Technical report

Nr.: FIN-011-2011

A Query Decomposition Approach for Relational DBMS using Different Storage Architectures

Andreas Lübcke, Veit Köppen, and Gunter Saake

Workgroup Databases

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-011-2011

A Query Decomposition Approach for Relational DBMS using Different Storage Architectures

Andreas Lübcke, Veit Köppen, and Gunter Saake

Workgroup Databases

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Andreas Lübcke
Postfach 4120
39016 Magdeburg
E-Mail: andreas.luebcke@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 21.12.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

A Query Decomposition Approach for Relational DBMS using Different Storage Architectures

Andreas Lübcke, Veit Köppen, and Gunter Saake
School of Computer Science,
Otto-von-Guericke-University Magdeburg, Germany
{andreas.luebcke,veit.koeppen,gunter.saake}@ovgu.de

Abstract: Database systems range from small-scale stripped database programs for embedded devices with minimal footprint to large-scale OLAP applications. For relational database management systems, two storage architectures have been introduced: a) row-oriented architecture and b) column-oriented architecture. In this paper, we analyze the workload for database systems to select the most suitable architecture for each workload. We present a query decomposition approach to evaluate database operations with respect to their performance according to the storage architecture. Decomposed queries are mapped to workload patterns which contain aggregated database statistics. Further, we develop decision models which advise the selection of the optimal storage architecture for a given application domain.

1 Introduction

Administration and optimization of database systems is a costly task [WKKS99]. Therefore, database-management-system (DBMS) vendors and researchers developed self-tuning techniques to continuously and automatically tune DBMSs [IBM06, WHMZ94]. Interestingly, almost all approaches target at row-oriented DBMSs (*row stores*) [CN07].

New requirements for database applications (e.g., extraordinary data growth, real-time data warehousing) came up in recent years. Therefore, DBMS vendors and researchers developed new technologies. Column-oriented DBMSs (*column stores*) [Aba08, ABH09, ZBNH05] were developed to process aggregates (and other typical data-warehouse operations) more efficiently on exploding data volumes than the long-established row stores (cf. also Section 2). Researcher investigate approaches that combine transactional and analytical processing functionality [SB08, VMRC04, ZAL08, Pla09, KN11][1] to support analyses on up-to-date data (best in real-time) as well as new approaches for decision analyses in new domains like sensor networks [BGS01] and mobile devices [Man04]. New approaches are developed to satisfy new changed requirements for database applications, thus the number of candidates in the decision process for physical database design has also increased. Moreover, new application fields imply a more complex decision process to find the suitable DBMS for a certain use case.

[1] Most approaches for both OLTP and OLAP are main-memory-based, thus they are only suitable for large server environments.

In this paper, we introduce a new approach of workload-statistics aggregation and maintenance whereby we illustrate the performance of relational DBMSs with different architectures (row or column) concerning a given workload. First, we show that query-syntax-based workload analyses, as described in [CN98], are not suitable to select the optimal storage architecture. Second, we define workload patterns based on database operations that support cost estimations with a basis of comparison. Third, we introduce a workload decomposition algorithm that enables us to analyze query parts. Workload patterns represent decomposed workloads to compare the performance of database operations for column and row stores. These workload patterns include all statistics needed for cost estimations. We simulate the statistic gathering process with an exemplary workload. Finally, we show that our approach is feasible for both architectures.

2 Challenges for Physical Design Process

In this section, we illustrate the increased complexity of the physical design process in the data-warehouse (DWH) domain. Formerly, the dominant architecture for (large-scale and commercial) relational DWH was row store[2] which is similar according to their functionality. There are only slight differences in functionality (e.g., TID concept, clustered and non-clustered data storage, data access via primary key, tuple-wise access based on pages) between different row-store implementations, thus they are comparable in the manner of query execution. The data storage of row stores is tuple-wise organized, i.e., tuples are stored sequentially. In contrast, column stores partition tuples column-wise (vertical), thus the values of columns are stored sequentially.

TPC-H benchmark results[3] show that column stores are faster than row stores for typical DWH workloads [AMH08, SAB+05]. However, column stores perform worse on tuple operations and updates because after vertical partitioning, column stores have to reconstruct tuples to proceed. Thus, we assume that there are application fields for row and column stores in the DWH domain with respect to new requirements like real-time DWHs [SB08, ZAL08] or dimension updates [VMRC04].

	Standard TPC-H		Adjusted TPC-H	
Query #	**MySQL**	**ICE**	**MySQL**	**ICE**
TPC-H Q15	00:00:08	00:00:01	00:00:08	00:00:02
TPC-H Q16	00:00:09	00:00:01	00:00:12	00:00:24

Table 1: Influence of operations to DBMS performance [Lüb10] (query execution times in hh:mm:ss).

We compare and estimate performance for a given workload for row and column stores to select the optimal storage architecture for a use case (e.g., real-time data warehousing). For the comparison, we consider different optimization techniques (e.g., different index structures) in each architecture. Unfortunately, several optimization techniques cannot be compared for both architectures because they are architecture-specific (e.g., self-tuning, vector-based operations). Self-tuning approaches [CN07] (indexes etc.) are well

[2]The authors are aware that column stores are already proposed in 1979 [THC79] and Sybase releases proprietary products more than 15 years ago.

