Leveraging Enterprise Application Characteristics to Optimize Incremental Aggregate Maintenance in a Columnar In-Memory Database

Stephan Müller(✉), Paul Möller, and Hasso Plattner

Hasso Plattner Institute,
University of Potsdam, Potsdam, Germany
{stephan.mueller,paul.mueller,hasso.plattner}@hpi.uni-potsdam.de

Abstract. An analysis of database workloads generated by enterprise applications revealed a mixed workload of short-running transactional and long-running analytical queries. With the latter type of queries containing many aggregate operations, we implemented an efficient aggregate caching mechanism. But the incremental materialized view maintenance is very costly for aggregate queries joining multiple tables. To overcome this problem, we analyzed the characteristics of enterprise applications with respect to the creation of business objects and their persistence in the database layer. We evaluated how the detected patterns can be leveraged to reduce the join operations between the main and delta partitions of the involved tables in a columnar in-memory database. The resulting performance improvements are significant and close to using the caching mechanism with a denormalized schema.

1 Introduction

Until recently, enterprise applications have been separated into online transactional processing (OLTP) and online analytical processing (OLAP). The drawbacks of this separation are complex and costly ETL processes, not up-to-date and redundant data. Further, the analytical applications are often limited in their flexibility due to pre-calculated data cubes with materialized aggregates.

With the rise of columnar in-memory databases (IMDB) such as SAP HANA [1], Hyrise [2] and Hyper [3], this artificial separation is not necessary anymore as they are capable of handling mixed workloads, with transactional and analytical queries on a single system [4]. In fact, a modern enterprise application executes a mixed workload with both – transactional *and* analytical – queries [5]. While the transactional queries are mostly inserts or single selects, the analytical queries are often comprised of costly data aggregations [6]. Having the possibility to run flexible, adhoc analytical queries directly on transactional data with sub-second response times will further lead to an increased workload of aggregate queries.

To speed up the execution of analytical queries with aggregates, *materialized views* have been proposed [7]. Accessing tuples of a materialized aggregate is

W.-S. Han et al. (Eds.): DASFAA 2014, LNCS 8505, pp. 102–116, 2014.
DOI: 10.1007/978-3-662-43984-5_8,

always faster than an aggregation on the fly. The overhead of materialized view maintenance to ensure consistency for changing base data has to be considered, though [8]. Apart from temporary transactional inconsistencies, a downtime is not acceptable in during materialized view maintenance in mixed workload environments.

While existing materialized view maintenance strategies are applicable in columnar IMDBs [9], their specific architecture is well-suited for a novel strategy of caching aggregate queries and applying incremental view maintenance techniques [10]. This is because the storage of columnar IMDBs can be separated into a read-optimized main storage and a write-optimized delta storage. Since the main storage is highly-compressed and not optimized for inserts, all data changes of a table are propagated to the delta storage in order to ensure high throughput. Periodically, the delta storage is combined with the main storage in a process called *merge operation* [11]. The materialized aggregates do not have to be invalidated when new records are inserted to the delta storage, because they are only based on records from the main storage. Instead, the final, consistent aggregate query result, is retrieved by aggregating the newly inserted records of the delta storage on the fly and combining them – using a SQL `UNION ALL` statement – with the materialized aggregate.

One challenge of the proposed aggregate caching mechanism and the involved incremental materialized view maintenance is to handle aggregate queries that are based on joins of multiple tables. These queries require a union of joining all permutations of delta and main partitions of the involved tables, excluding the already cached joins between the main partitions. For a query joining two tables, three subjoins are required, and query joining three tables already requires seven subjoins. This may result in very little performance gains over not caching at all the query on the main partitions. After analyzing the characteristics of enterprise applications, we identified several schema design and data access patterns that can be leveraged to optimize the overall database performance. While these business semantics could potentially be applied to several other aspects for data processing in a columnar IMDB, this paper focuses on an approach to reduce the incremental view maintenance by explicitly leveraging business semantics of applications.

