Using Object-Awareness to Optimize Join Processing in the SAP HANA Aggregate Cache

Stephan Müller
Hasso Plattner Institute
Potsdam, Germany
stephan.mueller@hpi.de

Anisoara Nica
SAP SE
Waterloo, Canada
anisoara.nica@sap.com

Lars Butzmann
Hasso Plattner Institute
Potsdam, Germany
lars.butzmann@hpi.de

Stefan Klauck
Hasso Plattner Institute
Potsdam, Germany
stefan.klauck@hpi.de

Hasso Plattner
Hasso Plattner Institute
Potsdam, Germany
hasso.plattner@hpi.de

ABSTRACT

The introduction of columnar in-memory databases, along with hardware evolution, has made the execution of transactional and analytical workloads on a single system both feasible and viable. Yet, doing analytics directly on the transactional data introduces an increasing amount of resource-intensive aggregate queries which can slow down the overall system performance in a multi-user environment. To increase the scalability of a system in the presence of multiple such queries, we propose an aggregate cache in the general delta-main architecture that provides an efficient means to handle costly aggregate queries by applying incremental materialized view maintenance and query compensation techniques. Handling aggregate queries based on joins of multiple tables however is still a challenge as query compensation can be very expensive in the delta-main architecture of columnar in-memory databases. Our analysis of enterprise applications has revealed several data schema and workload patterns that can be leveraged for addressing performance of query processing using the aggregate cache. We contribute by presenting an approach to transport the application object semantics into the database system, becoming object-aware, and optimize the query processing using the aggregate cache by applying partition pruning and predicate pushdown in such general delta-main architecture. Our experimental validation using customer data and workloads confirms that this type of optimizations enables efficient usage of the aggregate cache for an even higher share of aggregate queries as one mean to scale the system.

1. INTRODUCTION

The separation of enterprise applications into online transactional processing (OLTP) and online analytical processing (OLAP) induces drawbacks including stale and redundant data, and inflexible analytics due to pre-calculated data cubes. A closer look reveals that this separation is mostly artificial as both systems have the same number of inserts – unless the OLAP system already abstracts from the base data – and a high share of analytical queries with costly aggregations. To deal with aggregate queries, both systems employ different approaches. While OLAP systems make extensive use of materialized views [29, 33], we see that the handling of aggregates in OLTP systems is often done within the application by maintaining predefined summary tables. This leads to an increased application complexity with risks for violating data consistency and to a limited throughput of insert and update queries as the related summary tables must be updated in the same transaction [14, 25].

With the ongoing trend of columnar in-memory databases (IMDBs) such as Hyrise [11], SAP HANA [9], and Hyper [16], this artificial separation is no longer necessary as they are capable of handling mixed workloads, with transactional and analytical queries, in a single system [24]. In columnar IMDBs, the storage is separated into a highly compressed, read-optimized *main* storage and a write-optimized *delta* storage, both implemented as columnar data stores. New records are inserted to the delta storage and periodically *merged* to the main storage [17]. Having a single IMDB for transactional and analytical workloads however imposes one central challenge: While modern hardware enables the execution of arbitrary complex computations in a short time by parallelization, this means that one query can saturate an arbitrary large machine [30]. Especially the execution of expensive aggregations that may be done by many hundreds of users in parallel is problematic and requires means to keep the system scalable.

Despite the aggregation capabilities of columnar IMDBs [24], access to tuples of a materialized aggregate – which we define as a materialization of a query which contains aggregate functions – is always faster than aggregating on the fly. However, the overhead of materialized view maintenance to ensure consistency for modified base data has to be considered and involves several challenges [12]. It turns out that the main-delta architecture is well-suited for the *aggregate cache*, a novel strategy of dynamically caching aggregate queries and applying incremental view maintenance techniques [21] for maintaining the cache and answering queries using the aggregate cache. In the general main-delta architecture,

10.5441/002/edbt.2015.57

only the delta storage is updated when data is modified, for example, when new records are inserted. In our design of the aggregate cache, the materialized aggregates are only defined on records from the main storage. Hence, the materialized aggregates do not have to be invalidated when new records are inserted, updated, or deleted in the delta storage. When a query result is computed using the aggregate cache, the final, consistent query result is delta-compensated, on the fly, by aggregating the newly inserted records of the delta storage and combining them with the previously cached aggregate of the main storage.

One challenge of the aggregate cache in the main-delta architecture is to achieve high performance for relevant classes of application queries which include aggregates based on joins of multiple tables. These queries require expensive delta-compensations based on subjoins of all permutations of delta and main partitions of the involved tables, excluding the already cached join of the main partitions. For a query joining two tables, three extra subjoins are required for delta-compensation, and a query joining three tables already requires seven extra subjoins. This may result in very little performance gains over not using the aggregate cache. However, after analyzing the characteristics of several enterprise applications, we identified schema design and workload patterns that can be leveraged to allow pruning certain subjoins and therefore optimize the overall performance of join queries using the aggregate cache.

In this paper, we make the following contributions:

- We introduce the aggregate cache, a materialized aggregate engine implemented in SAP HANA, leveraging the main-delta architecture of columnar IMDBs, and describe its current architecture (see Section 2.1).
- We discuss the query processing using the aggregate cache and performance challenges related to main and delta compensations which are metrics for admittance in the aggregate cache (see Sections 2.2 and 2.3).
- We identify a class of join queries which normally do not qualify to be admitted in the aggregate cache and analyze their performance issues when using the aggregate cache. We propose a novel solution for increasing the performance for this class of queries exploiting the main-delta architecture and the application object semantics. These techniques are implemented as a prototype which extends the aggregate cache for join queries. The main contributions here are:
  - We analyze enterprise applications which benefit the most from the dynamic aggregate cache and identify several schema design and workload patterns imposed by the object semantics of these applications (see Section 3).
  - We give a formal definition of the join pruning problem in the aggregate cache and define matching dependencies among relations based on join attributes and temporal relationships as one possible design for efficiently using the aggregate cache for join queries (see Section 4).
  - We discuss our implementation for transporting application object semantics into the database to become *object-aware* which allows join pruning techniques to be applied for queries using the aggregate cache (see Section 5).
- We use the CH-benCHmark [10], a benchmark based on TPC-H and TPC-C, and a benchmark using real customer workloads from a production Enterprise Resource Planning (ERP) system to show performance results for (1) aggregate cache maintenance strategies; (2) data update overhead for tables referenced in the aggregate cache; (3) query processing using aggregate cache with and without join pruning (see Section 6).

2. AGGREGATE CACHE

The aggregate cache leverages the concept of the main-delta architecture in SAP HANA [9]. Separating a table into a main and delta storage has one main benefit: it allows to have a read-optimized main storage for faster scans and a write-optimized delta storage for high insert throughput. All records in the delta storage are periodically propagated into the main storage in an operation called *delta-merge* [17]. The fact that new records are added to the main storage only during a merge operation is leveraged by the aggregate cache which is designed to cache only the results computed on the main storage. For a current query using the aggregate cache, the records from the delta storage are aggregated on-the-fly which compensates the corresponding cache entry to build the result set of the query, a process we refer to as *delta compensation*. In the general main-delta architecture, records are not updated in place. Instead, the updated record is inserted in the delta partition whereas the old record in the main (or delta) partition is *invalidated*. Other database implementations with a delta storage or differential buffer such as C-Store, Sybase IQ, MonetDB/X100, Hyrise, or memory-optimized tables in SQL Server handle updates very similarly to the mechanism implemented in SAP HANA. During the

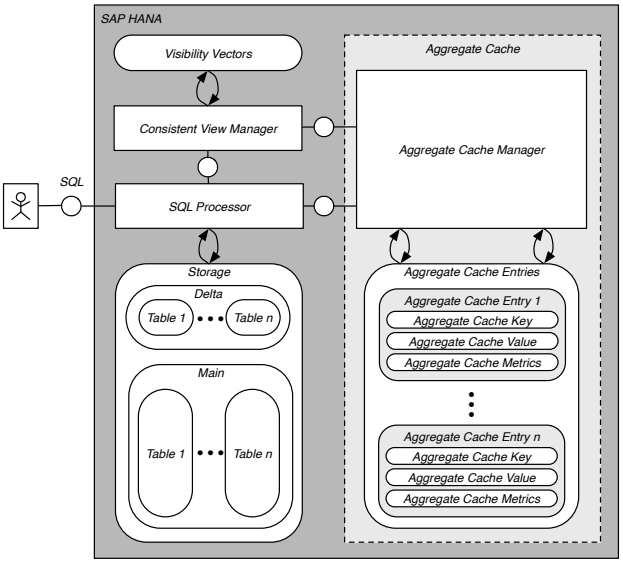

Figure 1: The architecture of the aggregate cache in SAP HANA.

next merge, all invalidated records can either be removed from the main storage or kept so that temporal query processing on historical data can be supported [15]. To handle invalidations in the main partition, we apply a *main compensation* process as described in Section 2.2.

