An Adaptive Aggregate Maintenance Approach for Mixed Workloads in Columnar In-Memory Databases

Stephan Müller **Lars Butzmann** **Stefan Klauck** **Hasso Plattner**

Hasso Plattner Institute
University of Potsdam, Germany
August-Bebel-Str. 88, 14482, Potsdam
Email: {firstname.lastname}@hpi.uni-potsdam.de

Abstract

The mixed database workloads generated by enterprise applications can be categorized into short-running transactional as well as long-running analytical queries with resource-intensive data aggregations. The introduction of materialized views can accelerate the execution of aggregate queries significantly. However, the overhead of materialized view maintenance has to be taken into account and varies mainly depending on the ratio of queries accessing the materialized view to queries altering the base data, which we define as insert ratio. On the basis of our constructed cost models for the identified materialized view maintenance strategies, we can determine the best performing strategy for the currently monitored workload. While a naive switching approach already improves the performance over staying with a single maintenance strategy, we show that an adaptive aggregate maintenance approach with inclusion of the workload history and switching costs can further improve the overall performance of a mixed workload. This behavior is demonstrated with benchmarks in a columnar in-memory database.

1 Introduction

Despite the accustomed association of online transactional processing (OLTP) and online analytical processing (OLAP) with separate applications, a modern enterprise application executes a mixed workload with both – transactional *and* analytical – queries [19, 20]. For example, within the available-to-promise (ATP) application, the OLTP-style queries represent product stock movements whereas the potentially very resource-intensive OLAP-style queries aggregate over the product movements to determine the earliest possible delivery date for requested goods by a customer [24]. To speed up the execution of OLAP-style queries with aggregates, a technique called *materialized views* has been proposed [23]. Throughout this paper, we use the term *materialized aggregate* for a materialized view whose creation query contains aggregations [22]. Accessing tuples of a materialized aggregate is always faster than aggregating on the fly. But the main drawback of introducing materialized views is the process of view maintenance which is necessary to guarantee consistency when the base data is changed [10]. Especially in mixed workload environments, where transactional throughput must be guaranteed, a downtime due to materialized view maintenance is not acceptable.

In-memory databases (IMDB) such as SAP HANA [19], Hyrise [9] or Hyper [13] are able to handle mixed workloads comprised of transactional and analytical queries on a single system. In contrast to traditional databases, their storage is separated into a read-optimized main storage and a write-optimized delta storage. Since the main storage is highly-compressed and not optimized for inserts, all data changes of a table are propagated to the delta storage to provide high throughput. Periodically, the delta storage is combined with the main storage in a process called *merge operation* [14].

This new storage architecture has implications on existing materialized view maintenance approaches which we have evaluated in our recent work [15]. We showed that IMDBs with a main-delta architecture are well-suited for a novel view maintenance strategy called *merge update* [15, 18]. Because of the main-delta separation, the materialized aggregates do not have to be invalidated when new records are inserted to the delta storage because the materialized aggregates are only based on data from the main storage. To retrieve the consistent, final query result, the newly inserted records of the delta storage are aggregated on the fly and are combined – using a SQL UNION statement – with the materialized aggregate table. While the merge update strategy outperforms other view maintenance strategies for workloads with high insert ratios, it is not the ideal choice for all workloads. Based on this premise, we showed in [17], that switching between materialized view maintenance strategies can increase the overall performance compared to staying with a single strategy. In this paper, we contribute by proposing more advanced switching approaches that include a smoothing of the monitored workload patterns and take switching costs into account. Further, we evaluate how these approaches can be applied to multiple materialized aggregate tables.

Although we assume that our findings can be transferred to a wide range of enterprise applications, we use the available-to-promise (ATP) application as it provides a mixed workload varying between high select ratios (when checking for possible delivery dates) and high insert ratios (stock movements) [24]. In our implementation, ATP relies on a single, denormalized table called *Facts* that contains all stock movements in a warehouse including past and future orders (Table 5a). Every movement consists of an unique transaction identifier, the date, the id of the product being moved, and the amount. The amount is positive if goods are put in the warehouse and negative if goods are removed from the warehouse. The materialized

aggregate based on this table is called *Aggregates* (Table 5b). The aggregate groups the good movements by date and product and sums up the total amount per date and product. The ATP application does not consider physical data updates and uses an insert-only approach. Logical deletes and updates are handled through differential inserts. We further manually define the materialized views and do not address the view selection problem [11] in the scope of this paper. We focused on the *sum* aggregation function as this is the dominant aggregate function for the TPC-H benchmark [1].

