Efficient Aggregate Cache Revalidation in an In-Memory Column Store

Stephan Müller, Lars Butzmann, Hasso Plattner
Hasso Plattner Institute
University of Potsdam, Germany
Email: {stephan.mueller, lars.butzmann, hasso.plattner}@hpi.uni-potsdam.de

***Abstract*—Modern enterprise applications do not separate between online transactional processing and online analytical processing anymore. To ensure fast response times for expensive analytical queries, we implemented an aggregate cache in an columnar in-memory database. The separation into read-optimized main and write-optimized delta storage is exploited to cache only the aggregates based on the main storage and aggregate all records in the delta storage on-the-fly. This works with insert-only workloads, but not with deletes and updates that invalidate records in the main storage and consequently invalidate a cached aggregate. In this paper, we introduce an approach to revalidate a cached aggregate using efficient bit vector operations. A revalidation is superior to an invalidation since the old cached aggregate is reused. A further contribution is an evaluation of the influence factors that determine whether to invalidate or revalidate a cached aggregate. Our implementation shows that an aggregate cache revalidation outperforms an invalidation when less than 50% of the relevant records are invalidated.**

Keywords-Aggregation, Materialized View Maintenance, In-Memory Data Base, Bit Vector

I. INTRODUCTION

Several decades ago, database vendors decided to separate online transactional processing (OLTP) and online analytical processing (OLAP) systems due to performance issues. They created systems that were only optimized for a specific workload, either OLTP or OLAP. With improving hardware, e.g., multi-core CPUs and terabytes of main memory, database management systems (DBMS) are changing [1]. Further, today's enterprise applications have mixed workloads running both, transactional and analytical, workloads [2]. An example is the available-to-promise check (ATP). Stock movements in a warehouse represent transactional queries whereas the ATP check itself is an analytical query aggregating over the product movements to determine the earliest possible delivery date for a customer [3]. Other applications that require a DBMS to handle mixed workloads are the dunning process where the application determines which customers have outstanding payments and accounting applications that calculate the profit and loss statement based on the aggregated records.

One technique to speed up the execution of expensive analytical queries is the use of *materialized views* [4]. A view defines a function from a set of base tables to a derived table and is recomputed every time the view is referenced. A materialized view stores the result of a view in the database and therefore does not require a recomputation. All materialized views that contain an aggregation [5] in its definition are called *aggregates* in this paper. Such pre-calculated results provide fast access to the data and reduce the overall load on the system. However, the benefit of speed comes with one tradeoff called *materialized view maintenance*.

Each time the underlying base tables are modified, a materialized view gets stale meaning the returned result is not up-to-date. The process of updating the materialized view in case of changes to the underlying base tables is called *materialized view maintenance*. This process was discussed in academia [6]–[8] and industry [9], [10]. However, the research is focused on data warehousing [8], [11], [12] and not on modern database architectures running mixed workloads. Compared to data warehouses, where maintenance downtimes may be feasible, transactional applications in mixed workload environments require high availability and throughput at any time.

In this paper, we introduce a mechanism to cache and revalidate analytical queries in the context of columnar in-memory databases (IMDBs). Columnar IMDBs got increased attention in the recent years since they are able to handle mixed workloads in a single system [1], [13]–[15]. Our work is based on [16] that introduced an aggregate cache for a columnar IMDB. The aggregate cache is a non-persistent caching engine inside the database. Like a materialized view, the aggregate cache consists of query results that are stored to speed up the access times. It leverages the main-delta architecture which separates a table into a read-optimized main storage and a write-optimized delta storage (cf. Figure 1). The idea of the aggregate cache is to cache only the main storage and unite the cached aggregate with the newly added records in the delta storage. This process is more efficient in many cases compared to calculating the complete result again. From the aggregate cache perspective, delete and update operations are equal and therefore called invalidations throughout this paper.

The aggregate cache guarantees that all results are up-to-date and it will never return a stale result. The aggregate cache consists of cache entries, each representing one unique aggregate query. A cache entry is created upon request and can be deleted upon request. Further, the cache entries are deleted in case the database shuts down or lacks main memory. For the future, we plan to include a mechanism into the cache that decides which queries are worth to cache and which are not.

For our evaluation, we have chosen a scenario of an ATP application. In our implementation, the application relies on a single, denormalized database table called Facts that contains all stock movements in a warehouse. Every movement consists

 ISBN: 978-1-61208-334-6

of an unique transaction identifier, the id of the product being moved, the date and the amount. The amount is positive if goods are put in the warehouse and negative if goods are removed from the warehouse. The aggregate that is based on the table Facts groups the stock movements by product and date and sums up the total amount per date and product. A detailed description of the ATP application can be found in our earlier work in [3]. We manually define the queries that will be cached and do not address the view selection problem [17] in the scope of this paper. We focus on the SUM and COUNT aggregation function as this is the dominant aggregate function in our introduced application. The aggregate cache also supports the standard SQL aggregate functions AVG, MIN, and MAX.

The paper is structured as follows: Section II gives a brief overview of related work in the area of materialized view maintenance. Section III explains the aggregate cache in detail including the algorithm and architecture. Section IV introduces the revalidation mechanism using the transaction manager and the incremental maintenance of a cached aggregate. In Section V, we analyze the cost factors of a revalidation. Section VI shows our experimental evaluation and Section VII concludes the paper with our main findings.

