, tuple reconstruction is part of several operations for column stores. We add these costs to the tuple operation pattern. We sustain the comparability of operations beyond the architectures because row stores are not affected by tuple reconstructions.

We assume further workload decomposition is not meaningful because administrative costs would affect the performance of existing systems as well as the comparability of performance issues between the architectures according to certain workload parts. These impacts would disadvantageously affect the usability of our pattern framework.

[1]Some column stores also support these join techniques.

3. QUERY DECOMPOSITION

In this section, we introduce the query decomposition approach. First, we illustrate the (re-) used DBMS functionality and how we gather necessary statistics from existing systems. Second, we introduce the mapping of decomposed query parts to our established workload patterns and show a decomposition result by example. Our approach is applicable to each relational DBMS. Nevertheless, we decide to use a closed source system for the following considerations because the richness of detail of optimizer/query plan output is higher and easier to understand. More detailed information will result in more accurate recommendation.

3.1 Query Plans

A workload decomposition based on database operations is necessary to select the optimal storage architecture (cf. [16]). Therefore, we use query plans [4] which exist in each relational DBMS. On the one hand, we reuse database functionality and avoid new calculation costs for optimization. On the other hand, we make use of system optimizer estimations that are necessary for physical database design [10].

Based on query plans, we gather statistics directly from DBMS and use the optimizer cost estimations. The example in Listing 1 shows an SQL query and we transform this to a query plan in Table 1 [19]. Table 1 already offers some statistics such as number of rows, accessed bytes by the operation, or costs. Nevertheless, Table 1 shows only an excerpt of gathered statistics. All possible values for query plan statistics can be found in [20] Chapter 12.10. Hence, we are able to determine the performance of operations on a certain architecture (in our example a row store) by statistics such as CPU costs and/or I/O costs. In addition to performance evaluation by several estimated costs, we can gather further statistics from query plans which may influence performance of an operation on a certain architecture, e.g., cardinality. For column stores, the operation cardinality can indirectly affect performance if the operation processes several columns, thus column stores have to process a number of tuple reconstructions, e.g., high cardinality means many reconstructions. Thus, we use meta-data to estimate influences of data itself on the performance, e.g., we can compute the selectivity of attributes.

3.2 From Query Plans to Workload Patterns

We have to map the gathered statistics from DBMS to our workload patterns. We use a second example [21] (Listing 2 and Table 2) to simulate a minimum workload instead of a single query. In the following, we illustrate the mapping approach by using the examples in Listing 1 and 2. In our name convention, we define a unique number[2] that identifies queries of the workload within our mapping algorithm, i.e., 1.X represents query 1 (Listing 1) and equally 2.X represents query 2 (Listing 2). Furthermore, we reuse the operation IDs from query plans (Table 1 and 2) in the second hierarchy

[2]In the following considerations, we start with 1 which represents the first query.

```sql
1 SELECT *
2 FROM employees e JOIN departments d
3 ON e.department_id=d.department_id
4 ORDER BY last_name;
```

Listing 1: Example SQL query (14-1) [19]

ID	Operation	Name	Rows	Bytes	Cost (%CPU)	...
0	SELECT STATEMENT		106	9328	7 (29)	...
1	SORT ORDER BY		106	9328	7 (29)	...
* 2	HASH JOIN		106	9328	6 (17)	....
3	TABLE ACCESS FULL	DEPARTMENTS	27	540	2 (0)	...
4	TABLE ACCESS FULL	EMPLOYEES	107	7276	3 (0)	...

Table 1: Textual query plan of SQL example (14-1) [19]

level (for X), e.g., 1.4 is the operation with `ID 4` of query 1 (cf. Table 1). In the following, we refer the CPU cost of Table 1 and 2.

The first query (Listing 1) is decomposed into four patterns. First, we see the data access operation of the `department (ID 3)` and the `employees (ID 4)` tables in the corresponding query plan in Table 1. The total cost for the data access operations is `5`. Second, the join operation `(ID 2)` is executed with a hash join algorithm. The hash join cost is only `1` because in Table 1 costs are iteratively sum up and the costs of its children `(5)` and its own cost `(1)` are summed up to `6` for `ID 2`. Third, the sort operation `(ID 1)` implements the ORDER BY statement with cost of `1`. The total costs of all processed operations are `7` now. Fourth, the select statement `(ID 0)` represents the projection and causes no additional cost (remain `7`). Following our name convention, the identifiers from `1.0` to `1.4` represent the operations of our first query (Listing 1) in Figure 1.

