Aggregates Caching in Columnar In-Memory Databases

Stephan Müller and Hasso Plattner

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

Abstract. The mixed database workloads found in enterprise applications are comprised of short-running transactional as well as analytical queries with resource-intensive data aggregations. In this context, caching the query results of long-running queries is desirable as it increases the overall performance. However, traditional caching approaches are inefficient in a way that changes in the base data result in invalidation or recalculation of cached results.
Columnar in-memory databases with a main-delta architecture are well-suited for a novel caching mechanism for aggregate queries that is the main contribution of this paper. With the separation into read-optimized main storage and write-optimized delta storage, we do not invalidate cached query results when new data is inserted to the delta storage. Instead, we use the cached query result and combine it with the newly added records in the delta storage. We evaluate this caching mechanism with mixed database workloads and show how it compares to existing work in this area.

1 Introduction

The classic distinction between online transactional processing (OLTP) and online analytical processing (OLAP) is no longer applicable in the context of modern enterprise applications [1],[2]. Instead of associating transactional or analytical queries with separate applications, a single modern enterprise application executes both – transactional *and* analytical – queries. Within the available-to-promise (ATP) application, for example, the OLTP-style queries represent product stock movements whereas the OLAP-style queries aggregate over the product movements to determine the earliest possible delivery date for requested goods by a customer [3]. Similarly, in financial accounting, every financial accounting document is created with OLTP-style queries, while a profit and loss statement needs to aggregate over all relevant documents with OLAP-style queries that are potentially very expensive [1].

To speed-up the execution of long-running queries, techniques such as *query caching* and the introduction of *materialized views* have been proposed [4]. However, the inherent problem with query caching and materialized views is that whenever the base data is modified, these changes have to be propagated to ensure consistency. While a database query cache can simply flush or invalidate the cache, a process known as *materialized view maintenance*, is well established in academia [5],[4],[6] and industry [7],[8] but with focus on traditional database architectures and data warehousing [9],[10],[11]. For purely analytical applications, a maintenance downtime may be acceptable, but this is not the case in a mixed workload environment as transactional throughput must always be guaranteed. Also, the recent trend towards columnar in-memory databases (IMDBs) that are able to handle transactional as well as analytical workloads on a single system [12],[13],[14] has not been considered.

A columnar IMDB for transactional and analytical workloads has some unique features and preferred modes of operating [1],[2]. To organize the attributes of a table in columns and to encode the attribute values via a dictionary into integers, known as dictionary encoding [15], has many advantages such as high data compression rates and fast attribute scans. But this organization comes at a certain price. In transactional workloads we have to cope with high insert rates. A permanent reorganization of the attribute vectors (columns) would not allow for a decent transactional performance, because new values appear and have to be included in the encoding process which complicates the request to keep the attribute dictionaries sorted. A way out of this dilemma is to split the attribute vectors of a table into a read-optimized main storage and a write-optimized delta storage. All new inserts, updates, and deletes are appended to the delta storage with separate unsorted dictionaries. At certain times the attribute vectors are merged with the ones in the main storage and a new dictionary (per attribute) is established [16]. Since the main storage is significantly larger than the delta, the insert performance becomes acceptable and the analytic performance is still outstanding [17].

The fact that we can handle transactional and analytical workloads in one system has tremendous benefits to the users of the system. Not only the freedom of choice what and how to aggregate data on demand but the instant availability of analytical responses on even large operational data sets will change how business will be run. A consequence of this desirable development will be a significant increase in the analytical workload with aggregate queries on the combined system.

To cope with the increase of analytical queries on transactional data, we propose an aggregate query caching mechanism that leverages the main-delta architecture of columnar in-memory databases. Because of the separation into main and delta storage, we do not invalidate cached aggregate queries when new records are inserted to the delta storage. Instead, we use the cached results of the aggregate queries in the main storage and combine them with the newly inserted records in the delta storage.

After discussing related work in Section 2, we outline the algorithm and architecture of our implementation in Section 3. In Section 4 we evaluate our caching mechanism and conclude with an outlook on future work in Section 5.