II. Related Work

Materialized view maintenance has been analyzed in academia [6]–[8] and industry [9], [10]. Blakely et al. were one of the first to propose a concept of incremental view maintenance [6] and Zhou et al. introduced a lazy view maintenance approach using delta tables [10]. These approaches are the foundation of our work. However, all work was done using traditional relational databases or even data warehouses with fewer restrictions [8], [11], [12]. In the context of this paper, we focus on enterprise applications consisting of OLTP and OLAP queries that require high availability and high throughput. The aggregate cache leverages the main-delta architecture that has not been evaluated before. To the best of our knowledge, an efficient maintenance strategy that is able to handle mixed workloads by leveraging available data structures of a columnar IMDB does not exists so far.

III. Aggregate Cache

The aggregate cache leverages the concept of the main-delta architecture as introduced in [16]. Separating a table into a main and delta storage has one main benefit. The separation allows to have a read-optimized main storage for faster scans and a write-optimized delta storage for high insert throughput. All inserts are inserted into the delta storage and are periodically propagated into the main storage in an operation called merge [18]. The fact that the main storage is only growing with a merge operation is leveraged by the aggregate cache so that only the results of the main storage are cached. All records from the delta storage are aggregated on-the-fly and united with the corresponding cached aggregate.

A. Architecture

The aggregate cache is located inside the column store engine of SanssouciDB (cf. Figure 1). There is a single aggregate cache manager instance that manages all cache entries. A cache

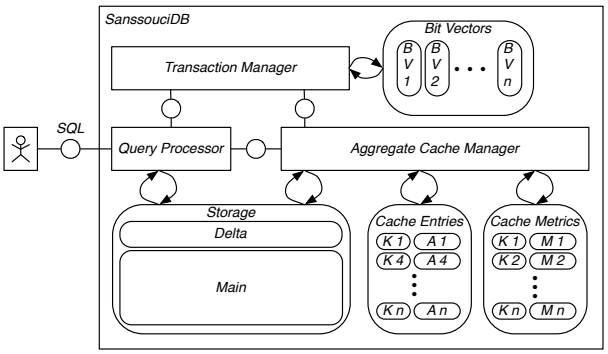

Figure 1. The internal architecture of SanssouciDB [2] with the main and delta storage, the aggregate cache manager and the transaction manager.

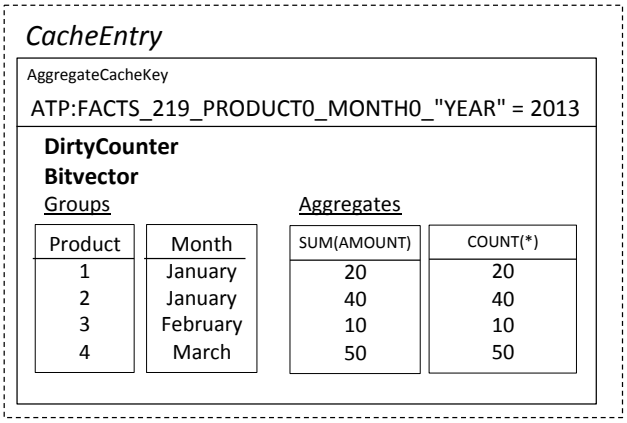

Figure 2. The structure of a cache entry for the ATP scenario.

entry consists of a key and a value. The key is composed of the table name, the group by columns, the where condition and the aggregates. The value is a complex structure consisting of the group by values and the corresponding aggregates. Additionally, relevant meta information about a cache entry, e.g. its creation time and its size, are stored in a structure called cache metrics. A detailed illustration is shown in Figure 2.

B. Insert-only Scenario

Insert-only scenarios, where the application does not change any previously inserted data, are beneficial for the aggregate cache because of a non-changing main storage. There, deletes and updates are replaced by logical updates using differential values. This restriction can be a result of legal requirements or the business logic behind an application. An insert-only scenario never invalidates the cache and therefore uses its full potential. With each merge operation, the cache entries can either be incrementally maintained using the records from the delta storage or invalidated in case they are not used anymore. The decision whether to maintain or invalidate a cache entry can be made based on a combination of known cache replacement algorithms, e.g. LRU or LRFU, and the benefit and resource requirements of a cache entry.

C. Cache Invalidation

Only a subset of applications have insert-only workloads. Most enterprise applications need to make changes to their

TABLE I. CONSOLIDATED BIT VECTORS OF TWO TRANSACTIONS RUNNING WITH TRANSACTION LEVEL SNAPSHOT ISOLATION. T_1 HAS DELETED THE RECORD WITH ID 2.

Facts				Bit Vector	
Id	Product	Date	Amount	T_1	T_2
1	Tire	3/1/2013	10	1	1
2	Tire	3/7/2013	20	0	1
3	Brakes	3/3/2013	5	1	1
4	Tires	3/12/2013	30	1	1

data which affect the cache entries. According to an analysis of Krueger et al. [18], their analyzed customer workload consisted of 8% delete and update queries. In contrast, the TPC-C benchmark [19] consists of 35% delete and update queries. Even a minor change in the main storage where only a single record is updated or deleted invalidates the cache entries that are based on that record. An invalidation equals a complete recalculation of the result which is the least efficient way to handle such behavior. In this paper, we present a solution that is able to reuse the stale cache entries by extracting information from the transaction manager.

1) Transaction Manager: Transactions are a main functionality of a RDBMS. The transaction manager is responsible that the database has a consistent state after each transaction and ensures the ACID properties. One mechanism to implement concurrent transaction handling is multi version concurrency control (MVCC) [20]. Using MVCC, multiple transactions can run simultaneously and each have their own visibility on the database. In SanssouciDB (cf. Figure 1), the transaction manager creates a bit vector representing the visibility of a table for an incoming query based on its transaction token.