[3]http://www.tpc.org/tpch/results/tpch_perf_results.asp

investigated for row stores but not for column stores. Column stores support a number of compression techniques and vector based operations [Aba08] which are not supported by row stores. New compression techniques are developed for row stores [OJP+11, Ora11] (e.g., Oracle Hybrid Columnar Compression on Exadata (HCC)) that are similar to compression techniques in column stores. Nevertheless, row stores have to decompress data for processing while column stores are capable to process on compressed data. The processing on compressed data is (beside aggressive compression) one of the major advantages of column stores. Moreover, column stores themselves increase the decision complexity because there is an amount of different approaches (e.g., column stores utilize either tuple-oriented or column-oriented query processors). Summarizing, there are more choices to be made for column than for row stores - this increases complexity (beyond choice of architecture).

```
1  CREATE VIEW revenue0(supplier_no, total_revenue) AS
2          SELECT l_suppkey,SUM(l_extendedprice * (1 - l_discount))
3          FROM lineitem
4          WHERE l_shipdate>=date '1993-05-01'
5          AND l_shipdate<date '1993-05-01'+interval'3'month
6          GROUP BY l_suppkey;
7  SELECT s_suppkey,s_name,s_address,s_phone,total_revenue
8  FROM supplier,revenue0
9  WHERE s_suppkey=supplier_no AND total_revenue=
10         (SELECT MAX(total_revenue) FROM revenue0)
11 ORDER BY s_suppkey;
12 DROP VIEW revenue0;
```

Listing 1: TPC-H query Q15

```
1  SELECT p_brand,p_type,p_size,COUNT(DISTINCT ps_suppkey)
2  AS supplier_cnt
3  FROM partsupp,part
4  WHERE p_partkey=ps_partkey AND p_brand<>'Brand#51'
5  AND p_type NOT LIKE 'SMALL␣PLATED%'
6  AND p_size IN(3, 12, 14, 45, 42, 21, 13, 37)
7  AND ps_suppkey NOT IN(
8          SELECT s_suppkey FROM supplier
9          WHERE s_comment LIKE '%Customer%Complaints%')
10 GROUP BY p_brand,p_type,p_size
11 ORDER BY supplier_cnt DESC,p_brand,p_type,p_size;
```

Listing 2: TPC-H query Q16

To show the complexity of storage-architecture decisions, we introduce an example based on the TPC-H benchmark [Tra10]. We use the DBMSs MySQL 5.1.37 (row store) and Infobright ICE 3.2.2 (column store) in our test setup. Our decision to use these two DBMSs is referable to the fact that both systems are based on the same DBMS-kernel [Inf08]. In the following, we motivate our work, thus we only present an excerpt of our study. We modify the number of returned attributes of the TPC-H queries `Q15` and `Q16`[4] (cf. List-

[4]These two queries illustrate typical results from our study, thus they are representative.

```
1  CREATE VIEW revenue0(supplier_no, total_revenue) AS
2          SELECT l_suppkey,SUM(l_extendedprice * (1 - l_discount))
3          FROM lineitem
4          WHERE l_shipdate>=date '1993-05-01'
5          AND l_shipdate<date '1993-05-01'+interval'3'month
6          GROUP BY l_suppkey;
7  SELECT *,total_revenue
8  FROM supplier,revenue0
9  WHERE s_suppkey=supplier_no AND total_revenue=
10          (SELECT MAX(total_revenue) FROM revenue0)
11 ORDER BY s_suppkey;
12 DROP VIEW revenue0;
```

Listing 3: Adjusted TPC-H query Q15

```
1  SELECT *,COUNT(DISTINCT ps_suppkey) AS supplier_cnt
2  FROM partsupp,part
3  WHERE p_partkey=ps_partkey AND p_brand<>'Brand#51'
4  AND p_type NOT LIKE 'SMALL␣PLATED%'
5  AND p_size IN(3,12,14,45,42,21,13,37)
6  AND ps_suppkey NOT IN(
7          SELECT s_suppkey FROM supplier
8          WHERE s_comment LIKE '%Customer%Complaints%')
9  GROUP BY p_brand,p_type,p_size
10 ORDER BY supplier_cnt DESC,p_brand,p_type,p_size;
```

Listing 4: Adjusted TPC-H query Q16

ing 1 and 2) to demonstrate influences of a single operation to the query performance. We choose the naive approach to return all attributes (cf. Listing 3 and 4; i.e., we increase the size of processed tuples). Note that we do not change the query structure. We assume that query-based workload analyses [CN98] are not sufficient to estimate performance behavior using different storage architectures if there is an influence by our changes. The results (cf. Table 1) show that there is only a negligent influence by our changes to `Q15`, i.e., the mutual performance of both DBMS is not affected. In contrast, the mutual performance of both DBMS alters for `Q16`. The differences are not obvious from the query structure or syntax (cf. Listing 1 and 2). We propose that the change of projection differentially alters the size of intermediate and final results of both queries. Hence, the performance differences are caused by different number of involved columns[5]. In other words, modifications to a single operation have different impacts on different queries. Our complete study can be found in [Lüb10]. Hence, we state that a general decision regarding the storage architecture is not possible based only on the query structure, as described in [CN98]. We have to analyze single operations of a query (e.g., join operation or tuple selection) to select the optimal storage architecture for each query.

[5]Projection does not change number of tuples but number of columns per tuple.

3 Workload Patterns

To select the optimal storage architecture, we have to analyze a given workload; therefore, we need workload-statistic representations. We decompose the workload to aggregate, to process, as well as to administrate the extracted statistics. In the following, we 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. Figure 1 illustrates the procedure of our decision process regarding the storage-architecture selection. In the following, we outline the design of our pattern framework.

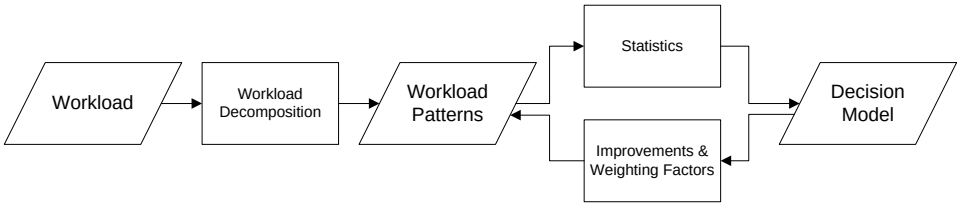

Figure 1: Workflow of the storage-architecture decision process.

3.1 Pattern Classes

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. In our approach, statistic representation is not limited to query-wise nor to overall point of view for analyses. Hence, we can determine where the majority of cost 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). 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 sort accelerates speed to find duplicates. Herein, we only consider explicit sort/order operations caused by workload or user.

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

Projection: Projection returns a subset of tuple attribute values and causes (normally) no additional cost 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. Operations which we assign to this pattern process data 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 in column stores and single column processing of the herein assigned operations, column stores perform well on aggregations (cf. Section 2). Hence, 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 computes 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 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 operator: The cube operator computes all feasible combinations 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 (i.e., two complete processing cycles).