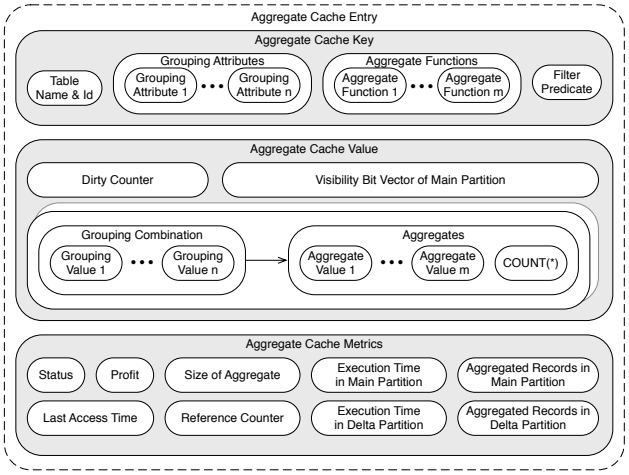

Figure 2: The structure of an aggregate cache entry, consisting of an aggregate cache key, an aggregate cache value, and aggregate cache metrics.

2.1 Architecture

As illustrated in Fig. 1, the aggregate cache is implemented inside the column store engine of SAP HANA [9]. The *aggregate cache manager* is the core component of the aggregate cache, managing aggregate cache entries.

An aggregate cache entry, depicted in Fig. 2, consists of a key, a value, visibility vectors, and profit metrics. The *aggregate cache key* is a unique identifier based on the query definition including the table name, table id, the grouping attributes, the aggregate functions, and the filter predicates of the related aggregate query. The *aggregate cache value*, the extent of the aggregate query, is a structure consisting of the grouping combinations and the corresponding aggregate functions: it contains the result set of the aggregate computed only on the main storage. The aggregate cache entry further contains dirty counters that indicates if records have been invalidated in the main partitions, and the visibility vectors of the main partitions at the time of last computation. The aggregate cache entry is first created during query processing (Fig. 3) and it is maintained during the delta-merge operations. *Aggregate cache metrics* are maintained for each entry including the aggregate's size, the number of aggregated records, execution times for delta and main compensations, maintenance times, and usage information. The metrics are required to calculate the profit of an aggregate cache entry to be used for dynamic cache admission, eviction, and maintenance decisions [20].

Query execution using the aggregate cache is shown in Fig. 3: the query executor delegates aggregate query blocks that qualify for the aggregate cache to the *aggregate cache manager*. The aggregate cache supports queries with *self-maintainable* aggregate functions [22] including SUM, COUNT, and AVG. When the aggregate cache matching process is not successful, the aggregate cache manager attempts to create a cache entry by executing the aggregate query on the main partitions with the global record visibility which is retrieved through the *consistent view manager*. If the aggregate is profitable enough for cache admission, the result is used to create an aggregate cache entry. In both cases, when aggregate entry is retrieved from the cache or it is just cached by the current transaction, the *main compensation* and the *delta compensation* must be applied.

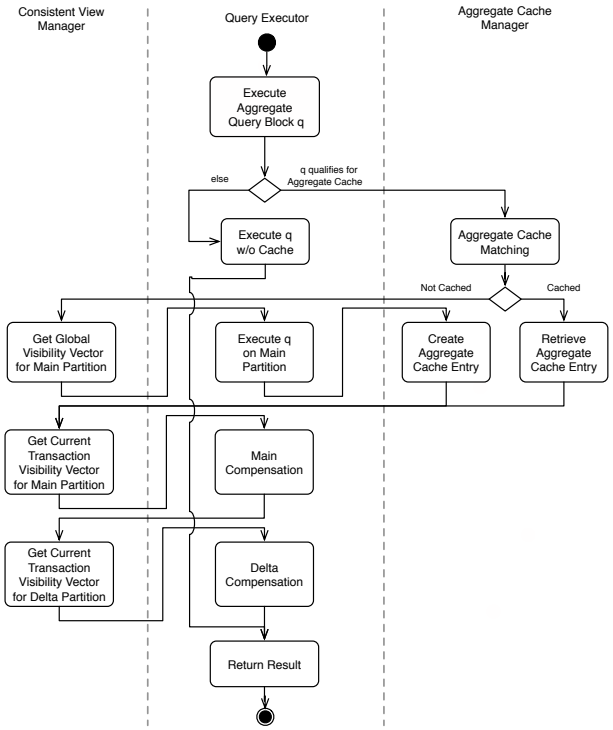

Figure 3: Query processing with the aggregate cache: creation and usage of aggregate cache entries including main and delta compensation during query execution.

2.2 Main Compensation

While updates and deletes of records in the delta storage are handled transparently and do not affect our caching algorithm, an aggregate cache entry can become inconsistent with respect to a record invalidation in the main storage, including deletes and updates of the current transaction. Instead of recalculating an aggregate cache entry with every record invalidation in the main storage, we employ an approach that uses bit vector comparison to efficiently detect invalidated records and apply them to aggregate cache entries in a process called *main compensation*. As illustrated in Fig. 3, we use the consistent view manager to retrieve current record visibilities during aggregate cache entry usage.

The record invalidation is handled through the consistent view manager (see Fig. 1) that creates a bit vector representing the visibility of records of a table for an incoming query based on its transaction token. When an aggregate query is cached, the current snapshot is captured using this visibility vector. When a query is executed using an aggregate cache entry, an efficient bit vector comparison of the current snapshot with the snapshot at the cache creation time is used, thereby detect invalidated records, and apply them for main compensation. The details of aggregate cache main compensation can be found in [19] and are omitted in this paper for simplification reasons.

2.3 Delta Compensation

As the last step in query execution (Fig. 3), any query using the aggregate cache must apply delta compensation operation which accounts for records in the delta storage visible to the current transaction. When the aggregate cache is based on multiple tables joins, the complexity of answering a query using the aggregate cache increases as the aggregate

cache is computed on the main partitions only, and the query must be compensated with all subjoins on deltas and mains. As a result, the profit of caching an aggregate query based on many tables may be very low because their performance using the aggregate cache is not superior to not using it. The techniques proposed in this paper have the main goal of extending the class of aggregate queries which qualify to be admitted into the SAP HANA aggregate cache.

The classical aggregate query joining a header table H, an item table I, and a dimension table D (see Section 3) on the join conditions $H[A] = I[A]$ and $I[B] = D[B]$ is $Q(H, I, D) = H \bowtie_{H[A]=I[A]} I \bowtie_{I[B]=D[B]} D$. In main-delta architecture, each table X consists of at least two partitions $\mathbb{P}(X) = \{X_{main}, X_{delta}\}$ which adds complexity when the result of the query $Q(H, I, D)$ is computed as the join processing must consider all subjoin combinations among these partitions. Theoretically, the subjoins on delta and main partitions of the tables referenced in $Q(H, I, D)$ are as depicted in Equation 1 and Fig. 4. The subscript numbers in Equation 1 of the subjoins match the subjoin numbers in Fig. 4. Based on the size of the involved table components, the time to execute the subjoins varies. Typically, the ratio between the sizes of main and delta partitions is 100:1. In our example the subjoins #5 and #8 require the longest time, since they involve matching the join condition of the mains of two large tables.

$$
\begin{aligned}
&Q(H, I, D) = \\
&(H_{delta} \bowtie_{H[A]=I[A]} I_{delta} \bowtie_{I[B]=D[B]} D_{main})_1 \\
&\ldots \cup (H_{main} \bowtie_{H[A]=I[A]} I_{main} \bowtie_{I[B]=D[B]} D_{delta})_5 \\
&\ldots \cup (H_{main} \bowtie_{H[A]=I[A]} I_{main} \bowtie_{I[B]=D[B]} D_{main})_8
\end{aligned} \tag{1}
$$

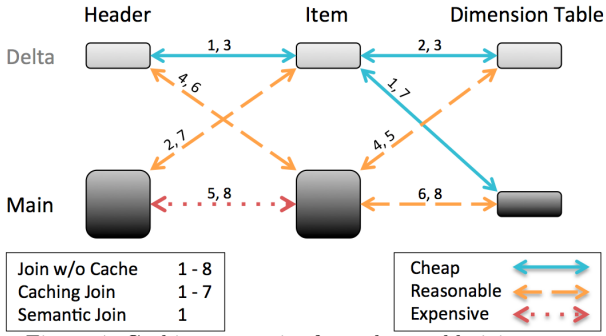

Figure 4: Caching strategies for a three table join query.