2) Visibility Bit Vector: Each table has four bit vectors, two for the main storage and two for the delta storage. For the aggregate cache, only the main storage bit vectors are relevant since a cache entry is only based on them. One bit vector contains the information about visible records (create bit vector) and the other bit vector the information about invalidated records (delete bit vector). The combination of both (using an exclusive OR) creates a bit vector which is called consolidated bit vector. It contains the actual visibility for a specific transaction. An example for two consolidated bit vectors is shown in Table I. Two transactions are running concurrently and query the table *Facts* with four records. In the beginning, all four records are visible to both transactions (consolidated bit vector 1111). Transaction T_1 deletes the record with Id 2. Consequently, the bit at index 2 changes from visible to not visible for all further operations inside T_1 (invalidated bit vector 0100 and consolidated bit vector 1011). Transaction T_2 started at the same time as T_1. Depending on the isolation level, T_2 can read the record with Id 2 (consolidated bit vector 1111 with transaction level snapshot isolation) or cannot read it after the delete (consolidated bit vector 1011 with statement level snapshot isolation).

IV. CACHE REVALIDATION

To prevent an invalidation of cached aggregates in case of invalidations in the main storage, the aggregate cache has to provide the functionality to calculate the modifications between the cache entry creation and its usage. Since the previously introduced transaction manager is responsible for the visibility of records in a table, the aggregate cache can leverage that information to extract the invalidations.

A. Bit Vector Comparison

With each cache entry creation, a consolidated bit vector is stored as a snapshot of the database (cf. Figure 2). To further optimize the usage of a bit vector, only the bits after applying the WHERE clause of a query are used. We call theses bits relevant bits. Additionally, a version counter called *dirty counter* is stored. The dirty counter is an integer value from the transaction manager that is incremented with each invalidation.

In Figure 3, the enhanced aggregate cache algorithm is displayed in detail. Compared to first version (without the grey colored boxes), the steps until the *AggCache Lookup* step are the same. In the next step, in case a cache entry was found, the *dirty counter* of the cache entry is compared with the current *dirty counter* of the table to determine if there has been an invalidation or not. In case the dirty counter has changed, the consolidated bit vector of the cache entry is compared to the current delete bit vector (retrieved from the transaction manager) using a bitwise AND to determine the relevant invalidations. The result is a bit vector containing all changes since the cache entry creation. If that bit vector has no bits set, e.g., the invalidated records did not affect the cache entry due to the filters in the WHERE clause, the cache entry is still up-to-date and nothing has to be done. If at least one bit is set, the aggregate query has to be executed on the main storage using only the bit vector containing the relevant invalidations. The output is a query result containing all the information that has to be subtracted from the cached aggregate to provide an up-to-date result. In IV-C, we explain how this information can be used for the incremental maintenance of cached aggregates and maintenance timings.

B. Bit Vector Compression

Storing a bit vector for each cache entry requires additional memory. To reduce the required amount, the aggregate cache can use different compression techniques. The characteristic of a bit vector is beneficial for compression due to its limitation of only two distinct values (0 and 1). If the characteristic of the application that is using the table is known, the aggregate cache can leverage that knowledge to choose the optimal compression techniques. Based on Abadi et al. [21], we propose three compression techniques that are most suitable for bit vectors. All techniques require to provide direct comparisons of two bit vectors without additional decompression overhead.

1) Prefix/Suffix Encoding: Prefix or suffix encoding is a simple compression technique where the first or last sequence of the same value is replaced by two single values, the value itself and its number of occurrences. For bit vectors, this technique is useful if the deleted values are at the beginning or end of the bit vector. Depending on the location of the deleted values, prefix or suffix encoding is applied.

2) Run-length Encoding: Like prefix encoding, run-length encoding replaces a sequence of the same value by two single values, the values itself and its number of occurrences. Unlike prefix encoding, this can be done for any sequence in the bit vector. This possibility makes run-length encoding

 ISBN: 978-1-61208-334-6

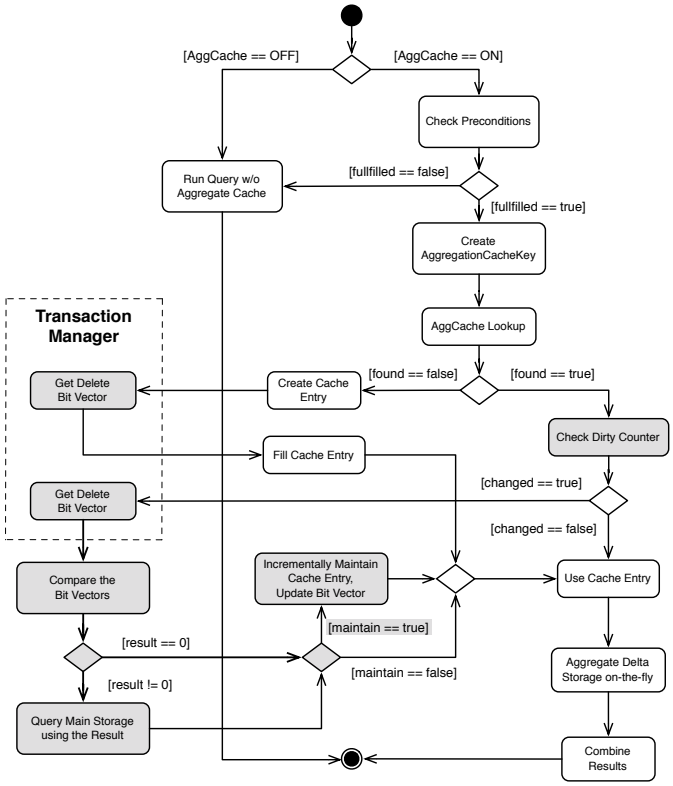

Figure 3. Activity diagram that visualizes the algorithm of the aggregate cache with the revalidation. The enhancements are colored in grey.

superior compared to prefix encoding. However, in case the number of deleted values is high and the bit vector shows the characteristic of an alternating pattern, the benefit of this technique decreases.

