Cache Management for Aggregates in Columnar In-Memory Databases

Stephan Müller, Ralf Diestelkämper, Hasso Plattner
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
Email: {stephan.mueller, ralf.diestelkaemper, hasso.plattner}@hpi.uni-potsdam.de

***Abstract*—Modern enterprise applications generate workloads of short-running transactional queries as well as long-running analytical queries. In order to improve the execution time of computationally intensive analytical queries we have introduced an aggregate cache that makes use of the typical main-delta architecture of columnar in-memory databases (IMDBs) to cope with data modifications. In this work, we contribute a cache management system for that aggregate cache, which bases the cache admission and replacement decisions on novel profit metrics. These metrics are tailored to the main-delta architecture of IMDBs. They ensure that expensive aggregates are stored in the cache while light weight query results are rejected. For the profit estimation of a cached aggregate the system also takes into account transactional data modifications triggered by the enterprise application. Along with the profit metrics we introduce an asynchronous cache management algorithm designed for the main-delta architecture as well as the transactional data modifications. We evaluate the cache management system on mixed, transactional and analytical workloads and real customer data.**

***Keywords*-Aggregates, Materialized Views, Cache Management, In-Memory Database, Column Store**

I. Introduction

In the past, transactional and analytical queries have been associated with two separate applications for transactional processing (OLTP) or analytical processing (OLAP). This distinction is no longer applicable for modern enterprise applications [1], [2] because they make use of both, online transactional and analytical queries. In a contemporary financials and controlling application typical OLTP-style queries insert new bookings, whereas OLAP-style queries aggregate the records for profit and loss statements.

The OLAP-style queries may take a significant time to be processed [1]. That is why we have developed an aggregate cache which leverages the main-delta architecture of columnar in-memory databases in order to speed up recurring analytical queries in a consolidated environment [3], [4]. In a columnar IMDB, a table is stored by column vectors instead of row tuples and resides in memory. Since inserts of new tuples are usually more expensive in a column store than in a row store database, each table has a highly-compressed, read-optimized main storage and a write-optimized delta storage. The delta storage persists the transactional manipulations made to the database table. It contains orders of magnitude less tuples then the main storage. However, when the delta storage reaches a certain threshold, a merge process is triggered [5] that merges the tuples of the delta storage into the main storage.

Our aggregate cache stores the aggregation result returned from the main storage when an analytical query is executed for the first time. In the following, we call the result aggregate. If the same query is executed again, the cached aggregate is combined with the on-the-fly calculated result on the delta storage and returned to the application. With this approach the aggregate cache can provide an up-to-date result and save significant computation and execution time overhead, because the delta storage is much smaller than the main storage and on-the-fly aggregations on it are relatively fast.

In this work, we present a cache management system for the aggregate cache. It ensures that the aggregate cache does not grow arbitrarily large. Additionally, it prevents the aggregate cache from keeping unused or computationally lightweight aggregates in the cache. For the identification of such aggregates, the cache management system makes use of a profit metric. The metric assesses the performance benefit obtainable from each aggregate if it remains in the cache. Existing profit metrics are calculated from multiple runtime metrics such as the access rate of a cached aggregate, the execution time to calculate the aggregate, and the aggregate's size. These metrics are not optimal for the aggregate cache because they do not distinguish between the calculation time on the main storage and calculation time on the delta storage. However, that is important for the aggregate cache since it cannot accelerate recurring queries whose calculation time mostly originates in the on-the-fly aggregation on the delta storage. Thus, novel profit metrics are required for the aggregate cache that consider the main-delta architecture and the mixed workload.

The aggregate cache is designed to concurrently handle the database requests from multiple users and applications. That is why the management system should avoid blocking behavior during query processing. Previous cache management systems performed synchronous cache management [6]–[8]. The synchronous management has caused blocking behavior for every processed query. That is why we introduce an asynchronous cache management algorithm, which evicts cached aggregates decoupled from the query processing.

During the merge phase of a base table, the current implementation of the aggregate cache removes those aggregates from the cache whose base table is merged. The consequence is that the aggregate cache has to recalculate the aggregates from scratch the next time they are required. Instead of evicting the aggregates, a sophisticated cache management system can incrementally revalidate the affected aggregates while the merge process is in progress. The incremental revalidation process is basically the same process that the cache manager performs on a cache hit, except that the aggregate cache updates the cache entry after it has been combined with the result from the on-the-fly aggregation on the delta storage. For many cached aggregates, the on-the-fly aggregation is lightweight compared to the complete recalculation. Thus, the cache performance can be significantly improved if cached aggregates are revalidated instead of evicted. That is why we describe an aggregate revalidation algorithm for the merge process.

Instead of using a mixed workload benchmark like the CH-Benchmark [9], we evaluate our cache management system on a financials and controlling scenario that is based on real world data and query templates from an operational enterprise system. For the evaluation purpose, we have implemented the cache management system in SanssouciDB, a columnar IMDB with main-delta architecture.

In the following section, we describe how our cache management system differs from other existing cache management systems. In Section 3, we give an architectural overview of the aggregate cache with our cache management extensions. We introduce the novel profit metrics in Section 4 and the asynchronous cache management algorithm in Section 5. In Section 6 we describe the revalidation algorithm for the merge process. We evaluate the profit metrics and the algorithms on the financials and controlling scenario in Section 7. In Section 8, we summarize our results and give an outlook on future work.

II. Related Work

Caches are not only applied in database systems but also in systems for disk buffering or client-server applications. Each of these systems requires a cache management system to maintain the cache.