The remainder of the paper is structured as follows: Section 2 gives a brief overview of related work. Section 3 explains view maintenance strategies in detail and describes workload patterns that motivate our research about switching maintenance strategies. Section 4 outlines algorithms for maintenance strategy switching before Section 5 benchmarks these and discusses the results. Section 6 provides an outlook on future work and concludes the paper with our main findings.

2 Related Work

Gupta gives a good overview of materialized views and related issues in [10]. Especially, the problem of materialized view maintenance has received significant attention in academia [6, 4]. Database vendors have also investigated this problem thouroughly [3, 25] but besides our earlier work [15], there is no work that evaluates materialized view maintenance strategies for mixed workloads. Instead, most of the existing research is focused on data warehousing environments [26, 1, 12, 16] where maintenance downtimes may be acceptable. Consequently, available DBMS only provide static view maintenance and support basic view maintenance strategies.

Chaudhuri et al. highlight in [7] the importance of automated physical database design including index and materialized view selection based on changing workloads. Agrawal et al. extend the definition of a workload by not only considering the ratios of query types within a workload, but also their sequence [2]. However, neither of them do address the problem of materialized view maintenance and how the optimal maintenance strategy can be chosen based on a changing workload.

3 Aggregate Maintenance Strategies

In [15], various aggregate maintenance strategies were presented and evaluated. It was shown that the insert ratio of a workload has the biggest influence on the execution performance. Figure 1 shows the aggregate maintenance and access times of different maintenance strategies for workloads with insert ratios between 0 and 1. For each single workload, the insert ratio was constant to allow comparisons between them. For each insert ratio, either *smart lazy incremental update* (SLIU) or *merge update* (MU) has the lowest workload execution time. SLIU performs best for read intensive workloads, since only seldom writes, which change the aggregate, require maintenance activities. For workloads with increasing writes (more than 40 percent inserts), MU outperforms the other strategies.

However, enterprise workloads are not characterized by constant insert ratios. Workloads change and therefore the best performing maintenance strategy

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

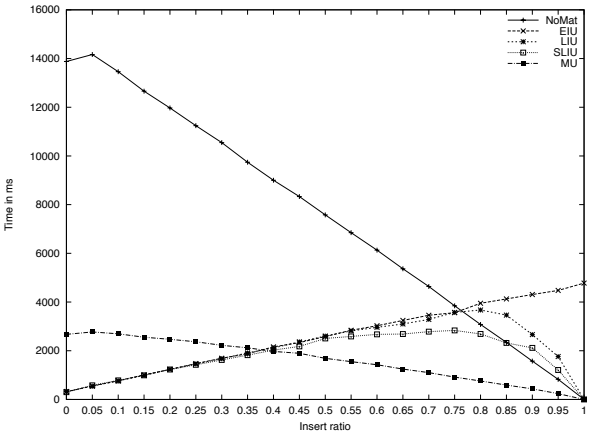


Figure 1: Aggregate maintenance and access time of different maintenance strategies for workloads with different insert ratios.

Table 1: Definition of symbols

Symbol	Definition
N_{total}	Total number of queries
N_{insert}	Number of insert queries
N_{select}	Number of select queries
N_{delta}	Number of records in delta storage
R_{select}	Select ratio
R_{insert}	Insert ratio
T_{select}	Time to select the aggregate
T_{delta}	Time to aggregate the delta storage
$T_{maintenance}$	Time to maintain the aggregate
T_{union}	Time to union two results
T_{dict}	Time to read from the dictionary structure

changes. The remainder of the section starts with a recap of SLIU and MU. The maintenance costs and switching costs are described to motivate the research of switching strategies. A definition of symbols used for the cost function is listed in Table 1. Additionally, patterns for changing workloads are listed.