2 Related Work

The caching of aggregate queries is closely related to the introduction of materialized views to answer queries more efficiently. To be more precise, a cached query result is a relation itself and can be regarded as a materialized view. Gupta gives a good overview of materialized views and related problems in [4]. Especially, the problem of materialized view maintenance has received significant attention in academia [18],[5],[6]. Database vendors have also investigated this problem thoroughly [7],[8] but to the best of our knowledge, there is no work that evaluates materialized view maintenance strategies in columnar in-memory databases with mixed workloads. Instead, most of the existing research is focused on data warehousing environments [9],[10],[11] where maintenance downtimes may be acceptable.

The summary-delta tables concept to efficiently update materialized views with aggregates comes close to our approach as the algorithm to recalculate the materialized view is based on the old view and the newly inserted, updated, or deleted values [19]. However, their approach updates the materialized views during a maintenance downtime in a warehousing environment and does not consider the newly inserted operational data during query processing time which is necessary in a transactional environment. Further, it does not take the main-delta architecture and the resulting merge process into account.

3 Aggregates Caching

In this section, we describe the basic architecture of our aggregate query caching mechanism and the involved algorithms. The cache is implemented in a way that is transparent to the application. Consequently, the caching engine has to ensure data consistency by employing an appropriate maintenance strategy.

While aggregate functions can be categorized into distributive, algebraic and holistic functions [20] we limit our implementation to distributive functions without the `distinct` keyword, such as `sum`, `min`, `max`, or `count` as they are most commonly found in analytical queries [17] and because they are self-maintainable with respect to insertions [19]. Updates and deletes require an extension of our algorithm as explained in Section 3.5. Since algebraic functions can be computed by combining a constant number of distributive functions, e.g., `avg = sum / count`, they can also be supported given the assumption that a cached aggregate query with an `avg` function is rewritten to include both the `sum` and `count` functions.

3.1 Architecture and Algorithm

The basic architecture of our aggregates caching mechanism is illustrated in Figure 1. With the columnar IMDB being divided into main and delta storage, the aggregates caching manager component can distinguish between these and read the delta storage explicitly and combine this result with the cached query result. The relations of cached aggregate queries are each stored in a separate database table. Further, a global cache management table (CMT) stores the meta data for each cached aggregate query including access statistics. Also, it maps the hash of the normalized SQL string to the database table that holds the cached results of the aggregate query.

Every parsed query with supported aggregate functions, is handled through the aggregates caching manager. To check if the query already exists in the

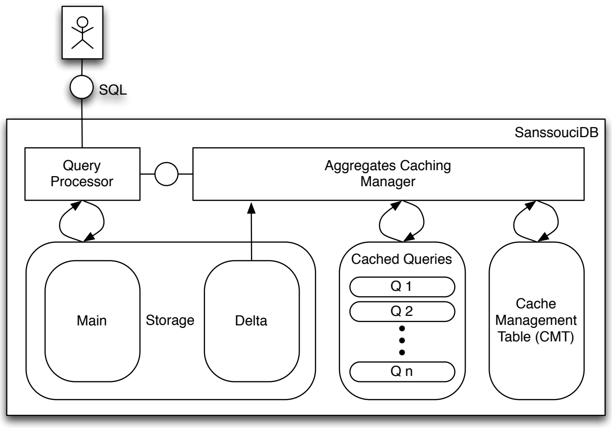


Fig. 1. Aggregates query caching architecture

cache, the hash value of the normalized SQL string is computed and looked up in the CMT. If the aggregates caching manager does not find an existing cache entry for the corresponding SQL query, it conveys the query without any changes to the underlying main and delta storage. After query execution, it is checked whether the query is suitable for being cached depending on the cache admission policy (cf. Section 3.6). If this is the case, the query result from the main storage is cached for further reuse. This is done by creating a separate table that only contains the results of the specific query. The name of the table equals the generated hash value of the SQL string and is referenced by the CMT.

Listing 1.1. A simple aggregate query

```
SELECT month, account, SUM(amount) FROM sales
WHERE year=2013 GROUP BY month, account
```

In case, the query is already cached, the original aggregate query (an example is shown in Listing 1.1) is executed on the delta storage. Listing 1.2 shows how the result from the delta storage is combined with the cached query result as persisted in the table `agg08f15e` (assuming that agg08f15e is the hash of the example query sql string) and returned to the query processor. We use a `UNION ALL` query to not eliminate duplicates, but aggregate them by applying the original aggregate query on the combined results.

