An Approach for Dynamic Materialization of Aggregates in Mixed Application Workloads

Stephan Müller
Hasso Plattner Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
stephan.mueller@hpi.uni-potsdam.de

Carsten Meyer
Hasso Plattner Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
carsten.meyer@hpi.uni-potsdam.de

Hasso Plattner
Hasso Plattner Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
hasso.plattner@hpi.uni-potsdam.de

***Abstract*—Aggregate queries are one of the most resource intensive operations for databases systems. Despite scanning and processing large data sets, they only return relatively small outputs, making them predestined for reuse in future queries. The challenge is to manage the infinite number of possible data areas that can be selected for caching, identify the relevant ones and materialize the corresponding aggregates on a reusable level of granularity. While analytical database systems often save materialized data in cubes of aggregated chunks this is not feasible in systems with transactional insert and change processes. We present the concept *dynamic materialized aggregate views (DMAV)* that materializes aggregates based on mixed application workloads. Using control tables, aggregate query structures as well as selected data areas are captured and kept track of. We evaluate data access characteristics and only materialize aggregates that are read often but at the same time have only few updates resulting in low aggregates maintenance costs.**

***Index Terms*—caching, data aggregation, dynamic view materialization, mixed-workload, OLAP, OLTP.**

I. Introduction

In the last decades, enterprise data management systems have evolved in two directions: transactional (OLTP) and analytical (OLAP) systems. Queries with aggregation functions that scan, read, and finally calculate large amounts of data are very resource intensive for both systems [1], [2], resulting in system-specific work-arounds: In OLTP systems, the applications maintain tables that contain materialized aggregates and deliver by magnitudes faster access to the desired abstraction level compared to aggregating on the original base table. On the downside, every change on the base data must be propagated to the materialized table. OLAP systems on the other hand often contain data schemas that are an aggregated abstraction of the underlying transactional data. The selection and definition of views is either done manually by an administrator or based on sophisticated materialized view selection algorithms. This enables fast response times for pre-defined queries. However, this setting inhibits flexible, ad-hoc queries, hence realistic analytical scenarios.

To overcome these and other drawbacks of the separation and to enable analytical processing directly on the transactional data, in-memory databases such as HYRISE [3] or SAP HANA [4] have evolved in academia and industry. The workload of the combined system is called *mixed workload* and consists of transactional as well as analytical data access patterns [5], [6]. Application scenarios such as dunning, cross selling and availability to promise [7] are an example for such workloads.

Although resource-intensive operations such as aggregations can now be processed directly on the base data and provide acceptable response times, reading a pre-aggregated result is always more efficient, especially in multi-user scenarios and when the same or similar queries are executed repeatedly. When considering the materialized aggregate maintenance costs on the other hand, an on-the-fly aggregation strategy can be more efficient. Nevertheless, the application should not deal with the maintenance of materialized aggregate tables and operate on the base data directly. We believe that the decision of which parts of a table should be materialized as an aggregate should be made in the database layer and not within the application for two reasons: First, this significantly simplifies application development as the aggregations are executed transparently. Second, the materialization of aggregates should be based on the actual workload and not on a static basis.

In this paper we introduce the concept of *dynamic materialized aggregate views*. This concept selectively materializes aggregates, based on the characteristics of a mixed workload, independent from the application. To our best knowledge, there are no techniques available that allow materialization and reuse of generic aggregates in transactional databases which are based on dynamic changing data characteristics. Analyzing these workloads, one can find application-dependent data ranges that do not change. Financial statements of past periods, point of sales data or historic production orders are often described as *cold data*. Still, they can be relevant for analytical queries. This paper proposes to analyze a given mixed workload in order to find those aggregates that are frequently requested and are based on cold data. Stored in dedicated control tables, the database shall be able to keep track of those *hot aggregates*. Instead of calculating hot aggregates over and over again they are materialized and can then be used to answer matching queries faster while, at the same time reducing CPU and memory bandwidth allocation.

 ISBN: 978-1-61208-247-9

We contribute by providing a generic framework that identifies and characterizes aggregate queries in a mixed workload. We describe how historical data relevance and change frequency can be evaluated in order to find important data regions, so called hot spots. Our concept will be provided for dynamically materialized aggregates on database level, as opposed to application based hard coded materialized tables.

The remainder of the paper is structured as follows: Section II provides background information and evaluates related work on materialization in analytical databases, optimization techniques and the nature of aggregate functions. Section III introduces the analysis framework that is used to identify aggregate queries and extract their structure as well as tracking hot spot areas. The core concept of DMAVs is presented and explained using an example in Section IV. It will be shown how different aggregate queries respectively their structural characteristics are stored in a relation and used for view matching and materialization. The number of materialized aggregates will not only be limited to the relevant ones but will also consider changes in the transactional base data. Different ways of optimizing the definition of data areas (hot spots) will be presented in Section V. Finally, in Section VI we preliminarily evaluate our concept with a mixed workload.

II. Background and Related Work

In this section, we present background information for aggregation queries and evaluate related work in the areas of clustering hot and cold data, materialized view selection and maintenance, and cache replacement techniques.

A. Data aggregations

Aggregates are the result of a query that has an aggregation function f() applied to an attribute *A* of an arbitrary relation r. The granularity of aggregates depends on the cardinality of its grouping attributes *G*. The structure of an aggregate query consists of three main aspects:

1) The aggregate function and the attribute *A*
2) The group by clause and its fields *G*
3) The relation *r* and its data selection (where clause structure)

In relational algebra the aggregation operation over a schema (A1, A2, ... An) of relation r is written as follows: **G1, G2, ..., Gm g f1(A1), f2(A2), ..., fk(An') (r)**

Let us assume that we have a table named "Sales" with three columns, namely "Product_Id", "Year" and "Rev". We wish to find the maximum Rev of each Product_Id in Year = 2011. This is accomplished by:
Product_IdgMax(Rev)(Select * Sales where Year = 2011). The relation "r" (Select * Sales where Year = 2011) could also be a relation joined over multiple tables. The table of the attribute "A" is called the *base table* and all rows selected in relation "r" are the *base data* of the aggregates. A materialized aggregate view is defined by the grouping and selection of aggregates on the base data. This data is then materialized using another table in the database.