3.1 Smart Lazy Incremental Update

Using the *smart lazy incremental update (SLIU)* strategy, the maintenance is done when processing selects querying the materialized aggregate. Thereby, the where-clause of the query is evaluated and only aggregates contained in the result set are maintained. Hence, after processing a select, the requested aggregates are up to date. In order to be able to maintain the aggregate during a select, one has to store the changes caused by inserts since the last maintenance point. This is done in a dictionary, called proxy structure, storing the difference between the materialized aggregate and the current correct aggregate for each combination of grouping values.

Table 2 shows the starting point of a SLIU maintenance scenario with a materialized view containing fresh aggregates based on the base table (c.f. Tables 2a, 2b). The corresponding proxy structure, shown in Table 2c, is empty.

Incoming inserts are not immediately included in the materialized aggregate (c.f. Table 3b). The resulting changes for the aggregate are temporarily stored in the proxy structure as shown in Table 3c.

When querying the aggregates table, the maintenance is triggered. Thereby, only requested aggregates are updated using the proxy structure. Af-

Table 2: SLIU maintenance: setup

(a) Snapshot of initial base table

Facts							
Main				Delta			
ID	Date	Prod	Amt	ID	Date	Prod	Amt
1	1/1/2013	1	100				
2	1/1/2013	1	-50				
3	1/1/2013	2	30				
4	1/1/2013	2	60				
5	1/2/2013	1	-10				

(b) Fresh materialized aggregate

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-10
1/1/2013	2	90

(c) Empty proxy structure

Proxy structure of Aggregates		
Key		Value
Date	Prod	SUM(Amt)

Table 3: SLIU maintenance: after three inserts

(a) Snapshot of base table after three inserts

Facts							
Main				Delta			
ID	Date	Prod	Amt	ID	Date	Prod	Amt
1	1/1/2013	1	100				
2	1/1/2013	1	-50				
3	1/1/2013	2	30				
4	1/1/2013	2	60				
5	1/2/2013	1	-10				
				6	1/2/2013	1	20
				7	1/1/2013	3	50
				8	1/1/2013	3	-10

(b) Materialized aggregate

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-10
1/1/2013	2	90

(c) Proxy structure

Proxy structure of Aggregates		
Key		Value
Date	Prod	SUM(Amt)
1/2/2013	1	-10
1/1/2013	3	40

ter selecting the aggregates for product 1, the corresponding aggregates are up-to-date and the proxy struture contains no entries for those grouping values with product 1 (c.f. Tables 4a, 4b).

Equation 1 shows the costs for a single query using SLIU. The first summand describes the costs for read accesses on the materialized aggregate. T_{select} is the average time for a single read of an aggregate. This time is multiplied by the select ratio R_{select} to weight the costs, since they are not required for inserts. The costs to maintain the aggregate are calculated by the second summand. The costs of a single maintenance activity are $T_{dict}+T_{maintenance}$. The number of single maintenance activities increases with an increasing insert ratio R_{insert}, since each insert demands a maintenance activity when the corresponding aggregate is requested. However, with an increasing number of inserts, the maintenance process can be optimized. The calculation of the whole maintenance costs is therefore divided into two scenarios. With an insert ratio R_{insert} smaller than or equal to 0.5, the maintenance costs $R_{insert} * (T_{dict} + T_{maintenance})$ are linear. With an insert ratio greater than 0.5, the average maintenance costs decrease due to two facts. First, the possibility of combining multiple values in the proxy structure with the same grouping attributes. Second, a "bulk" maintenance where all relevant values from the proxy structure are processed together. This improvement is expressed by the optimization function in Equation 2.

$$costs_{SLIU} = R_{select} * T_{select} + optimization(R_{insert}) * R_{insert} * (T_{dict} + T_{maintenance}) \quad (1)$$

$$optimization(x) = \begin{cases} 1 & 0 <= x <= 0.5 \\ 2 - 2x & 0.5 < x <= 1 \end{cases} \quad (2)$$

Algorithm 1 Tear down for smart lazy incremental update strategy

```
1: procedure SLIU_Tear_Down(mat_aggregate)
2:     for all rows row in the proxy_structure of
       mat_aggregate do
3:         ⟨update the value of the mat_aggregate table at
           row.key by row.value⟩
4:     end for
5:     ⟨delete proxy_structure⟩
6: end procedure
```

Setup A proxy structure has to be created to store the temporary changes caused by inserts.