Listing 1.2. Combining the cached aggregate query with results from the delta storage

```
SELECT month, account, SUM(amount) FROM
        (SELECT * FROM agg08f15e
        UNION ALL
        SELECT month, account, SUM(amount)
        FROM sales_delta
        WHERE year=2013 GROUP BY quarter, account)
GROUP BY month, account
```

3.2 Aggregates Maintenance Strategies

To ensure consistency, cached aggregates have to be maintained accordingly. The timing of existing materialized view maintenance strategies can be distinguished between *eager* and *lazy*. While eager strategies immediately propagate each change of base tables to the affected materialized views [5], lazy (or deferred) strategies maintain materialized views no later than the time the materialized view is queried [8]. Independently of the timing, one can divide maintenance strategies into *full* and *incremental* ones. Full strategies maintain the aggregate by complete recalculation using its base tables. Incremental strategies store recent modifications of base tables and explicitly use them to maintain the views. Based on the fact that an incremental calculation of aggregates is always more efficient than a full recalculation [6], we focus on incremental strategies, despite the fact that some aggregate functions cannot be maintained incrementally [19].

The proposed aggregate query caching mechanism does neither maintain the cached aggregate at insert time nor at query time. Instead, it is done incrementally during the delta merge process. Since it is possible to predict the query execution time of in-memory databases very accurately [21], we create cost models for each maintenance strategy. The costs are based on a simplified workload model that consists of a number of writes N_w into the base table and a number of reads N_r of the cached aggregate query.

Eager Incremental Update (EIU) Since the cached aggregate query is maintained after each insert, the cost for accessing the aggregate query is just a single read. The maintenance costs are tied to a write into the base table. As it is an incremental strategy, the costs consist of the read time T_{RA} to retrieve the old value and the write time T_W for the new value into the cached aggregate table.

Lazy Incremental Update (LIU) All maintenance is done on the first read accessing the cached aggregate query. The maintenance costs $N_{w_k} \cdot (T_{RA} + T_W)$ and cost to read the requested aggregate T_{RA} are combined into one function. The maintenance costs depend on the number of writes with distinct grouping attribute values per read N_{w_k} which is influenced by the order of the queries in a workload and the distribution of the distinct grouping attributes.

Merge Update (MU) The costs of a read T_{r_k} is the sum of an access to the cached aggregate query T_{RA} and an on-the-fly aggregation on the delta table whereas T_{RD_k} defines the costs for the aggregation for the k^{th} read. The merge update strategy updates its materialized aggregate table during a merge process. Therefore, the tuples in delta storage have to be considered. The merge time T_m for the number of cached aggregates N_A is the sum of a complete read of the cached aggregate query tables $N_A \cdot T_{RA}$, a read of the delta T_{RD_k}, and the write of the new aggregate $(N_A + N_{newWD}) \cdot T_W$. Equation 1 shows the calculation of the total execution time based on the time for reads and the merge.

$$T_{total} = N_m \cdot T_m + N_r \cdot T_{r_k} \tag{1}$$

3.3 Optimal Merge Interval

The costs of our aggregates caching mechanism and the MU maintenance strategy mainly depend on the aggregation performance on the delta storage which decreases linearly with an increasing number of records [22]. However, the merge operation also generates costs that have to be considered. In the following, we propose a cost model which takes the costs for the merge operation and the costs for the aggregation on the delta storage into account. Similarly to the cost model for the merge operation introduced by Krüger et al. [16], our model is based on the number of accessed records to determine the optimal merge interval for one base table of a materialized view.

Equation 2 calculates the number of records $Costs_{total}$ that are accessed during the execution of a given workload. A workload consists of a number of reads N_r and a number of writes N_w. The number of merge operations is represented by N_m. The first summand represents the accesses that occur during the merge operations. Firstly, each merge operation has to access all records of the initial main storage $|C_M|$. Secondly, previously merged records and new delta entries are accessed as well [16]. This number depends on the number of writes N_w in the given workload divided by two (since the number of records in the delta increases linearly). The second summand determines the number of accesses for all reads N_r on the delta. As before, the delta grows linearly and is speed-up by the number of merge operations N_m.

$$Costs_{total} = N_m \cdot (|C_M| + \frac{N_w}{2}) + N_r \cdot \frac{\frac{N_w}{2}}{N_m + 1} \tag{2}$$