After discussing related work in Sect. 2, we describe the identified enterprise applications characteristics in Sect. 3. Section 4 describes the aggregate cache and strategies to reduce the number of joins for cached queries. We then outline in Sect. 5 how the database engine can obtain information about application characteristics. Our benchmarks in Sect. 6 support the significant speedup potential and Sect. 7 concludes the paper with the main contributions and an outlook on future work.

2 Related Work

A database can have different design goals depending on the application and its characteristics. The CAP theorem is an example of how different design

trade-offs have to be balanced [12]. In fact, there is an emergence of databases that are custom-built for specific applications such as Cassandra[1] or Amazon DynamoDB[2], each with its own design goals according to the characteristics of the application.

The enterprise application characteristics identified and discussed in this paper are used to reduce the incremental view maintenance inherent when introducing materialized views to speed-up analytical queries [8]. The maintenance of materialized views has received significant attention in academia [13,14] and industry [15,16], and the problem of incrementally maintaining aggregate queries with joins has been widely identified [17,18]. However, neither of these approaches use the characteristics of the application to reduce the maintenance effort.

3 Enterprise Application Characteristics

In this section we give an overview of identified enterprise application characteristics, that can be utilized to speedup processing of join queries for the aggregate cache. Two aspects are of essential relevance: what are common patterns of database schema design and workloads.

3.1 Schema Design

In different 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 transaction. E.g., for a sale in a financials system it stores who made the purchase and when the transaction took place. In materials management the header stores attributes common to a single movement of goods like who initiated the movement and also the time it took place.

To each header table entry, there are a number of corresponding tuples in an *item* table. Item entries 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.

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 and lengths (e.g., product names). *Configuration* tables enable system adoption to customer specific needs and business processes.

[1] Distributed key value store focusing on scalability and high availability, http://cassandra.apache.org/.

[2] Managed NoSQL database focusing on cost efficiency, http://aws.amazon.com/dynamodb/.

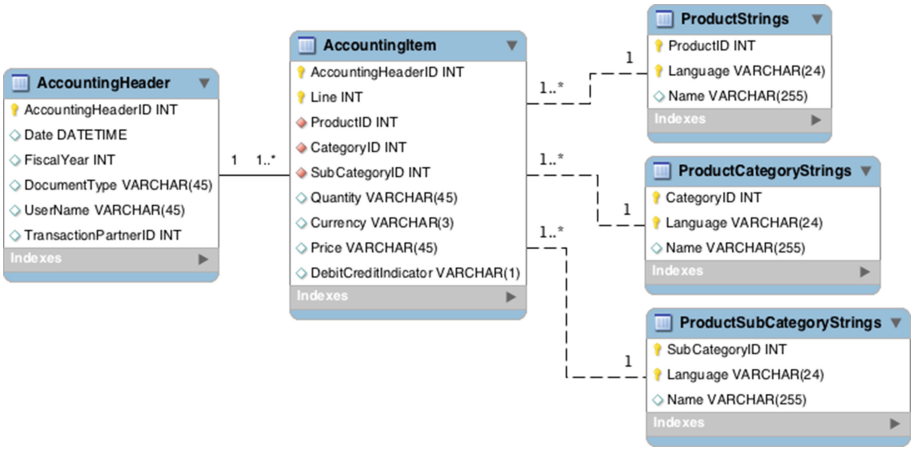

Fig. 1. Simplified schema extract of a financials application.

Figure 1 shows a simplified extract of an example schema of a SAP financials application from an international company producing consumer goods. An accounting header entry refers to a single business action, e.g. a sale or a purchase. It includes the specific time, what kind of accounting document this is, what system user entered the document and with whom the transaction took place. The accounting item table lists all invoiced or billed items. The three text tables store the real names in different languages for the involved products and other item properties.

3.2 Workload Patterns

We also see patterns in how the previously described schemas are used. There is a high insert load from enterprise systems persisting business transactions. Each transaction is represented by one header and a number of item tuples. Therefore the header and item tables have a high insert load and a large tuple count.