2.3.1 Join without Aggregate Cache

Join queries referencing partitioned tables are of the form $Q(R_1, \ldots, R_t) = R_1 \bowtie_{c_1(R_1,R_2)} \cdots \bowtie_{c_{t-1}(R_{t-1},R_t)} R_t$, where each table R_i has the partitioning $\mathbb{P}(R_i) = \{R_{i,1}, \ldots, R_{i,k_i}\}$, for all $i \in \{1, \ldots, t\}$. Without caching some of the subjoins, the database engine needs to compute all possible join combinations of the involved number of tables t and partitions $\mathbb{P}(R_i)$ to build a complete result set. The result of Q is a union of all $k_1 \times \ldots \times k_t$ subjoins i.e., $Q(R_1, \ldots, R_t) = \bigcup_{(j_1,j_2,\ldots,j_t) \in \mathbb{J}_{noCache(Q)}} R_{1,j_1} \bowtie_{c_1(R_1,R_2)} \cdots \bowtie_{c_{t-1}(R_{t-1},R_t)} R_{t,j_t}$, with $\mathbb{J}_{noCache}(Q) = \{1, \ldots, k_1\} \times \cdots \times \{1, \ldots, k_t\}$.

To evaluate $Q(H, I, D)$ from Equation 1, joining three tables with two partitions each, that adds up to a total of $2^3 = 8$ subjoins to be unified: $Q(H, I, D) = \bigcup_{(j_1,j_2,j_3) \in \mathbb{J}_{noCache}(3)} (H_{j_1} \bowtie_{H[A]=I[A]} I_{j_2} \bowtie_{I[B]=D[B]} D_{j_3})$, where $\mathbb{J}_{noCache}(3) = \{delta, main\} \times \{delta, main\} \times \{delta, main\}$.

2.3.2 Join with Aggregate Cache

When using the aggregate cache, the result set from joining all main partitions is already cached (i.e., $R_{1,main} \bowtie \cdots \bowtie R_{t,main}$) and the total number of subjoins computed for delta compensation is reduced to $2^t - 1$: $\mathbb{J}_{withCache}(t) = \mathbb{J}_{noCache}(t) \setminus \{main\}^t$. For our example from Fig. 4, the subjoin #8 does not need to be recomputed as it is cached. However, all other subjoins in Equation 1 are evaluated during delta compensation.

3. ENTERPRISE APPLICATION CHARACTERISTICS

In this section, we give an overview of enterprise application characteristics, that can be utilized to speedup processing of join queries in the aggregate cache. We have analyzed several enterprise applications including financial and managerial accounting, materials management, and customer relationship management and found out that they all share schema design and workload patterns.

3.1 Schema Design Patterns

In all analyzed application domains, we identified tables with similar design patterns, namely *header*, *item*, *dimension*, *text*, and *configuration* tables.

A *header* table describes common attributes of a single business object. In a financial accounting application, for example, this includes attributes such as the fiscal year and the type of the particular business transaction. In materials management the header tuple stores attributes such as the warehouse origin and destination, and the date and time of a goods movement.

To each header tuple, there are a number of corresponding tuples in an *item* table. Item tuples represent entities that are involved in a business transaction. For instance, all products and the corresponding amount for a sale or materials and their amount for a goods movement are stored in the items table. A header tuple and all corresponding item tuples are also referred to as a *business object* since they are modeled as part of a business transaction.

Additionally, attributes of the header and item tables refer to keys of a number of smaller tables. Based on their use case we categorize them into *dimension*, *text*, and *configuration* tables. *Dimension* tables manage the existence of entities, such as accounts and materials. Especially companies based in multiple countries have *text* tables to store strings for dimension table entities in different languages (e.g., product names). *Configuration* tables enable system adoption to customer specific needs and business processes.

3.2 Application Workload Patterns

According to the table classifications, different workload patterns occur. Not surprisingly, there is a high insert load on tables that contain transactional data (i.e., header and item) compared to dimension, text, and configuration tables.

In many domains, entire *static business objects* are persisted in the context of a single transaction. Therefore, the header and corresponding item tuples are inserted within the same transaction and never changed thereafter. In financial applications, it is even required from a legal perspective that *booked* transaction cannot be deleted, but only changed with the insertion of a counter booking transaction.

In some domains such as customer relationship manage-

ment and sales, items may be added to a header at a later point in time. This could be the case when a customer adds products to an order. As [24] analyzed a number of enterprise systems, there is only a small amount of updates and deletes compared to inserts and selects on the header and item tables.

Analyzing aggregate queries of the examined applications, a join between header and their corresponding item tuples is very common. Additionally, the analytical queries extract item properties, text strings, and calculation rules from dimension, text, and configuration tables. Those three table categories do have a number of properties in common: There are rarely inserts, updates, or deletes and they contain only a few entries compared to header and item tables.

In the next section, we briefly discuss partition pruning techniques and introduce matching dependencies, and then, in Section 5, we describe how each mentioned enterprise application characteristic can be captured in the application design to allow very efficient query processing with aggregate cache by leveraging the join partition pruning techniques.

4. PARTITION PRUNING AND MATCHING DEPENDENCIES

In this section, we first formally define join pruning for partitioned tables, discuss how these techniques can be applied to columnar tables, and then introduce the concept of matching dependencies which can be leveraged to model and enforce object-aware, temporal relationships.

Each column of a table in SAP HANA is dynamically partitioned into main and delta storages, both columnar stores, hence the columnar tables have a natural mix of vertical partitioning (i.e., columns) and horizontal partitioning (i.e., delta and main). Traditional techniques for partition pruning could be applied to this type of tables during query processing [13, 26]. Formally, horizontal partitioning of a table R is a set of disjoint subsets $\{R_1, ..., R_n\}$ of R such that $R = R_1 \bigcup R_2 \bigcup ... \bigcup R_n$. A table partitioned based on a specified partitioning scheme, must be processed during query execution by accessing each of its partitions based on the query semantics [23]. As some of the partitions may not be relevant to the query, partition pruning methods can be applied to avoid accessing irrelevant data. *Logical partition pruning* refers to methods of pruning based on the definitions of the partitioning scheme (usually applied during query optimization), while *dynamic partition pruning* is a method of pruning based on runtime properties of the data not on the static partitioning scheme (usually applied at query execution time). For dynamic partition pruning, the execution plan can be built with extra physical operators which will allow partition pruning during query execution based on properties which hold for the current instance of the database.

DEFINITION 1. **Join Pair-Wise Partition Pruning** *by a join operator $\bowtie_q$. Let $\{R_1, ..., R_n\}$ be a horizontal partitioning for a table R. Let $\{S_1, ..., S_m\}$ be a horizontal partitioning for a table S. We say that the pair (R_j, S_k) is logically pruned by the join operator $\bowtie_{q(R,S)}$ if and only if $R_j \bowtie_{q(R,S)} S_k = \emptyset$ for any instances of the tables R and S. Let $\{R_1^i, ..., R_n^i\}$ be an instance of the table R, R^i, and $\{S_1^i, ..., S_m^i\}$ be an instance of the table S, S^i. We say that the pair of instances (R_j^i, S_k^i) is dynamically pruned by the join operator $\bowtie_{q(R,S)}$ if and only if $R_j^i \bowtie_{q(R,S)} S_k^i = \emptyset$.*

A simple example of dynamic partition pruning for a join $R \bowtie S$ is pruning all subjoins of the form $R_j \bowtie S_k$ if the partition R_j is empty at the query execution time.

One type of dynamic join partition pruning is based on the range values of the join attributes in each partition (see Example 1). This type of partition pruning is relevant to our solution for addressing performance problems of join queries using the aggregate cache. Note that successful pruning is achieved when the value ranges of the join attributes do not overlap among partitions.