We also decompose the second example (Listing 2) into four operation types (cf. Table 2). First, `IDs 3, 7, and 8` represent the data access operations and cause total costs of `14`. Second, the optimizer estimates both hash joins `(ID 2 and 6)` with no (additional) costs because their costs are only composed by the summed costs of their children (`ID 3, 4` and `ID 7, 8`). Third, the GROUP BY statement in Listing 2 is implemented by hash-based grouping operations (`ID 1` and `ID 5`). The cost of each HASH GROUP BY is `1` and the total costs of this operation type are `2`. Fourth, the projection `(ID 0)` and the sum operation represented by select statement causes again no additional costs. If the sum operation causes costs then it will be represented by a separate operation (`ID`). Following our name convention, the identifiers from `2.0` to `2.8` represent the operations of the second query (Listing 2) in Figure 1. The view (`ID 2.4`) is not represented in our workload pattern because its costs are already mapped by its child operations (`ID 2.5-2.8`).

In our examples, we summarize single operations of similar types (five for example query two). In the following, we list the five operation types and assign them to our workload patterns and their sub-patterns that we introduced in Section 2. The join operations of our example queries `ID 1.2, 2.2,` and `2.6` are assigned to the non-vector based join pattern. We assign the operations with `ID 1.3, 1.4, 2.3, 2.7,` and `2.8` to the data access sub-pattern of the tuple operation pattern. We also assign the projections (`ID 1.0` and `2.0`) and the sort operation `(ID 1.1)` to the tuple operation pattern. Finally, we assign the group by operations (`ID 2.1` and `2.5`) to the group by sub-pattern within the aggregation and grouping pattern. We present the result in Figure 1 whereby we only show ID and cost of each operation for reasons of readability. We state that the we do not need to directly extract statistics from existing systems. Our pattern framework is system independent, thus we are also able to use already extracted (or aggregated) data as well as estimated values.

3.3 Operations in Column Stores

We state that we do not need a separate decomposition algorithm for column stores, i.e., the query plan operations of column stores can be also mapped to our workload patterns. Representatively, we illustrate the mapping of C-Store/Vertica query plan operations introduced in [25] and map them to our workload patterns as follows:

Decompress: Decompress is mapped to the data access pattern. This operation decompresses data for subsequent operations in the query plan that cannot process on compressed data (cf. [1]).

Select: Select is equivalent to the selection of relational algebra with the exception that the result is represented as bitstring. Hence, we map it to the filtering pattern.

Mask: Mask process on bitstrings and returns only those values whose associated bits in the bitstring are 1. Consequently, we map mask to the filtering pattern.

Project: Projection is equivalent to the projection of relational algebra, thus this operation is mapped to the projection pattern.

Sort: This operation sorts the columns of a C-Store projection according to a (set of) sort column(s). This technique is equivalent to sort operations on projected tuples, i.e., we can map this operation to the sort/order pattern.

Aggregation Operators: These operations compute aggregations and groupings like in SQL [1], thus we directly map these operations to the corresponding sub-pattern in the aggregation & grouping pattern.

Concat: Concat combines C-Store projections sorted in the same order into a new projection. We regard this operation as tuple reconstruction and map it to the corresponding pattern.

Permute: This operation permutes the order of columns in C-Store projections according to the given order by a join index. It prevents additional replication overhead that would emerge through creation of join indexes and C-Store projections in several orders. This operation is used for joins, thus we map its cost to the join pattern.

Join: We map this operation to the join pattern and distinguish two join types. First, if tuples are already reconstructed then we process them as row stores, i.e., we map this join type to the non-vector based join pattern. Second, the join operation only processes columns that are needed to evaluate the join predicate. The join result is a set of pairs of positions in the input columns [1]. This join type can process on compressed data as well as it can use vector based join techniques, thus, we map this join type to the vector based join pattern.

Bitstring Operations: These operations (AND, OR, NOT) process bitstrings and compute a new bitstring with respect to the corresponding logical operator. These operations implement the concatenation of different selection predicates. Therefore, we map these operations to the filtering pattern.