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 at the same point in time. E.g. sales or goods movement transactions are persisted as a whole in the database. In some domains such as sales order management, items may be added or changed at a later point in time, e.g. when a customer adds products to his order. As [4] 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. Looking at aggregation join queries, we can almost always see that header entries are joined with their corresponding item entries.

Additionally, the analytical queries extract strings from dimension or text tables. Item tuple values are aggregated according to methods described in configuration tables. The number of involved smaller tables varies between none to

five. 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.

Starting in Sect. 4.2 we describe how each mentioned characteristic allows to reduce the number of table joins necessary when processing a query with a materialized view.

4 Optimizing Incremental Aggregate Maintenance

In this section we give a brief overview of how the aggregate cache utilizes the main-delta architecture to handle mixed workloads as explained in [10] and how enterprise application characteristics can be applied to improve the incremental maintenance of aggregation queries involving a join of multiple tables.

4.1 Architecture Overview

As depicted in Fig. 2, the query processor handles reads and writes to main and delta storage through the SQL interface from the application and delegates aggregate queries to the aggregates caching manager. In case the cache management table (CMT) indicates that the current query has not been cached yet, the query is processed on the main and delta storage. The query result set from the main is being cached and an entry in the CMT is created. Finally the unified result sets from main and delta are delivered back to the application.

As all new inserts are stored in the delta, an already cached query only needs to be executed on the delta storage. The final result set is obtained by unifying

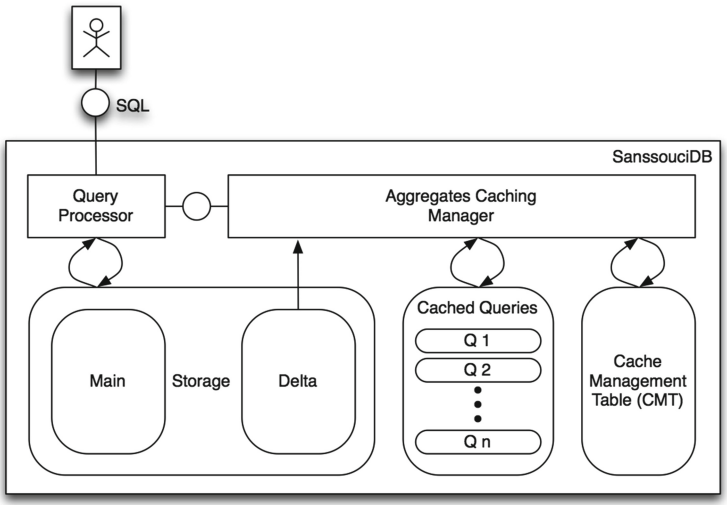

Fig. 2. Aggregate cache architecture.

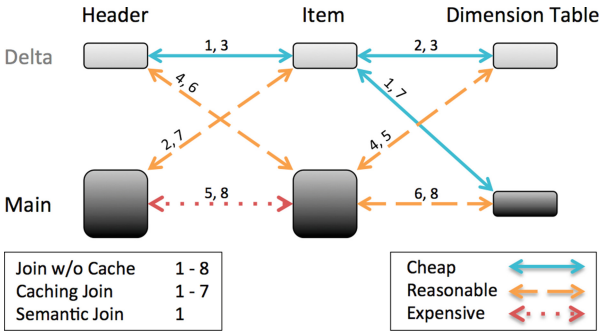

Fig. 3. Caching strategies for a three table join query.

the results from the delta with the cached entry that holds the content of the main storage. Since the delta is far smaller than the main and retrieving a cached result takes little time, the aggregate cache can speedup analytical queries by order of magnitudes. As there are only few updates and deletes in enterprise workloads as outlined in Sect. 3, we focus on insert only workloads in this paper.