Third, the ***join pattern*** matches all join operations of a workload. Join operations are costly tasks for DBMSs. We propose this pattern to show different join techniques between column and row stores (e.g., join processing on compressed columns or on bitmaps). Within this pattern, we evaluate 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 [Aba08, Lüb08]).

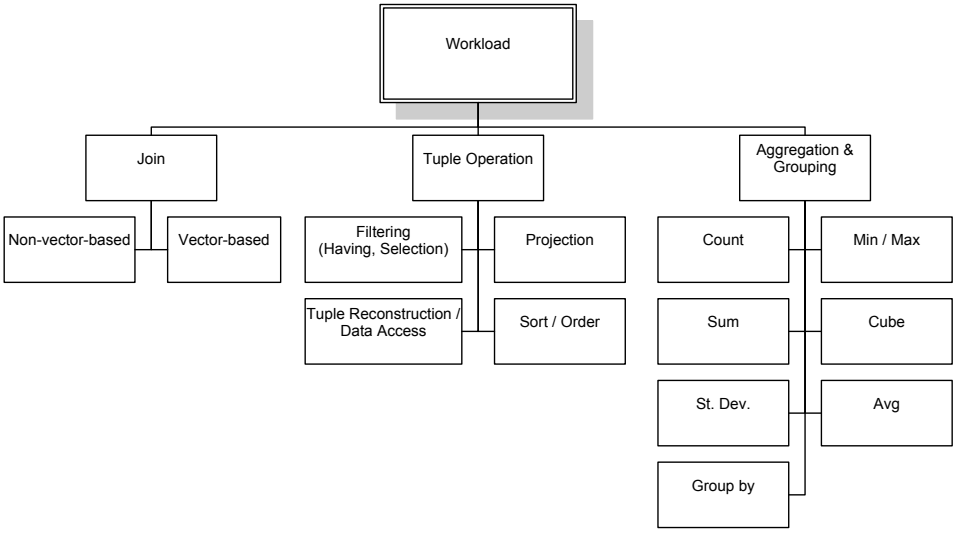

Figure 2: Workload patterns based on operations.

Non-vector-based: This pattern matches "classic" join techniques (from row stores[6], e.g., nested loop or merge join) 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 processing on bit-vectors/bitmaps is a distinction between an amount of join techniques. Hence, we assume that there is no necessity to map each join technique into its own sub-pattern. Figure 2 shows all introduced patterns and their relation to each other.

3.2 Dependencies between Patterns

Database operations are not always independent from each other. We 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 have to be implemented to further decompose join operations and gather necessary statistics, thus administrative cost for tuning will be noticeably increased. To a side-effect, each DBMSs have to be extended with this new system-specific decomposition and the comparison of join techniques belonging to different architectures are no longer possible (system-independence is lost again).

[6]Some column stores also support these join techniques (especially if tuple-oriented query processor is used).

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 sorting is caused by the optimizer (e.g., for sort-merge join or duplicate elimination). Therefore, we assign all cost of grouping to the GROUP BY (or CUBE) pattern including the sort cost to sustain comparability.

Third, tuple reconstruction is part of several operations for column stores. We add this cost to the tuple operation pattern and maintain 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 cost affects the performance of existing systems as well as the comparability of performance issues between the architectures according to certain workload parts. These impacts disadvantageously affect the usability of our pattern framework.

4 Query Decomposition

In this section, we introduce our approach to decompose the workload. 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 result in more accurate recommendations.

4.1 Query Plans

A workload decomposition based on database operations is necessary to select the optimal storage architecture (cf. Section 2). Therefore, we use query plans [ABC+76] which exist in each relational DBMS. On the one hand, we reuse database functionality and avoid new computation cost for optimization. On the other hand, we make use of system optimizer estimations that are necessary for physical database design [FST88].

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

Listing 5: Example SQL query (14-1) [Ora10a]

Based on query plans, we collect statistics directly from a DBMS and use the optimizer cost estimations. The example in Listing 5 shows an SQL query and we transform this to a query plan in Table 2 [Ora10a]. Table 2 already offers some statistics such as number of rows, accessed bytes by the operation, or cost. Nevertheless, Table 2 shows only an excerpt of gathered statistics. All possible values

for query plan statistics can be found in [Ora10b, 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 cost and/or I/O cost[7].

In addition to performance evaluation by several estimated cost, we gather further statistics from query plans which influence performance of an operation on a certain architecture (e.g., cardinality of attributes). For column stores, the operation cardinality indirectly affects 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 (e.g., compute the selectivity of attributes) to estimate influences of data itself on the performance.

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 2: Textual query plan of SQL example (14-1) [Ora10a]