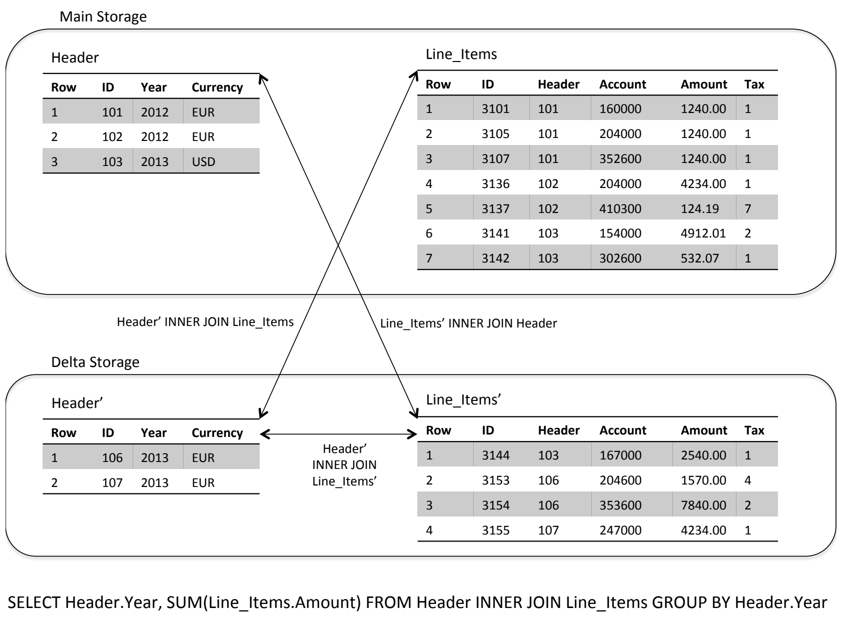
$$Costs'_{total} = |C_M| + \frac{N_w}{2} - \frac{N_r \cdot N_w}{2 \cdot N_m^2 + 4 \cdot N_m + 2} \tag{3}$$

$$N_m = \frac{\sqrt{2 \cdot |C_M| \cdot N_w \cdot N_r + N_w^2 \cdot N_r} - 2 \cdot |C_M| - N_w}{2 \cdot |C_M| + N_w} \tag{4}$$

The minimum is calculated by creating the derivation (Equation 3) of our cost model and by obtaining is root (Equation 4). N_m represents the number of merge operations. Dividing the total number of queries by N_m returns the optimal merge interval.

3.4 Join Operations

When processing aggregate queries with join operations, the complexity of the caching mechanism and the involved MU maintenance strategy increases. Instead of combining the cached result with the query result on the delta storage, the join of every permutation has to be computed before these results can be combined. In Figure 2, we have illustrated the involved tables in the main and delta partition of a simple aggregate query including a join of two tables. While the cached query result is based on a join of the `header` and `line_items` table in the main partition, we have to compute the joins of `header'` and `line_items'` tables in the delta partition, and additionally the joins between `header'` and `line_items` as well as `line_items'` and `header`. When the cached aggregate query consists


Main Storage
Header
Line_Items
Header' INNER JOIN Line_Items
Line_Items' INNER JOIN Header
Delta Storage
Header'
Line_Items'
Header' INNER JOIN Line_Items'

Header

Row	ID	Year	Currency
1	101	2012	EUR
2	102	2012	EUR
3	103	2013	USD

Line_Items

Row	ID	Header	Account	Amount	Tax
1	3101	101	160000	1240.00	1
2	3105	101	204000	1240.00	1
3	3107	101	352600	1240.00	1
4	3136	102	204000	4234.00	1
5	3137	102	410300	124.19	7
6	3141	103	154000	4912.01	2
7	3142	103	302600	532.07	1

Header'

Row	ID	Year	Currency
1	106	2013	EUR
2	107	2013	EUR

Line_Items'

Row	ID	Header	Account	Amount	Tax
1	3144	103	167000	2540.00	1
2	3153	106	204600	1570.00	4
3	3154	106	353600	7840.00	2
4	3155	107	247000	4234.00	1

```
SELECT Header.Year, SUM(Line_Items.Amount) FROM Header INNER JOIN Line_Items GROUP BY Header.Year
```

Fig. 2. Aggregate queries with join operations

of three or more joined tables, the necessary join operations between delta and main storage increase exponentially. The number of necessary joins based on the number of tables t in the aggregate query can be derived as $JoinOps = t^2 - 1$.