4.2 Joins

Based on the header/item composition relationship and the use cases for the financials (see schema in Sect. 3) and materials management systems, we focus on inner joins, mostly with an equality operator in the join-predicate (equi-join) [19]. We define an aggregation query joining t tables between relations R with join conditions C as

$$Q_{Agg}(t) = R_1 \bowtie_{C_1} R_2 \ldots \bowtie_{C_t} R_{t+1}$$

Each table consists of two partitions $\mathbb{P} = \{main, delta\}$. For a query involving a join of two tables, the database engine internally has to process more than just one join in order to retrieve a complete result set. The mains of both tables need to be joined, both deltas and both main-delta combinations of the two tables.

In the following subsections we show how to handle joins with different aggregate caching strategies. All variants are compared based on an example query involving a header, item and dimension table as depicted in Fig. 3. Each number represents a subjoin that needs to be unified with the `UNION ALL` SQL operator. The analytical queries of the financials application always included a join between the large header and item tables, and a varying number of smaller configuration and text tables.

4.3 Join Without Cache

Without caching, the database engine needs to run the join on all possible main-delta combinations $\mathbb{J}_{noCache}$ of all involved tables to build a complete result set:

$$\mathbb{J}_{noCache}(t) = \mathbb{P}^t$$

To evaluate Q_{Agg} joining t tables, that adds up to a total of 2^t subjoins to be unified:

$$ResultSet\,(Q_{Agg}) = \bigcup_{(p_1,p_2,\ldots,p_t)\in\mathbb{J}_{noCache}} \left(R_{1_{p_1}} \bowtie_{C_1} R_{2_{p_2}} \ldots \bowtie_{C_t} R_{t+1_{p_{t+1}}}\right)$$

As depicted in Fig. 3, for a join query involving three tables, this would mean unifying the result sets of eight sub joins.

Based on the size of the involved table components, the time to execute the subjoins varies. 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.

4.4 Caching Join

When using the aggregate cache, the result set from joining all main partitions is already calculated and the total number of subjoins is reduced to $2^t - 1$:

$$\mathbb{J}_{withCache}(t) = \mathbb{J}_{noCache}(t) \setminus \{main\}^t$$

For our example from Fig. 3, the subjoin #8 does not need to be rerun based on the cached result set. Since the database does not know anything about the semantics of the involved tables and therefore their usage characteristics, it has to assume there could potentially be newly inserted tuples in the delta of the dimension table, that create a new match for the join of the header-main and item-main. Based on their size, that subjoin requires a lot of time though. The header-main/item-main join needs to be run even more often, there more dimension, text or configuration tables are involved. Depending on the overhead induced by the caching mechanism (incremental update during merge, check of cache admission policy, ...), the regular caching join may not improve performance for analytical queries with three or more tables.

In case we have a cached query involving only a header and item table, only the header-delta/item-delta, header-main/item-delta and header-delta/item-main subjoins need to be computed. Since deltas get merged before they get to large, those subjoins take little time. Therefore the caching join delivers a speedup for analytical queries limited to joins involving only two tables.

4.5 Semantic Join

In this subsection we show which table components need to be joined, when the database is aware of the enterprise application characteristics introduced in Sect. 3. In Sect. 5 we explain how the database can become aware of those characteristics.

Let us assume a query joining a header and item table with a present cached result set representing the joined mains. As static business objects are inserted in the context of a single transaction, the header tuple and the corresponding

item tuples are inserted together. If there was no merge yet, both tuples that will match the join condition are both in the delta part of their table. Therefore we only need to run the header-delta/item-delta join and unify the results with the cached entry. The main-delta combinations of header and item table can be avoided. Same holds true for the subjoins #2, #4, #6 and #7 of our example from Fig. 3, since the header and item tuples that belong together are either all in the mains *or* deltas.

If there has not been an insert, update or delete on the dimension table in a long time, the delta of that table is empty. For inner joins, empty table components do not need to be included since they will not contribute to the result set. Therefore the subjoins #2 and #3 can be avoided. This elimination method could also be applied if there would be a greater number of involved dimension, text or configuration tables with empty deltas.