B. Clustering data in hot and cold areas

Categorizing data areas according to their usage is especially relevant in mixed workload databases. Cold data that is not accessed for updates or maybe even reads can be handled in other ways than hot data that is accessed more frequently. In [8] Funke et al. measure the *temperature* of data, based on memory page granularity. They store attributes column-wise, each attribute vector cut into chunks the size of a memory page. For every chunk they determine its usage from the number of page access but do not distinguish between read and write access. They categorize the usage into *hot*, *cooling*, *cold*, and *frozen* areas. In this paper, chunk based granularity for usage characteristics is not sufficient as hot spots must be described and captured on row level granularity.

C. Materialized view selection and maintenance

Materializing views in order to speed up the processing of queries is not new [9], [10]. Today, all major analytical database systems support the definition of de-normalized, pre-aggregated views [11], [12], [13] for resource-intensive queries. Although static view selection is a common approach in OLAP systems, it contradicts with the dynamic nature of decision support analysis as well as the diverse usage scenarios of enterprise standard software. Kotidis et al. show that the time frame needed to update those views becomes more and more the limiting factor, as the cost per disk decreases while the number of views increases [2]. In their paper they propose a concept for data warehouse systems called *DynaMat* that materializes results based on a goodness measure that is extracted from the workload demand. This measures the relevance of each materialized multi-dimension range fragments and limits the data amount to a certain disk space. In [14] Zhou et al. propose a different approach, but address the same problem. They use key parameters in the where-clauses and an associated control table to filter only the most relevant data. The values (filters) in the control table, dynamically change the number of materialized rows of the view.

This paper builds upon the idea of a workload demand [2] as well as on the concept of control tables [14]. Adapted to a mixed workload, the approach of this paper does not rely on predefined views. Instead, it extracts arbitrary aggregate queries and their structure from a given workload demand. Multi-parameter control tables are used to filter includable and excludable data areas eligible for materialization. This will ensure that only those aggregates are materialized that are selected frequently and do not have frequent updates in their base records, minimizing view maintenance costs.

In [15] Deshpande et al. propose a way to build aggregates based on chunks of aggregates. The idea of chunk based caching [16] requires a simple correspondence between chunks (called closure property) at different levels of aggregation. Without having defined hierarchies, respectively hierarchical dimensions that are used to select aggregates, there is no possibility to aggregate low level chunks up to high level chunks. However, materializing dynamic aggregates shall not

 ISBN: 978-1-61208-247-9

require hierarchical dimensions, but adapt to a mixed, changing workload. Still, in section V we will look into ways how simple hierarchies in dimensional- & fact- tables can be used to harmonize the definition of data areas.

The problem of view matching has already received considerable attention in research. Larson and Yang were the first to describe view matching algorithms for SPJ queries and views [9], [10]. In [12] Goldstein and Larson extended SPJ view matching for aggregate queries. They show conditions a view has to fulfill in order to compute a query. A general survey on view matching can be found in [13] by A. Y. Halevy. This paper will only explain those steps, additionally needed to identify aggregate queries that can be answered by a DMAV.

D. Cache replacement

In order to measure and manage the data area definitions (hot aggregates, hot change data) according to their usage and cost, there must be a replacement and admission mechanism for materialization filters in the control tables. In [17] Scheuermann et al. discuss a cache manager (LNC-RA) that employs cache replacement and admission policy that aims at maximizing the query execution cost savings. In comparison to a vanilla LRU algorithm, this one promises to be a good algorithm for managing control tables. Even though the control tables do not manage the caching of data pages but the materialization of aggregates, similar costing functions can be used. With a cost-based ranking, introduced in V-A filters in control table can be compared.

III. Analysis Framework

The framework that is used to analyze the workload is explained within this section. It is shown how the framework is used to identify aggregates including their structure as well as capture relevant data areas as the basis for DMAV. As a generic framework the basic idea is to analyze and identify arbitrary workload characteristics by parsing SQL statements being executed on a database. Today, all major databases (Oracle, DB2, SQL Server, MySQL) are able to trace SQL workloads. This paper shows algorithms that find and save characteristics as well as additional workload information, such as the number of executions, rows processed and the elapsed time in a fact table of an analytical star schema. The framework proposed in this paper was implemented as an database external Microsoft .Net application that can read SQL statements either from exported CSV files, or directly access specific log tables in the database. The application can parse those SQL statements and write commands (facts) into the star schema. As this paper provides a concept for materializing aggregates, the question how this functionality can be integrated into a DBMS is not addressed.

A. Commands

Commands are a substantial component of the analysis framework. Each command describes a certain characteristic of a database table or field that was identified by the parsing code. The used parser is able to distinguish between different

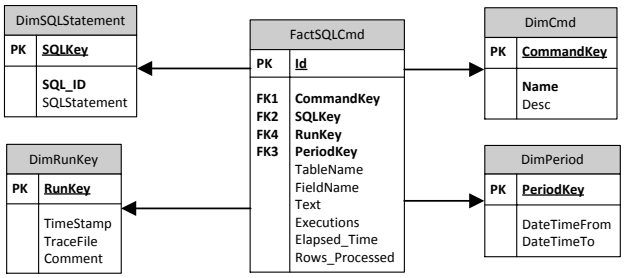

Fig. 1. Star Schema Analysis Framework

statement types (select, insert, delete, update, etc.) , detect projected columns, tables and join relations, predicates and group-by clauses. In general all aspects of an SPJG query can be identified and tracked. The characteristics parsed and needed for the the identification of dynamic, materialized aggregated view are defined as follows:

AGR_SUM SUM aggregates
AGR_AVG AVG aggregates
AGR_CNT COUNT aggregates
AGR_GRP GROUP BY table field
AGR_WCL WHERE clause with values
AGR_WCP WHERE clause w/o field values
UPD_WCL WHERE clause in update statements with values
UPD_WCP WHERE clause in update statements, w/o values
AGR_GCL GROUP BY clause in aggregate queries

B. Star schema

The analysis star schema as illustrated in Figure 1 allows to capture all the parsed information into relational tables. Next to the fact table there are four dimensional tables that describe each captured fact.