4.2 From Query Plans to Workload Patterns

In order to benefit from the collected statistics, we map them to our workload patterns. We use a second example [Ora10c] (Listing 6 and Table 3) to simulate a minimum workload instead of a single query. In the following, we illustrate the mapping approach by using the examples in Listing 5 and 6. In our name convention, we define a unique number[8] that identifies a query within our mapping algorithm. Furthermore, we reuse the operation IDs from query plans (Table 2 and 3) in the second hierarchy level to identify operations within queries (i.e., 2.6 represents the second query (cf. Listing 6) and its HASH JOIN (cf. Table 3)). In the following, we refer the CPU cost from Table 2 and 3.

```
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 6: Example SQL query (11-9) [Ora10c]

The first query (Listing 5) is decomposed into four patterns. First, we see the data access operation of `employees (ID 3)` and `department (ID 4)` tables in the corresponding query plan in Table 2. The total cost for the data access operations is `5`. Second, the join operation `(ID 2)` is executed with a hash-join algorithm. Due to bottom-up summation in the given query plans, the hash-join cost is only `1` because up to this point the cost (cf. Table 2) is `6` and the cost of its children is already `5` (i.e., cost of children `(5)` summed up with

[7] We receive query plans directly from DBMS optimizer (e.g., EXPLAIN PLAN) or use sample workloads with operation-type distribution and corresponding cost.

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

own cost of 1 results in total cost of 6). Third, the sort operation (ID 1) implements the ORDER BY statement with cost of 1. The total cost of all processed operations is 7. Fourth, the select statement (ID 0) represents the projection and causes no additional cost (remain 7). The identifiers from 1.0 to 1.4 represent all operations of the first query (Listing 5) in Figure 3.

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 3: Textual query plan of SQL example (11-9) [Ora10c]

We also decompose the second example (Listing 6) into four operation types (cf. Table 3). First, IDs 3, 7, and 8 represent data access operations and cause total cost of 14. Second, the optimizer estimates both hash joins (ID 2 and 6) with no (additional) cost because their cost is only composed by the summed cost of their children (ID 3, 4 and ID 7, 8). Third, the GROUP BY statement in Listing 6 is implemented by hash-based grouping operations (ID 1 and ID 5). The cost of each HASH GROUP BY is 1 and the total cost of this operation type is 2. Fourth, the projection (ID 0) and the sum operation represented by select statement causes again no additional cost[9]. If the sum operation causes cost then it is represented by a separate operation (ID). We also classify the view (ID 4) as projection. If a view implements joins (or other complex operations) they are separately distinguished in the query plan. The identifiers from 2.0 to 2.8 represent all operations of the second query (Listing 6) in Figure 3.

In the following, we summarize single operations of similar types (five for example query two). We list the five operation types and assign them to the workload patterns and their sub-patterns that we introduce in Section 3. 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, 2.0, and 2.4) 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 3 whereby we only show ID and cost of each operation for reasons of readability.

However, we are able to compare performance of operations as well as to describe the behavior of operations on different architectures with the stored information in workload patterns. Therefore, we use (a combination of) different metrics in our decision models like CPU or I/O cost to evaluate the performance with respect to certain cost criteria (e.g., minimal I/O cost). We derive heuristics and rules for the design process of relational

[9] We state that the sum operation is already processed on-the-fly while grouping.

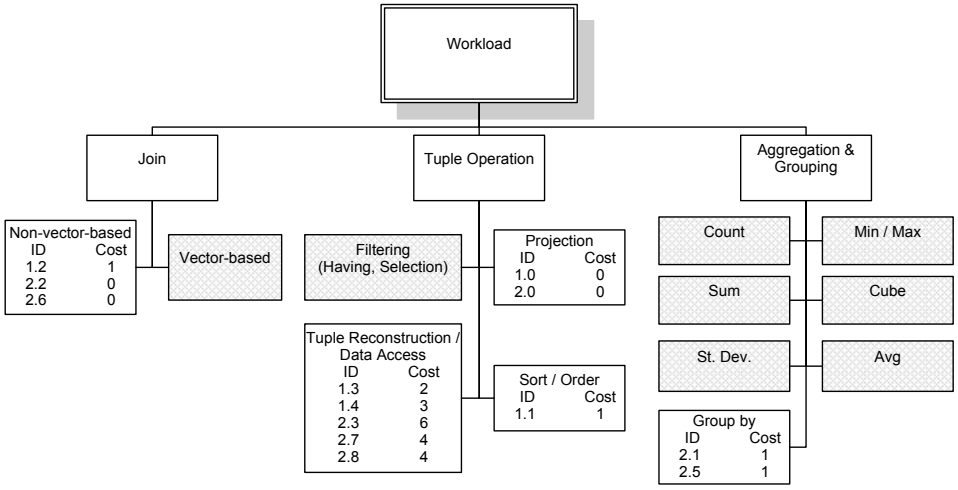

Figure 3: Workload patterns with cost of operations for the row store example workload

databases from operations behavior evaluated by statistics. Heuristics and rules further improve our decision models (i.e., reduce complexity of design decision process and/or give previews for design decision without holistic workload computation). Furthermore, we state that our design decision (process) is not static because we can periodically repeat the process to react on workload changes. We suggest an integration with self-tuning/alerter tools [CN07, SGS03] that continuously monitor the workload and update statistics. Due to the system-independent and transparent framework design, we are also able to use already extracted (or aggregated) data as well as estimated values (i.e., the statistics do not have to be extracted from existing systems). Furthermore, our approach is transparent to any workload type. We can evaluate the performance for on-line analytical processing (OLAP) and on-line transactional processing (OLTP) workloads just as mixed workloads with OLAP and OLTP parts. The transparency of our approach is necessary to also consider new requirements (e.g., real-time DWH) for database and data-warehouse systems.

4.3 Representation for Column Stores

For our approach, we do not need a separate decomposition algorithm for column stores (i.e., the query plan operations of column stores can also be mapped to our workload patterns) because only the naming in column stores for operations differ from the typical naming in row stores but the abstracted functionality is equal. Representatively, we illustrate the mapping of C-Store/Vertica query plan operations introduced in [SAB$^+$05] and map these operations to our workload patterns as follows:

Decompress: Decompress is mapped to our data access pattern. This operation decompresses data for subsequent operations in the query plan that cannot be processed on

compressed data (cf. [Aba08]).

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 our filtering pattern.

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

Project: Projection is equivalent to the projection of relational algebra. Thus, we map this operation to our projection pattern.

Sort: This operation sorts 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 map this operation to our sort/order pattern.