In disk buffering systems, the cache is used to provide fast access to data on disk. Disk buffering systems have to manage a limited size of cache space. Hence, they favor frequently and recently accessed data blocks over rarely touched blocks in order to maximize the system's performance. In the past, many algorithms have been proposed to most profitably manage the cache. Least-frequently-used (LFU) [10], least-recently-used (LRU) [11], k-least-recently-used (LRU-K) [12], 2Q [13], MultiQueue [14] and least-recently-frequently-used (LRFU) [15] are just a few to be mentioned. It takes only minimal effort to adjust these algorithms to work with the aggregate cache. However, they do not consider the cached aggregate's size, the execution time for a cache hit and the execution time for a cache miss. These parameters are assumed to be equal or at least almost equal in a disk buffering system, but can significantly differ in a cache system for database aggregates. For example, we have two analytical queries which have been executed similarly frequently in the recent past. The first analytical query has a processing time of several seconds when it is not cached, but runs only a couple milliseconds when it is cached. The second aggregate query may take only several hundred milliseconds overall execution time when it is not cached and about the same time when it is cached. The above mentioned algorithms would not prefer one query over the other. Our cache management system should prefer the first query over the second in order to maximize the saved processing time.

Another popular application for caches and, hence, cache management systems lies in client-server systems. A client retains information received from the server via a network in order to avoid redundant data transfer and network contention. Client-side caches have also been introduced for database systems [16]. Opposite to these caches our aggregate cache resides on the server. The server side cache can serve multiple tenants working on a single consolidated database system. In this way, multiple clients can profit from a single cached aggregate on the server.

A couple of cache management systems for materialized views and query results have already been introduced and implemented in the past. In the following, we first provide a non-comprehensive overview over previous cache management systems and then distinguish them from our aggregate cache.

Scheuermann et al. introduce WATCHMAN [6] system as one of the first approaches to manage database query results. It makes cache admission and replacement decisions based on the execution frequency, the execution time, and the result set size of a query. The authors show that WATCHMAN significantly improves the cache performance over an at that time sophisticated disk buffer management algorithm LRU-K. In a follow-up project the group around Scheuermann extended the original WATCHMAN system described in [6] to support subqueries and to consider update costs for the result sets [17].

Kotidis et al. implement a view management system called DynaMat for data warehouses. It dynamically manages materialized aggregates [7]. The authors evaluate four metrics for cache admission and replacement, the frequency metric, the execution time metric, the result set size metric and a combination of the three of the previous metrics. They show that the combined metric, which is similar to the WATCHMAN metric, performs best.

Park et al. design a caching mechanism for OLAP systems, which is able to reuse partial results for related queries in drill-down and roll-up sequences [8]. The cache admission and replacement algorithm is extended in order to take into account the profit of a query result for multiple related queries. Related queries are part of the same drill-down or roll-up sequences and were either executed in the recent past or will very likely be executed in the near future. The reuse of partial results saves their system expensive random accesses to the disk. Therefore, the system performance can significantly profit from reusing partial and overlapping query results residing in memory. In our setup, however, the data is already stored in memory and random accesses to disk are no limiting factor any more. Additionally, Park et al. are limited to matching canonical drill-down and roll-up sequences. Real world enterprise workloads contain more complex analytical queries with subqueries and joins. That is why, in the recent years, more sophisticated research has been done on reusing partial views mitigating a canonical query schema [18]. However, this is out of scope of this work.

Opposite to all the above cache management systems our management system administers an aggregate cache on an IMDB with main-delta architecture. It considers the costs to do an on-the-fly aggregation on the delta storage for every query answered from the cache. That was unnecessary in the above system setups, because the cached aggregate was delivered as is.

Additionally, none of the above systems are designed for mixed workloads in which data modifications can occur at any time. Some of them consider bulk data modifications during dedicated maintenance intervals but they cannot process combined online and analytical workloads. In contrast, our

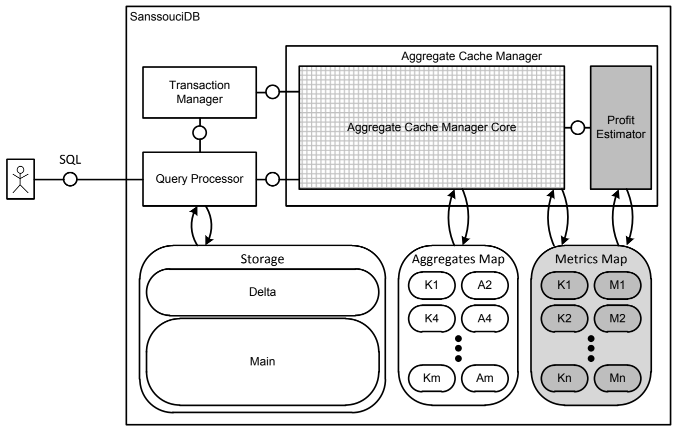

Figure 1. The Aggregate Cache Manager inside SanssouciDB

cache management system is designed to deal with mixed workloads.

III. Aggregate Cache Architecture

The aggregate cache and the cache management system are fully integrated into SanssouciDB. Figure 1 gives an architectural overview over the cache management system. The white components are already existing components, while the gray parts are cache management extensions that are implemented for this work. The textured gray component is extended in this work in order to support cache management. Figure 1 shows that the aggregate cache consists of three major components, the aggregate cache manager, the aggregates map, and the metrics map.

The aggregate cache manager consists of the cache manager core and the profit estimator. The cache manager core stores the cached aggregates in the aggregate map and updates the runtime information in the metrics map. It receives cacheable aggregates from the query processor and delivers cached aggregates back to the query processor if a query can be answered from the cache. Before a cached aggregate is delivered to the query processor, the cache manager core checks whether the cached aggregate is still up to date. It asks the transaction manager if rows were invalidated in the main storage since the point in time the aggregate was created. In case rows were invalidated, the aggregate cache updates the cached aggregate before it delivers the aggregate to the query processor.

The profit estimator is an extension to the existing aggregate cache. It calculates the profit for every cached aggregate by applying a profit metric on the runtime information residing in the metrics map.

IV. Profit Metrics

The better the profit metric assesses the benefit of a cached aggregate, the better the aggregate cache can perform. A higher cache performance, in turn, leads to a better system performance. Before we introduce the novel metrics, we assess existing profit metrics in buffer management systems and previous query result caches in the context of the aggregate cache.