EXAMPLE 1. **Dynamic join partition pruning based on range values.** *Let $\{R_1, R_2\}$ be a horizontal partitioning of $R(A)$. Let $\{S_1, S_2\}$ be a horizontal partitioning of $S(A)$. A pair (R_1, S_2) is pruned by the join operator $\bowtie_{R[A]=S[A]}$ if it can be determined that the instances S^i and R^i are such that $R_1^i \bowtie_{R[A]=S[A]} S_2^i = \emptyset$.*

One runtime criteria for determining that the pair (R_1^i, S_2^i) is pruned by $\bowtie_{R[A]=S[A]}$ could be based on the current range values of the attribute A in the relations R and S. Note that the tuples with NULL value on A will not participate in the join.

Let $max(R_1^i[A]) = max\{t[A] | t \in R_1^i\}$,
$min(R_1^i[A]) = min\{t[A] | t \in R_1^i\}$,
$max(S_2^i[A]) = max\{t[A] | t \in S_2^i\}$,
$min(S_2^i[A]) = min\{t[A] | t \in S_2^i\}$.
If $max(R_1^i[A]) < min(S_2^i[A])$ or
$max(S_2^i[A]) < min(R_1^i[A])$ then $R_1^i \bowtie_{R[A]=S[A]} S_2^i = \emptyset$.
Proof: If $max(R_1^i[A])$ and $min(R_1^i[A])$ are defined as above, then $R_1^i = \sigma_{min(R_1^i[A]) \leq R[A] \leq max(R_1^i[A])}(R)$.
Similarly, $S_2^i = \sigma_{min(S_2^i[A]) \leq S[A] \leq max(S_2^i[A])}(S)$.
Then $R_1^i \bowtie_{R[A]=S[A]} S_2^i =$
$R_1^i \bowtie_{q(R,S)} S_2^i = \emptyset$ with $q(R,S) = (R[A] = S[A] \wedge$
$min(R_1^i[A]) \leq R[A] \leq max(R_1^i[A]) \wedge$
$min(S_2^i[A]) \leq S[A] \leq max(S_2^i[A]))$
because the join predicate $q(R,S)$ is a contradiction if $max(R_1^i[A]) < min(S_2^i[A])$ or $max(S_2^i[A]) < min(R_1^i[A])$.

4.1 Matching Dependencies

Matching dependencies are well studied in the literature, for example in [8], and can be used for defining extra relationships between matching tuples of two relations. The matching dependencies extend functional dependencies and were originally introduced with the purpose of specifying matching rules for object identifications [7]. However, matching dependencies can be defined as well in a database system, and can be used to extend functional or inclusion dependencies supported in RDBMSs. They can be used to impose certain constraints on the data, or they can be dynamically determine for a query; they can be used for semantic transformations (i.e, query rewrite), and optimization of the query execution. We adopt here a variant of the definition for matching dependencies introduced in [8].

DEFINITION 2. **A matching dependency** *MD on two relations (R, S) is defined as follows: The matching dependency $MD = (R, S, (q_1(R,S), q_2(R,S))$, where q_1 and q_2 are two predicates, is defined as a constraint of the form:*

$$\forall r \in R \wedge \forall s \in S : q_1(r[A], s[A]) \implies q_2(r[B], s[B]) \quad (2)$$

Note that if a matching dependency $MD = (R, S, (q_1(R,S), q_2(R,S))$ holds, it can be used for query

optimization, e.g., join pruning, semantic transformations, as the following equality holds for any instance of R and S.

$$R \bowtie_{q_1(R,S)} S = R \bowtie_{q_1(R,S) \wedge q_2(R,S)} S$$

Section 5 details specific matching dependencies defined to model object-aware semantic constraints among tables, and how they can be used for dynamic join pruning for partitioned tables in this context.

5. OBJECT-AWARE JOINS

We discuss in this section some practical design problems of how matching dependencies can be defined, enforced, and used for dynamic join partition pruning as well as join predicate push downs in a RDBMSs. We also discuss how specific semantic constraints among relations can be defined using MDs. While *object-awareness* can refer to various semantic constraints, we focus on temporal locality with regards to record insertion in this paper.

Matching dependencies can be used to impose constraints on two relations which are usually joined together in queries: if two tuples agree on some attributes, then they must agree on some other attributes as well [8]. An example: if two tuples agree on the product attribute, then they must agree on the product category attribute as well. By adding a temporal attribute such as an auto-incremented transaction id, we can use this type of constraint to model temporal locality semantics among relations.

As discussed in Section 3, specific application scenarios have naturally the following semantic constraints among pairs of tables: if a tuple r is inserted in the table R, then a matching tuple s (where $r[A] = s[A]$, $A \subseteq attr(R)$ and $A \subseteq attr(S)$) is inserted in the table S in *the same transaction* as r is inserted, or within *a small range of transactions* from r. To model this type of semantic constraints, MDs can be used. The MDs themselves, as defined here, are strong constraints which are enforced in the database. The constraint that records in related tables are inserted in transactions close to each other, is a temporal soft-constraint. When this temporal constraint holds, using the proposed MDs will guarantee dynamic pruning as matching tuples reside all in delta store or all in main store. If the temporal soft-constraint doesn't hold, the dynamic pruning will not be possible. In both cases, the join pruning using these MDs will be correct. An interesting future work is to model (and dynamically discover) this type of soft-constraints without using strong MDs which require extra storage.

The following design can be imposed to define the MDs between two tables R and S which will allow dynamic partition pruning for join queries using the aggregate cache.

A new column $R[tid_R]$ is added which records the temporal property of the tuples in R as they are inserted into R. We set $r[tid_R]$ to the auto-incremented transaction identifier (generally available in an IMDB) during which the new tuple r is inserted, a value larger than any existing value already in the column $R[tid_R]$. For the table S, which is joined with the table R on the matching predicate $R[A] = S[A]$, a new column $S[tid_R]$ is added which is set, at the insert time, to the value of $R[tid_R]$ of the unique matching tuple in R, if at most one matching tuple exists, e.g. $R[A]$ is the primary key of R. While this does not constrain s to be inserted at a later time than r, the MD captures the temporal relation between matching tuples in r and s. This scenario is used for our benchmarks described in Section 6 for which the corresponding MD defined in Equation 3 holds.

$$\begin{aligned} MD_{R,S} = \quad & (R[A, tid_R], S[A, tid_R], (R[A] = S[A]), \\ & (R[tid_R] = S[tid_R])) \end{aligned} \tag{3}$$

In the current prototypical implementation which extends the class of aggregate queries supported by the SAP HANA aggregate cache with join aggregates, MDs are enforced on the application level during record insertion. Theoretically, MDs can be implemented in the database if the database supports general MDs as new type of constraints as proposed in [8]. Then, our specific MDs can be defined as part of the meta data and can be enforced similarly to other constraints such as checking for referential integrity.

5.1 Join Pruning

The matching dependency $MD_{R,S}$ from Equation 3 can be used to perform dynamic pruning for the joins $R \bowtie_{R[A]=S[A]} S$. Let's assume that the tables R and S are partitioned as described in Example 1: $R = (R_1, R_2)$ and $S = (S_1, S_2)$, with S_1 and R_1 containing the most recent tuples of R and S, respectively. Also, the matching dependency from Equation 3 holds. The dynamic pruning described in Example 1 can be attempted. Equation 4 shows the derived join predicate which must evaluate to false for pruning a subjoin.

$$\begin{aligned} & R_1 \bowtie_{R[A]=S[A]} S_2 \\ & \text{using } MD_{R,S} \text{ from Eq. 3} \\ & = R_1 \bowtie_{R_1[A]=S_2[A] \wedge R_1[tid_R]=S_2[tid_R]} S_2 \\ & = R_1 \bowtie_{q(R_1,S_2)} S_2 \\ & \text{where } q(R_1, S_2) \text{ uses min()/max() as in Example 1} \\ & q(R_1, S_2) = \\ & R_1[A] = S_2[A] \wedge R_1[tid_R] = S_2[tid_R] \wedge \\ & min(R_1[tid_R]) \leq R_1[tid_R] \leq max(R_1[tid_R]) \wedge \\ & min(S_2[tid_R]) \leq S_2[tid_R] \leq max(S_2[tid_R]) \end{aligned} \tag{4}$$

If $q(R_1, S_2)$ can be proven to be a contradiction then $R_1 \bowtie_{R[A]=S[A]} S_2 = \emptyset$. The above technique for dynamic pruning must be done during runtime and it will be always correct as long as $MD_{R,S}$ holds. For example, a prefilter condition defined as in Equation 5, if true, assures that $q(R_1, S_2)$ is a contradiction hence the subjoin $R_1 \bowtie_{R[A]=S[A]} S_2 = \emptyset$ can be dynamically pruned.

$$\begin{aligned} & max(R_1[tid_R]) < min(S_2[tid_R]) \vee \\ & min(R_1[tid_R]) > max(S_2[tid_R]) \end{aligned} \tag{5}$$

In the case of tables in a columnar IMDB, $min()$ and $max()$ can be obtained from current dictionaries of the respective partitions. The pruning will succeed if the prefilter from Equation 5 is true. Otherwise, the pruning will correctly fail if, for example, $MD_{R,S}$ holds but S_2 contains matching tuples from R_1 i.e., the prefilter is false in his case. For an empty partition R_j, we define $min()$ and $max()$ such that the prefilter is true for all join pairs (R_j, S_k).