DimCmd All commands with their description and version
DimPeriod The time span information of a period
DimRun Holds information for each executed analysis run
DimSQLStatement Stores the actual SQL query information

The analyzer parses each query, looks for certain characteristics, and then creates a fact. The sum aggregate parsing code (AGR_SUM) for example creates a new fact every time it finds a sum aggregate function in a query, while AGR_GRP creates a fact for every group-by table field found in an aggregate function. The captured facts, provided by the workload, which are measurable, are:

TableName Name of the table
FieldName Name of the table field
Executions Number of executions
RowsExecuted Number of rows read, updated, etc.
ElapsedTime Total time in ms used for execution
Text Used to save string values (e.g. values of predicates)

C. Ways to mine those data

Once the workload is parsed and captured in the analysis schema, it is possible to query and mine this data according to certain questions:

 ISBN: 978-1-61208-247-9

TABLE I
WORKLOAD ANALYSIS

Field	Value
SQLKey	5832
Base table T_B	BSEG
View predicate P_V	BSEG.VBELN = VBRK.VBELN
Exec	64
Rows	658
Elapsed Time	87036ms
Aggregate A	AGR_SUM(BSEG.DMBTR)
GroupBy G	BSEG.PSWSL-&&-BSEG.GJAHR
Select predicates P_S	NE(BSEG.AUGDT)-&&-BT(VBRK.FKDAT)-&&–EQ(BSEG.KOART)-&&-EQ(BSEG.SHKZG)-&&-EQ(BSEG.BUKRS)
Filters (Select predicate values) P_{SV}	('0001-01-01','2001-11-01','2002-01-31','D', 'S','6000'), (...), (...)

- How often has which aggregate function been executed,
- On what tables do they base,
- What group-by fields have been used, and
- How do the where-clause predicates look like?

All these questions can be ranked by the provided metrics (executions, rows processed and elapsed time) and additionally joined as required using the SQLKey. In Table I it can be seen, how the identified commands are joined into a usable result set. The set is grouped on: T_B, P_V, A, G, and P_S. From that information, structural information as well the data areas (filters) can be extracted. Besides, all required infrastructure (view definitions, aggregate structure) for DMAVs are derived.

IV. DYNAMIC MATERIALIZED AGGREGATE VIEWS

The idea of DMAV is that aggregates of generic aggregate queries are selectively materialized, limited to hot spot data areas. Therefore a data structure keeping track of generic aggregate functions is needed. As there are no predefined aggregate views, even ad-hoc executed queries are persisted to a central *structure control table* T_S.

Additionally to this structure there are *data control tables* T_D for every base table that holds the hot spot data (as include and exclude filters), dynamically adjusted to the workload. Both structures, T_D and T_S are required to materialize aggregated results, based on the group-by and selection attributes of the aggregate query. They are also both used for view matching (see Section IV-F). Only when a query matches both structure (T_S) and data T_D, it can be answered by the corresponding V_{dA} view. Because V_{dA} contains (is grouped by) the selection attributes of the aggregate query, (see Section: IV-E3) it is possible for the aggregate query to directly select on the V_{dA} instead of the base relation. Because of this grouping the result of an aggregate query cannot be saved to the V_{dA} during runtime but is materialized asynchronously. The concept of DMAV is split into two phases. A periodically executed *update phase* (see Figure 3) that runs asynchronous to the actual database and a second phase (see Figure 2), which is active during query runtime (*online phase*). This separation provides the key requirements that the materialized aggregates can be invalidated and maintained when the base data changes, adapting to a changing workload.

The main task of the online phase is view matching. Every aggregate query is matched against T_S and T_D. When it hits a hot spot, this means the query can be answered by the DMAV V_{dA}. Otherwise the query has to be answered by the base tables. At the same time, unknown structures or newly selected data areas (filters) are added to T_S and to T_D. In both ways (hit or miss) the usage, cost, and the size of the result of a filter is captured in T_D. Thus, a changing workload is recognized as new structures and new filters during the online phase. However, as new aggregates are not directly materialized to the according V_{dA} view during runtime, those new filters are marked as invalid. During the online phase, every change statement to a base table is tracked and captured as an exclude filter in T_S. Additionally, all affected include-filters are invalidated (see: Table II).

Based on change metrics and the system load, the update phase of a T_D is triggered periodically. Those metrics shall not be considered in further detail. Still, they could be dependent on the number and usage of new filters and structures as well as available system resources. Obviously, every T_D table that is updated puts a load on the system and therefore should only be executed when necessary. However, every T_D can be updated separately. Hence, it is easy to scale with the workload demand of different base tables.

During the update phase a V_{dA} view gets materialized. Before that step, new filters in T_S and aggregate structures in T_D are evaluated and optimized. As can be seen in Figure 3 this includes four steps: cost ranking, variable distribution, time correlation, and structural harmonization. Cost ranking calculates a profit for each filter, based on usage statistics captured during online phase: *EXEC* (number of executions), *COST* (time of execution) as well as *FUSE* (first use time) are captured by the runtime manager and the numbers used for profit calculation (see: Section V-A). This ensures that only the relevant filters are part of the control table, vacant includes and excludes are cleaned up and the memory consumption used for materialized aggregates is kept small.