This only leaves the subjoin #1, between the header-delta, item-delta, and the main of the small dimension table. Using the semantic chaching strategy, an aggregation query

$$Q_{Agg_{HID}} = H \bowtie_{C_1} I \bowtie_{C_2} D$$

between a header H, item I and dimension table D is reduced to process the single subjoin

$$Changes(Q_{Agg_{HID}}) = H_{delta} \bowtie_{C_1} I_{delta} \bowtie_{C_2} D_{main}$$

compared to

$$ResultSet(Q_{Agg_{HID}}) = \bigcup_{(p_1,p_2,p_3)\in\mathbb{P}^3} (H_{p_1} \bowtie_{C_1} I_{p_2} \bowtie_{C_2} D_{p_3})$$

without an aggregate caching mechanism. Since all involved table components are small, the subjoin can be executed with little effort.

The concept of the semantic join can also be applied to extendable business objects such as sales orders, with item tuples possibly being added to an existing header tuple at a later point in time. In that case we additionally have to include the subjoin matching header-main, item-delta, and the mains of the smaller static tables.

For static business objects, the semantic join always only executes one subjoin using the header-delta, item-delta and dimension-, text- and configuration-table-mains. Next to the schema usage characteristics it requires a different method of handling the merge process as outlined in the following subsection.

4.6 Merge

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. When employing a semantic join between a header and an item table, there are two ways to merge. One way is to synchronize the merge of both tables. This way the tuples that match the join condition will always be all in

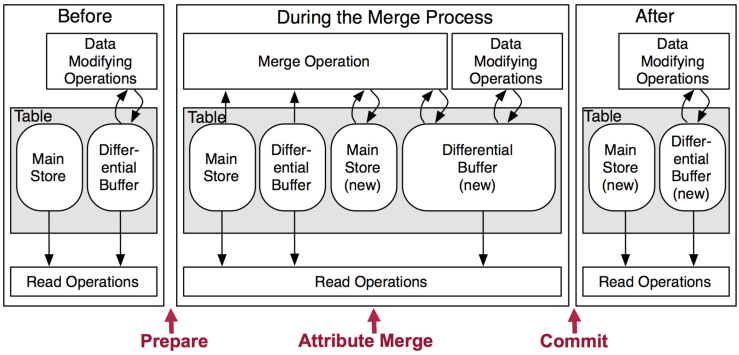

Fig. 4. Stages of the delta merge process.

the delta *or* main storage. Figure 4 shows the three phases of a merge process [11]. Specifically, the prepare steps that switch inserts to run into new blank delta storages need to happen in between the same two SQL queries. Depending on the database architecture, this may be a challenging implementation task without the introduction of a lock that cues transactional queries.

On the other hand, the header and item table could be merged independently by maintaining the aggregate cache at the same time. For static business objects the incremental update would need to process all subjoins that include the delta that is being merged. If we merge the header-delta of our example from Fig. 3, that would be #1. When rerunning a query after the header table has been merged, the item tuples of the merged header would not find a join partner using the inner join. But they also should not find a partner since they are already considered in the cached result set. For extendable business objects, the incremental update would only be done for merging the item table, since the semantic join also processes the item-delta/header-main subjoin for every analytical query.

5 Annotating Enterprise Application Characteristics

In this section we list schema usage characteristics the database requires to process the semantic join from Sect. 4.5. For each information aspect we introduce a number of ways of explicit and implicit character, how the database could obtain that information. In this Section we limit our annotation examples to static business objects, even though similar methods can be used to extendable business objects.

5.1 Empty Delta

To avoid the subjoins with the deltas of dimension, text, and configuration tables, the database engine needs to know that they are empty. That simple check should be a trivial implementation for most database architectures.

5.2 Associations

The caching engine needs to know which table attributes are used as join condition to the key of other tables. There are three methods with different strengths and weaknesses.

First, *foreign keys* could be defined on database level during the design time of the schema [19]. They are a well established mean in many database systems. A column is marked to match the key of another table. New inserts, updates and deletes are checked for data integrity of defined foreign keys. The checking mechanism may decrease transactional throughput performance.