Tear down All records from the proxy structure have to be included into the materialized aggregate. Algorithm 1 explains the required steps in detail.

3.2 Merge Update

The *merge update (MU)* strategy leverages the existence of a delta storage in a columnar IMDB. Using this strategy, the materialized aggregate always consists of the aggregated main storage. Values from the delta storage, which have been inserted after a merge operation, are not included in the materialized aggregate. Instead, when querying the aggregate, the data stored in the delta is aggregated on the fly and combined with the materialized aggregate to represent the fresh aggregate. With each merge operation [14], the values from the delta storage are aggregated and the materialized aggregate table is updated accordingly.

Table 5a shows a table consisting of a main and delta storage. The materialized aggregate (c.f. Table 5b) stores the values of the main storage. When querying the aggregate, the result (c.f. Table 5c) is calculated by combining the materialized aggregate with the on the fly aggregated delta.

Table 4: SLIU maintenance: after querying product 1

(a) Materialized aggregate

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-20
1/1/2013	2	90

(b) Proxy structure

Proxy structure of Aggregates		
Key		Value
Date	Prod	SUM(Amt)
1/1/2013	3	40

Algorithm 2 Tear down for merge update strategy

```
1: procedure MU_Tear_Down(mat_aggregate)
2:     base_table ← ⟨get the base table of mat_aggregate⟩
3:     delta ← ⟨get all rows in delta of base_table⟩
4:     aggr_delta ← ⟨aggregate the rows in delta as per
                    mat_aggregate create statement ⟩
5:     ⟨combine mat_aggregate table with aggr_delta⟩
6: end procedure
```

The merge update strategy only creates costs when requesting an aggregate. However, since it has to access the delta storage, these costs are higher compared to an aggregate access using SLIU and therefore have to be included. Equation 3 shows the costs T_{select} for accessing the aggregate, T_{delta} for aggregating on the delta and the costs to combine both results T_{union}.

$$costs_{MU} = R_{select} * (T_{select} + T_{delta} + T_{union}) \quad (3)$$

Setup After a strategy switch to MU, the materialized aggregate is up to date and therefore includes all records of the main and delta storage. Hence, the values from the delta storage have to be subtracted from the materialized aggregate, so that it only contains aggregated main storage records. Alternatively, a merge can be performed to combine the records of the delta storage and the main storage. In that case, the materialized aggregate stays the same.

Tear down The values from the delta storage have to be included into the materialized aggregate. This is done by aggregating the records of the delta storage and combine it with the materialized aggregate. Algorithm 2 explains the process in detail.

The introduced parameters, e.g. time for a select T_{select} and time to maintain the aggregate $T_{maintenance}$ depend on the underlying hardware. The calibrator introduced in [15] helps to determine these values.

3.3 Workloads

Workloads are characterized by queries differing in type and complexity. As our research focuses on aggregate maintenance, we study workloads containing queries that request or change aggregates. Our models distinguish two kinds of queries: single inserts changing the base table and selects querying single aggregate values. Resulting, the workload can be described by the terms *insert ratio* respectively *select ratio*. The insert ratio R_{insert} specifies the number of insert queries in relation to the total number of queries (Equation 4). Consequently, the select ratio is $1 - R_{insert}$. These ratios change during a workload depending on the business application.

$$R_{insert} = \frac{N_{insert}}{N_{total}} \quad (4)$$

$$R_{select} = 1 - R_{insert} \quad (5)$$

There is no typical workload for enterprises as they have different business applications implying different database schema and queries. We use a randomized workload pattern, called random walk, as general pattern for enterprise workloads. Additionally, more regular patterns like periodic, linear and hard switching changes are employed. In the following, we characterize these patterns with its configuration parameters.