3) Cluster Encoding: Cluster encoding divides a bit vector into clusters of fixed size. Clusters that only contain equal values (only 0 or only 1) can be compressed and replaced by a single value. Clusters with mixed values (containing 0 and 1) are not replaced. The result is a new smaller vector. To remember which clusters have been compressed, an additional bit vector is created. Since the size of the cluster is not fix, it can be chosen based on the table size and the application characteristic.

The resulting compression ratio of each technique depends on the characteristic of the bit vector. A bit vector is influenced by the filter conditions in the WHERE clause that determines the ratio of relevant bits compared to the table size. Second, the application characteristic, e.g., are there bulk deletes or single deletes randomly distributed over the table. The automatic determination of the optimal compression technique based on the bit distribution is part of our future work.

C. Incremental Maintenance Timing

The maintenance of a cache is always a challenge. One crucial requirement for cache maintenance is to know what has changed. Using the introduced mechanism from Section IV-A, the aggregate cache is able to determine the modification that occurred between the cache entry creation and the current point in time. Having that information enables the aggregate cache to incrementally maintain a cache entry. In the following, we introduce four different maintenance timings.

1) Immediate Maintenance: Immediate maintenance, also known as eager maintenance, maintains all cache entries with each invalidation in the main storage. A maintenance is done even if the cache entry is not accessed in the meantime. The benefit is that the cache entries are always up-to-date and do not require any additional calculation when the cache is accessed. The disadvantage is a significant overhead for workloads with higher insert ratios since more time for maintenance is required than a cache access can save. Additionally, the maintenance has to be done for all cache entries that are based on the modified table which increases the amount of maintenance with an increasing number of cache entries. If n cache entries are based on a table and one record is invalidated, n cache entries have to be maintained. In our previous work, we have evaluated this timing in the context of materialized views and showed that it is inferior to other maintenance timings [22].

2) Deferred Maintenance: Deferred maintenance, also known as lazy maintenance, maintains a cache entry with each cache access. As a result, a cache entry is up-to-date after each cache access. With this maintenance timing, the cost of a maintenance is with the actual cache access, and not with the delete or update. In their work, Zhou et al. [10] introduced lazy maintenance in the context of materialized views for Microsoft SQL Server, with the same motivation of shifting the costs towards the the view access. They also proposed to perform the maintenance in periods of lower loads to reduce the actual overhead during the view access.

3) Periodical Maintenance: Periodical maintenance is a lazy form of deferred maintenance. The maintenance can be based on different criteria. One criteria is a certain number of cache accesses. In that case, a cache entry is maintained every N cache accesses. N can be chosen depending on the workload characteristic. A second criteria is a certain threshold of the invalidation ratio. If that ratio is reached, a maintenance is performed. With a periodical maintenance, the cache entry is not up-to-date at all times, as shown in the scenario where an execution on the main storage is required with each cache access.

4) Merge Operation Maintenance: This strategy does not maintain the cache entry between two merge operations. After a cache entry creation, it always revalidates the cache entry using the additional execution on the main storage to return the correct query result. Due to the missing maintenance, this strategy should only be used for cache entries where the base data never gets invalidated. In that case, the additional main storage run is not required. A second use case for this strategy can be a less strict freshness of the cache entry. If an application only requires a rough estimation that can be based on 'older' data, this approach is sufficient. So far, we have not implemented a functionality to return old cache entries.

D. Merge Operation

The aggregate cache relies on the main-delta architecture. With the periodical merge operation, the delta storage is merged into the main storage and all deleted values in the main

 ISBN: 978-1-61208-334-6

storage are removed. The aggregate cache has to react to that change. There are three possibilities how the aggregate cache can react. Each cache entry can have its own strategy based on the decision of the aggregate cache. First, a cache entry can be invalidated and is removed from the aggregate cache. Second, a cache entry can be incrementally maintained. In that case, the delta storage has to be added to the cache entry and the invalidated records from the main storage have to be subtracted from the cache entry. Third, the old value of the cache entry is ignored and the aggregate query recalculates the cache entry after the merge operation has completed.

E. Aggregate Functions

Aggregate functions can be divided into two categories, functions that are self-maintainable and functions that are not self-maintainable [23]. Looking at the standard aggregation function, SUM, COUNT, AVG (using SUM and COUNT) are self-maintainable with regards to inserts, deletes and updates. MIN and MAX are only self-maintainable with regards to inserts, but not with regards to deletes and updates because the information about the second highest/lowest value is not available. However, to overcome this issue, the database can store the n highest/lowest values, also known as MaxN/MinN. Using a data structure to store additional values enables an incremental maintenance until the data structure is empty and has to be refilled.