After analyzing enterprise workloads, we found out that aggregates for accounting, sales, purchasing, stocks etc. always need a join of the transaction header and the corresponding line items. Interestingly, new business objects such as sales orders or accounting documents are always inserted as a whole, therefore the new header and the new line items are persisted in the delta storage. Using these semantics of the business objects can reduce the number of necessary join operations from three to just one (a join of header and line items in the delta). In case a business object can be extended after the initial insert, the header entry could already be merged into the main storage. Consequently, we would need an additional join of the `line_items'` table in the delta with the `header` table in the main.

3.5 Updates and Deletes

The presented algorithm is valid for an insert-only approach which handles logical updates or deletes by inserting differential values to the delta storage. When updating a tuple, and only inserting the new, updated value to the delta storage, the algorithm needs to be extended. We have identified two possible solutions: We can either retrieve the old value from main storage, calculate the differential value and insert this value in the delta storage and flag it accordingly, so that the merge process does not consider this tuple. Or, to avoid an adaption of the merge process, we could also maintain a separate data structure that holds the differential values for all updates or deletes and include these values in the delta aggregate query.

`Min` and `max` functions are not self-maintainable and therefore, for every update or delete, we have to perform additional checks. For deletes, we have to check if the deleted tuple is a `min` or `max` tuple. For updates, we have to check if the updated value is higher than a cached `max` aggregate or lower than a cached `min` aggregate. If that is the case, the cached `min` or `max` aggregate has to be invalidated and recalculated from the main and delta storage.

Despite the inherent overhead, we believe that this process is viable, because the percentage of updates and deletes is very low in enterprise applications [16].

3.6 Cache Management Strategies

In order to limit the needed memory space and reduce the inherent computational overhead of the caching algorithm, we only want to admit the most profitable aggregate queries to the cache. The query cache management takes place at query execution time for cache admission and replacement, and during the merge process to determine which aggregate queries to incrementally maintain or to evict from the cache.

We have identified two approaches to determine whether to cache an aggregate query after it has been executed: The first way is to measure the execution time of the aggregate query and only cache queries that are above a system-defined threshold. Another way is to calculate the profit of using a cached query over an on-the-fly aggregation. The definition of the profit for query Q_i can be described with the execution time for the aggregation on the main storage $AggMain_i$ and delta storage $AggDelta_i$ divided by the time to access a cached aggregate query $AggCached_i$ and the execution time of the aggregation on the delta storage $AggDelta_i$.

$$profit(Q_i) = \frac{AggMain_i + AggDelta_i}{AggCached_i + AggDelta_i} \tag{5}$$

This profit metric will change when the delta storage grows, but it is a good initial indicator to decide which queries to admit to the cache. When the cache size reaches a system-defined size limit, we can replace queries with lower profits or execution times by incomings queries with higher profits or execution times.

During the merge process, it has to be decided which cached aggregate query to incrementally update or evict from the cache. For this process, we can use another metric that includes the average frequency of execution λ_i of query Q_i which is calculated based on the K_ith last reference and the difference between the current time t and the time of the last reference t_K:

$$\lambda_i = \frac{K_i}{t - t_K} \tag{6}$$

The profit of a query Q_i can then be extended as follows:

$$profit(Q_i) = \frac{\lambda_i \cdot (AggMain_i + AggDelta_i)}{AggCached_i + AggDelta_i} \tag{7}$$

4 Evaluation

We implemented the concepts of the presented aggregates caching mechanism in SanssouciDB [17] but believe that an implementation in other columnar IMDBs with a main-delta architecture such as SAP HANA [12] or Hyrise [14] will lead to similar results. Instead of relying on a mixed workload benchmark such as the CH-benchmark [23], we chose an enterprise application that generates a mixed workload to the database with real customer data. The identified financial accounting application covers OLTP-style inserts for the creation of accounting documents as well as OLAP-style queries to generate reports such as a profit and loss statement. The inserts were generated based on the original customer data set covering 330 million records in a denormalized single table. We then extracted 1,000 OLAP-style aggregate queries from the application and validated these with domain experts. The query pattern of the aggregate queries contain at least one aggregate function with optional group by clauses and predicates. Further, nested subqueries are supported. Mingling both query types according to the creation times (inserts) and typical execution times (aggregate queries) yielded a mixed workload which our evaluations are based upon.