3.3.1 Random Walk

Enterprise workloads differ and cannot be described by a single workload pattern. To match many different scenarios, we use a configurable randomized workload. The insert ratio of the workload randomly increases or decreases after constant time frames. Configuration parameters influence the exact behavior, e.g. how fast the insert ratio changes or how high the probability of consecutive phase with insert ratio increases respectively decreases are. Additionally, upper and lower bounds of the ratio can be set. This way, we can setup highly unpredictable workloads to test our switching strategies.

Table 5: MU: calculation of the fresh aggregate

(a) Snapshot of base table

Facts							
Main				Delta			
ID	Date	Prod	Amt	ID	Date	Prod	Amt
1	1/1/2013	1	100				
2	1/1/2013	1	-50				
3	1/1/2013	2	30				
4	1/1/2013	2	60				
5	1/2/2013	1	-10				
				6	1/2/2013	1	20
				7	1/1/2013	3	50
				8	1/1/2013	3	-10

(b) Materialized aggregate: based on main storage

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-10
1/1/2013	2	90

(c) On the fly calculated fresh aggregate

Result when querying Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	10
1/1/2013	2	90
1/1/2013	3	40

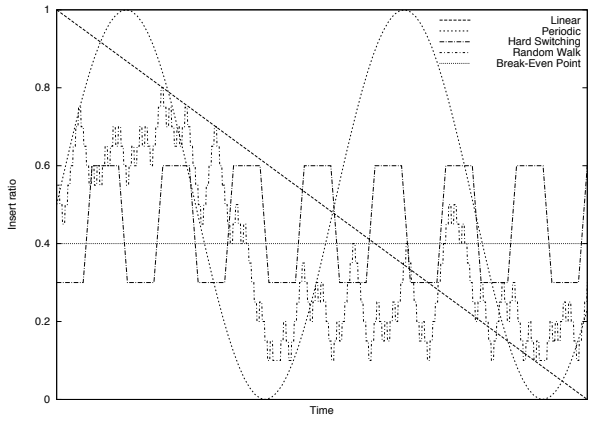


Figure 2: Different workload patterns we used for our evaluation.

3.3.2 Periodic Pattern

The periodic pattern is the first regular workload behavior. The insert ratio behaves like a sinus curve: it periodically increases to the configured maximum and decreases to its minimum. The periodic pattern can be further configured with the length of period and the amplitude. This pattern can match a typical customer-based workload with peaks during the day and lows during the night.

3.3.3 Linear Pattern

The insert ratio increases respectively decreases linearly. It is beside the hard switching pattern a simple changing behavior of the insert ratio. This pattern does not reflect any specific business application. It is rather used to show the benefit of switching the maintenance strategy in a simple scenario. However, linear insert ratio changes are often part of more complicated enterprise workloads.

3.3.4 Hard Switching Pattern

The insert ratio of the hard switching pattern jumps between certain values. The time of a constant ratio value can be configured. This pattern can reflect extreme changes of workloads. Enterprises with hourly batch jobs and businesses with several query peaks per day have workloads matching this pattern.

3.3.5 Examples

Figure 2 shows an example for each workload patterns. The linear pattern has a constantly decreasing insert ratio. The periodic pattern consists of two sinus periods with an amplitude between 0 and 1. The hard switching pattern jumps between 0.3 and 0.6. The random walk starts at 0.5, goes up to 0.8 and stays between 0.1 and 0.5 in the second half. Compared to the other three examples, the insert ratio of the random walk is not smooth. Additionally, the break-even point of the merge update and smart lazy incremental update strategy is included (cf. Section 3).

3.4 Multiple Materialized Aggregates

In [15], we have concentrated our work on a single materialized aggregate. However, enterprise applications typically work with multiple materialized aggregates depending on the current scenario.

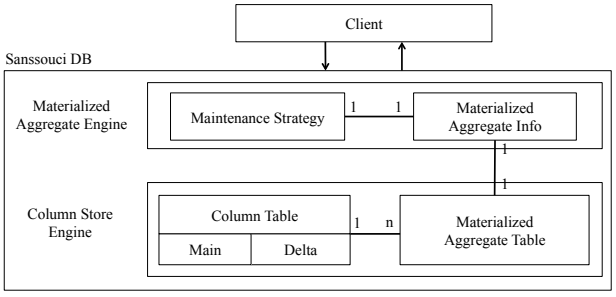


Figure 3: Internal architecture of Sanssouci DB including the novel materialized aggregate engine.