Aggregation operators: These operations compute aggregations and groupings equivalent to SQL [Aba08], thus we directly map these operations to the corresponding sub-pattern in our 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 emerges through creation of join indexes and C-Store projections in several orders. This operation is used for joins, thus we map its cost to our join pattern.

Join: We map this operation to the join pattern and distinguish two join types. First, if the 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 on columns that are needed to evaluate the join predicate. The join result is only a set of pairs of positions in the input columns [Aba08]. 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 our 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 (e.g., tuple reconstruction for column stores) for hybrid DBMSs to our pattern. For a future (relational) hybrid storage architecture, such an operation could be necessary to map the cost for conversions between row- and column-oriented structures and vice versa.

Threats to validity. We know that hundreds of relational DBMSs exist. We state that we cannot test all existing systems to claim generality of the approach nor can we formally prove all different implementations of general concepts based on the relational data model. Nevertheless, each relational DBMS is referable to the relational data model as well as relational algebra operations are based on the relational data model. The relational data model serves in this sense as an interface between implementation of database operations and general concept concerning the relational data model, thus we can map each database operation of relational DBMSs to our framework. Furthermore, we argue that (internal) database operations of each DBMS that use SQL can be mapped to relational algebra operations.

5 Evaluation

We decide to simulate the workload with the standardized TPC-H benchmark (2.8.0, scale factor 1) to show the usability of our approach. We use the DBMSs Oracle 11gR2 Enterprise Edition and Infobright ICE 3.3.1 for our experiments. 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[10] TPC-H queries namely `Q2, Q6,` and `Q14` to the workload patterns, see Figure 4. The results for the remaining queries can be found in Appendix A[11].

The query structure, syntax, and execution time are not sufficient to estimate the query-performance behavior on different storage architectures. We introduce an approach based on database operations that provides analyses to find long running operations (bottlenecks). Moreover, we want to figure out reasons for bad (or good) performance behavior of operations in DBMSs, thus we have to use additional metrics. We select the I/O cost[12] to compare DBMSs and summarize the optimizer output in Table 4. We state that I/O cost is a reasonable cost metric but not sufficient to select the optimal storage architecture. We will show this effect for I/O cost with a negation example in the following. 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 4), thus the root operation of TPC-H query `Q2` has the `ID 2.0` in Figure 4. All values in Table 4 are given in Kbytes. The given values are input cost of each operation except the table access cost because no information on input cost to table access operations are available. Note, the granularity of Oracle's cost measurements is on the byte level whereas the measurements of ICE are on the data pack (65K) level. Nevertheless, we used the default data block size 8kbytes in our Oracle installation; that is the smallest accessible unit.

In Figure 4, we present the workload patterns with I/O cost of the corresponding TPC-H queries. As mentioned before, the projection operation causes no additional cost. Hence, the I/O cost in Table 4 and Figure 4 represent the size of final results. The stored infor-

[10] The queries show typical results for the TPC-Benchmark in our test environment.
[11] Please cf. [Tra10] for the complete schema and query description.
[12] I/O cost is a best practice cost metric.

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

Table 4: Accessed Kbytes by query operations of TPC-H query Q2, Q6, and Q14.

mation 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 compute I/O cost per query in Kbytes. For the three selected queries, all results and intermediate results are smaller than the available main memory, thus no data has to be reread subsequently. We suppose, the DBMS with minimal I/O cost performs best (as we mentioned before, I/O cost is a good cost metric). Oracle reads 1452.133 Kbytes for query `Q2` and takes 8.14 seconds. ICE needs 41 seconds and accesses 2340 Kbytes. The results for `Q2` fulfill our assumption. Our assumption is also 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 have previously shown that I/O cost alone is not a sufficient metric to estimate the behavior of database operations and further, we suggest that each single cost metric is not sufficient. 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 (re-)read stepwise[13]. We assume that sets of cost metrics are needed to sufficiently evaluate the behavior of database operations. Therefore, one needs tool support as we propose in this paper.

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

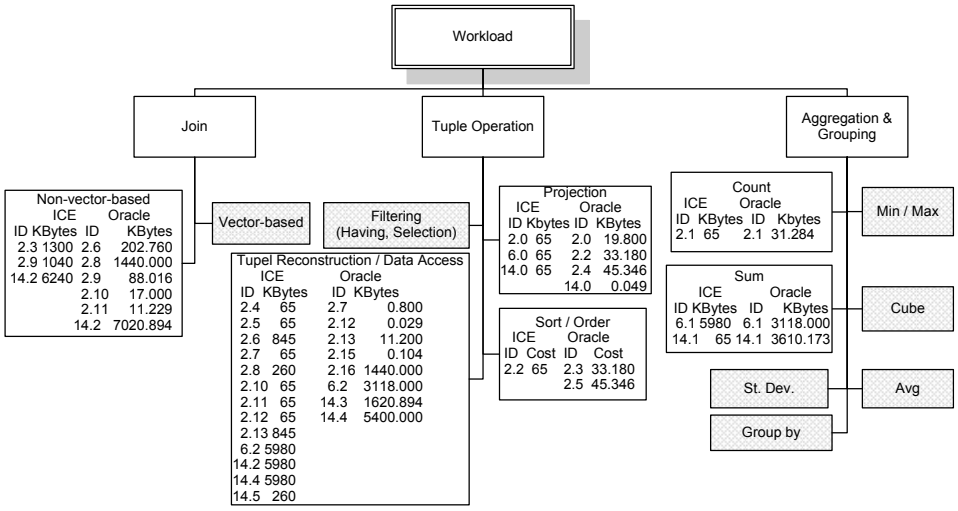

Figure 4: Workload graph with mapped I/O cost of TPC-H query Q2, Q6, and Q14.

We also want to evaluate our approach with the column-store solutions that use cost-based optimizer, thus we are able to receive more expressive results. We requested the permission to use such systems for our evaluation but until now the decision is pending. Meanwhile, we change our system setup (from MySQL 5.1.37 to Oracle 11gR2 and ICE 3.2.2 to ICE 3.3.1) due to two issues. First, we want to show that our results are not only valid for MySQL but also for DBMSs (in our case Oracle) that are capable for DWHs in practice. Second, ICE 3.2.2 had some issues while processing subqueries and/or nested queries which are referable to the underlying MySQL-kernel[14]. These issues are fixed in ICE 3.3.1 which is the current version as we redo our experiments.

6 Related Work