4.1 Aggregates Caching

The strength of a caching mechanism is to answer reoccurring queries. To compare our approach to a standard query cache that gets invalidated whenever the base data changes, we have created a benchmark based on a mixed workload of 10,000 queries with 90% analytical queries and 10% transactional insert queries. The 9,000 analytical aggregate queries were randomly generated from the 1,000 distinct queries. The average execution time on a 40 core server with 4 Intel Xeon E" 7 4870 CPU each having 10 physical cores and 1 TB of main memory when using no cache was 591ms which dropped down to 414ms with a standard query cache. The average execution time of the aggregates cache was at 74ms, outperforming the standard query cache by nearly a factor of six.

With an increasing number of distinct aggregate queries, the performance of the proposed aggregates caching mechanisms decreases linearly. With a workload of 100% distinct aggregate queries, where no cache reuse takes place, we measured the overhead of the aggregates caching mechanism. When caching every incoming aggregate query, this overhead was at 7% compared to not using the cache, mainly due to the execution time of creating the table that holds the results of the cached aggregate query.

4.2 Aggregates Maintenance Strategies under Varying Workloads

To compare the aggregates caching mechanisms and the involved maintenance strategy to the strategies described in Section 3.2, we have changed the benchmark to varying read/write ratios and a workload of 1,000 queries. A read represents an analytical query with an aggregation and a write represents an insert to the base table which contains one million records. The results as depicted in Figure 3 reveal that when using no materialization (NoMat), the time to execute the workload decreases with an increasing ratio of inserts because an

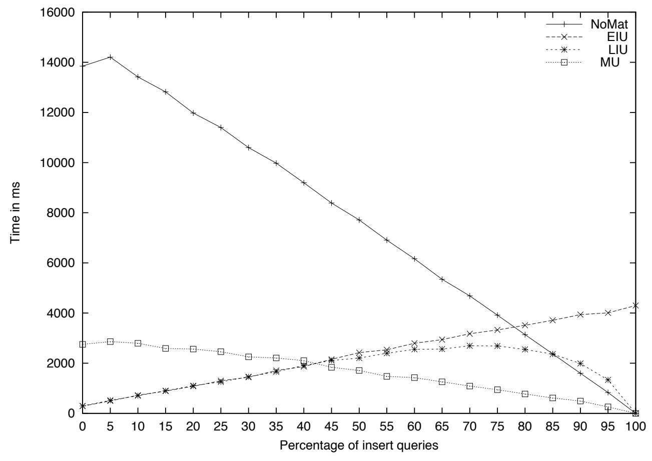


Fig. 3. Measuring the total time of a workload with a varying ratio of inserts.

on-the-fly aggregation is more expensive than inserting new values. The EIU and LIU strategies use materialized aggregates to answer selects and perform much better with high select ratios than no materialization. EIU and LIU have almost the same execution time for read-intensive (less than 50% inserts) workloads. Reads do not change the base table and the materialized aggregates stay consistent. Hence, maintenance costs do not dominate the execution time of the workload and the mentioned strategies perform similarly. With an increasing number of inserts, the performance of EIU decreases nearly linearly while LIU can condense multiple inserts within a single maintenance step. The MU maintenance strategy, which the proposed aggregates query caching mechanism is based on, outperforms all other strategies when the workload has more than 40% insert queries. The low performance for read-intensive workloads is based on the fact, that both, the cached aggregate and the delta storage have to be queried and even an empty or small delta implies a small overhead with the current implementation.

4.3 Merge Interval

To validate the cost model for the optimal merge interval, introduced in Section 3.3, we have created a benchmark and compared it to our cost model. The benchmark executed a workload of 200,000 queries with 20% selects and a varying base table size of 10M, 20M, and 30M records. We have used different merge intervals with a step size of 3,000 queries starting with 1,000 and compared the best performing merge interval to the one predicted by our cost model. The results reveal that the values predicted by our cost model have a mean absolute error of 10.6% with the remark that our approximation is limited by the chosen step size.

4.4 Object-Aware Join Operations

To evaluate the overhead of joining two tables when using the aggregates caching mechanism, we have split the single, denormalized table into a table that contains 30M header records and a table with 311M item records. The workload as presented in Section 4.1 was adjusted accordingly so that the queries contain a join operation of the header and items table. With the aggregates caching mechanism, the time needed to join these two tables, divided in main and delta partitions, increases with a growing number of records in the delta storage, as shown in Table 1. Leveraging the business semantics of the chosen financial application which states that header and belonging item records are always inserted together, we can employ the object-aware join which reduces the number of necessary joins from three to one (cf. Section 3.4). This reduces the execution times significantly by a factor of up to 15.