F. Joins

Joins are a concept that combines records of multiple tables into one result. The combination is based on a common field, the join condition, among the tables. Even though the aggregate cache implementation does not support joins yet, we suggest a concept how the handling of joins can be done. Equally to the algorithm for a single table, a cache entry of a join query consists only of the result on the main storages. To return the complete result, three joins are required which are combined with the cache entry. First, both delta storages of the two tables have to be joined. Second, the delta storage of table A has to be joined with the main storage of table B. Third, the delta storage of table B has to be joined with the main storage of table A. The join between the two main storages is the most expensive join because most records are in the main storage (a ratio of >100:1 [24]). The three other joins involve the smaller delta storages and therefore are cheaper.

Teorey et al. [25] introduced three types of relationships for relational databases: one-to-one, one-to-many, and many-to-many. The most frequently used type is the one-to-many relationship which is used to normalized database tables. For invalidations, the location of the invalidations is important. In case the invalidations only happen in one table, a single join is required to perform a maintenance. In case the invalidations happen in both tables (in a two table scenario) three joins are required. The third join makes sure that no values are removed twice, e.g., if records with the same join key from both tables are invalidated. In summary, a maintenance can vary between one join and three joins depending on the location of the invalidation. However, further knowledge about the application can reduce the complexity. An further implementation and evaluation has to verify this assumption and are part of the future work.

V. Cost Analysis Invalidation vs. Revalidation

In most cases, a cache revalidation is beneficial compared to a complete recalculation because less data has to be accessed. However, a revalidation is not always the best solution, especially after large bulk deletes. Comparing the revalidation process with the recalculation process shows that the only difference is the input bit vector used for the aggregate query. The aggregate query itself, the query processing steps and the used data structures are the same. As a result, the only cost factor is the number of accessed records in the main storage. For a revalidation, it is the number of relevant invalidations in the main storage. For a recalculation, it is the number of relevant visible records in the main storage. For a decision in favor of an invalidation or revalidation, the aggregate cache has to determine the *InvalidationRatio* respectively *Benefit*. The *InvalidationRatio* describes the relation of relevant invalidated records to relevant visible records at cache creation (cf. Equation 1). If less than half of the records are invalidated, it is beneficial to revalidate the cache entry. If more than half of the records are invalidated, an invalidation performs better (cf. Equation 3). The parameter α represents a factor which is required to combine the changes with the groups the cached result. Combining the two lists of results has a linear complexity. The factor will be further analyzed in Section VI-A.

$$InvalidationRatio = \frac{numSetBits(Bv_{visible} \wedge Bv_{delete})}{numSetBits(Bv_{visible})} \quad (1)$$

$$Benefit = \frac{1}{2} - InvalidationRatio - \alpha \quad (2)$$

$$Strategy(Benefit) = \begin{cases} \text{Revalidation} & Benefit \geq 0 \\ \text{Recalculation} & Benefit < 0 \end{cases} \quad (3)$$

The function $numSetBits$ returns the number of set bits for a given bit vector. $Bv_{visible}$ is the bit vector for relevant visible records at cache creation time, e.g. 00111011. Bv_{delete} is the current delete bit vector, e.g. 11100101. The bitwise AND would be 00100001 with 2 bits set. The resulting *InvalidationRatio* is 0.4 and consequently a revalidation is more beneficial.

VI. Experimental Evaluation

We implemented the aggregate cache in SanssouciDB but believe that the implementation in other columnar IMDBs with a main-delta architecture such as SAP HANA [26] will lead to similar results. Figure 1 illustrates the architecture of our implementation. The data and workloads we used are based on customer data and are parametrized to simulate different scenarios and patterns. The basic schema from our ATP scenario is shown in Table I with four columns.

All experiments and benchmarks have been conducted on a server featuring 8 CPUs (Intel Xeon E5450) with 3GHz and 12MB cache each. The entire machine was comprised of 64GB of main memory. Every benchmark in this section is run at least three times and the displayed results are the median of all runs.

 ISBN: 978-1-61208-334-6

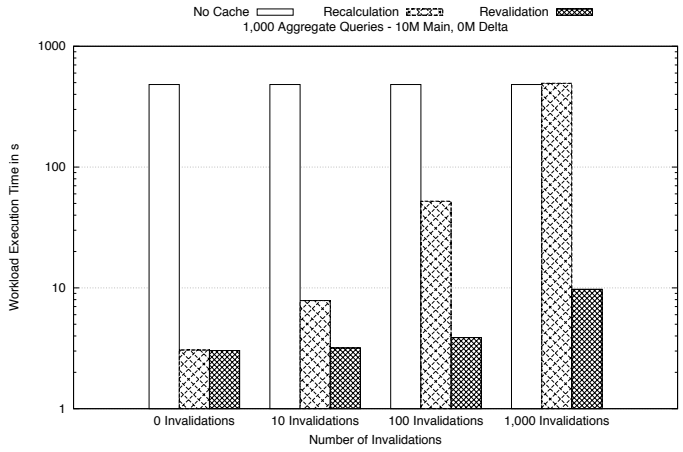

Figure 4. A revalidation outperforms an invalidation. The benefit of a revalidation increases which an increasing number of deletes.