In Section V-C distribution of variables and correlation between current date and filter-variables V-D is used to predict future aggregate selections. In order to improve the view matching, filters are extended during optimization. Thus, include filters are not only based on captured workload selections, but also anticipate aggregates that have not been, but are probable to be selected. E.g.: a correlation between an attribute "posting date" of an exclude filter and the current date and time indicates that changes in a base table always happen in a certain time frame. Harmonization of structures (Section V-B) tries to find selection predicate (P_S) that can be replaced by one unifying attribute in order to keep the number of diverse structures and materialized views V_{dA} small. Finally, the optimized and harmonized filters in control table T_D are used to limit the number of materialized aggregates in view V_{dA}.

 ISBN: 978-1-61208-247-9

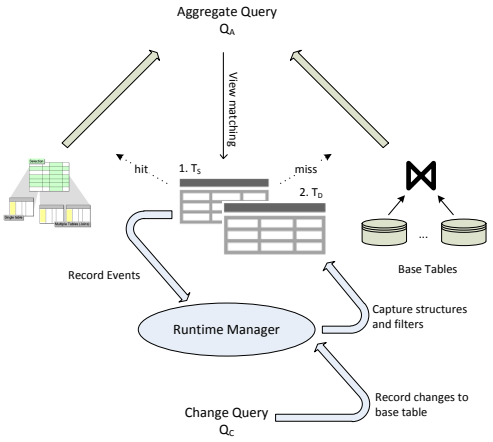

Fig. 2. Runtime Architecture

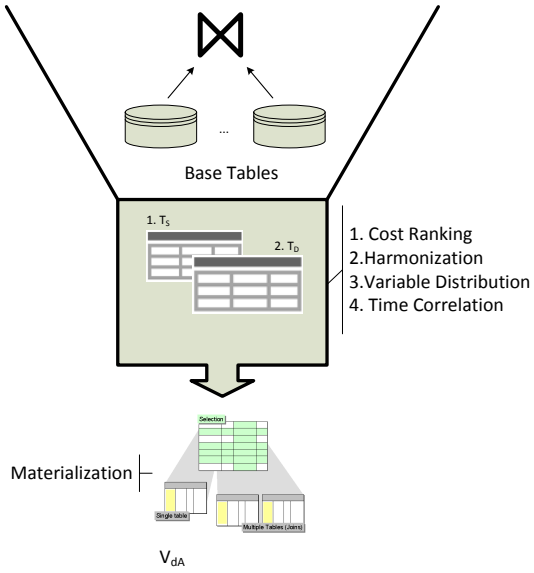

Fig. 3. Update Architecture

The basic steps of each phase are as follows:

1) online phase
- View matching of aggregate queries
- Provide aggregates from V_{dA} or refer to base tables
- Capture filter usage in T_D
- Add missed aggregates in T_S as structures
- Add missed data areas in T_D as include filter
- Add changed data areas in T_D as exclude filter
- Invalidate aggregates that clash with change areas
- Trigger update phase

2) Update phase
- Parse historic workload: identify, save characteristics
- Parse new structures and data areas
- Analyze, rank workload characteristics
- Create materialized aggregate views V_{dA}
- Create control tables T_D
- Optimize control table T_D
- Materialize aggregates V_{dA}

A. Global Structure table T_S

T_S is designed as one global table holding the structures of all aggregate queries that shall be used in a dynamic materialized view V_{dA}. A structure is defined by the aggregate function, its group-by and its predicate-fields. Aggregate queries that have the same predicates, but different group-by fields, can share the same structure, as long as every group-by is covered. However, two aggregate queries, even if they have the same function and grouping, must be handled as two separate structures when they have different selection predicates. This is because two aggregate queries, one selecting using attribute year, the other with a selection on product id produce overlapping, incomparable results.

1) Data Schema: The data schema of T_S consists of the following fields:

SID (int) PK, referenced in T_D
T_B (char) Name of the base table
P_V (char) Predicate that defines base view V_B
A (char) The name of the function and the attribute f(A)
G (char) The group-by clause G_1,G_2...G_n (attribute names)
P_S (char) The where clause of the relation (attribute names)

B. Control table T_D

For every base table, used to build aggregates there is one control table T_D that contains filters, limiting the number of aggregates that are dynamically materialized in its corresponding view V_{dA}. Filters stored in T_D define data ranges that are read (includes) and changed frequently (excludes).

T_D only stores the value characteristics of data areas but not the structure. Therefore, every include filter also references a structure (SID) persisted in T_S . The combined information is needed during materialization, as well as during view matching.

For simplicity reason it is assumed that where-clauses only consists of predicates that are linked with ANDs and that the predicates are all equality predicates. In [14] it is shown how the data structure must be adapted in order to support other predicate types, such as range and expression.

1) Data Schema: There is one dedicated control table for every base table in T_S. Its data schema depends on the number and types of the P_S attributes of the corresponding structures.

FID (int) Primary key - filter id
SID (int) Secondary key, referencing a structure in T_S
SIGN (bit) (I/E) Sign for (I)nclude and (E)xclude filters
FUSE (datetime) A timestamp when filter was first used
LUSE (datetime) A timestamp when filter was last used
EXEC (int) Number of executions of the filter
COST (int) Time used by database cursor for parsing, executing, fetching data of queries referencing the filter
MAT (boolean) A control sign that marks a filter validity
P_{SV} (as original) Filter attributes, store the values of the selection predicates (parameters), derived from P_S in T_S.

 ISBN: 978-1-61208-247-9

2) Hot read data: The runtime manager tracks the usage (number of executions) and cost (elapsed time) for every aggregate query that matches an existing include-filter. Additionally, every aggregate query that does not match an existing filter is captured by the runtime manager as a new include filter in T_D. This filter is saved as invalid until it gets materialized during the update phase.