A. Existing Buffer Profit Metrics

We start with the metrics originating in disk buffering systems. The symbols used to describe the following metrics are explained in Table I:

1) **LRU:** The LRU metric defines the profit of cached aggregates by their last access [11].

$$profit_{LRU}(q) = \frac{1}{t - last\_access_q} \tag{1}$$

2) **LRU-K:** An extension of the LRU metric is the k-recently used metric which considers the k most recent accesses of a cached result [12].

$$profit_{LRU-K}(q) = \begin{cases} \frac{1}{t - kth\_access_q}, & \text{if } \geq \text{k accesses} \\ 0, & \text{else} \end{cases} \tag{2}$$

3) **LFU:** The least frequently used metric (LFU) rates the cached aggregates by their recurrence. The recurrence increases constantly with every access [10].

$$profit_{LFU}(q) = recurrence_q \tag{3}$$

4) **LRFU:** The least frequently recently used metric (LRFU) rates aggregates by their recurrence and recency of access. It actually is not a single metric, but a spectrum of metrics covering the range between the LRU and the LFU metric [15].

$$profit_{LRFU}(q) = \left(\frac{1}{2}\right)^{\lambda \cdot (t - last\_access_q)} \quad \text{with } 0 \leq \lambda \leq 1 \tag{4}$$

For $\lambda = 0$ $profit_{LRFU}(q) = profit_{LFU}(q)$ and for $\lambda = 1$ $profit_{LRFU}(q) = profit_{LRU}(q)$. This is proven in [15]. If λ lies in between 0 and 1 the LRFU metric returns a profit between the LFU profit and the LRU profit. Such a profit is desirable, because the recurrence and the recency with which a cached aggregate is accessed are two important indicators for the benefit of a cached aggregate.

The above metrics only consider access rates and frequencies of cached aggregates. This is sufficient for disk buffering systems, because they manage disk blocks of equal size and similar disk fetch times. In database result caches or aggregate caches like ours, these metrics do not perform well, because the aggregate sizes can differ significantly. Some may contain only a couple groups and only a few columns while others have hundreds or even thousands of groups and several dozens to hundreds columns. Additionally, the time needed to calculate the query result or the cached aggregate can differ substantially. Some calculations finish after a few milliseconds while others process for several seconds or even longer.

B. Existing Query Cache Profit Metrics

The profit metrics of previous query result caches reflect the above thoughts, since they consider the result set size as well as the execution time of the cached query.

1) **WATCHMAN:** The WATCHMAN metric considers the k last references as well as the aggregate size and the execution time of a query [6].

$$profit_{\text{WATCHMAN}}(q) = profit_{LRU-K}(q) \cdot \frac{t_q}{result\_size_q} \tag{5}$$

2) **DynaMat:** Similarly to the WATCHMAN metric, the DynaMat metric takes the result size and the

TABLE I. DEFINITION OF SYMBOLS

Symbol	Definition
t	current time
$last\_access_q$	time of the last access to aggregate q
kth_access_q	time of the kth last access to aggregate q
$recurrence_q$	number of times aggregate q has been referenced
t_q	calculation time for an aggregate q on the main and delta storage without cache
t_{main}	calculation time for an aggregate q only on the main storage
$t_{cache+\Delta}$	calculation time for an aggregate q on the cache and the delta storage
$result\_size_q$	memory space required to store an aggregate q that is calculated on the main storage
$delta_q$	number of aggregated tuples in the delta storage relevant for aggregate q
$main_q$	number of aggregated tuples in the main storage relevant for aggregate q
$inval_q$	number of invalidated tuples in the main storage affecting aggregate q
Δ_q	number of tuples involved in the delta compensation

calculation time of an aggregate into account. They are combined with the recurrence of the query [17].

$$profit_{DynaMat}(q) = profit_{LFU}(q) \cdot \frac{t_q}{result\_size_q} \quad (6)$$

These metrics worked well in previous query result caching systems [6], [7]. However, the aggregate cache differs in at least two substantial characteristics from the previous caches, which should be reflected in the profit metric. First, the cache is tailored to the main-delta architecture of columnar IMDBs. Second, the cache has to deal with invalidated tuples on the main storage.

C. Novel Profit Metrics for the Aggregate Cache

The above metrics only consider the overall query execution time. They do not distinguish between processing an aggregation on the main storage and on the delta storage. The differentiation is important for the performance of the cache as the following example demonstrates. The aggregate cache keeps only the aggregation result from the main storage and performs an on-the-fly aggregation on the delta storage for every incoming query. If a query only touches tuples in the delta storage and the overall execution time of the query is considerably long, the cache does not speed up this query. However, the above metrics assign the associated aggregate a high profit, because the overall execution time is long and the cached (empty) aggregate has a small size.

The following four metric extensions are tailored to the aggregate cache architecture. They distinguish between the aggregation on the main storage and the delta storage.

1) **TAR:** The tuples aggregated ratio (TAR) metric represents the ratio of the tuples aggregated on the main storage and the tuples aggregated on the delta storage for a query q. The ratio yields more profitable results, if the number of aggregated delta tuples is low or the count of processed main tuples is high. That is desirable, because the tuples on the delta are aggregated whenever q is processed, whereas the tuples on the main are aggregated only when the aggregate is cached.

$$profit_{TAR}(q) = \frac{main_q}{\Delta_q + 1} \quad (7)$$

Note that a "+1" is added to the tuples touched in the delta storage in order to avoid a zero division. For the simple case of just having a single table, $\Delta_q = delta_q$.