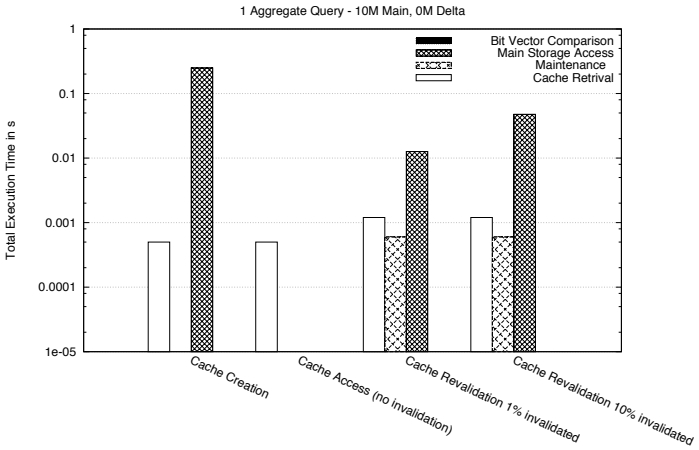

Figure 5. A benchmark that visualizes the run-time analysis of the different cache operations.

A. Invalidation vs. Revalidation

To show the benefit of a revalidation, we use four workloads with different invalidation behavior. A single OLAP query is executed 1,000 times. The different workloads vary in their number of invalidations. To have comparable execution times, the time for the delete operations will not be included into the measurement. The deferred maintenance timing is the default for all benchmarks. As seen in Figure 4, we compare the recalculation and revalidation strategy with the non-caching strategy to make the results more plausible. The non-caching strategy has equal query execution times since no caching is done and the OLAP query has to run 1,000 times. For the other two strategies, the query execution times vary depending on the invalidation behavior. With an increasing number of deletes, the difference between a revalidation and an recalculation increases, in favor for a revalidation. If a delete operation is always between two cache accesses (1,000 invalidations), a recalculation looses all its benefits compared to a non-caching strategy.

B. Run-Time Analysis

The runtime of accessing a cached aggregate can be divided into four steps: *a*) Bit vector comparison *b*) Main storage access and aggregate calculation *c*) Maintenance *d*) Cache entry retrieval All four steps are required for the revalidation strategy using a deferred maintenance timing. Steps 2 and 4 are used for the cache entry creation as well as cache entry recalculation (both are identical operations). The goal of this experiment is to measure the detailed costs for a cache entry creation respectively recalculation, and a cache entry revalidation. We also measure the costs for a cache access without a revalidation. A revalidation is measured using two scenarios with invalidation ratios of 1% and 10%.

Figure 5 shows the absolute results of the experiment with a logarithmic y-axis. The time to retrieve the cache entry is nearly the same for all. The maintenance costs for the revalidation strategy is also equal and not influenced by the invalidation ratio. Surprisingly, the comparison of bit vectors is so fast that the logarithmic y-axis cannot display it (the actual value 1ns). The main cost factor is the main storage access. For the revalidation, the time for accessing the main storage is the only time which increases with an increasing invalidation ratio. Comparing all four operations, the cache creation respectively recalculation is by far the most expensive operation.

C. Invalidation Ratio

Figure 5 indicated that the performance of the revalidation depends on the number of invalidated records. This experiment measures the execution time of a revalidation for invalidations ratios ranging from 0% to 70%. The results are compared to a recalculation. Based on the introduced *InvalidationRatio* from Section V, we assume that both approaches are break-even (have the same performance) when the invalidation ratio is approximately 0.5. Further, we measure the influence of the result size using aggregate queries with 100 and 10k groups.

The experiment in Figure 6 confirms that the performance increases linearly with an increasing number of invalidations. The results also confirm the *Benefit* we have introduced with Equation 3. For small group size of 100, the ratio is approximately 0.5. At a ratio of 0.5 respectively 50%, a revalidation has the same performance as a recalculation. This point is also known as the break-even point. For the query with the larger group of 10k, the break-even point has shifted towards a smaller ratio. The performance of the revalidation is influenced by the size of groups since the revalidation has to match the groups of a cache entry to the groups of a revalidation. This process has linear complexity because each item of the group is accessed once (using a hash-based approach).

D. Non-Relevant Invalidations

As explained in Section IV-A, the consolidated bit vector includes the information about relevant visible records. Figure 7 shows the benefit of that information in case non-relevant records are invalidated. Having this optimized bit vector, the aggregate cache is able to skip the main storage access in case the bit vector comparison produces a result containing only 0 (Figure 3). In contrast, the standard bit vector is not aware of that information and has to access the main storage until the query processor finds out that the result is empty. However,

 ISBN: 978-1-61208-334-6

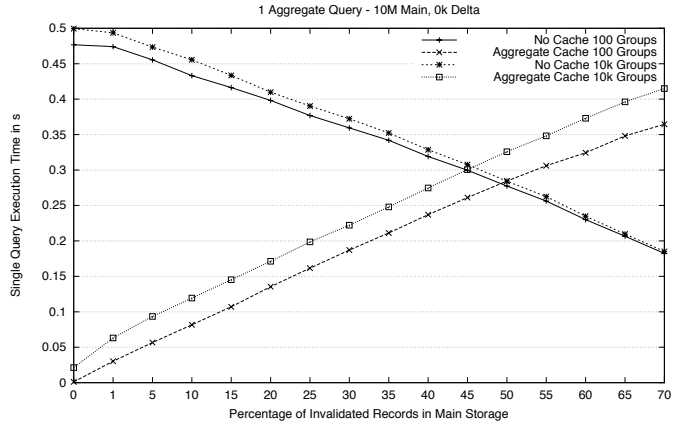

Figure 6. An experiment analyzing the influence of the number of invalidated records and the number of aggregate groupings on a revalidation.