When the database is aware of the enterprise application characteristics (Section 3) based on their object semantics, join partition pruning can be used to efficiently execute join queries with or without the aggregate cache. We refer to this type of joins as *semantic* or *object-aware* joins.

Let us consider the join query as discussed in Section 2.3 $Q(H, I) = H \bowtie_{H[PK]=I[FK]} I$ joining a header table H and item table I on the join condition, that the primary key $H[PK]$ matches the foreign key $I[FK]$. The matching dependency defined in Equation 6 captures this object-

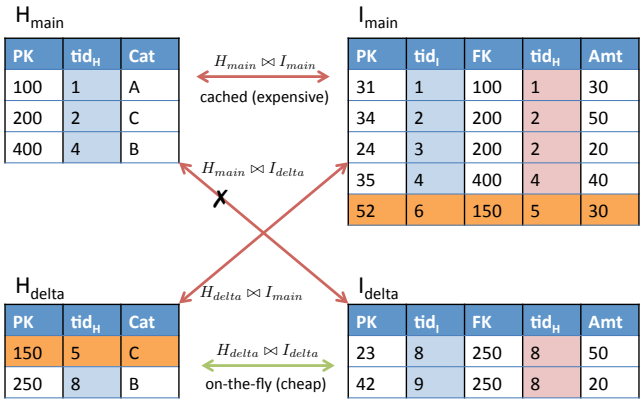

Figure 5: Join pruning example between a header and item table with main and delta partitions.

aware semantic constraint, where the attributes $H[tid_H]$ and $I[tid_H]$ are new attributes especially added for the MD:

$$MD_{H,I} = (H, I, (H[PK] = I[FK]), (H[tid_H] = I[tid_H])) \quad (6)$$

The MD is enforced during record insertion in the item table I by setting the attribute $I[tid_H]$ to $H[tid_H]$ of the matching tuple in the header table H. As illustrated in Fig. 5, the item table I has two temporal attributes: $I[tid_H]$ is used to capture the MD with the header table H and $I[tid_I]$ can be used for MDs with other tables that join on the primary key of I.

After an insert into H and I, if there was no merge operation yet, all new matching tuples are in the delta partitions. Therefore, for a delta compensation, we only need to compute the subjoin $H_{delta} \bowtie I_{delta}$ and unify the results with the cached aggregate $(H_{main} \bowtie I_{main})$. Dynamic pruning for the remaining subjoins $H_{main} \bowtie I_{delta}$ and $H_{delta} \bowtie I_{main}$ can be performed if the prefilter condition as defined in Equation 5 holds:

$$max(H_{main}[tid_H]) < min(I_{delta}[tid_H]) \longrightarrow H_{main} \bowtie I_{delta} = \emptyset$$
$$max(I_{main}[tid_H]) < min(H_{delta}[tid_H]) \longrightarrow H_{delta} \bowtie I_{main} = \emptyset$$

Fig. 5 depicts an example of join dynamic pruning for the subjoin $H_{main} \bowtie_{H[PK]=I[FK]} I_{delta} = \emptyset$ as the prefilter $min(I_{delta}[tid_H]) > max(H_{main}[tid_H)$ (i.e., $8 > 4$) is true. However, the subjoin $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ cannot be pruned: the prefilter $max(I_{main}[tid_H]) < min(H_{delta}[tid_H])$ (i.e., $5 < 5$) is false. Fig. 5 highlights the matching tuples in H_{delta} and I_{main} which prevent the join pruning for $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$.

5.2 Delta Merge Operation

The incremental maintenance of the aggregate cache takes place during the online merge process which propagates the changes of the delta storage to the main storage [17]. When employing an object-aware join between a header and an item table, if the timing of the delta merge processes could be adjusted for the two tables then the join pruning success rate for delta-compensation and maintenance operations could be maximized. While the dynamic join pruning will always be correct, join pruning is more likely to succeed when the merge processes of related transactional tables are synchronized, rather than when the tables are merged independently, because there is little overlap between delta and main partitions. The example from Fig. 5 shows the case when one of joins $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ cannot be pruned because table I has been merged before H while the join $H_{main} \bowtie_{H[PK]=I[FK]} I_{delta}$ is pruned successfully.

5.3 Join Predicate Pushdown

In case the join pruning does not succeed, we can still leverage the temporal information through the enforced matching dependencies to optimize join processing. Consider the example depicted in Fig. 5, with an overlap of matching tuples in the H_{delta} and I_{main} partitions, which in turn implies that join pruning between H_{delta} and I_{main} cannot succeed.

Based on the matching dependencies, a query optimizer should be able to infer new predicates that can then be pushed down as local filter predicates to the respective partitions, H_{delta} and I_{main}, before evaluating the subjoin. In our example, the subjoin $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ can be rewritten using $MD_{H,I}$ from Eq. 6 and using the runtime domain properties of the attributes $H[PK, tid_H]$ and $I[FK, tid_H]$ follows:
$(\sigma_{f(H)} H_{delta}) \bowtie_{H[PK]=I[FK] \wedge H[tid_H]=I[tid_H])} (\sigma_{f(I)} I_{main})$
with local predicates defined as:
$f(I) = (I[tid_H] >= min(H_{delta}[tid_H]))$ and
$f(H) = (H[tid_H] <= max(I_{main}[tid_H]))$.

Especially the evaluation of $f(I)$ on the I_{main} partition seems to be promising since we do not have to do a full table scan for every potential join partner of H_{delta} but can limit the partition to only consider the relevant records. In the example from Fig. 5, we would only need to check all records for which $f(I) = (tid_H >= 5)$ is true, since $5 = min(H_{delta})[tid_H]$. Similarly, $f(H) = (tid_H <= 5)$, as $5 = max(I_{main}[tid_H]$.

5.4 Applying Join Pruning to Multi Partitions

Up to this point, we have only considered a table to be partitioned into delta and main storage as it is the case in the general main-delta architecture. However, tables can be further partitioned using specific partitioning schemes, for example, as proposed in [25], for *data aging* or *archiving*. We consider a scenario where the columnar tables H and I are partitioned based on the age of the tuples into one *hot* and one *cold* partition. Given the case, that the hot and cold partitioning is static, we can employ a mix of logical and dynamic partition pruning. Thus, the tables H and I each have four partitions $X^c_{main}, X^c_{delta}, X^h_{main}, X^h_{delta}$, where X is any of the tables H or I. There are several interesting properties in this scenario:

- The cold partition X^c_{delta} contains only the updated tuples from X^c_{main} if any. X^c_{delta} is empty in general.
- New tuples are inserted in the hot delta partition X^h_{delta} only.
- The delta-merge operation affects only the hot partition X^h_{main} which is much smaller than $X^h_{main} \cup X^c_{main}$.
- The subjoins on cold and hot partitions of the form $I^c_v \bowtie H^h_w$ with $v, w \in \{main, delta\}$, are always empty, given a consistent aging definition on related tables. These subjoins can be logically pruned. Dynamic pruning can also be applied, almost always, for subjoins between any cold and hot partitions $X^c_v \bowtie Y^h_w$ with $v, w \in \{main, delta\}$.

- There are two aggregate caches defined for subjoins on cold and hot partitions, respectively: $H^c_{main} \bowtie I^c_{main}$, and $H^h_{main} \bowtie I^h_{main}$. The delta-merge operation executed most often affects only the aggregate cache built on hot partitions $H^h_{main} \bowtie I^h_{main}$, hence this is the one which needs to be rebuilt after each merge. The aggregate cache built on cold partitions will be rebuilt very rarely, when tuples are aged into the cold partitions.

To evaluate a query using these aggregate caches, many of the subjoins used for the main-main and delta-main compensation can be pruned. In particular, the subjoins referencing both cold and hot partitions can be partially pruned logically, given a consistent aging definition.