Figure 3 shows the architecture of SanssouciDB [21]. Each column table can have multiple materialized aggregates. Each aggregate has meta information and its own maintenance strategy. This independence is important for our materialized aggregate engine and the switching strategies. As a result, the engine is able to choose the optimal maintenance strategy for each aggregate individually. The required information about the number of accesses on the aggregate and the number of modifications is therefore stored in the meta information.

Figure 4a shows the SQL statements of three materialized aggregates: *Aggregates_Total*, *Aggregates_Q3* and *Aggregates_2013*. They all have the same base table *Facts*. The first aggregate has two grouping attributes and no where clause, meaning that all new inserts affect the aggregate. The second aggregate has a where clause including a date (year 2013) and an amount filter. In the ATP scenario, amounts greater than 0 refer to incoming products. Therefore, approximately half of the inserts affect the aggregate. The third aggregate has, compared to the previous ones, the most restricted where clause. It has filters on the date (third quarter of 2013) and the amount (all outgoing products). Since the third quarter is already over, only a few inserts queries affect the aggregate.

These three aggregates only represent a subset of the aggregates that real enterprises have.

Figure 4b shows one example for an insert query and one example for accessing the aggregate.

4 Maintenance Strategy Switching

As shown in Figure 1, changing insert ratios imply a change of the maintenance performance. Hence, it is preferable to change the maintenance strategy according to workload changes. [17] presented a simple switching algorithm henceforth called naive switching. It was shown that naive switching outperforms static maintenance strategies for workloads with varying insert ratios. However, naive switching with certain configurations can be unfavorable for workloads with insert ratios oscillating around the break-even point. That is why, new switching algorithms are introduced to avoid unnecessary maintenance strategy switches which can decrease the overall performance.

4.1 Naive Switching

Naive switching follows the idea to switch to the best performing strategy as early as possible. Therefore, it monitors the current workload for a configurable number of queries called window. The number of inserts and selects is counted. When the end of a window is reached, the number is divided by the window size to calculate the ratios. Given the insert ratio,

```
CREATE MATERIALIZED VIEW Aggregates_Total AS
SELECT date, product, SUM(amount)
FROM Facts
GROUP BY date, product;

CREATE MATERIALIZED VIEW Aggregates_2013 AS
SELECT date, SUM(amount)
FROM Facts
WHERE date >= 1/1/2013 AND date <= 12/31/2013
    AND amount > 0
GROUP BY date;

CREATE MATERIALIZED VIEW Aggregates_Q3 AS
SELECT product, SUM(amount)
FROM Facts
WHERE date >= 6/1/2013 AND date <= 9/30/2013
    AND amount < 0
GROUP BY product;
```

(a) Three materialized aggregate creation queries.

```
INSERT INTO Facts
(id, date, product, amount)
VALUES (1, 7/30/2013, 1, 100);

SELECT date, product, amount
FROM Aggregates
WHERE product = 1;
```

(b) An insert into the base table and a select on the aggregate.

Figure 4: Example SQL queries of the ATP scenario.

the optimal maintenance strategy can be obtained by evaluating the cost functions of the single strategies. The naive strategy switches to the best performing strategy after each end of a window. The window size thereby controls two things: On the one hand, how long a maintenance strategy stays active until the next switch is possible, namely at least during the next window. On the other hand, how precise the calculated insert ratio is, because the greater the window the more precise the insert ratio.

In the following, the naive switching algorithm is explained in detail using the example workload in Figure 5a. Assume that the workload starts with SLIU as configured maintenance strategy. In the first window, an insert ratio of 0.3 is measured, meaning SLIU was the optimal maintenance strategy for the first window. That is why, SLIU is used for the second window, too. However, the insert ratio of the second window is 0.5 so that MU would have been the better choice. Hence, naive switching changes the maintenance strategy for the third window. For the windows three to six, MU stays the optimal maintenance strategy. Resulting, naive switching keeps MU until the seventh window. At the end of window seven, an insert ratio of 0.3 is measured and the used maintenance strategy is changed back to SLIU. Summarizing, the maintenance strategy is changed twice: after the second and seventh window. The workload was not executed with the best performing strategy during window two and seven.