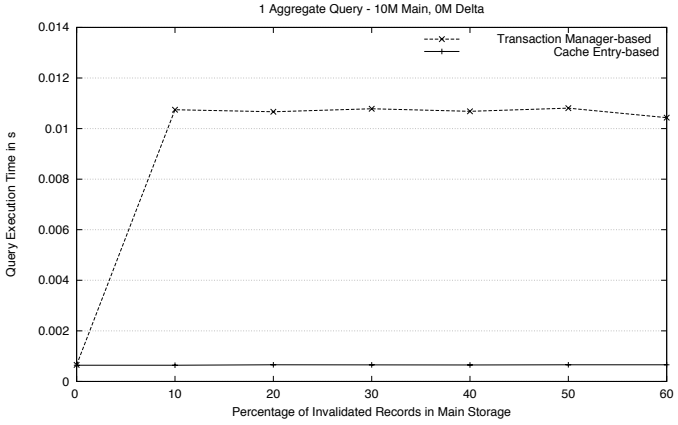

Figure 7. An experiment using on-relevant delete operations.

the experiment shows that despite an increasing number of invalidations on the main storage, the query execution time using the standard bit vector is constant.

E. Maintenance Timing

In this paper, four maintenance timings for the aggregate cache are proposed (Section IV-C). Each timing has different maintenance costs that are influenced by three cost factors. In the following experiment, we focus on one cost factor. Based on the analysis of workloads we had access to, the ratio of invalidations to aggregate queries is the driving factor for the maintenance costs.

The experiment (cf. Figure 8) has five different aggregate queries which are executed 200 times each. The invalidations are distributed uniformly and always invalidate records that are part of the cache entries. The periodic timing has an n of 20 that revalidates a cache entry every 50 accesses.

As expected, all four timings have the same execution time in case no invalidations happen. For 10 invalidations, which occur every 100 aggregate queries, all timings create maintenance costs. The merge timing always has to access the main storage after the first invalidation, but does not maintain the cache entry. As a result, the costs are high, even though the

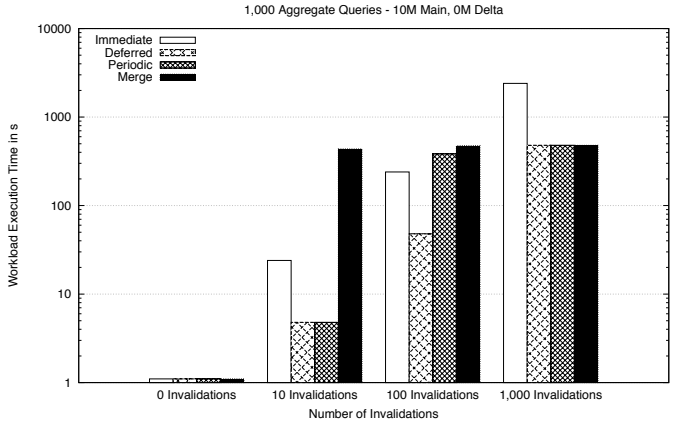

Figure 8. A comparison of the four proposed maintenance timings.

number of invalidations is low. The eager timing has to maintain five cache entries with every invalidation. This results in 50 revalidations. The deferred and periodic timing have lowest maintenance costs since they both require 10 revalidations. For the periodic timing, it is the best case since the revalidation of a cache entry is actually done right after the invalidation. With 100 invalidations (every 10 aggregate queries), the costs of the merge timing increase further. The costs for the eager and deferred time increase linearly. The periodic timing has a more than linear increase because the timing of the maintenance is not optimal. With 1,000 invalidations (one every aggregate query), the costs for the deferred, periodic and merge timing are the same. The eager timing has five times the maintenance costs of the other timings caused by the five cache entries. With only a single entry, the costs would be equally to the others.

In conclusion, the deferred maintenance timing is superior over the other timings. The costs of maintenance using an eager timing increase with an increasing number of cache entries. A periodic or merge timing might be applicable for workloads with a high number of invalidations, but the performance does never beat the deferred timing.

F. Mixed Workload Benchmark

The CH-benCHmark created by Cole et al. [27] is a mixed workload benchmark combing the TPC-C and TPC-H benchmark. Based on their work and the available data generator, we created scenarios using three different scale factors (1, 10, 50), Since the aggregate cache does not support joins yet, we are only able to use five of the OLAP queries.

In Figure 9, the revalidation algorithm is compared with the recalculation and a non-caching strategy (for better comparability). A scale factor of 50 creates 60M records in the order line table. The number of queries was fix for all three scale factors. The workload contains 3561 inserts, 230 updates, and a total of 380 aggregate queries. Each of the five aggregate queries was executed 76 times. The revalidation algorithm uses the lazy maintenance timing as this is the best performing timing (cf. Figure 8).

The results show that a revalidation outperforms a recalculation for all scale factors. A revalidation is up to 5

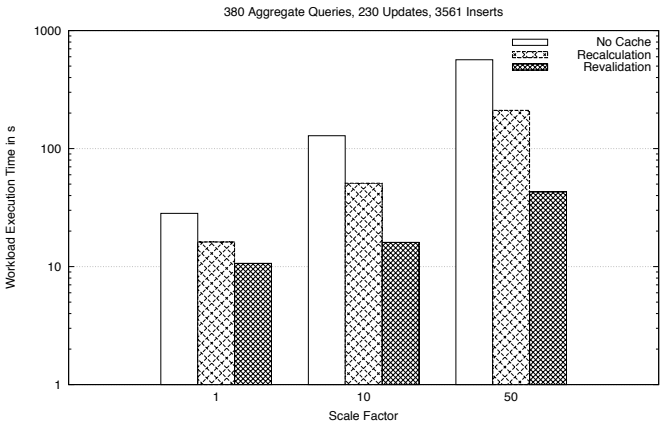

Figure 9. The CH-benCHmark [27] results.

times faster (scale factor 50). The results indicate that the performance advantage increases with an increasing scale factor. This validates the efficiency of the proposed revalidation algorithm.