Another way would be to use a *domain specific language (DSL) to model objects on database level*. The database would create the `CREATE TABLE` statements from the abstract data modeling language. The DSL supports syntax to explicitly express associations between objects. Listing 1.1 shows an example syntax similar to the CDS-DDL[3] from SAP HANA [1]. An AccountingItem can, but does not have to be associated with a Product.

```
entity AccountingItem{
    Product:  association[0..1] of Product;
    Quantity: int}
entity Product{
    Name:     string}
```

Listing 1.1. DSL example to model objects and associations on DB level.

A third way would be to look at meta data repositories of present systems. Some enterprise application landscapes keep schema information in a central place. One example of those repositories is the SAP Data Dictionary. Each table column has a specific domain. Such a domain can be defined by a data type, a value range or the column of another table. The latter case indicates an association used as join condition.

5.3 Single Transaction Inserts

As explained in Sect. 3, static business objects are inserted in the context of a single transaction. There are two fundamentally different ways to communicate the insert behavior to the database. The schema could be annotated at design time or the database access could be restricted to insert entire business objects.

Design Time Annotation. During data modeling phase the designer defines how the schema will be used. This might be a challenging programming paradigm for environments where a large number of developers are involved. Application programmers might not know about the restrictions implied by the data modelers and be surprised about the errors returned by the database. The annotation could be done in two ways.

[3] Core Data Services - Data Definition Language, a DSL to model objects on SAP HANA.

The *DSL for data modeling*, as introduced in Sect. 5.2 could support syntax to explicitly model a composition relationship. That relation is stronger than an association, meaning that entity *foo* consist of some entities *bar*. The composition relationship implies that they are inserted in the context of single transaction. In Listing 1.2 the AccountingHeader is composed of a number of AccountingItems.

```
entity AccountingHeader{
    FiscalYear: int;
    Items:      composition[1..*] of AccountingItem}
entity AccountingItem{
    Product:    association[0..1] of Product;
    Quantity:   int}
```

Listing 1.2. DSL example to model composition relationship.

Another way would be to slightly extend the in many databases already present concept of *SQL constraints*. Typically they are defined on a schema level, checked on SQL statement level and sometimes with a leaner execution time on transaction level. The available constraint enforcement levels need to be extended with a new *transaction* level, that explicitly checks for constraint consistency within a transaction. It checks if a transaction inserting new tuples is valid by itself. The defined foreign keys are validated among the tuples that are inserted together.

High Level APIs for Data Manipulation. Inserts into databases are typically done by using SQL commands. The database could restrict data manipulation to higher level APIs. Those commands could e.g. look like `StoreAccountingObject()`, `RegisterMaterialMovement()` or `ReleaseSalesOrder()`. In that case all information of header and item tuples would be inserted with a single command. By restricting data manipulation to such higher level APIs, inconsistent data states could also be prevented, that could otherwise be caused by improper usage of SQL commands. The application developer would access those ORM[4] like methods directly on the database.

In the context of currently available database technology, one implementation strategy would be to do all data manipulation with *Stored Procedures* (SPs). The database would offer SPs to persist entire objects. The procedure describes in detail how the object attributes are transformed into tuples for different tables. A SP for e.g. an invoice may store a AccountingHeader tuple and multiple AccountingItem tuples in the corresponding tables.

Another way would be to use a *DSL for business object persistence and manipulation on database level*. That DSL would also only offer high-level commands as previously mentioned. One example would be the CDS-DML[5] currently in development for SAP HANA.

[4] Object Relational Mapper, a framework to easy access to relational databases from object oriented programming languages.

[5] Core Data Services - Data Manipulation Language.

6 Benchmarks

In this section we evaluate the potential speedup of the semantic join (see Sect. 4.5) compared to the join not using schema usage characteristics, the caching mechanism used with a fully denormalized schema and using no caching mechanism at all.