The naive switching algorithm evaluates only the last window and does not consider the costs for switching to the optimal strategy. Hence, it is not the best switching strategy for specific benchmark scenarios, especially when the optimal aggregate maintenance strategy changes for each window. Figure 5b shows such a workload. Naive switching reacts on each workload change. However, since it takes one window to adjust the maintenance strategy, a non-optimal strategy is used for each window.

4.2 History-Aware Switching

History-aware switching is an extension to the naive switching strategy. It includes not only the insert ratio of the last window to calculate the optimal maintenance strategy, but also the insert ratios of lapsed windows. By including multiple windows, history-aware switching calculates a smoothed insert ratio. In this way, unnecessary switches in the case of strongly varying workloads can be prevented. History-aware switching uses Brown's simple exponential smoothing [5].

$$s_1 = x_0$$
$$s_t = \alpha x_t + (1-\alpha)s_{t-1} \quad 0 < \alpha < 1 \tag{6}$$

Given a starting value x_0 and a smoothing factor α, a smoothed value s_t is calculated based on the previous input x_{t-1} and the previous smoothed value s_{t-1} (Equation 6). The smoothing factor α determines the level of smoothing. The bigger α, the lower the level of smoothing. α can be chosen based on the level of information that is known in advance, e.g. historic data from previous days or weeks.

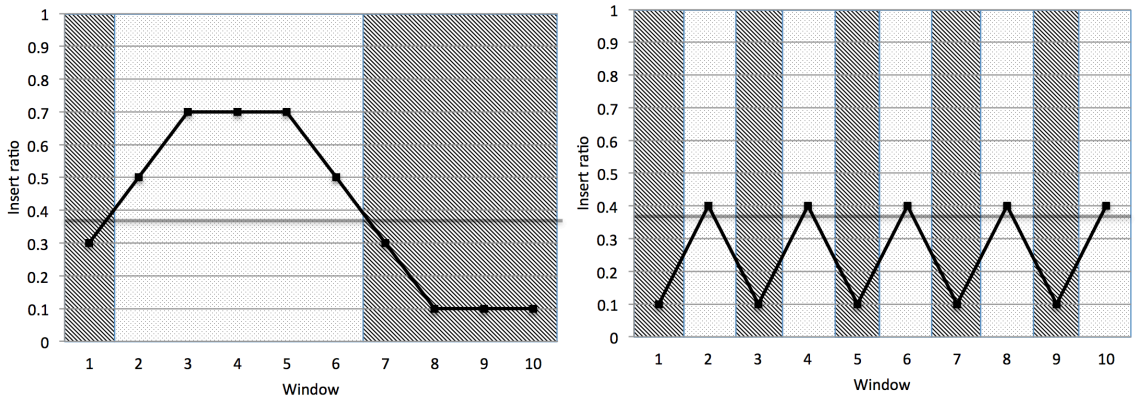
4.3 Cost-Aware Switching

Switching between two maintenance strategies creates tear down and setup costs as explained in Section 3. Neither naive nor history-aware switching consider theses costs. Cost-aware switching takes the costs to switch to another strategy into account to prevent switches, which waste more time for switching than they gain for optimized maintenance performance. Algorithm 3 describes the procedure that determines if a switch is favorable or not. Therefore, the switching costs for each maintenance strategy have to be known. Additionally, a data structure to keep track of savings is required. As long as the current strategy is the fastest strategy, nothing happens. As soon as another strategy is faster than the current strategy, the cost difference between the current strategy and the other strategy is added to the savings data structure. In case the savings are smaller than the switching costs, nothing changes. In the other case when the savings are greater than the switching costs, the system switches to the new strategy and the data structure is reset.