2) **ETR:** The execution time ratio (ETR) metric is the proportion the main storage processing time and the delta storage processing time of a query q. It favors queries that have a short processing time on the delta storage and a long processing time on the main storage. We want to cache the aggregates of these queries, because they are answered quickly from the cache, but they need a significant amount of time if they are calculated from the main storage.

$$profit_{ETR}(q) = \frac{t_{main}}{t_{cache+\Delta}} \quad (8)$$

3) **TAD:** The tuples aggregated difference (TAD) metric considers the number of tuples that do not have to be aggregated when a query q is answered with the help of the aggregate cache. The tuples on the delta storage are aggregated on a cache hit and miss to answer q. The tuples on the main storage are not aggregated on a cache hit, because the aggregation result is cached by the aggregate cache. Their count is the number of tuples saved on a cache hit. The higher it is, the higher is the profit of q's aggregate.

$$profit_{TAD}(q) = (main_q + \Delta_q) - \Delta_q = main_q \quad (9)$$

4) **ETD:** The execution time difference (ETD) metric reflects the time saved when a query q is executed with the aggregate cache. The processing time on the delta storage is needed to answer q in both cases, when the q's aggregate is cached and when it is not cached. Therefore, the time saved when q's aggregate is cached is the main processing time. The higher this main processing time is, the higher is the profit of q's aggregate.

$$profit_{ETD}(q) = (t_{main} + t_{cache+\Delta}) - t_{cache+\Delta} = t_{main} \quad (10)$$

Manipulative database transactions like deletes and updates can invalidate rows in the main storage [19]. Thus, the deletes and updates potentially have an impact on the cached aggregates. In case the deleted rows are part of a cached sum the rows are added up and then subtracted from the cached sum. This on-the-fly process occurs whenever a query is answered from the aggregate cache.

To address the invalidation of rows in the main storage and the resulting compensation process in the cache, we define the following compensation factor that is based on the number of invalidated tuples relevant for aggregate q, $inval_q$, and the number of aggregated tuples $main_q$ in the main storage:

$$icomp(q) = \frac{1}{2} - \frac{inval_q}{main_q} \tag{11}$$

If more than 50% of the aggregated tuples are being invalidated, the profit becomes negative; which indicates that an on-the-fly aggregation on the main storage is more cheaper than using the cached aggregate.

The novel profit metrics combine one of the four main-delta metric extensions with the invalidation compensation extension, the LRFU metric, and the size of the cached aggregate. That allows them to assess the profit of each cached aggregate more precisely than the existing profit metrics.

1) **AC-TAR:** The aggregate cache tuples aggregated ratio metric (AC-TAR) is a combination of the LRFU metric (cf. Equation 4), the tuples aggregated ratio metric (cf. Equation 7), the invalidation compensation (cf. Equation 11), and the size of the aggregate associated with query q.

$$profit_{AC-TAR}(q) = profit_{LRFU}(q) \cdot icomp(q) \cdot \frac{profit_{TAR}(q)}{result\_size_q} \tag{12}$$

2) **AC-ETR:** The aggregate cache execution time ratio metric (AC-ETR) combines the LRFU metric (cf. Equation 4), the execution time ratio metric (cf. Equation 8), the invalidation compensation (cf. Equation 11), and the size of the aggregate associated with query q.

$$profit_{AC-ETR}(q) = profit_{LRFU}(q) \cdot icomp(q) \cdot \frac{profit_{ETR}(q)}{result\_size_q} \tag{13}$$

3) **AC-TAD:** The aggregate cache tuples aggregated difference metric (AC-TAD) is a combination of the LRFU metric (cf. Equation 4), the tuples aggregated difference metric (cf. Equation 9), the invalidation compensation (cf. Equation 11), and the size of the aggregate associated with query q.

$$profit_{AC-TAD}(q) = profit_{LRFU}(q) \cdot icomp(q) \cdot \frac{profit_{TAD}(q)}{result\_size_q} \tag{14}$$

4) **AC-ETD:** The aggregate cache execution time difference metric (AC-ETD) is assembled from the LRFU metric (cf. Equation 4), the execution time difference metric (cf. Equation 10), the invalidation compensation (cf. Equation 11), and the size of the aggregate associated with query q.

$$profit_{AC-ETD}(q) = profit_{LRFU}(q) \cdot icomp(q) \cdot \frac{profit_{ETD}(q)}{result\_size_q} \tag{15}$$

V. Cache Management Algorithm

The algorithm evicts aggregates from the cache that do not improve the overall system performance. Such aggregates may be empty aggregates or aggregates based on meanwhile invalidated rows. On-the-fly recalculation of these aggregates is cheaper than retaining the aggregates in the cache. The algorithm also has to remove the least profitable aggregates from the cache in case the system is running out of memory. Many analytical queries may be processed in parallel along with even more transactional operations. That is why, the algorithm should not block the system's progress with every incoming analytical query.

For the given reasons, we propose a cache management algorithm which maintains the aggregate cache asynchronously to the query processing. In previous systems the aggregate cache was maintained with every incoming analytical query [6]–[8]. We simply add the queries to the aggregate cache and trigger a maintenance routine to evict the least profitable queries in time intervals and when memory space is running low.

The regular cache maintenance and the aggregates revalidation task during the merge phase requires the algorithm to be split into three parts: The first part updates the cache metrics, the second part evicts queries from the cache, and the third part removes entries from the metrics map.

1) Cache Metrics Update: This first part of the algorithm updates the information stored in the metrics map. If the aggregate matching the executed query already exists as an entry in the metrics map, the entry is updated. This process requires a read lock on the metrics map, because another process in the system may concurrently remove entries from the map. If there is no matching entry in the map, the procedure creates a new entry and fills it with the information obtained from the system. That requires a write lock on the metrics map, since other procedures handling other analytical queries concurrently may add new entries to the metrics map. However, the algorithm inserts the entry to a hash map so that the procedure has an average complexity of O(1) [20]. So in comparison to the other parts of our algorithm, the update is processed very quickly and does not cause noticeable blocking behavior.

2) Cache Trimming: Trimming the cache is more complex than updating the metrics for a single aggregate. The procedure is described in Algorithm 1. It is executed periodically and when the system's memory space is low.