VII. Conclusion

In this paper, we have introduced an efficient strategy to revalidate cached aggregates in an in-memory column store. Based on the existing insert-only implementation, we have explained the idea of the aggregate cache and motivated the necessity of a cache revalidation strategy for mixed workloads with updates and deletes. Using the available information from the transaction manager, the aggregate cache is able extract the information of invalidated records. A bit vector containing snapshot information of the database is added to the cache entry. To keep the cache entry size as small as possible, we have proposed three compression techniques that reduce the required amount of memory for a bit vector. We have compared the process of a recalculation with a revalidation and created a cost function to determine the optimal decision between the two. In our evaluation using an ATP scenario consisting of transactional as well as analytical queries, the experiments reveal that a revalidation outperforms an recalculation up to an invalidation ratio of 50%. The influencing factor of the revalidation is the number of relevant invalidated records. Our optimization to include the filter conditions into the bit vector reduce the amount of maintenance significantly.

In our future work, we plan to include the support for joins into the aggregate cache. Existing admission and evictions strategies can be extended by run-time information of the aggregate cache manager, for example the execution time or the result size. Also, an evaluation of our algorithm using other database architectures without a main-delta architecture is subject to further research.

References

[1] H. Plattner, “A common database approach for oltp and olap using an in-memory column database,” in SIGMOD, 2009, pp. 1–2.

[2] ——, “Sanssoucidb: An in-memory database for processing enterprise workloads,” in BTW, 2011, pp. 2–21.

[3] C. Tinnefeld, S. Müller, H. Kaltegärtner, S. Hillig, L. Butzmann, D. Eickhoff, S. Klauck, D. Taschik, B. Wagner, O. Xylander, A. Zeier, H. Plattner, and C. Tosun, “Available-to-promise on an in-memory column store,” in BTW, 2011, pp. 667–686.

[4] D. Srivastava, S. Dar, H. Jagadish, and A. Levy, “Answering queries with aggregation using views,” in VLDB, 1996.

[5] J. M. Smith and D. C. P. Smith, “Database abstractions: Aggregation,” Commun. ACM 1977.

[6] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, “Efficiently updating materialized views,” in SIGMOD, 1986, pp. 61–71.

[7] A. Gupta and I. S. Mumick, “Maintenance of materialized views: Problems, techniques, and applications,” IEEE Data Eng. Bull. 1995.

[8] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek, “Efficient view maintenance at data warehouses,” in SIGMOD, 1997.

[9] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin, “Materialized views in oracle,” in VLDB, 1998, pp. 659–664.

[10] J. Zhou, P.-A. Larson, and H. G. Elmongui, “Lazy maintenance of materialized views,” in VLDB, 2007, pp. 231–242.

[11] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom, “View maintenance in a warehousing environment,” in SIGMOD, 1995, pp. 316–327.

[12] H. Jain and A. Gosain, “A comprehensive study of view maintenance approaches in data warehousing evolution,” SIGSOFT Softw. Eng. Notes 2012.

[13] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, “Hyrise: a main memory hybrid storage engine,” in VLDB, 2010, pp. 105–116.

[14] A. Kemper, T. Neumann, F. F. Informatik, T. U. Mnchen, and D-Garching, “Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots,” in ICDE, 2011.

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, “Monetdb: Two decades of research in column-oriented database architectures,” IEEE Data Eng. Bull. 2012.

[16] S. Müller and H. Plattner, “Aggregates caching in columnar in-memory databases,” in 1st International Workshop on In-Memory Data Management and Analytics (IMDM), in conjunction with VLDB 2013, Riva del Garda, Trento, Italy, 2013.

[17] H. Gupta, “Selection of views to materialize in a data warehouse,” in ICDT, 1997.

[18] 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,” in VLDB, 2012.

[19] F. Raab, “TPC-C - the standard benchmark for online transaction processing (OLTP),” in The Benchmark Handbook, 1993.

[20] P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” ACM Comput. Surv., vol. 13, no. 2, Jun. 1981, pp. 185–221.

[21] D. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’06. New York, NY, USA: ACM, 2006, pp. 671–682.

[22] S. Müller, L. Butzmann, K. Howelmeyer, S. Klauck, and H. Plattner, “Efficient view maintenance for enterprise applications in columnar in-memory databases,” in EDOC, 2013, pp. 249–258.

[23] I. S. Mumick, D. Quass, and B. S. Mumick, “Maintenance of data cubes and summary tables in a warehouse,” in SIGMOD, 1997.

[24] H. Plattner and A. Zeier, In-memory data management: an inflection point for enterprise applications. Springerverlag Berlin Heidelberg, 2011.

[25] T. J. Teorey, D. Yang, and J. P. Fry, “A logical design methodology for relational databases using the extended entity-relationship model,” ACM Comput. Surv., vol. 18, no. 2, Jun. 1986, pp. 197–222.

[26] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “SAP HANA database: data management for modern business applications,” SIGMOD, 2011.

[27] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas, “The mixed workload ch-benchmark,” in Proceedings of the Fourth International Workshop on Testing Database Systems, ser. DBTest ’11. New York, NY, USA: ACM, 2011, pp. 8:1–8:6.

 ISBN: 978-1-61208-334-6