6. EXPERIMENTAL EVALUATION

We first present experimental results for aggregate cache maintenance (in Section 6.1) on the current SAP HANA implementation, performed in a mixed workload of updates and aggregate queries.

Secondly, we assess the performance of query execution without and with aggregate cache for the class of join aggregate queries for which dynamic pruning is performed during delta-compensation. The experimental results are obtained on a prototype implementation which extends the aggregate cache for join queries. For these experiments, we use two benchmarks, the CH-benCHmark [10] based on TPC-H[1] and TPC-C[2], and a benchmark built based on data and workloads from a financial and managerial accounting application of a production ERP system. Opposed to standardized benchmarks such as TPC-C or TPC-H, the second benchmark especially reflects the characteristics of enterprise applications, generating mixed workloads. For this benchmark, the schema contains three tables: a header table *Header* with 35 million tuples, an item table *Item* with 330 million tuples, and a dimension table *ProductCategory* with less than 2000 tuples.

```
SELECT D.Name AS Category, SUM(I.Price) AS
    Profit
FROM  Header AS H,
      Item AS I,
      ProductCategory AS D
WHERE I.HeaderID = H.HeaderID
      AND I.CategoryID = D.CategoryID
      AND D.Language = 'ENG'
      AND H.FiscalYear = 2013
GROUP BY I.CategoryID
```

Listing 1: Benchmark sample query

We modeled a mixed OLTP/OLAP workload, based on input from interviews and workload traces with an industry customer. The analytical queries simulate multiple users, using a profit and loss statement analysis tool. The SQL statements calculate the profitability for different dimensions including the product category (as mentioned in Section 3) by aggregating debit and credit entries. Listing 1 shows a simplified sample query that calculates how much profit the company made with each of its product categories. The inserts were replayed by using the timestamps in the base data. Deletes and updates were not part of our evaluation workload because they only had a relative low presence in the analyzed ERP production system workloads. All benchmarks were run on a server with 64 Intel Xeon X7560 processor cores and 1 TB of main memory.

[1]http://www.tpc.org/tpch/
[2]http://www.tpc.org/tpcc/

6.1 Maintenance Strategies

We first discuss how our aggregate cache (defined on the main partitions) performs in a mixed workload of inserts and aggregate queries compared to using materialized views with classical maintenance strategies. The statements in this workload reference a single table. Materialized views are defined on main and delta partitions and must be maintained for any delta store changes. Traditional maintenance strategies ensure that a materialized view is always up-to-date when used during query execution: *eager incremental* strategy maintains the materialized views with every insert operation [2], while *lazy incremental* strategy keeps a log of insert operations and maintains the materialized views before it is used [32]. The aggregate cache is defined on the main partition only as presented in this work - and delta-compensation is done at the query time (as shown in Fig. 3). In this experiment, the delta-merge operation is not performed. The insert rates in this experiment bear upon an individual materialized aggregate. In other words, they reflect the number of base data inserts affecting this particular materialized aggregate in relation to the number of times this aggregate is used by read-only queries.

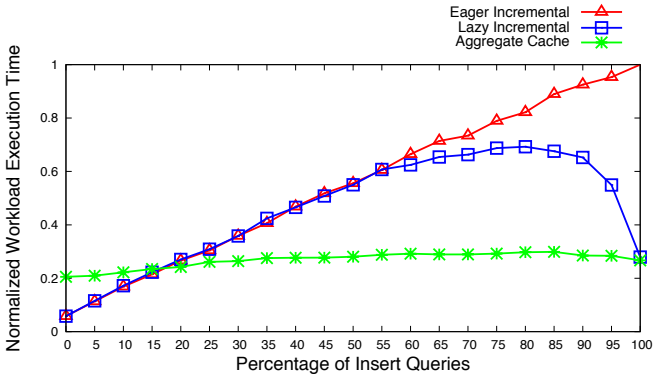

Figure 6: Mixed workload performance using the SAP HANA aggregate cache compared to using materialized views with classical maintenance strategies with varying insert ratios.

The results are depicted in Fig. 6 and reveal that in *write-heavy* scenarios, the materialized view maintenance overhead is very high because materialized views are maintained for any delta changes, either by maintaining the materialized view with every base data modification (eager), or before a read-only query (lazy). Read-only queries using the aggregate cache have an overhead for delta-compensation which is much smaller, in this scenario, compared to the maintenance overhead of materialized views. In a *read-mostly* workload, the materialized view maintenance overhead is marginal as changes do occur very infrequent and the materialized view can directly be used without maintenance by the read-only queries. With an increasing insert ratio however, their maintenance costs increase while our aggregate cache delivers nearly constant execution times due to the fact that the aggregate cache is defined on main stores. Yet, read-only queries using the aggregate cache have an overhead for delta-compensation even if delta store is very small. For insert ratios above 15 percent, this compensation overhead is outweighed by the maintenance overhead by the

classical strategies, with the aggregate cache being superior. The shift to a read-mostly overall workload, as described in [25], is not based on number of statements, but on the high percentage of the read-only statements' execution time out of the total workload execution time, which does not necessarily contradict with this experiment.

6.2 Memory Consumption Overhead

In our scenario, we have three tables (header, item, and one dimension table) that need to be extended with the temporal information in order to prune the subjoins. In total, this adds up to the following five additional attributes:

- Header table: $Header[tid_{Header}]$
- Item table: $Item[tid_{Item}, tid_{Header}, tid_{ProductCategory}]$
- Dimension table: $ProductCategory[tid_{ProductCategory}]$

The measured memory consumption, for delta stores, with 2.7 thousand header tuples, 270 thousand item tuples was 78,553 KB compared to 69,507 KB without the temporal information. This is an overhead of 13 percent only for the delta partitions. In main partitions, based on our dataset with 35 million header and 330 million item records, this results in an overhead of 10 percent because of better compression applied to main stores only.

6.3 Insert Overhead

To ensure the matching dependencies of records with foreign keys, every insert operation involving a foreign key attribute needs to find the related *temporal* attribute of the matching tuple. To quantify this overhead, we have measured the time for the look-up of the $Header[tid_{Header}]$ attribute for every insert of a record in the $Item$ table.

The results show that the record insertion in the $Item$ table without the tid_{Header} lookup, and without any referential integrity checks takes about 50 percent of the record insertion time with referential integrity checks. The lookup-up of the matching tid_{Header} value in the $Header$ table takes 20 percent of the time of referential integrity checks. When the number of records in the $Header$ table increase, the look-up slightly increases up to 30 percent. However, this look-up can be combined with the required integrity check for newly inserted records with foreign keys that must find the existing primary key record. Also, we argue that with a shift to a read-mostly workload in enterprise systems [25], the impact of the insert overhead can be regarded as negligible compared to resource-intensive aggregate queries.

6.4 Join Pruning Benefit

To measure the benefit of our proposed join pruning approach, we have created three experiments in which we compare the following four different join query execution strategies:

- *Uncached aggregate query*: this executes an aggregate query without using the aggregate cache as described in Section 2.3.1,
- *Cached aggregate query without pruning*: while the main partition is cached, all remaining subjoins including any delta partitions must be computed for the delta-compensation as described in Section 2.3.2,
- *Cached aggregate query with empty delta pruning*: as an optimization to the previous strategy, we omit subjoins with empty delta partitions as it is the case with the $ProductCategory$ dimension table, and
- *Cached aggregate query with full pruning*: this strategy uses the dynamic pruning concept as described in Section 5.

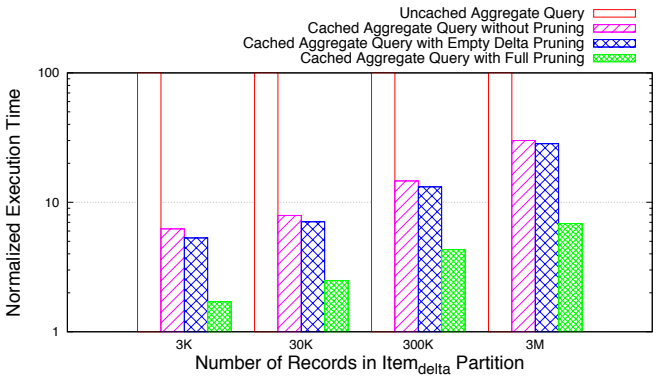

Figure 7: Join performance with different join query execution strategies based on different delta sizes of $Item_{delta}$ and $Header_{delta}$.