3) Hot change data: Changes to base table data, such as inserts, updates, and deletes can invalidate materialized aggregates. In order to minimize the aggregates maintenance effort, aggregates based on frequently changing base data are excluded from materialization. Typically, the *hot change* data area shifts over time. Once the data ages, the frequency of change transactions is reduced. In our framework, every change event is recorded by the runtime manager as exclude filter(s). First of all, it must be determined how a change statement can effect an existing aggregate. Changes to attributes that are not part of the view structure V_{dA} can be neglected because they neither change the value of the aggregated attributes, nor does it change the grouping attributes. Changes to a grouping attribute always effect aggregates of two groups: the group of aggregates with the old data and the new one. Inserts and deletes on the other hand can always effect aggregates and results in the creation of an exclude filter. Secondly, the data area that is changed must be identified when selected with a primary key, updates and deletes effect a single row, and when selected with a non-unique attribute they effect a range of rows. In order to make that comparable with the existing include filters (hot read data), every change must be *normalized* to the filter-attributes, as defined in P_S.

4) Data Hotspots: Hotspots are data areas in a base table that are frequently used for aggregations but do not change frequently. Those hotspots change over time and adapt to the workload. The example in Table II shows how hotspots are identified in T_D. The values are extracted from three different aggregate queries (see SID column). SID1 selects its aggregates using attributes Product and Quarter. SID2 uses Product and Month and SID3 aggregates just over the parameter Month. The exclude filters are independent from the SID. The exclude filter FID7 for example could be derived from an insert of Product A in Quarter 3 and Month 8. Because of this, filter FID1 and FID5 are no hotspot areas anymore, even though they have been used to select aggregates. The same happens with FID6 because of FID9. FID9 means that there was a change in Month 7 and therefore aggregates of that selection do not qualify for materialization.

C. Dynamically materialized view V_{dA}

While the structure and filter control tables are maintained *online* during runtime, DMAVs are created and materialized only during the update phase. For every aggregate structure there is one V_{dA} defined and materialized. The algorithm that materializes the views is based on the view structure derived from T_S and the filters provided by T_D. It is important to note that the aggregates of V_{dA} are not only grouped by the G_i attributes of the structure but also by the P_i attributes.

TABLE II
HOTSPOT EXAMPLE OF T_D

FID	SID	SIGN	Product	Quarter	MONTH
1	1	I	A	3	
2	1	I	A	2	
3	1	I	A	1	
4	2	I	A		7
5	2	I	A		8
6	3	I			7
7		E	A	3	8
8		E	B	3	8
9		E	B	3	7

Aggregates are not specific to a certain filter, but comply to all filters. In fact, they can be used among different aggregate queries and different (overlapping) data selections. On the condition that there is a valid, matching filter, different aggregate queries (with different parameter values) can select directly on the aggregated view V_{dA}, instead of the base view. This allows the reuse of aggregates that have overlapping filters V_B.

1) Data Schema: The data schema of an aggregate view consists of (a) column(s) for the aggregated value A_i, (b) the group-bys G_i and (c) the selection attributes P_i.

D. Example for dynamic view materialization

The following example will show two aggregate queries (Q1 and Q2) as well as two change queries (U1 and I1) that represent a mixed workload. Based on a simple table schema, consisting of three tables (one fact table , two dimension tables) it is explained, how the query structure and its read and changed data ranges (includes and excludes) are extracted into the structure table T_S and a control table T_D. The data schema is as follows:

Product (pid, pname, pcategory)
DateH (did, dyear, dquarter, dmonth)
Sales (pid, did, rev)

Suppose Q1 and Q2 (see Listing 1 and 2) are two parameterized SPJG (Select, project, join and group-by) aggregate queries from a given mixed workload. In the first step of the analysis, the base table T_B:(Sales), the view predicate P_V:(Sales.pid = Product.did AND Sales.did = DateH.did) as well as the aggregate function and field A:(sum(Sales.rev)) are identified and written to the T_S table (as illustrated in Figure 3). For both queries they are the same. However their selection-predicates P_S are different; consequently, two separate structures have to be created. In Table III, Q_1 has SID=1 and Q_2 has SID=2. Their grouping attributes are written to G.

By collecting that information, the data structures of the materialized views can be defined as:

V_{dA1} (SUM(rev), pcategory, dyear dmonth)
V_{dA2} (SUM(rev), pname, dyear, dquarter)

 ISBN: 978-1-61208-247-9

TABLE III
STRUCTURE CONTROL TABLE T_S

SID	T_B	P_V	A	G	P_S
1	Sales	Sales.pid = Product.did AND Sales.did = DateH.did	SUM(rev)	dmonth	EQ(pcategory)-&&-EQ(dyear)-&&-EQ(dmonth)
2	Sales	Sales.pid = Product.did AND Sales.did = DateH.did	SUM(rev)	dquarter-pname	EQ(pname)-&&-EQ(dyear)

TABLE IV
DATA CONTROL TABLE T_D FOR TABLE SALES: TD_SALES

FID	SID	SIGN	FUSE & LUSE	EXEC	COST	MAT	PNAME	PCATEGORY	DYEAR	DMONTH
1	1	I				true/false		2	2011	3
2	1	I				true/false		4	2011	3
3	2	I				true/false	'A'		2011	
3	2	I				true/false	'A'		2010	
4	-	E				true/false	'B'	1	2010	9
4	-	E				true/false	'A'	2	2010	9
4	-	E				true/false	'A'	1	2011	12

Listing 1. Aggregate Query Q_1

```
SELECT d.dmonth, SUM(s.rev)
FROM Sales AS s, Product AS p, DateH AS d
WHERE s.pid = p.pid
  AND s.did = d.did
  AND (p.cat = 2 OR p.cat = 4)
  AND d.dyear = 2011
  AND d.dmonth = 3
GROUP BY d.dmonth
```

Listing 2. Aggregate Query Q_2

```
SELECT d.dquarter, p.pname, SUM(s.rev)
FROM Sales AS s, Product AS p, DateH AS d
WHERE s.pid = p.pid
  AND s.did = d.did
  AND d.dyear = 2011
  AND p.pname = 'A'
GROUP BY d.dquarter, p.pname
```

Having captured the structure of the aggregate query in T_S, the control table T_D is created. As illustrated in Table IV, the distinct set of attributes (A_i, G_i and P_{Si} of both structures SID= 1 and SID= 2) defines its data structure. Afterwards the values of the selection predicates P_{SV} (parameters) are inserted as include filters FID1-3 in Table III.