For the evaluation we use a real customer data set of an SAP financials application of an international-operating company producing consumer goods. The schema – limited to the benchmark relevant tables and columns – looks similar to the one illustrated in Fig. 1. The data set consists of 35 million AccountingHeader tuples, 310 million AccountingItem tuples and the text tables have each less than 2000 entries.

We modeled a mixed OLTP/OLAP workload, based on input from interviews with that customer. The analytical queries simulate multiple users, using a profit and loss statement (P&L) analysis tool. The SQL statements calculate the profitability for different dimensions like product category and subcategory (as mentioned in Sect. 3) by aggregating debit and credit entries. Listing 1.3 shows a simplified sample query that calculates how much profit the company made with each of its product categories. We simulate a drill down into the (P&L) by applying a specific dimension value as filter and then grouping by another dimension.

```
SELECT pc.Name AS Category, SUM(i.Price) AS Profit
FROM   AccountingHeader AS h,
       AccountingItem AS i,
       ProductCategory AS pc
WHERE i.AccountingHeaderID = h.AccountingHeaderID
      AND i.CategoryID = pc.CategoryID
      AND pc.Language = 'ENG'
GROUP BY i.CategoryID;
```

Listing 1.3. Simplified benchmark sample query.

All benchmarks are run on a server with 64 Intel Xeon QPI[6] enabled processor cores and 1 TB of RAM running SansoucciDB [5], an in-memory column-oriented research database.

6.1 Delta Size

The speed up of the aggregate caching mechanism greatly depends on the number of records in the delta storage. The smaller the delta in respect to the main storage, the less tuples need to be aggregated when rerunning cached queries. How large the peak delta size is just before merging, depends on the insert rate and how long it takes to merge the table.

Figure 5 shows the speedup factor of the different caching strategies outlined in Sect. 4.2 compared to the caching mechanism running on a single, denormalized table. For the denormalized caching, the speedup is calculated by comparing

[6] Quick Patch Interconnect, a direct communication system for processor cores that replaces the Front Side Bus (FSB).

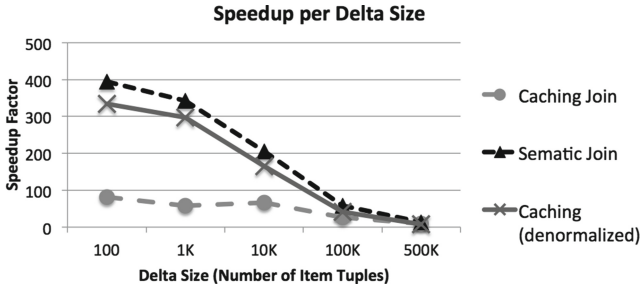

Fig. 5. Aggregation with header-item join benchmark.

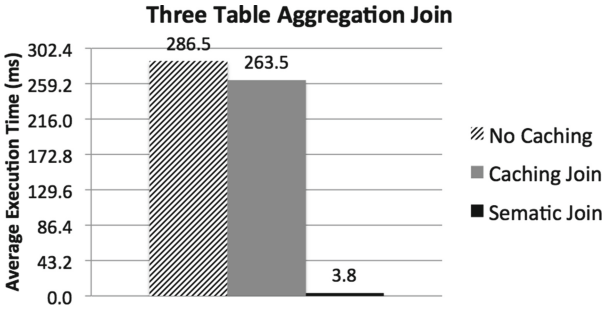

Fig. 6. Benchmark for aggregation queries joining header, item, and one dimension table.

it to the runtime on the denormalized table without caching. For this specific benchmark we only use a two table join between the header and item table. In that case the strategy not leveraging enterprise application characteristics also performs better by magnitudes since it never has to do the header-main/item-main subjoin.

The semantic join enables a speedup of greater than 200 for item deltas smaller than 10 thousand tuples and greater than 50 with less than 100 thousand tuples. Even for larger deltas with half a million entries, cached queries are calculated thirteen times faster than without caching (0.12 compared to 1.58 s).