First, the procedure iterates over the cache metrics map M and appends each entry whose associated aggregate is cached in a separate list ca. For this task it acquires a read lock on M. This process has a complexity of O(n), where n is the number of entries in the cache metrics map. Therefore, the lock time scales linearly with the number of records in M. The collection process does not block the update of information in existing entries, because that only requires a read lock on M. However, it blocks the insertion of new entries into the metrics map, because the insertion of new entries requires an exclusive write lock. The insertion of new entries is only necessary, when a unknown query gets cached. In that case, the query has to be processed on both, the main and the delta storage. That processing time exceeds the time to collect all cached aggregates in most cases so that the blocking behavior is hardly noticeable.

As the next step, the procedure sorts the entries in the list ca

by profit in ascending order. Since most of the profit metrics introduced in Section IV require a timestamp, the a current timestamp is handed to the profit estimator that encapsulates the profit metric. The sorting is the most costly part in the cache trimming procedure. It has a complexity of O(n log(n)), where n is the number of cached entries. However, it does not require any lock, because it works on a temporary list. Thus, it cannot cause any blocking behavior.

Given this sorted list of cached entries the procedure starts removing entries from the aggregate map A. It fetches entries from the front of the list ca until the new cache size is reached or the profit of an entry is bigger than zero. Recapture that the profit becomes negative if more than 50% of the rows that the cached aggregate is based on are deleted. The removing of cache entries requires locks on both the aggregate map A and the metrics map M. The cache trimming procedure obtains and releases them for the eviction of each entry in order to avoid blocking behavior.

If new aggregates are added to the cache, while the procedure is sorting or removing entries from the list ca. They are ignored for the current trimming process.

Algorithm 1 Cache Trimming Procedure

```
Require: aggregate_map $A$, metrics_map $M$,
    current_cache_size $cs$, target_cache_size $ts$
 1: procedure TRIM_CACHE($A$, $M$, $cs$, $ts$)
 2:     cached_aggregates $ca \leftarrow$ [ ]
 3:     $M$.acquire_read_lock()
 4:     for all metrics_map_entry $\langle k, v \rangle$ in $M$ do
 5:         if $v$.is_cached then
 6:             $ca$.append($\langle k, v \rangle$)
 7:         end if
 8:     end for
 9:     $M$.release_read_lock()
10:     timestamp $t \leftarrow$ current_time()
11:     profit_estimator.sort($ca$, $t$)
12:     for all metrics_map_entry $\langle k, v \rangle$ in $ca$ do
13:         if profit_estimator.profit($v$) $> 0$ then
14:             if $cs < ts$ then
15:                 break
16:             end if
17:         end if
18:         $A$.acquire_write_lock()
19:         $M$.acquire_read_lock()
20:         $cs \leftarrow cs - v$.result_size
21:         $M$.invalidate($k$)
22:         $A$.evict($k$)
23:         $M$.release_read_lock()
24:         $A$.release_write_lock()
25:     end for
26: end procedure
```

3) Metrics Trimming: Similarly to the cache entries, the metric entries are trimmed in periodic intervals. For the metrics trimming task, all metric map entries for uncached aggregates are obtained and stored in a list. A read lock on the metrics map is required for this operation. It has a complexity of O(n), where n is the number of entries in the cache metrics map. The read lock may block the insertion of unknown aggregates to the cache, but as described previously, query processing should not noticeably be blocked.

The list of uncached aggregates is sorted by LRU in order to find the metric map entries that have not been used for the longest period of time. They are evicted from the metrics map one after another until the defined threshold is reached. For every entry eviction, a write lock on the metrics map is acquired and released after the entry is removed from the metrics map. This fine grained lock handling avoids blocking behavior.

VI. Incremental Revalidation Algorithm for the Merge Process

The incremental revalidation algorithm for the merge process updates those cached entries whose underlying base table is merged. It is integrated with the merge process described in [21]. During the merge prepare phase, the algorithm identifies and collects all cached aggregates that are based on the table being merged. It marks the aggregates to indicate that they need revalidation.

When the merge process is in progress, the algorithm orders the cached aggregates by their profit in descending order, so that the most profitable aggregates are at the beginning of the list. It incrementally revalidates the aggregates beginning from the head of the list until the merge process is finished. Directly after the revalidation the aggregates are marked as revalidated.

When the merge process is committed, the algorithm removes all aggregates which have not been updated from the cache, because they do not represent the aggregation result on the new main storage any more.

VII. Evaluation

For the evaluation, we implemented the algorithms and profit metrics in SanssouciDB [21], an IMDB with main-delta architecture. However, we are confident that our algorithm and metrics yield similar results when implemented in other IMDBs such as SAP HANA [22] or Hyrise [23]. We evaluate our algorithms and metrics with a financial accounting application. Other than mixed workload benchmarks such as the CH-benchmark [9] the application works on a database with real customer data. It also generates a mixed workload with OLTP-style inserts for the creation of accounting documents and OLAP-style queries for the calculation of reports like profit and loss statements. Therefore, the financial accounting application suits our evaluation purposes.

The application's database contains 22 million records in a single, denormalized table. We generated inserts based on these records for our workload. We also extracted 100 OLAP-style aggregate queries from the application and validated these with domain experts. They contain at least one aggregation function. From these 100 distinct OLAP-style queries, we create an analytical workload with 1000 queries, which we use for the following experiments. The server for the benchmarks has 4 Intel Xeon processors with a total of 40 physical cores and 1 TB of main memory.