Listing 3. Change Query I_1

```
INSERT INTO Sales (pid, did, rev)
     VALUES (2,34,898.99)
```

Listing 4. Change Query U_1

```
UPDATE Sales
   SET rev = 99.99
 WHERE
1st Option: sid = 55
2nd Option: p.pid='1' and d.dyear='2011
```

Listing 3 and 4 are two changing queries: an insert and an update statement with two typical selection predicates. As all their selection predicates cannot be captured in T_D, they need to be normalized, as shown in Listing 5 and 6, and described in Section IV-C.

Listing 5. Normalized Exclude Query I1 language

```
INSERT into control_table
SELECT p.pname, p.pcategory, d.dyear, d.dmonth
FROM Sales s, Product p, DateH d
WHERE s.pid = p.pid AND s.did = d.did AND s.sid = <id>
GROUP BY ...
```

Listing 6. Normalized Exclude Query U1

```
INSERT into control_table
SELECT p.pname, p.pcategory, d.dyear, d.dmonth
FROM Sales s, Product p, DateH d
WHERE s.pid = p.pid AND s.did = d.did AND
1st Option: s.sid = 55
2nd Option: p.pid='1' AND d.dyear='2011'
GROUP BY ...
```

E. View maintenance and materialization V_{dA}

1) Maintenance: Compared to traditional, fully materialized views, dynamic materialization using the control table T_S allows for more efficient maintenance as only the hotspot parts of the view are materialized. Further, as aggregates of base data that is subject to changes is excluded, re-materialization is kept to a minimum.

Many different incremental view maintenance algorithms that compute changes of the base relation to the corresponding views [18] have been discussed in research. The concept used in this paper is built upon the control table T_D. Before a view is materialized, every filter is marked as invalid (mat = false). This status information is also used during view matching in Section IV-F. Aggregate queries that match an invalid filter cannot use the aggregate view, but must be answered by the

TABLE V
DYNAMIC, MATERIALIZED VIEW V_{DA2} FOR STRUCTURE SID 2

REV	PNAME	DYEAR	DQUARTER
2499.12	A	2010	1
3189,81	A	2010	2
3747.15	A	2010	3
1806.07	A	2010	4

 ISBN: 978-1-61208-247-9

base relation. This is true until the view is re-materialized during update phase where all invalid filters are set valid again.

2) Materialization: Let V_B denote a query expression on a base table T_B joining other tables using a predicate P_V. V_{Bi} may be referred to as a base view, defined by the structure SID=i in T_S. A_i shall be its aggregated function on an attribute of V_{Bi}. $G_i.*$ are all of its grouping attributes and $P_i.*$ are the parameters of the where-predicate. For each structure (SID) a dynamic materialized view V_{dAi} is defined. For each V_{Bi} the materialization is controlled by the control table T_D, its own include-control-predicate $P_{ICi}(V_{Bi},T_D)$ and an exclude-control-predicate $P_{EC}(V_{Bi},T_D)$. The algorithm in Listing 7 shows the generic definition of a materialized view. The exists- and not-exists-clause in the definition confine the rows, actually materialized in V_{dAi}, to those satisfying the control predicates $P_{ICi}(V_{Bi},T_D)$ and not to those of $P_{EC}(V_{Bi},T_D)$.

Listing 7. Algorithm for Materialization of $_{dA}$

```
FOR ALL structures as i in T_S DO
    CREATE VIEW AS
       SELECT A_i, G_i.*, P_i.* FROM V_Bi
       WHERE
       EXISTS
                SELECT 1 FROM T_D WHERE P_ICi(V_Bi,T_D))
       AND NOT EXISTS
                SELECT 1 FROM T_D WHERE P_EC(V_Bi,T_D)))
       GROUP BY G_i.*, P_i
END FOR
```

In Listing 8 the view definition for SID=2, V_{dA2} is shown. Td_Sales shall be the control table T_D, holding all include and exclude filters. However, most interestingly are the include and exclude predicate controls ($P_{ICi}(V_{Bi},T_D)$, $P_{EC}(V_{Bi},T_D)$) that ensure that only aggregates over the hotspot data areas are materialized.

Listing 8. Materialized View V_{dA} for SID=2

```
SELECT SUM(s.rev), G_i.*, P_i.*
FROM V_Bi AS bV
WHERE EXISTS (
      SELECT *
      FROM Td_Sales f
      WHERE  bV.pcategory = f.pcategory AND bV.pname = f.pname AND
             bV.dyear = f.dyear AND f.sign = I AND f.sid = 1 AND f.
            mat = false) --includes
AND NOT EXISTS (
      SELECT *
      FROM Td_Sales f
      WHERE  bV.pname = f.pname AND bV.ddyear = f.dyear AND f.sign
            = E) --excludes
GROUP BY G_i.*,P_i.*
```

3) Aggregate granularity: The granularity of an aggregated result set depends on the number and cardinality of its grouping attributes. Aggregates with several group-by attributes have the advantage that they can be reused to calculate coarse-grained ones. On the other hand this increases the size of the result set. This trade-off is considered during cost ranking in Section V-A that calculates the profit of an filter depending on the size of the result set. In the proposed concept, the granularity of the materialized view V_{dA} is also determined by the number and cardinality of the predicate attributes of the base relation. As the aggregates in V_{dA} are not specific for one parameter selection (filter) but should be reusable among overlapping selections (filters), of the same structure, the aggregates are additionally grouped by predicate attributes. This way, different aggregates with different groupings can be selected on the same materialized view.

F. View matching

The question whether an aggregate query or a sub query can be answered by a view (V_{dA}), is a well-known view matching problem. In [12] Goldstein et al. already looked at SPJ queries having an aggregate function followed by a group-by operation. Still, there is one major aspect that makes view matching for DMAVs different to other algorithms: query containment of the aggregate query in the view cannot be tested before execution time. As outlined in [14], the part of view matching that checks if the parameters of a query match a valid include-filter and not an exclude-filter in T_D has to be postponed to execution time. This is being done using a guard condition as described in Listing 9.