Several column stores are proposed for OLAP applications [Aba08, LLR06, SWES08, ZBNH05]. However, 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. [AMH08] compare row and column store with respect to performance on the star-schema benchmark. They simulate the 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. Thereby, Abadi et al. use a classical DWH benchmark that does not consider new requirements in this domain like dimension updates or real-time DWH. In this paper, we do not directly compare optimization techniques of row and column stores. Instead, we propose a frame-

[14]This is another argument for not using MySQL because no information was available in which version fixes will be implemented

work to detect strengths and weaknesses of row and column-oriented architecture with respect to the performance for given workloads. We do not discuss earlier approaches like DSM [CK85], hybrid NSM/DSM schemes [CY90], or PAX [ADHS01] because the differences to state-of-the-art column stores have been already discussed (e.g., Harizopoulus et al. [HLAM06]).

There are systems available which attempt to fill the gap between column and row stores. C-Store [Aba08] uses two distinct storage areas to overcome update problems of column stores. A related approach brings together a column store approach and the typical row-store domain of OLTP data [SBKZ08]. However, we do not develop hybrid solutions that attempt to fill this gap for now. Our approach recommends the optimal architecture for a certain application.

There exist a number of design advisors which are related to our work (e.g., IBM DB2 Configuration Advisor [KLS+03]). The IBM Configuration Advisor recommends pre-configurations for databases. Zilio et al. [ZRL+04, ZZL+04] introduce an approach that collects statistics like our approach directly from DBMSs. The statistics are used to advise index and materialized view configurations. Similarly, Bruno and Chaudhuri [BC06, BC07] 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 according to their architecture. Additionally, our approach aims at architectural decisions contrary to the mentioned approaches which 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 [Ing09]. In cooperation with Vectorwise, Ingres is developing a new storage manager ColumnBM for the new Ingres/Vectorwise. However, the integration of the new architecture into the existing environment remains unclear [Ing09].

7 Conclusion

In recent years, column stores have shown good results for DWH applications and often outperformed row stores. However, new requirements (cf. Section 1) arise in the DWH domain that cannot be satisfied only by column stores. The new requirements also demand for row-store functionality (e.g., real-time DWHs need (sufficient) quick update processing). Thereby, the complexity of design process increases because we have to choose the optimal architecture for given applications. We show with an experiment that workload analyses based on query structure and syntax are not sufficient to select the optimal storage architecture. Consequently, we propose a new approach based on database operations. We introduce workload patterns which contain all workload information beyond the architectures (e.g., statistics and operation cost). We also present a workload decomposition approach based on existing database functionality that maps operations of a given workload to our workload patterns. We illustrate 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 state that our presented approach is transparent to any workload and any storage architecture based on the relational data model. In our evaluation, we prove the usability of our approach. Additionally, we demonstrate the comparability of different systems using different architectures even if systems provide different information with respect to their query execution. We also state that the richness of detail of system optimizer in closed source systems is higher than in open source systems. Decision processes can be periodically repeated to monitor if workload changes (e.g., new sample workloads or new applications) effect the previous design decision, thus the storage architecture selection is not static. We see two practical implications for our approach. First, we use our approach to select the most suitable relational storage architecture during system design as we use sample workloads for the prediction. Second, we implement our approach to work on existing systems as alerter (or monitor) that analyzes system workload continuously to inform the (database) administrator if the current workload is better supported by another storage architecture. 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 implement a new DBMS to export periodically statistics and operation cost which map to our workload patterns. This way, we will not affect performance of analyzed systems by prediction computation. Second, we adapt existing approaches [BC06, LGB09] to automatically collect and map statistics to workload patterns which we can directly transfer into a graph structure (query graph model). With both strategies, we are able to store the necessary statistics for storage architecture decision without running systems. Additionally, we will present studies for storage architecture decision based on aggregated or estimated values (i.e., not directly from DBMS/optimizer). To develop our decision model based on our workload patterns, we will perform detailed studies on OLAP, OTLP, and mixed workloads. We will use existing systems to gather expressive values for predictions. Finally, we will derive physical design heuristics and rules from our gathered values to extend our decision model.

8 Acknowledgment

We thank Martin Kuhlemann, Martin Schäler, and Ingolf Geist for helpful discussions and comments on earlier drafts of this paper.

References

[Aba08] Daniel J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Cambridge, MA, USA, 2008. Adviser: Madden, Samuel.

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and

Vera Watson. System R: Relational Approach to Database Management. *ACM TODS*, 1(2):97–137, 1976.

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB '01*, pages 169–180. Morgan Kaufmann Publishers Inc., 2001.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD '08*, pages 967–980, 2008.

[BC06] Nicolas Bruno and Surajit Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB '06*, pages 499–510. VLDB Endowment, 2006.

[BC07] Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE '07*, pages 826–835, 2007.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD '85*, pages 268–279. ACM, 1985.