A. Delta Storage Tuples

In the first experiment, we vary the number of tuples in the delta storage and compare the four aggregate cache metrics AC-ETD, AC-TAD, AC-ETR, and AC-TAR with the existing profit metrics LRU, WATCHMAN (WM), and DynaMat (DYN). The results are displayed in Figure 2. As performance measure we use the workload execution time. The lower it is, the better a metric performs. We set the eviction threshold to 80%. As a consequence the cache is trimmed to 80% of its

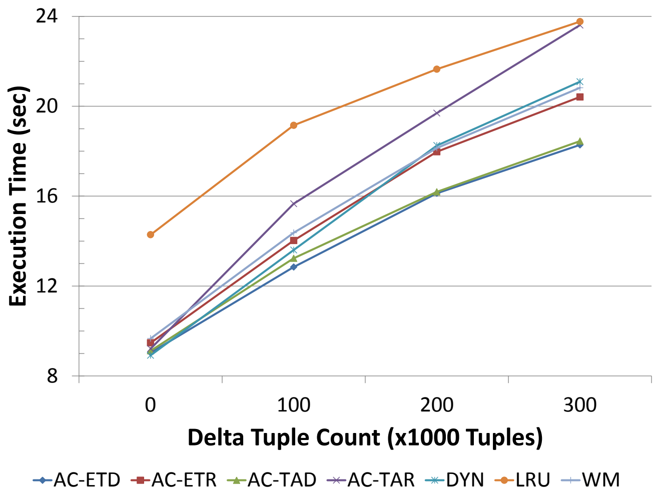

Figure 2. Comparison of Profit Metrics with Varying Delta Sizes. (Cache Size 200kB, Eviction Threshold 80%)

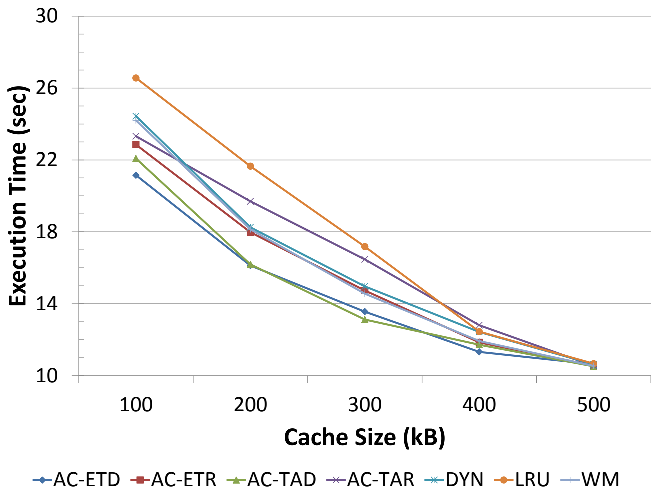

Figure 3. Comparison of Profit Metrics on Different Cache Sizes. (Delta Tuple Count 200,000, Eviction Threshold 80%)

maximum size when the maximum size of 200kB is reached. 200kB are sufficient to cache about 40 to 50 aggregates. The LRFU decay factor λ is set to 0.0001. This value has proven to perform best in our experiments.

The results show that the execution times of all profit metrics increase with a growing delta storage. The reason is that the aggregations on the delta storage become more expensive with an increasing delta size. When the delta storage is empty, all metrics, except the LRU metric, show similar performance results. That is the case, because the delta access time is very low and almost equal for all queries. With increasing delta size, the AC-ETD and AC-TAD metrics more and more outperform the other metrics. When the delta contains 200,000 tuples or more, the two metrics perform at least two seconds better than the existing WATCHMAN and DynaMat metrics. That is a performance gain of at least 11%. The AC-ETD and AC-TAD metrics constantly yield workload execution times that are five to six seconds faster than the LRU metric, independent from the delta size.

The ratio metrics AC-ETR and AC-TAR perform one and a half to six seconds worse than AC-ETD and AC-TAD metrics. The reason is that the ratio metrics assign high profit to all aggregates that consider only few tuples on the main and the delta storage. For example, an aggregate a is computed over 2 tuples in the delta storage and 100 tuples in the main storage. An aggregate b is calculated over 2,000 tuples in the delta storage and 50,000 tuples in the main storage. When the size and access history is equal for a and b, the AC-TAR metric assigns a profit to a that is twice as high as b's profit. Consequently, the AC-TAR metric favors lightweight aggregates like a. However, caching b is better for the cache performance, because the time to calculate b on the main storage is higher than the time to compute a. The AC-ETR metric also assigns a higher profit to a, but the profit is less than twice as high, since the aggregation time does not scale linearly with the number of tuples aggregated. That is why the AC-ETR metric performs up to three seconds faster than the AC-TAR metric.

Since the AC-TAR metric favors lightweight aggregates the performance of the metric decreases to the performance of the LRU metric when the delta storage contains 300,000 tuples. The LRU metric shows the worst results because assessing an aggregate's profit on the last access only is not sufficient. The AC-TAD and AC-ETD metric constantly have results that are five to six seconds faster. That is a performance benefit of more than 35%.

B. *Cache Size*

In a second experiment, we evaluate the influence of the cache size on the performance of the profit metrics. The results are shown in Figure 3. The execution time is the same for all metrics when the cache size is 500kB. Since the aggregates have a total size of 450kB, they all fit into the cache of 500kB size. Therefore, results from the runs with 500kB cache size show the optimal aggregate cache performance because no aggregate is evicted in any of the runs. In operational systems, the cache can grow several hundred gigabytes large because the systems process more than a hundred distinct queries. At a cache size of 400kB, the results of all metrics are similar because most of the aggregates fit into the cache. The cache metric has hardly any influence on the cache performance.

When the cache size is smaller than 400kB, the performance of all metrics decreases. However, the decrease significantly differs between the metrics. The LRU metric shows a performance decrease of 11 seconds or 100% in case the cache size decreases from 500kB to 200kB. In comparison, the performance decrease of the AC-ETD and the AC-TAD metrics is only five seconds or 45%. That is less than half the performance decrease. The WATCHMAN, DynaMat, and AC-ETR metric have a decrease of seven seconds or 63%. The AC-TAR metric has a decrease of 81%.