Finally, we state that our approach can be used for each relational DBMS. Each relational DBMS is referable to the relational data model, so these DBMSs are based on the relational algebra in some manner too. Thus, we can reduce or map those operations to our workload patterns; in worst case, we have to add an architecture-specific operation for hybrid DBMSs to our pattern, e.g., tuple reconstruction for column stores. For a future (relational)

```sql
1 SELECT c.cust_last_name, SUM(revenue)
2 FROM customers c, v_orders o
3 WHERE c.credit_limit > 2000
4 AND o.customer_id(+) = c.customer_id
5 GROUP BY c.cust_last_name;
```

Listing 2: Example SQL query (11-9) [21]

ID	Operation	Name	Rows	Bytes	Cost (%CPU)	...
0	SELECT STATEMENT		144	4608	16 (32)	...
1	HASH GROUP BY		144	4608	16 (32)	...
* 2	HASH JOIN OUTER		663	21216	15 (27)	...
* 3	TABLE ACCESS FULL	CUSTOMERS	195	2925	6 (17)	...
4	VIEW	V_ORDERS	665	11305		...
5	HASH GROUP BY		665	15960	9 (34)	...
* 6	HASH JOIN		665	15960	8 (25)	...
* 7	TABLE ACCESS FULL	ORDERS	105	840	4 (25)	...
8	TABLE ACCESS FULL	ORDER_ITEMS	665	10640	4 (25)	...

Table 2: Textual query plan of SQL example (11-9) [21]

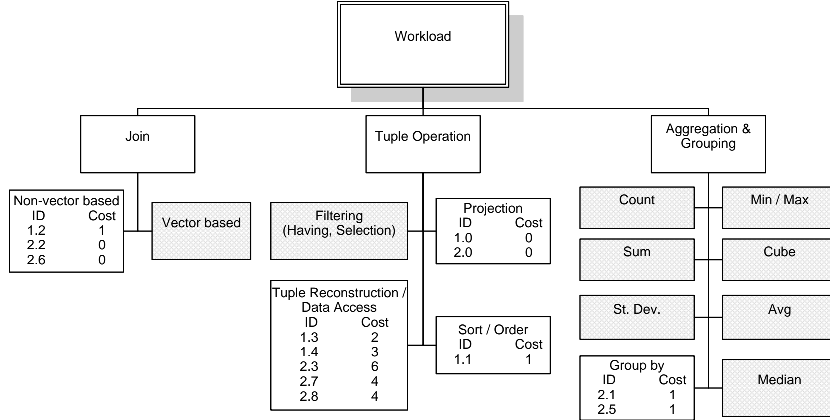

Figure 1: Workload patterns with cost of operations for the row store example workload

hybrid storage architecture, such an operation could be necessary to map the cost for conversions between row- and column-oriented structures and vice versa.

4. DEMONSTRATING EXAMPLE

We decide to simulate the workload with the standardized TPC-H benchmark (2.8.0) to show the usability of our approach. We use the DBMSs Oracle 11gR2 Enterprise Edition and Infobright ICE 3.3.1 for our experiments[3]. We run all 22 TPC-H queries and extract the optimizer statistics from the DBMSs. For reasons of clarity and comprehensibility, we only map three representative TPC-H queries namely `Q2, Q6,` and `Q14` to the workload patterns.

The query structure, syntax, and execution time are not sufficient to estimate the query behavior on different storage architectures. We introduced an approach based on database operations that provides analyses to find long running operations (bottlenecks). Moreover, we want to figure out reasons for the behavior, thus we have to use additional metrics. We select the I/O cost to compare the DBMSs and summarize the optimizer output in Table 3. Following our previous name convention, we define the query `IDs` according to their TPC-H query number, i.e., we map the queries with the `IDs 2, 6`, and `14`. The operations are identified by their query plan number (IDs in Table 3), thus the root operation of TPC-H query `Q2` has the `ID 2.0` in Figure 2. All values in Table 3 are given in Kbytes. The given values are input costs of each operation except the table access costs because no information on input costs to table access operations are available. Note, the granularity of Oracle's costs measurements is on the byte level whereas the measurements of ICE are on the data pack (65k) level.