Table 1. Aggregate cache execution times with join queries

Records in delta	Execution times in ms			Speedup factor
	No join	Join	Object-aware join	
0	2.69	4.01	2.95	1.36
1,000	3.24	61.87	4.39	14.10
10,000	5.32	112.89	7.81	14.46
25,000	8.79	247.29	15.65	15.80
50,000	14.58	362.40	23.85	15.20

5 Conclusions

In this paper, we have proposed a novel aggregate query caching strategy that utilizes the main-delta architecture of a columnar IMDB for efficient materialized view maintenance. Instead of invalidating or recalculating the cached query when the base data changes, we combine the cached result of the main storage with newly added records that are persisted in the delta storage. We have compared and evaluated the involved materialized view maintenance strategy to existing ones under varying workloads. Also, we have created a cost model to determine the optimal merge frequency of records in the delta storage with the main storage. To optimize the caching mechanism, we have discussed cache admission and replacement strategies, and an object-aware join mechanism. Further, we have outlined how physical updates and deletes can be handled efficiently. For evaluation, we have modeled a mixed database workload based on real customer data and the financial accounting application, revealing that our aggregates cache outperforms a simple query cache by a factor of six.

One direction of future work is the investigation of transactional properties when handling updates and deletes. Also, we plan to examine ways to persist the business semantics for object-aware join operations and to evaluate additional enterprise applications.

Acknowledgements The authors would like to thank the SAP HANA team for the cooperation including many fruitful discussions.

References

1. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD. (2009)
2. Plattner, H.: Sanssoucidb: An in-memory database for processing enterprise workloads. In: BTW. (2011)
3. Tinnefeld, C., Müller, S., Kaltegärtner, H., Hillig, S., Butzmann, L., Eickhoff, D., Klauck, S., Taschik, D., Wagner, B., Xylander, O., Zeier, A., Plattner, H., Tosun, C.: Available-to-promise on an in-memory column store. In: BTW. (2011) 667–686
4. Gupta, A., Mumick, I.S., et al.: Maintenance of materialized views: Problems, techniques, and applications. Data Engineering Bulletin (1995)
5. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD. (1986)
6. Agrawal, D., El Abbadi, A., Singh, A., Yurek, T.: Efficient view maintenance at data warehouses. In: SIGMOD. (1997)
7. Bello, R.G., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in oracle. In: VLDB. (1998)
8. Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy maintenance of materialized views. In: VLDB. (2007)
9. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View maintenance in a warehousing environment. In: SIGMOD. (1995)
10. Agrawal, D., El Abbadi, A., Singh, A., Yurek, T.: Efficient view maintenance at data warehouses. In: SIGMOD. (1997)
11. Jain, H., Gosain, A.: A comprehensive study of view maintenance approaches in data warehousing evolution. SIGSOFT (2012)
12. 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)
13. Kemper, A., Neumann, T.: Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In: ICDE. (2011)
14. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: Hyrise: a main memory hybrid storage engine. In: VLDB. (2010)
15. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD. (2006)
16. 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)
17. Plattner, H., Zeier, A.: In-memory data management: an inflection point for enterprise applications. Springer Verlag (2011)
18. Buneman, O.P., Clemons, E.K.: Efficiently monitoring relational databases. ACM Transactions on Database Systems (1979)
19. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. In: SIGMOD. (1997)
20. Gray, J., Bosworth: Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In: ICDE. (1996)
21. Schaffner, J., Eckart, B., Jacobs, D., Schwarz, C., Plattner, H., Zeier, A.: Predicting in-memory database performance for automating cluster management tasks. In: ICDE. (2011)
22. Manegold, S., Boncz, P., Kersten, M.: Generic database cost models for hierarchical memory systems. In: VLDB. (2002)
23. Cole, R., Funke, F., Giakoumakis, L., Guy, W., Kemper, A., Krompass, S., Kuno, H., Nambiar, R., Neumann, T., Poess, M., Sattler, K.U., Seibold, M., Simon, E., Waas, F.: The mixed workload CH-benCHmark. In: DBTest. (2011)