The first experiment as illustrated in Fig. 7 measures the execution times of the four different join approaches based on five different delta sizes of the $Item$ table ranging from 300 thousand to 3 million records. The delta partition of the $Header$ table contains approximately one tenth of the $Item_{delta}$ table records and the delta partition of the $ProductCategory$ table is empty. The workload for this benchmark contains 100 aggregate join queries similar to the query in Listing 1. Fig. 7 shows the average normalized execution times of these queries. We see that for small delta sizes, a query using the cached aggregate can be answered by an order of magnitude faster than when not using the aggregate cache. With an increasing number of records in $Item_{delta}$ and $Header_{delta}$ the query execution time increases regardless of the applied join pruning strategy because the newly inserted records in the delta partitions have to be aggregated during the delta-compensation to compute the query results. While the empty delta pruning delivers performance improvements of around 10 percent, the execution times using the full pruning approach is, on average, four times faster than using the cached aggregates without any dynamic join pruning.

In the second experiment, whose results are illustrated in Fig. 8, we have created a mixed workload consisting of insertions of records into $Header$ and $Item$ tables and the execution of aggregate join queries. The starting point is an empty delta partition of both the $Header$ and $Item$ tables. The benchmark then starts the insertion of records in both tables including the look-ups of *tid* attributes. At the same time, we monitor the execution times for aggregate queries executed with the four different strategies. The benchmark has varying frequencies of aggregate queries with respect to the number of inserts which is realistic in an enterprise application context. For example, we can see that there are many aggregate queries at the point of time when $Item_{delta}$ contains around 1 million records.

The results in Fig. 8 show that while the empty delta pruning has minor performance advantages over not pruning

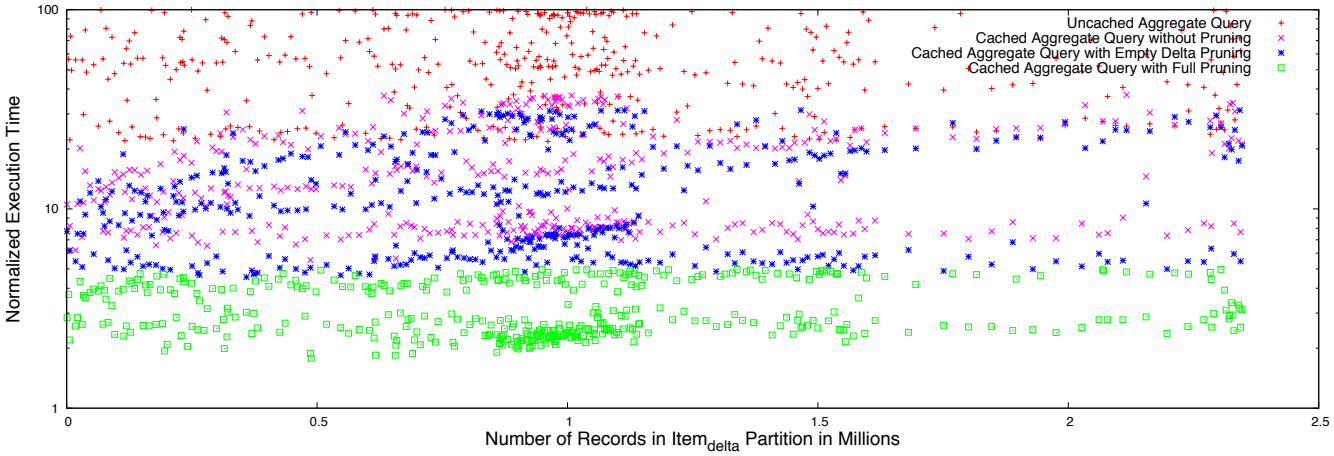

Figure 8: Join performance with different join query execution strategies based on growing delta sizes.

at all, our proposed join pruning approach outperforms both when the delta partitions have non-trivial sizes. We also see that the runtime variance of queries with or without the aggregate cache but without any pruning is very high. This can be explained by a high concurrent system load which, due to the complexity of the monitored aggregate queries, results in variable execution times.

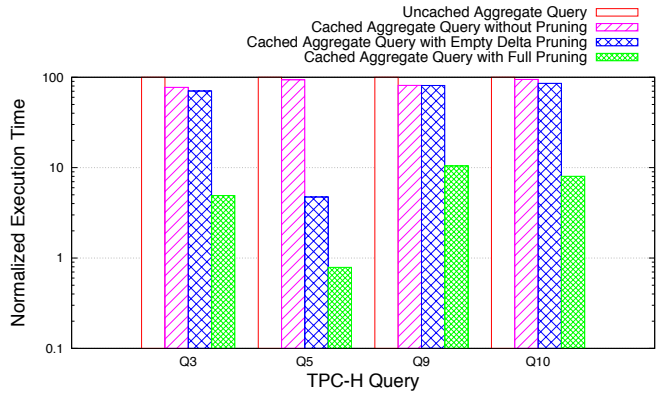

Figure 9: Join performance with different join query execution strategies of TPC-H queries based on CH-benCHmark [10].

As a third experiment for the join pruning benefit, we have taken four analytical TPC-H queries of the CH-benCHmark [10] and analyzed their performance with the four join approaches. The four queries (Q3, Q5, Q9, and Q10) were selected because they are fully supported by the aggregate cache and join more than three tables as in our previous benchmarks. We chose the scale factor 200 for this experiment, which yields 60 million records in the *orderline*, 20 million records in the *orders* table and less records in the remaining tables according to Funke et al. [10]. As proposed in the CH-benCHmark setup, we have populated the delta partitions of the *orders*, *neworder*, *orderline*, and *stock* tables with five percent of total records per table (i.e., the *orderline* table contains 3 million records in the delta and 57 million records in the main), reflecting a mixed workload.

The results are illustrated in Fig. 9 and reveal that for aggregate queries joining more than three tables, the benefit of the aggregate cache is only marginal if delta-compensation during the query execution doesn't use dynamic join pruning. Pruning empty delta partitions yields a minor improvement while the full join pruning approach can accelerate query execution by up to an order of magnitude compared to an uncached aggregate query.

6.5 Join Predicate Pushdown Benefit

In cases when the join pruning is not successful, we can still leverage the temporal relation between the partitions modeled using the MD constraints. In this experiment, we measure the execution time of the subjoins between $Header_{delta}$ and $Item_{main}$ partitions by using the predicate pushdown explained in Section 5.3. We have three different setups with a varying total number of records in $Item_{main}$, while $Header_{delta}$ has a constant number of 100 thousand records. The results as illustrated in Fig. 10 show that with an increasing number of records participating in the join (i.e., matching the join conditions), the performance of the delta-compensation decreases. By using our predicate pushdown concept, we can see that it can accelerate the join query execution up to a factor of four, especially if the number of matching records is low compared to the overall table size.

6.6 Applying Join Pruning to Multi Partitions

To benchmark the performance of the join pruning approach in the presence of multiple partitioned tables as outlined in Section 5.4, we have created an experiment with the $Header$ and $Item$ tables, partitioned in a hot-cold ratio of 1:3 as proposed in [25]. We execute five different aggregate queries with different selectivities, aggregating 100 thousand to 25 million records in the hot partition and measure their performance with different join strategies.

The results are illustrated in Fig. 11 and reveal several insights. First of all, we see that an uncached aggregate query is slightly faster in a partitioned environment, because the scan effort can be reduced. Second, we see that the performance of using a cached aggregate query without pruning is worse in a partitioned environment because of the additional subjoins that are required for delta-compensation. The performance of the full join pruning approach is superior in both partitioning scenarios, speeding up query execution by an order of magnitude.

7. RELATED WORK

Materialized views have received significant attention in academia [1, 2, 12, 32], especially in data warehousing envi-

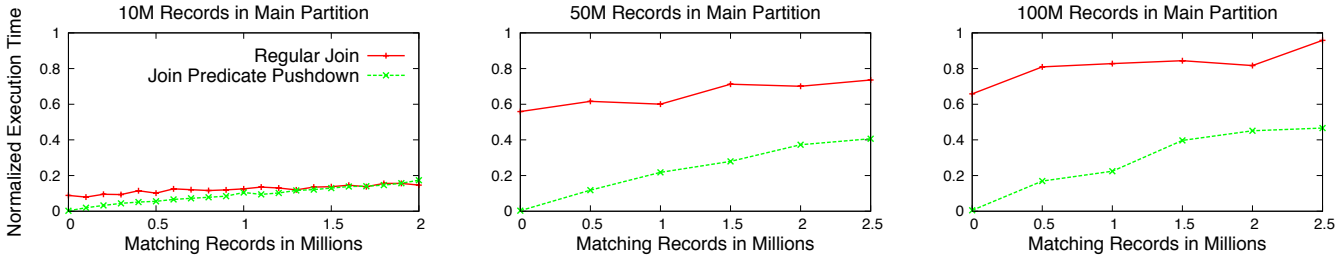

Figure 10: Query execution performance when the subjoin $Header_{delta} \bowtie Item_{main}$ cannot be pruned: with and without the predicate pushdown, based on different $Item_{main}$ partition sizes, and varying in the number of matching records between $Header_{delta}$ and $Item_{main}$.