In Figure 2, we present our workload patterns with I/O costs of the corresponding TPC-H queries. As we mentioned before, the projection operation causes no additional costs. Hence, the I/O costs in Table 3 and Figure 2 represent the size of final results. The stored information can be analyzed and aggregated in decision models with any necessary granularity. In our example, we only sum up all values of the data access pattern for each query to calculate the I/O costs per query in Kbytes. For these three queries, all results and intermediate results are smaller than the available main memory, thus no data has to be reread subsequently. Oracle reads 1452.133 Kbytes for query `Q2` and takes 8.14 seconds. ICE needs 41 seconds and access 2340 Kbytes. We suppose, the DBMS with minimal I/O cost performs best. Our assumption is confirmed for query `Q14`. Oracle accesses 7020.894 Kbytes and computes the query in 22.55 seconds whereas ICE computes it in 3 seconds and reads 6240 Kbytes. Nevertheless, we cannot prove our assumption for query `Q6`. Oracle (3118 Kbytes) accesses less data than ICE (5980) Kbytes but ICE (2 seconds) computes this query ten times faster than Oracle (22.64 seconds). Hence, we cannot figure out a definite correlation for our sample workload.

We state that only I/O cost is not sufficient to estimate the behavior of database operations. However, I/O cost is one important metric to describe performance behavior on different storage architectures because one of the crucial achievements of column stores is the reduction of data size (i.e., I/O cost) by aggressive compression. The I/O cost also gives an insight into necessary main memory for database operations or if operations have to access the secondary memory. Hence, we can estimate that database operations are completely computed in main memory or data have to be reread/read stepwise[4].

[3] We also wanted to evaluate our approach with the DBMSs solutions from Vertica and Sybase because both DBMSs use cost-based optimizer and we would be able to receive more expressive results. We requested the permission to use the systems for our evaluation but until now the decision is pending.

[4] We remind of the performance gap (circa 10^5) between main memory and HDDs.

Operation	Oracle			ICE		
	Q2 (8.14sec)	**Q6** (22.64sec)	**Q14** (22.55sec)	**Q2** (41sec)	**Q6** (2sec)	**Q14** (3sec)
Data Access	*ID7:*0.8;*ID12:*0.029;*ID13:*11.2; *ID15:*0.104;*ID16:*1440	*ID2:*3118	*ID3:*1620.894; *ID4:*5400	*ID4:65;ID5:65;ID6:*845;*ID7:65ID8:*260; *ID10:*65;*ID11:*65;*ID12:*65;*ID13:*845	*ID2:*5980	*ID3:*5980; *ID4:*260
Non-vector based join	*ID6:*202.760;*ID8:*1440;*ID9:*88.016; *ID10:*17;*ID11:*11.229		*ID2:*7020.894	*ID3:*1300;*ID9:*1040		*ID2:*6240
Sort	*ID3:*33.18;*ID5:*45.346			*ID2:65*		
Count	*ID1:*31.284			*ID1:65*		
Sum		*ID1:*3118	*ID1:*3610.173		*ID1:*5980	*ID1:*65
Projection	*ID0:*19.800	*ID0:*0.020	*ID0:*0.049	*ID0:65*	*ID0:*65	*ID0:*65

Table 3: Accessed Kbytes by query operations of TPC-H query Q2, Q6, and Q14.