[CN98] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin "What-if" index analysis utility. In *SIGMOD '98*, pages 367–378. ACM, 1998.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *VLDB '07*, pages 3–14. VLDB Endowment, 2007.

[CY90] Douglas W. Cornell and Philip S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Softw. Eng.*, 16(2):248–258, 1990.

[FST88] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical Database Design for Relational Databases. *ACM TODS*, 13(1):91–128, 1988.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06*, pages 487–498. VLDB Endowment, 2006.

[IBM06] IBM. An Architectural Blueprint for Autonomic Computing. White Paper, June 2006. Fourth Edition, IBM Corporation.

[Inf08] Infobright Inc. Infobright Community Edition. White Paper, September 2008. ICE 3.2.2 documentation pack.

[Ing09] Ingres/Vectorwise. Ingres/VectorWise Sneak Preview on the Intel Xeon Processor 5500 series-based Platform. White Paper, September 2009.

[KLS+03] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In *BTW '03*, pages 620–629, 2003.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE '11*, pages 195–206, 2011.

[LGB09] Andreas Lübcke, Ingolf Geist, and Ronny Bubke. Dynamic Construction and Administration of the Workload Graph for Materialized Views Selection. *Int. Journal of Information Studies*, 1(3):172–181, 2009.

[LLR06] Thomas Legler, Wolfgang Lehner, and Andrew Ross. Data mining with the SAP NetWeaver BI Accelerator. In *VLDB '06*, pages 1059–1068. VLDB Endowment, 2006.

[Lüb08] Andreas 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.

[Lüb10] Andreas Lübcke. Challenges in Workload Analyses for Column and Row Stores. In *22nd Workshop "Grundlagen von Datenbanken"*, volume 581. CEUR-WS.org, 2010.

[Man04] Andreas S. Maniatis. The Case for Mobile OLAP. In *EDBT Workshops*, pages 405–414, 2004.

[OJP+11] Kerry Osborne, Randy Johnson, Tanel Pder, Kerry Osborne, Randy Johnson, and Tanel Pder. Hybrid Columnar Compression. In *Expert Oracle Exadata*, pages 65–104. Apress, 2011. 10.1007/978-1-4302-3393-0_3.

[Ora10a] Oracle Corp. Oracle Database Concepts 11g Release (11.2). 14 Memory Architecture (Part Number E10713-05), March 2010.

[Ora10b] Oracle Corp. Oracle Performance Tuning Guide 11g Release (11.2). 12 Using EXPLAIN PLAN (Part Number E10821-05), March 2010.

[Ora10c] Oracle Corp. Oracle Performance Tuning Guide 11g Release (11.2). 11 The Query Optimizer (Part Number E10821-05), March 2010.

[Ora11] Oracle Corp. Hybrid Columnar Compression (HCC) on Exadata. White Paper, October 2011.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD '09*, pages 1–2. ACM, 2009.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB '05*, pages 553–564. VLDB Endowment, 2005.

[SB08] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *IDEAS '08*, pages 49–58, 2008.

[SBKZ08] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *BIRTE '08*, 2008.

[SGS03] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: Continuous Query-driven Index Tuning. In *VLDB'04*, pages 1129–1132. VLDB Endowment, 2003.

[SWES08] Dominik Ślęzak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.

[THC79] Martin J. Turner, Rex Hammond, and Paul Cotton. A DBMS for large statistical databases. In *VLDB '79*, pages 319–327. VLDB Endowment, 1979.

[Tra10] Transaction Processing Performance Council. TPC BENCHMARK^{TM} H. White Paper, April 2010. Decision Support Standard Specification, Revision 2.11.0.

[VMRC04] Alejandro A. Vaisman, Alberto O. Mendelzon, Walter Ruaro, and Sergio G. Cymerman. Supporting dimension updates in an OLAP server. *Information Systems*, 29(2):165–185, 2004.

[WHMZ94] Gerhard Weikum, Christof Hasse, Alex Moenkeberg, and Peter Zabback. The COMFORT Automatic Tuning Project, Invited Project Review. *Information Systems*, 19(5):381–432, 1994.

[WKKS99] Gerhard Weikum, Arnd Christian König, Achim Kraiss, and Markus Sinnwell. Towards Self-Tuning Memory Management for Data Servers. *IEEE Data Eng. Bulletin*, 22:3–11, 1999.

[ZAL08] Youchan Zhu, Lei An, and Shuangxi Liu. Data Updating and Query in Real-Time Data Warehouse System. In *CSSE '08*, pages 1295–1297, 2008.

[ZBNH05] Marcin Zukowski, Peter A. Boncz, Nils Nes, and Sándor Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bulletin*, 28(2):17–22, 2005.

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB '04*, pages 1087–1097. VLDB Endowment, 2004.

[ZZL+04] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *ICAC '04*, pages 180–188, 2004.

A TPC-H: Query-wise Summary

In the following tables (Table 5 to 26), we present our results for the complete TPC-H benchmark [Tra10] (2.8.0, scale factor 1). We present our results query-wise for both systems (Oracle vs. ICE). The value for each pattern is the summation of operation cost (cf. Section 3).

Workload Pattern	**Query Q1**			
	Oracle (22.82sec)		**ICE** (25sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	5789.7K	156321.522	6012.7K	5980
Group By	5789.7K	156321.5K	5916.6K	5980
Projection	5	0.135	4	65

Table 5: Accessed data of tpc-h query Q1 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q2			
	Oracle (8.14)		**ICE** (41sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	10.2K	1452.133	1048.6K	1040
Non-vector	12K	1759.005	4521.9K	4455
Tuple Reconstruction			162.3K	195
Sort	316	78.526	460	65
Count	158	31.284	460	65
Projection	416	98.326	100	65

Table 6: Accessed data of tpc-h query Q2 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q3			
	Oracle (30.97sec)		**ICE** (3sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	3984.7K	89977.195	7712K	7670
Non-vector	4204.7K	98117.269	9240.6K	9165
Tuple Reconstruction			177.6K	195
Group By	501.7K	30102.72	30.5K	65
Sort	501.7K	30102.72	11.6K	65
Count	501.7K	24082.172	11.6K	65
Projection	10	0.48	10	65

Table 7: Accessed data of tpc-h query Q3 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q4			
	Oracle (27.29sec)		**ICE** (2min 33sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	3100.9K	68386.986	7515.9K	7475
Non-vector	3100.9K	68386.986		
Filtering			9019.1K	8970
Group By	58K	3016.028	52.5K	1495
Projection	5	260	5	65

Table 8: Accessed data of tpc-h query Q4 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q5			
	Oracle (32.66sec)		**ICE** (4sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	6389.5K	119631.12	7908.1K	7865
Non-vector	7777.1K	243824.536	14378.3K	14300
Tuple Reconstruction			1392.5K	1430