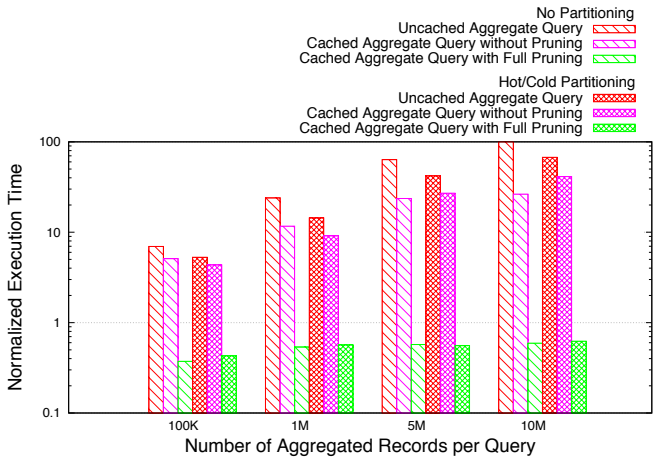

Figure 11: Join performance of different join query execution strategies with unpartitioned and hot/cold partitioned tables. The underlying aggregate queries vary in the number of aggregated records.

ronments [33, 1, 22]. Our techniques for using materialized views are different along multiple dimensions.

First of all, the maintenance timing is not bound to an update of the base data [2], nor it is deferred no later than querying the materialized view [32, 28, 27, 5]. Instead, we maintain the materialized view during the online merge process [17] as our aggregate cache is defined on main partitions. This enables high insert rates and does not imply a maintenance downtime which is not tolerable in mixed workload environments as opposed to data warehouses [4]. Secondly, we do not rely on redundant storage of base data changes, as others do with *auxiliary tables* or *summary tables* [18, 31, 22, 32]. Our delta storage is the primary storage for all inserts and updates performed between two delta merge processes. Our algorithm to calculate the consistent query result of queries using the aggregate cache is similar to the summary-delta tables method introduced in [22], but we do not distinguish between a refresh and propagate phase.

For partitioned tables, several join optimization techniques have been proposed. One of them is to dynamically partition the relations based on workload [26] for improved performance. Another approach is to do logical pruning for horizontally partitioned tables [13]. However, the latter approach is limited to the scenario when the horizontal partitioning attribute matches the join attributes used in the query whereas our implementation supports, by leveraging matching dependency methods, arbitrary join attributes. Also, this approach does not apply to the dynamic partitioning in the general main-delta architecture which we address through dynamic join partition pruning.

While there is an emergence of application-specific databases such as Amazon Dynamo [6] or Google Bigtable [3], we are not aware of a materialized view maintenance and query compensation approach for a general purpose DBMS that leverages the semantics of an enterprise application to increase the performance of aggregate queries using materialized views.

8. CONCLUSIONS AND FUTURE WORK

With the growing requirements of enterprise applications, combining transactional and analytical workloads on a single system, the aggregate cache, a dynamic materialized aggregate engine implemented in SAP HANA, enables the handling of an even higher throughput of aggregate queries generated by multiple parallel users as one mean to scale the system. As admittance in the aggregate cache is directly dependent on the performance of the query execution using the cache, we analyze a special class of aggregate join queries which can be very expensive to compensate. Joins of partitioned tables are challenging in general, but slow down the incremental materialized view maintenance and query compensation of the aggregate cache in particular.

Our analysis of enterprise applications revealed several patterns for their schema design and usage. Most importantly among them, business objects are persisted using a header and item table with additional rather static dimension tables. Moreover, our application workload analysis showed that related header and item records are often inserted within a single transaction or at least within a small time window.

To transport these enterprise application object semantics characteristics into the database, becoming *object-aware*, and optimize the join processing in the aggregate cache, we exploit the concepts of *matching dependencies* and *join pruning* that potentially eliminate expensive joins of partitioned tables. This is achieved by adding temporal attributes at insertion time and use them during run time to dynamically prune subjoins with an empty result set. In addition, we use techniques for *join predicate pushdown*, also based on *matching dependencies*, that can further optimize join processing with aggregate cache when join pruning does not succeed.

The experimental results show that while our approach induces a small overhead for record insertion, the query processing with the aggregate cache using the pruning approach outperforms the non-pruning approach by an average factor of four in the case of three joined tables, and up to an order of magnitude when joining more than three tables or using additional hot and cold partitioning. The join predicate

pushdown can optimize a join, in case the pruning does not succeed, up to a factor of four.

One direction of future work includes improving the performance of delta-compensation process for join queries when invalidations are detected in the main storage in case of updates. While the presented join pruning techniques will always deliver correct results, and deletes do not negatively impact the performance of our solution, we are investigating ways to improve the pruning success rate for data updates by keeping track of updates in the delta storage in a separate negative-delta partition. To this end, another interesting future work is to model (and dynamically discover) the temporal soft-constraints among relations without using strong matching dependencies which require extra storage.

9. REFERENCES

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *ACM SIGMOD*, pages 417–427, 1997.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD*, pages 61–71, 1986.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[4] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. In *ACM SIGMOD*, pages 65–74, 1997.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD*, pages 469–480, 1996.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS*, pages 205–220, 2007.

[7] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[8] W. Fan. Dependencies revisited for improving data quality. In *ACM PODS*, pages 159–170, 2008.

[9] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, pages 45–51, 2011.

[10] F. Funke, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, A. Nica, M. Poess, and M. Seibold. Metrics for measuring the performance of the mixed workload CH-benCHmark. In *TPCTC*, pages 10–30, 2012.

[11] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, pages 105–116, 2010.

[12] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[13] H. Herodotou, N. Borisov, and S. Babu. Query optimization techniques for partitioned tables. In *ACM SIGMOD*, pages 46–60, 2011.

[14] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *ACM PODS*, pages 113–124, 1995.

[15] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, pages 1173–1184, 2013.

[16] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *IEEE ICDE*, pages 195–206, 2011.

[17] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, pages 61–72, 2011.

[18] W. Lehner, R. Sidle, H. Pirahesh, and R. W. Cochrane. Maintenance of cube automatic summary tables. In *ACM SIGMOD*, pages 512–513, 2000.

[19] S. Müller, L. Butzmann, and H. Plattner. Efficient aggregate cache revalidation in an in-memory column store. In *DBKDA*, pages 66–73, 2014.

[20] S. Müller, R. Diestelkämper, and H. Plattner. Cache management for aggregates in columnar in-memory databases. In *DBKDA*, pages 139–147, 2014.

[21] S. Müller and H. Plattner. Aggregates caching in columnar in-memory databases. In *International Workshop on In-Memory Data Management and Analytics (IMDM), VLDB Workshop*, 2013.

[22] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *ACM SIGMOD*, pages 100–111, 1997.

[23] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

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

[25] H. Plattner, M. Faust, S. Müller, D. Schwalb, M. Uflacker, and J. Wust. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, pages 1722–1729, 2014.

[26] N. Polyzotis. Selectivity-based partitioning. In *ACM CIKM*, pages 720–727, 2005.

[27] D. Quass and J. Widom. On-line warehouse view maintenance. In *ACM SIGMOD*, pages 393–404, 1997.

[28] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: asynchronous incremental view maintenance. In *ACM SIGMOD*, pages 129–140, 2000.

[29] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.

[30] J. Wust, M. Grund, K. Hoewelmeyer, D. Schwalb, and H. Plattner. Concurrent execution of mixed enterprise workloads on in-memory databases. In *DASFAA*, pages 126–140, 2014.

[31] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *ACM SIGMOD*, pages 105–116, 2000.

[32] J. Zhou and P. Larson. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

[33] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *ACM SIGMOD*, pages 316–327, 1995.