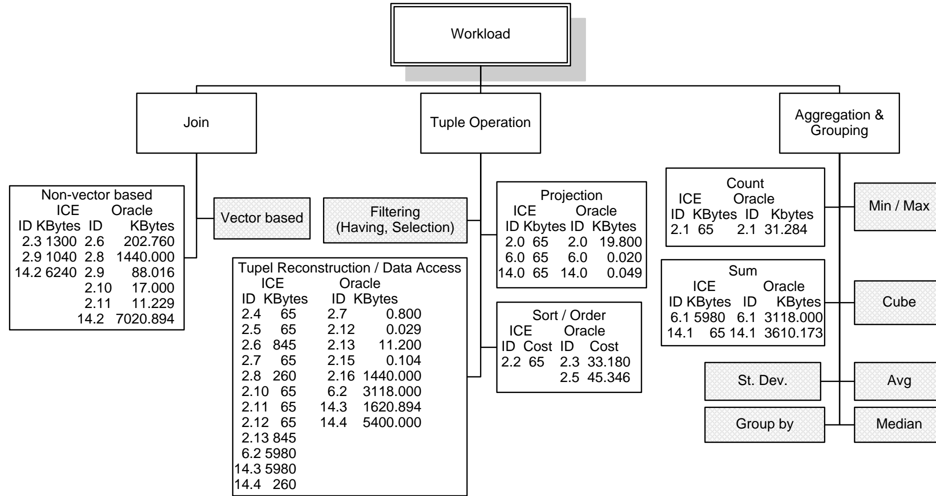

Figure 2: Workload graph with mapped I/O costs of TPC-H query Q2, Q6, and Q14.

5. RELATED WORK

Several column stores have been proposed [1, 14, 30] for OLAP applications. But all systems are pure column stores and do not support any row store functionality. Thus, a storage architecture decision between row and column store is necessary. Abadi et al. [2] compare row and column store with respect to performance on the star schema benchmark. They simulate column store architecture by indexing every single column or vertical partitioning of the schema. They show that using column store architecture in a row store is possible but the performance is poor. In this paper, we do not compare end to end performance of DBMSs or architectures. We support sound and comparable analyses based on database operations across different DBMSs with our approach. We do not discuss approaches like DSM [8], hybrid NSM/DSM schemes [9], or PAX [3] because the differences to state-of-the-art column stores have been already discussed, e.g., Harizopoulus et al. [11].

There are systems available which attempt to fill the gap between a column and a row store. C-Store [1] uses two different storage areas to overcome the update problems of column stores. A related approach brings together a column store approach and the typical row store domain of OLTP data [24]. However, we do not develop hybrid solutions that attempt to fill this gap for now.

There exist a number of design advisors which are related to our work, e.g., IBM DB2 Configuration Advisor [13]. The IBM Configuration Advisor recommends pre-configurations for databases. Zilio et al. [28, 29] introduce an approach that gathers statistics like our approach directly from DBMSs. The statistics are used to advise index and materialized view configurations. Similarly, Chaudhuri et al. [5, 6] present two approaches which illustrate the whole tuning process using constraints such as space threshold. However, these approaches operate on single systems instead of comparing two or more systems. In contrast to the mentioned approaches, our approach do not consider tune configurations, indexes, etc.

Another approach for OLAP applications is Ingres/Vectorwise which applies the Vectorwise (formerly MonetDB/X100) architecture into the Ingres product family [12]. In cooperation with Vectorwise, Ingres developes a new storage manager ColumnBM for the new Ingres/Vectorwise. However, the integration of the new architecture into the existing environment remains unclear [12].

6. CONCLUSION

In recent years, column stores have shown good results for DWH applications and often outperformed established row stores. However, new requirements arise in the DWH domain that cannot be satisfied only by column stores. The new requirements demand also for row store functionality, e.g., real-time DWHs need sufficient update processing. Thereby, the complexity of design process increases because we have to choose the optimal architecture for given applications. We showed with an experiment that workload analyses based on query structure and syntax are not sufficient to select the optimal storage architecture. Consequently, we suggested a new approach based on database operations. We introduced workload patterns which contain all workload information beyond the architectures, e.g., statistics and operation cost. We also presented a workload decomposition approach based on existing database functionality that maps operations of a given workload to our workload patterns. We illustrated the methodology of our decomposition approach using an example workload. Subsequently, we state that a separate decomposition algorithm for column stores is not needed. We stated that our presented approach is transparent

to any workload and any storage architecture based on the relational data model. In the evaluation, we proved the usability of our approach. Additionally, we demonstrate the comparability of different systems using different architectures even if the systems provide different information with respect to their query execution. The decision process can be periodically repeated, thus the storage architecture selection is not static. Moreover, our approach can be used for optimizer (decisions) in hybrid relational DBMS that has to select the storage method for parts of data.

In future work, we will investigate two strategies to implement our workload patterns in a prototype. First, we utilize a new DBS to export periodically statistics and operation costs which we map to our workload patterns. This way, we will not affect performance of analyzed systems by prediction computation. Second, we adapt existing approaches [5, 17] to automatically gather statistics, e.g., mapping statistics and workload patterns directly into a graph structure (query graph model). Additionally, aggregated or estimated values from other sources can be stored. We will perform detailed studies on OLAP, OTLP, and mixed workloads to gather expressive values for predictions.