C. *Eviction Threshold*

The results in Figure 4 show the impact of the eviction threshold on the profit metrics. When the eviction threshold is 0%, all aggregates are evicted from the cache once the maximum cache size of 200kB is reached. Then, the profit metrics have no influence on the cache performance so that all of them show the same performance. In general, the bigger the threshold gets, the better all of the profit metrics perform, except from the AC-TAR metric.

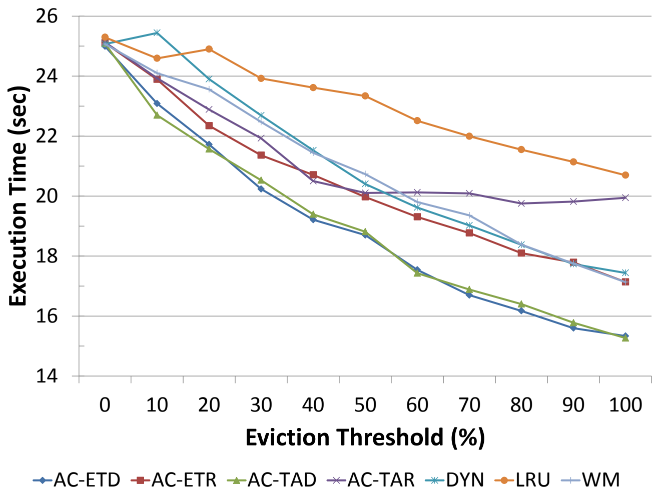

Figure 4. Comparison of Profit Metrics for Different Eviction Thresholds. (Delta Tuple Count 200,000, Cache Size 200kB)

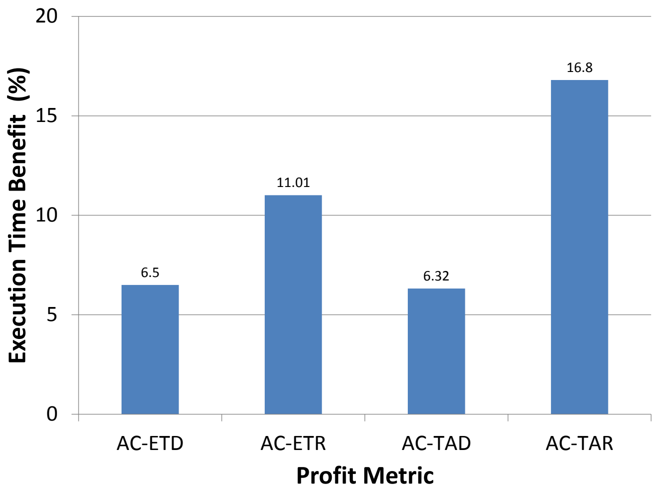

Figure 5. Impact of the Invalidation Compensation on the Profit Metrics. (Delta Tuple Count 200,000, Cache Size 200kB, Eviction Threshold 80%)

The AC-TAR metric does not improve the cache performance, when the threshold gets bigger than 50%. That has two reasons. On the one hand, the metric tends to favor lightweight aggregates. On the other hand, costly aggregates can run multiple times between two consecutive runs of the cache trimming procedure. The bigger the intervals, the more likely is it, that the queries reoccur. Their aggregates get cached on the first occurrence. On the following occurrences, the queries are answered from the cache.

The AC-ETD and the AC-TAD metrics show the best results for an eviction threshold between 10% and 100%. They are up two seconds faster than any of the other metrics. The AC-ETR, WATCHMAN, and DynaMat metrics show similar results when the threshold is bigger than 50%. In case the threshold is smaller, the AC-ETR metric is up to one second faster than the WATCHMAN and DynaMat metrics. The LRU metric shows the lowest performance. The time difference to the AC-ETD and the AC-TAD metrics grows from at least one second at an eviction threshold of 10% to more than five seconds at an eviction threshold of 100%.

D. Invalidation Compensation

The fourth experiment describes the impact of the invalidation compensation for deleted tuples in the main storage. The results are presented in Figure 5. We cache all queries, before we invalidate one million tuples in the main storage. After the invalidation, we reduce the cache size to 200kB and execute the analytical workload with the novel aggregate cache profit metrics. Once the metrics have the invalidation compensation factor $icomp(q)$; once they do not have it. The results in Figure 5 show that the system's performance increases by up to almost 17% when the metrics with the invalidation compensation factor are applied. The percentage increase is higher for the AC-TAR metric and the AC-ETR metric compared to the AC-ETD metric and the AC-TAD metric because the overall performance of the AC-TAR and the AC-ETR metrics is not as good as the one of the AC-ETD and AC-TAD metrics.

E. Cache Revalidation During Merge Process

In the last experiment, we analyze the performance impact of the incremental aggregate revalidation during the merge

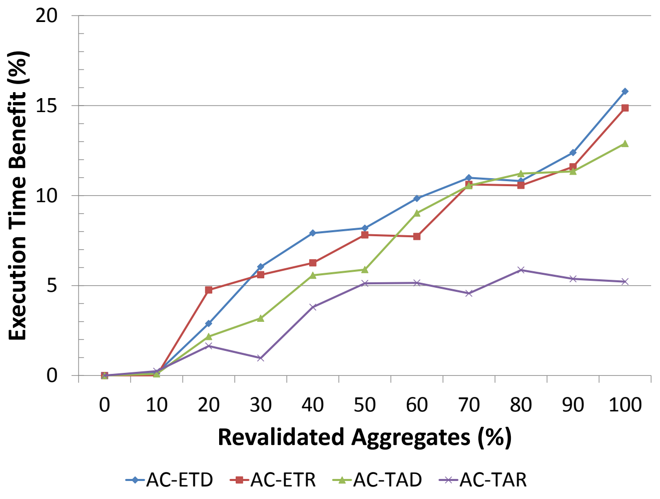

Figure 6. Impact of the Incremental Aggregate Revalidation during the Merge Process dependent from the Profit Metrics.
(Delta Tuple Count 200,000, Cache Size 200kB, Eviction Threshold 80%)

process. In an operational system, where more than 100 distinct queries are executed, the revalidation of all affected aggregates can take more time than the merge process. Since, in our scenario, the merge process takes more time than the revalidation of the 100 queries, we manually limit the revalidation of the aggregates to a certain percentage of all cached aggregates. When half of the workload is processed, we trigger the merge process.