6.2 Three Tables

For an aggregation query joining three tables as illustrated in Fig. 3, the caching mechanism has to join the large header-main and item-main (see Sect. 4.2). In this benchmark we use deltas with 50,000 items and their corresponding header tuples. The dimension table consists of 150 entries. Figure 6 shows the importance of utilizing schema usage characteristics once there are three or more tables involved. The analytical queries of the analyzed customer typically involve three to seven tables. Since the semantic caching strategy only joins rather small table

components, its execution time remains faster by an order of magnitudes, even if more tables are involved.

7 Conclusions and Future Work

With growing requirements on data analysis, the aggregate cache enables IMDBs to handle an even higher aggregation query throughput in enterprise system environments with mixed workloads. However, with queries joining two or more tables, the benefit of the aggregate cache is reduced as the needed incremental view maintenance is very expensive.

Our analysis of enterprise applications revealed several patterns with respect to schema design and resulting workloads. Most importantly among them, it is a very common practice to split business objects into a header and item table and schemas having many small rather static tables. These patterns can be leveraged to reduce the incremental view maintenance and run more efficient aggregate caching strategies. Especially for small delta sizes, they enable a speed-up by order of magnitudes.

With having a clear understanding of the speedup potential of the caching mechanism for aggregation queries joining tables, one direction of future work is to predict the runtime improvements for a columnar IMDB with a main-delta architecture. Based on cardinalities of main and delta partitions, unique value count, filter selectivity, and possibly other metrics, a cost model of the cache admission policy should decide what aggregate queries with joins are most valuable to be cached.

References

1. Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: SAP HANA database: data management for modern business applications. In: SIGMOD (2011)
2. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: Hyrise: a main memory hybrid storage engine. In: VLDB, pp. 105–116 (2010)
3. Kemper, A., Neumann, T., Informatik, F.F., München, T.U.: Hyper: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, D-Garching (2011)
4. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD, pp. 1–2 (2009)
5. Plattner, H.: SanssouciDB: an in-memory database for processing enterprise workloads. In: BTW (2011)
6. Smith, J.M., Smith, D.C.P.: Database abstractions: aggregation. ACM Commun. **20**, 405–413 (1977)
7. Srivastava, D., Dar, S., Jagadish, H., Levy, A.: Answering queries with aggregation using views. In: VLDB (1996)
8. Gupta, A., Mumick, I.S.: Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng. Bull. **18**, 3–18 (1995)
9. Müller, S., Butzmann, L., Höwelmeyer, K., Klauck, S., Plattner, H.: Efficient view maintenance for enterprise applications in columnar in-memory databases. In: EDOC (2013)

10. Müller, S., Plattner, H.: Aggregates caching in columnar in-memory databases. In: 1st International Workshop on In-Memory Data Management and Analytics (IMDM), in conjunction with VLDB (2013)
11. Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., Zeier, A.: Fast updates on read-optimized databases using multi-core CPUs. In: VLDB (2012)
12. Brewer, E.A.: Towards robust distributed systems. In: PODC (2000)
13. Buneman, O.P., Clemons, E.K.: Efficiently monitoring relational databases. ACM Trans. Database Syst. **4**, 368–382 (1979)
14. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, pp. 61–71 (1986)
15. Bello, R.G., Dias, K., Downing, A., Feenan, Jr., J.J., Finnerty, J.L., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in oracle. In: VLDB, pp. 659–664 (1998)
16. Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy maintenance of materialized views. In: VLDB, pp. 231–242 (2007)
17. Gupta, H., Mumick, I.S.: Incremental maintenance of aggregate and outerjoin expressions. Inf. Syst. **31**(6), 435–464 (2006)
18. Larson, P.A., Zhou, J.: Efficient maintenance of materialized outer-join views. In: 2007 IEEE 23rd International Conference on Data Engineering, pp. 56–65. IEEE (2007)
19. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2nd edn. Prentice Hall Press, Upper Saddle River (2008)