7. REFERENCES

[1] D. J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Cambridge, MA, USA, 2008. Adviser: Madden, Samuel.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD '08*, pages 967–980, New York, NY, USA, 2008. ACM.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB '01*, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM TODS*, 1(2):97–137, 1976.

[5] N. Bruno and S. Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *VLDB '06*, pages 499–510. VLDB Endowment, 2006.

[6] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE '07*, pages 826–835, 2007.

[7] S. Chaudhuri and V. Narasayya. Autoadmin “what-if” index analysis utility. In *SIGMOD '98*, pages 367–378, New York, NY, USA, 1998. ACM.

[8] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD '85*, pages 268–279, New York, NY, USA, 1985. ACM.

[9] D. W. Cornell and P. S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Trans. Softw. Eng.*, 16(2):248–258, 1990.

[10] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM TODS*, 13(1):91–128, 1988.

[11] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06*, pages 487–498. VLDB Endowment, 2006.

[12] Ingres/Vectorwise. Ingres/VectorWise sneak preview on the Intel Xeon processor 5500 series-based platform. White Paper, September 2009.

[13] E. Kwan, S. Lightstone, K. B. Schiefer, A. J. Storm, and L. Wu. Automatic database configuration for DB2 Universal Database: Compressing years of performance expertise into seconds of execution. In *BTW '03*, pages 620–629, 2003.

[14] T. Legler, W. Lehner, and A. Ross. Data mining with the SAP NetWeaver BI Accelerator. In *VLDB '06*, pages 1059–1068. VLDB Endowment, 2006.

[15] A. Lübcke. Cost-effective usage of bitmap-indexes in DS-Systems. In *20th Workshop "Grundlagen von Datenbanken"*, pages 96–100. School of Information Technology, International University in Germany, 2008.

[16] A. Lübcke. Challenges in workload analyses for column and row stores. In *22nd Workshop "Grundlagen von Datenbanken"*, volume 581. CEUR-WS.org, 2010.

[17] A. Lübcke, I. Geist, and R. Bubke. Dynamic construction and administration of the workload graph for materialized views selection. *Int. Journal of Information Studies*, 1(3):172–181, 2009.

[18] A. Lübcke, V. Köppen, and G. Saake. A decision model to select the optimal storage architecture for relational databases. RCIS, France, MAY 2011. IEEE. to appear.

[19] Oracle Corp. Oracle Database Concepts 11g Release (11.2). 14 Memory Architecture (Part Number E10713-05), March 2010.

[20] Oracle Corp. Oracle Performance Tuning Guide 11g Release (11.2). 12 Using EXPLAIN PLAN (Part Number E10821-05), March 2010.

[21] Oracle Corp. Oracle Performance Tuning Guide 11g Release (11.2). 11 The Query Optimizer (Part Number E10821-05), March 2010.

[22] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD '09*, pages 1–2, New York, NY, USA, 2009. ACM.

[23] R. J. Santos and J. Bernardino. Real-time data warehouse loading methodology. In *IDEAS '08*, pages 49–58, New York, NY, USA, 2008. ACM.

[24] J. Schaffner, A. Bog, J. Krüger, and A. Zeier. A hybrid row-column OLTP database architecture for operational reporting. In *BIRTE '08*, 2008.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB '05*, pages 553–564. VLDB Endowment, 2005.

[26] A. A. Vaisman, A. O. Mendelzon, W. Ruaro, and S. G. Cymerman. Supporting dimension updates in an OLAP server. *Information Systems*, 29(2):165–185, 2004.

[27] Y. Zhu, L. An, and S. Liu. Data updating and query in real-time data warehouse system. In *CSSE '08*, pages 1295–1297, Washington, DC, USA, 2008. IEEE Computer Society.

[28] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *VLDB '04*, pages 1087–1097. VLDB Endowment, 2004.

[29] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 Design Advisor. In *ICAC '04*, pages 180–188, 2004.

[30] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Eng. Bulletin*, 28(2):17–22, June 2005.