Listing 9. Guard Condition

```
EXISTS(SELECT 1 FROM T_d WHERE SID=i AND MAT=true AND SIGN='I' AND
     hotkeys = @parameters)
NOT EXISTS(<exclude-filter>)
```

A guard condition is only executed when there is a valid structure for a query. If the guard condition is positive it is served from V_{dA}, otherwise it must be calculated from the base relation. Besides, aggregate queries can be treated as a SPJ query followed by a group-by. Accordingly, as described in [12] an aggregation query can be computed from a view V_{dA} if there is a structure in T_S that meets the following requirements.

1) All columns required by compensating predicates (if any) are available in the view output.
2) The view contains no aggregation or is less aggregated than the query, i.e., the groups formed by the query can be computed by further aggregation of groups output by the view.
3) All columns required to perform further grouping (if necessary) are available in the view output.
4) All columns required to compute output expressions are available in the view output.

All of these requirements can be satisfied by using the structure control table T_S. They can already be checked during optimization time. Then during execution time the SID of a query is known and the view matching can be limited to the guard condition check.

V. Control Table Management & Optimization

Both tables, T_D and T_S are managed during the online phase and optimized during update phase. During the online phase, the goal is to answer as many queries as possible from the view, because they will be answered a lot faster from V_{dA} than from the base relation. At the same time the runtime manager tracks new queries, usage/ hit statistics, and change events. This allows adapting to new query patterns as well as efficiently utilize the system resources during the update and optimization phase. During this phase, new filters in T_D and structures in T_S are materialized while invalidated filters are refreshed and validated again. Before that the control tables are optimized. The reason for optimization is to materialize only

relevant aggregates, reduce the overhead of different aggregate structures and to anticipate future selections and improve the hit ratio. Therefore:

- Every filter is ranked based on a cost function,
- Different query structures of one base table are scanned, harmonized, and merged, and
- Future selection parameters are anticipated and added as filters to T_D.

A. Cost ranking

Every filter that is stored in a control table has a certain cost in terms of storage and materialization resources. Whether a new filter that is captured during online phase replaces existing ones shall depend on the calculated profit that can be compared amongst each other. Following are some metrics that are available for each filter. They are either tracked by the runtime manager or calculated.

- c_i: Sum of time used by the database cursor for parsing, executing, fetching data of all queries that reference the same filter (without materialization)
- FU_i: Time stamp when the filter was used the first time
- LU_i: Time stamp when the filter was used the last time
- s_i: Max. granularity of a view V_{dA}
- e_i: Number of hits (see:EXEC in T_D)

The goal of cost ranking is to have a number that allows the comparison of materialized and un-materialized filters and structures. Based on that number it can be decided which filters to admit to V_{dA} and which ones to replace.

$$profit(i) = \frac{h_i \cdot c_i}{s_i} - (c_i \cdot MAT) \quad (1)$$

$$h_i = \frac{e_i}{t - FU_i} \quad (2)$$

$$s_i = \prod_{j=1}^{n} AG_j \quad (3)$$

$$profit(SID) = \sum_{i=1}^{n} profit(FID_i) \quad (4)$$

In (1) the profit of an include-filter is calculated based on the average hit rate of the filter h_i, the cost of a filter c_i to be retrieved without materialization and the product of the attribute granularity of all view attributes. In order to compare materialized and non-materialized filters, the cost of a onetime materialization is subtracted at the end.

The average hit rate h_i of filter i (2) tells how many times a filter has been hit in average since it was first screened. Including the current time t, is guaranteed that aging of filters is considered. A filter, hit ten times in the last hour has a higher hit rate than a filter, hit ten times in the last two hours. (3) S_i is the maximum possible granularity of the view V_{dA} as defined by its structure. Therefore the cardinality of each attribute must be multiplied. The profit of a structure (4) is calculated by the sum of all filters that reference the structure.

$$relev(i) = \frac{h_i}{t - LU_i} \quad (5)$$

Exclude filters are ranked on their time dependent relevance (5). The overall average hit ratio of the exclude filter divided by its actuality (last time used).

B. Structure harmonization

For each aggregate structure, there is one view V_{dA} defined. They cannot serve aggregate queries with different structures as they have a different predicates clause, different predicate attributes as well as different aggregate granularity. However, if different predicate structures could be harmonized to one structure, this would reduce the number of required aggregates, and improve reuse of materialized aggregates. The example in Listing 10 shows two aggregates (see line 1 and 2). Both queries require their own structure and filters. Accordingly, there would be two materialized views.

In the mentioned example there is one hierarchical dimension: Date(did-year-month-day). The base table is joined with that table on the PK attribute "did". Thus, aggregates can be selected using all attributes of Date. The idea of structure harmonization is to consolidate the selection predicates of one dimension to its unique (composite) key. As can be seen in line 4 of Listing 10, the real selection of data is swapped out to a sub query wrapped with an IN operator. With the unique dimension key "did" every selection in dimension "Date" can be expressed without overlapping sets.

Listing 10. Predicate harmonization

```
1 WHERE did=d.did AND d.day=x
2 WHERE did=d.did AND d.year=y AND d.day=z
3
4 WHERE did IN (Select did from Date WHERE
5 1st Option: d.year=y and d.day=z
6 2nd Option: d.month=x )
```

Rewriting aggregate queries into such a harmonized form would also simplify the data schema of control table T_D to one attribute "did". The number of filters, however, would increase significantly: E.g.: Assumed that there is one unique key per day, a filter of type: year=2012 would be changed to 365 filters of type "did". Instead, if the key was based on a chronological interval scale, a range control filter could express year=2012 as did=from:1-to:365.

Structure harmonization/ replacement can only be done between attributes of the same dimension table. The new attribute must be on a lower level than the old one. Additionally, only, if there is an overlapping free, n:1 relation between those attribute this harmonization works.