Group By	7.4K	844.854	7243	5980
Sort	50	3.65	5	65
Projection	25	2.85	5	65

Table 9: Accessed data of tpc-h query Q5 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q6			
	Oracle (22.24sec)		**ICE** (2sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	155.9K	3118	18087.9K	5980
Sum	155.9K	3118	114.2K	5980
Projection	1	0.02	1	65

Table 10: Accessed data of tpc-h query Q6 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q7			
	Oracle (29.48sec)		**ICE** (4sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	3249.8K	63543.499	7908.1K	7865
Non-vector	3678.7K	102714.005	15685.4K	15600
Tuple Reconstruction			443.8K	455
Filtering			220.8K	260
Group By	5.6k	617.493	5.9K	65
Projection	1.5K	152.504	4	65

Table 11: Accessed data of tpc-h query Q7 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q8			
	Oracle (29.95sec)		**ICE** (3sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	6619.1K	154010.467	8234.9K	8190
Non-vector	6700.3K	159438.031	22090.3K	21970
Tuple Reconstruction			494K	520
Group By	2446	364.454	2603	65
Projection	732	109.068	2	65

Table 12: Accessed data of tpc-h query Q8 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q9			
	Oracle (37.05sec)		**ICE** (7sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	7511.2K	183432.882	8757.7K	8710
Non-vector	9212.7K	277307.627	33723.7K	33540
Tuple Reconstruction			1743.8K	1755
Group By	297.1K	38924.947	348.8K	390
Projection	42.5K	5571.823	175	65

Table 13: Accessed data of tpc-h query Q9 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q10			
	Oracle (29.18sec)		**ICE** (10sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	2305.6K	75690.021	7973.4K	7930
Non-vector	2402.9K	79190.306	9476.6K	9425
Group By	97.21K	20900.15	114.7K	130
Sort	97.21K	20900.15	37.9K	65
Count	97.21K	17400.59	37.9K	65
Projection	97.23K	20903.73	20	65

Table 14: Accessed data of tpc-h query Q10 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q11			
	Oracle (5.06sec)		**ICE** (1sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	810K	14470.029	1960.7K	1950
Non-vector	1610K	52070.029	2091.4K	2080
Tuple Reconstruction			64.2K	65
Group By	32K	1728	31.7K	65
Sort	832K	15648	29.8K	65
Projection	64K	2496	752	65

Table 15: Accessed data of tpc-h query Q11 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q12			
	Oracle (26.85sec)		**ICE**(6sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	1511.7K	33479.167	7515.9K	7475
Non-vector	1511.7K	33479.167	7515.9K	7475
Tuple Reconstruction			30.9K	65
Group By	11.7K	726.281	30.9K	65
Projection	2	0.126	2	65

Table 16: Accessed data of tpc-h query Q12 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q13			
	Oracle (5.24sec)		**ICE** (22sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	1575K	87625	1699.3K	1690
Non-vector	1575K	87625	1699.3K	1690
Tuple Reconstruction			1483.9K	1495
Group By	201.4K	7956.248	1533.9K	195
Sort	100.7K	1309.256	150k	195
Projection	201.4K	2618.512	42	65

Table 17: Accessed data of tpc-h query Q13 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q14			
	Oracle (22.55sec)		**ICE** (3sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	273.7K	7020.894	6274.1K	6240
Non-vector	273.7K	7020.894	6274.1K	6240
Tuple Reconstruction			75.9K	5980
Sum	73.7K	3610.173	75.9K	5980
Projection	1	0.049	1	65

Table 18: Accessed data of tpc-h query Q14 - Number of rows and I/O cost in Kbytes.

Workload Pattern	**Query Q15**			
	Oracle (2.18sec)		**ICE** (2sec + 1sec for View creation)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	228.7K	5311.797	6078.1K	6045
Non-vector	20K	1020	130.7K	130
Tuple Reconstruction			1	65
Group By	218.7K	4591.797	451.9K	520
Sort	20K	1020		
Projection	20K	1320	1	65

Table 19: Accessed data of tpc-h query Q15 - Number of rows and I/O cost in Kbytes. Note, ICE needs committed view creation before querying this one.

Workload Pattern	**Query Q16**			
	Oracle (3.93sec)		**ICE** (1sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	800.5K	7234	1111.1K	1105
Non-vector	951.9K	14528.665	1111.1K	1105
Tuple Reconstruction			118.3K	65
Filtering			118.3K	130
Group By	129.8K	14284.704	118.3K	130
Sort	15K	735	18.3K	65
Projection	129.8K	14284.704	18.3K	65

Table 20: Accessed data of tpc-h query Q16 - Number of rows and I/O cost in Kbytes.

Workload Pattern	**Query Q17**			
	Oracle (24.06sec)		**ICE** (1sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	6001.4K	84022.41	6274.2K	6240
Non-vector	6001.4K	84022.41	6274.2K	6240
Tuple Reconstruction			587	65
Sort	5.9K	243.663		
Sum	5.9K	77.259	587	65
Projection	5.9K	243.676	1	65

Table 21: Accessed data of tpc-h query Q17 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q18			
	Oracle (31.56sec)		**ICE** (9sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	6001.2K	54011.02	7712K	7670
Non-vector	32	0.425	9215.2K	9165
Tuple Reconstruction			798	65
Group By	6001.2K	54010.935	399	65
Sort	5	0.265	57	65
Count	4	0.3	57	65
Projection	9	0.562	57	65

Table 22: Accessed data of tpc-h query Q18 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q19			
	Oracle (33.27sec)		**ICE** (11sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	239.1K	12899.256	18822.5K	18720
Non-vector	239.1K	12899.256	6274.2K	6240
Tuple Reconstruction			121	65
Filtering			96	130
Sum	357	29.988	121	65
Projection	1	0.084	1	65

Table 23: Accessed data of tpc-h query Q19 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q20			
	Oracle (31.56sec)		**ICE** (9min 43sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	869.9K	17397.8	7123.8K	7085
Non-vector	869.9K	17398.274	130.7K	130
Group By	4	0.388		
Sort	1	0.092	204	65
Projection	2	0.096	204	65

Table 24: Accessed data of tpc-h query Q20 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q21			
	Oracle (1min 2.48sec)		**ICE** (6h 3min 58sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	6511.2K	166481.649	13659.4K	13585
Non-vector	13218.7K	547594.066	15162.6K	15080
Tuple Reconstruction			12.9K	65
Group By	177.1K	32685.07	4141	65
Sort	3043K	115632.914	411	65
Count	100	4	411	65
Projection	300	23.6	100	65

Table 25: Accessed data of tpc-h query Q21 - Number of rows and I/O cost in Kbytes.

Workload Pattern	Query Q22			
	Oracle (5.34sec)		**ICE** (1sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	1509.8K	7717.578	392.1K	390
Non-vector	1500.5K	7513.77		
Filtering			588.2K	585
Group By	5	0.160	6384	65
Count	9.3K	203.808	7	65
Projection	1	0.032	7	65

Table 26: Accessed data of tpc-h query Q22 - Number of rows and I/O cost in Kbytes.