The results in Figure 6 show that the incremental revalidation yields a performance benefit of up to 15% when all aggregates are revalidated. The benefit increases with the percentage of incrementally revalidated aggregates. When the AC-TAR metric is applied, the benefit stagnates at around 5%, once more than 50% of the cached aggregates are revalidated. That has the following reason: In case no aggregates are revalidated, the cache is empty after the merge. Then, many expensive aggregates are cached and repeatedly accessed before the first cache trimming after the merge is executed. That reduces the execution time. In case all aggregates are revalidated, the lightweight aggregates that the AC-TAR metric assigns a high profit permanently content the cache.

VIII. Conclusion

We have introduced novel profit metrics and cache management algorithms tailored to the special main-delta architecture of modern IMDBs. In the evaluation section, we showed that the novel profit metrics yield up to 10% better results than existing metrics. The experiments on SanssoucciDB indicate that especially the AC-ETD and the AC-TAD metrics outperform all existing profit metrics.

We also showed that it is important to consider the invalidated tuples in the main storage for the profit metrics. In our experiments, the metrics with invalidation compensation factor perform 6% to 16% better than the same metrics without invalidation compensation factor.

The evaluation also indicates that the system's performance increase when the profitable aggregates are incrementally revalidated instead of invalidated during the merge process. The performance increases up to 15% when all cached aggregates are incrementally updated.

So far our evaluation was based on a single table. In the future, we want extend the evaluation to scenarios with multiple tables and even more complex queries including joins. We also want to evaluate the impact of different incremental update strategies with our cache. The cache management system can update the cached aggregates not only during merge phase, but whenever the aggregate is touched and an on-the-fly aggregation on the delta is performed.

References

[1] H. Plattner, "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database," in Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2009, pp. 1–2.

[2] ——, "SanssouciDB: An In-Memory Database for Processing Enterprise Workloads Architecture of SanssouciDB," in Datenbanksysteme für Business, Technologie und Web (BTW), 2011, pp. 2–21.

[3] S. Müller and H. Plattner, "Aggregates Caching in Columnar In-Memory Databases," in Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM), 2013, pp. 58–69.

[4] S. Müller, L. Butzmann, K. Howelmeyer, S. Klauck, and H. Plattner, "Efficient View Maintenance for Enterprise Applications in Columnar In-Memory Databases," in Proceedings of the IEEE International Enterprise Distributed Object Computing Conference (EDOC), 2013, pp. 249–258.

[5] F. Hübner, J.-H. Böse, J. Krüger, C. Tosun, A. Zeier, and H. Plattner, "A cost-aware strategy for merging differential stores in column-oriented in-memory DBMS," in Proceedings of the Workshop on Business Intelligence for the Real Time Enterprise (BIRTE), 2011, pp. 38–52.

[6] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Manager Intelligent Cache," in Proceedings of the International Conference on Very Large Data Bases (VLDB), 1996, pp. 51–62.

[7] Y. Kotidis and N. Roussopoulos, "DynaMat: A Dynamic View Management System for Data Warehouses," ACM SIGMOD Record, vol. 28, no. 2, 1999, pp. 371–382.

[8] C.-S. Park, M. H. Kim, and Y.-J. Lee, "Usability-based caching of query results in OLAP systems," Journal of Systems and Software, vol. 68, no. 2, 2003, pp. 103–119.

[9] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, K.-U. Sattler, and F. Waas, "The Mixed Workload CH-benCHmark," in Proceedings of the International Workshop on Testing Database Systems (DBTest), 2011, pp. 8:1–8:6.

[10] S. Maffeis, "Cache management algorithms for flexible filesystems," ACM SIGMETRICS Performance Evaluation Review, vol. 21, no. 2255, 1993, pp. 16–25.

[11] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in Proceedings of the International Conference on Very Large Data Bases (VLDB), 1985, pp. 127–141.

[12] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," ACM SIGMOD Record, vol. 22, no. 2, 1993, pp. 297–306.

[13] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in Proceedings of the International Conference on Very Large Databases (VLDB), 1994, pp. 439–450.

[14] Y. Zhou and J. F. Philbin, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in Proceedings of the USENIX Annual Technical Conference (USENIX ATC), 2001, pp. 91–104.

[15] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU (Least Recently / Frequently Used) Replacement Policy : A Spectrum of Block Replacement Policies," IEEE Transactions on Computers, vol. 50, no. Technical Report, 2001, pp. 1352–1361.

[16] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in Proceedings of the International Conference on Very Large Data Bases (VLDB), 1996, pp. 330–341.

[17] J. Shim, P. Scheuermann, and R. Vingralek, "Dynamic Caching of Query Results for Decision Support Systems," in Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), 1999, pp. 254–263.

[18] J. Goldstein and P.-A. k. Larson, "Optimizing queries using materialized views: a practical, scalable solution," ACM SIGMOD Record, vol. 30, no. 2, 2001, pp. 331–342.

[19] H. Plattner, A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases. Springer, 2013.

[20] K. Mehlhorn and P. Sanders, Algorithms and Data Structures: The Basic Toolbox. Springer, 2008.

[21] H. Plattner and A. Zeier, In-Memory Data Management: An Inflection Point for Enterprise Applications. Springer, 2011.

[22] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," ACM SIGMOD Record, vol. 40, no. 4, 2011, pp. 45–51.

[23] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE: a main memory hybrid storage engine," Proceedings of the VLDB Endowment, vol. 4, no. 2, 2010, pp. 105–116.