As a downside of this concept, materialized aggregates can become extremely fine grained. Even though the number is filtered by the control table, grouping by a unique dimension key "did" in V_{dA} results in very high numbers of results. At this point, it might happen that the results in views V_{dA} are not aggregated anymore but materialize the base relation.

 ISBN: 978-1-61208-247-9

C. Variable distribution

As there is a big training set of recorded selection parameters in T_S one can check for significant statistical distributions of the single attributes. Those selection characteristics could be used to automatically adjust filters and optimize the materialized view. A continuous distributed variable X (attribute) that has a similar probability for every captured value characteristic should not be limited during materialization to certain values. It is probable that the user might select another value next time, as there is no accumulation of value characteristics. For such attributes existing filters can be replaced so that aggregates are not filtered on that attribute anymore.

D. Time correlation

Often, there are dependencies between time and certain selection parameters. Products might be relevant only in certain seasons, account balances might only be interesting for the last twelve months and so forth. Finding those correlation can be done based on the captured filters in T_D. Not only include-filters could be anticipated in order to improve hit ration on V_{dA}, but also exclude-filters. Based on such knowledge, exclude filters can better adapt to hot change data as it changes over time.

VI. Experiment & Discussion

To evaluate our proposed concept, we have tested it rudimentarily with a mixed workload benchmark. The chosen benchmark [5] was setup on a MySQL database and executed twice with the same settings. The share of read-only OLTP queries was set to 95% and the share of OLAP queries for the mixed client was set to 40%.

In the first run, the benchmark performed all queries with the original schema. In the second run, the OLAP queries where optimized so that they did not accessed the base table anymore, but a DMAV V_{dA} instead. There were four different analytical queries with aggregate functions, two of them had additional sub queries with aggregate function. Hence, there were six different structures and V_{dA} views that the queries could use to select and aggregate. Every run was executed for 15min, plus 2min warm-up time that was not counted. During each run, database performance was monitored using MySQL Enterprise Monitor and the general db log. Afterwards the general log was analyzed using the Analysis Framework III.

Compared with the default benchmark, during the optimized run, the database could serve eight-times more select statements. Hence, the overall output performance increased, also for the OLTP queries. That was because the runtime cost (time in ms) for the OLAP queries was reduced by up to 99,96% each. Obviously, this comparison can only show relative improvements. Performed on a real application, the number of aggregate queries would be even higher. Besides, DMAVs would put an overhead on the database engine in terms of storage consumption and runtime performance. This was not considered in the experiment. Still, it became obvious that the performance can be improved by using DMAVs.

VII. Conclusion & Future Work

In this paper we presented *dynamic materialized aggregate views* (DMAV), a concept to selectively materialize arbitrary aggregate queries in a mixed workload environment. Built upon a central dictionary structure that holds generic aggregate structures and corresponding hot spot tables, it was shown how the execution of analytical queries in transactional databases can be improved. Intelligent selection and materialization of hot aggregates promises to be a way to handle such workload demands in future applications. Even though major parts of the concept can be transferred to asynchronous update phases or can be done during optimization phase it still puts an overhead on the database. Therefore, focus of future research addresses the problem of how and when to integrate the required steps of DMAV into query execution of a database.

References

[1] S. Chaudhuri and K. Shim, “Optimizing queries with aggregate views,” *EDBT*, 1996.
[2] Y. Kotidis and N. Roussopoulos, “DynaMat: a dynamic view management system for data warehouses,” *ACM SIGMOD Record*, 1999.
[3] M. Grund, J. Krüger, and H. Plattner, “HYRISE: a main memory hybrid storage engine,” *Proceedings of the PVLDB*, pp. 105–116, 2010.
[4] V. Sikka, F. Färber, W. Lehner, T. Peh, and C. Bornhövd, “Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth,” *SIGMOD*, pp. 731–741, 2012.
[5] A. Bog and J. Kruger, “A Composite Benchmark for Online Transaction Processing and Operational Reporting,” *BIRTE*, 2008.
[6] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, and Others, “The mixed workload CH-benCHmark,” in *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, 2011, p. 8.
[7] C. Tinnefeld, S. Müller, H. Kaltegärtner, S. Hillig, L. Butzmann, D. Eickhoff, S. Klkauck, D. Taschik, B. Wagner, O. Xylander, A. Zeier, H. Plattner, and C. Tosun, “Available-To-Promise on an In-Memory Column Store,” *BTW*, pp. 667–686, 2011.
[8] F. Funke, A. Kemper, and T. Neumann, “Compacting Transactional Data in Hybrid OLTP & OLAP Databases,” *VLDB*, 2012.
[9] P. Larson and H. Z. Yang, “Computing Queries from Derived Relations,” *VLDB*, 1985.
[10] H. Z. Yang and P.-A. Larson, “Query transformation for PSJ-queries,” *VLDB*, pp. 245–254, 1987.
[11] R. Bello, K. Dias, and A. Downing, “Materialized views in Oracle,” *VLDB*, 1998.
[12] J. Goldstein and P.-a. k. Larson, “Optimizing queries using materialized views: a practical, scalable solution,” *ACM SIGMOD Record*, vol. 30, no. 2, pp. 331–342, 2001.
[13] A. Halevy, “Answering queries using views: A survey,” *VLDB*, no. 1999, pp. 270–294, 2001.
[14] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding, “Dynamic Materialized Views,” *ICDE*, pp. 526–535, 2007.
[15] P. M. Deshpande and J. F. Naughton, “Aggregate Aware Caching for Multi-Dimensional Queries,” *EDBT*, pp. 167–182, 2000.
[16] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton, “Caching multidimensional queries using chunks,” *SIGMOD*, pp. 259–270, 1998.
[17] P. Scheuermann, “WATCHMAN: A Data Warehouse Manager Intelligent Cache,” *VLDB*, 1996.
[18] A. Gupta, “Maintaining views incrementally,” *ACM SIGMOD Record*, 1993.

 ISBN: 978-1-61208-247-9