5 Evaluation

To evaluate our presented concepts, we implemented the materialized aggregate engine in SanssouciDB [21] but we believe that they can be applied to other columnar IMDBs with a main-delta architecture such as SAP HANA [8]. Figure 3 illustrates the architecture of our implementation. The column store engine represents the persistence layer and contains the column table with its main and delta storage. The novel materialized aggregate engine is on top and is responsible for creating and maintaining materialized aggregates. All logic for the maintenance strategies as well as the switching strategies is included there.

For our benchmarks, we used a data set of an ATP scenario that is based on customer data which we parametrized to generate different workload characteristics and patterns. The base table size for all benchmarks is 1M records. We have chosen this size for faster data imports and because the base table



(a) The insert ratio passes the break-even point twice.

(b) The insert ratio jumps between 0.1 and 0.4.

Figure 5: Workloads with changing insert ratios.

Algorithm 3 Saving cost calculation

```
1:  switching_costs ← ⟨define switching costs for strategies⟩
2:  savings ← ⟨initialize savings for all strategies⟩
3:
4:  procedure CALCULATE_SAVINGS(costs_of_last_interval)
5:      current_strategy ← ⟨get current strategy⟩
6:      fastest_strategy ← ⟨retrieve fastest strategy from
    costs_of_last_interval⟩
7:      savings_per_strategy ← ⟨get current saving⟩
8:      if current_strategy is fastest_strategy then
9:          ⟨reset savings to 0⟩
10:     else
11:         ⟨update savings_per_strategy with the
            cost delta of current_strategy
            and fastest_strategy⟩
12:         savings ← ⟨savings for fastest_strategy
                       from savings_per_strategy⟩
13:         if savings for fastest_strategy are greater than
    switching_costs then
14:             ⟨switch to fastest_strategy⟩
15:             ⟨reset savings to 0⟩
16:         end if
17:     end if
18: end procedure
```

size has no influence on the performance since we use incremental view maintenance strategies (as shown in [15]). The materialized aggregate contains about 4,000 records (i.e. date - product combinations). The workloads consist of two query types: selects querying aggregates filtered by product, and inserts with about 1,000 different date - product combinations. Three queries to create a materialized view, one to insert a value into the base table and one to select the aggregate are shown in Figure 4. Each workload contains 20k queries divided into 200 phases of constant insert ratios. Between consecutive phases, the insert ratio can stay constant or increase respectively decrease by 5 percent. For the history-aware switching strategy, alpha is chosen to be 0.5. This is equivalent to a history of three intervals.

All 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.

The different switching strategies are compared with MU and SLIU, which use the same maintenance strategy all the time.

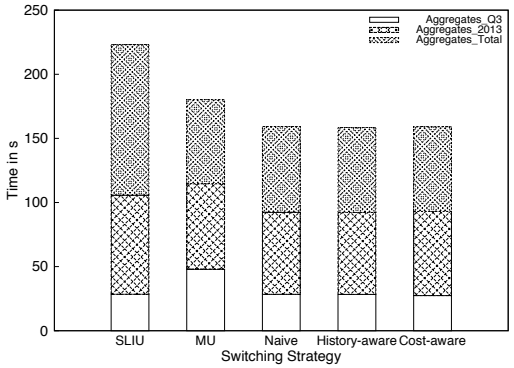


Figure 6: Performance of the switching strategies for multiple views.

5.1 Multiple Views

In Section 3.4, we have explained the need for multiple materialized aggregates in one system and on one table. The benchmark in Figure 6 evaluates the performance of the switching strategies for three different aggregates. The SQL for the aggregates *Aggregates_Total*, *Aggregates_2013* and *Aggregates_Q3* is shown in Figure 4a.

The results in Figure 6 show that individual switching strategies for aggregates perform better than static maintenance strategies. Since each aggregate has its own insert ratio caused by different materialized view definitions, the materialized view engine chooses the optimal strategy for each of them.

5.2 Basic Workload Patterns

The following benchmarks are based on our evaluation in [17], where we investigated the performance of a naive switching strategy for individual aggregates. Figure 7 shows the result of benchmarking the linear, periodic and hard switching workload patterns which were introduced in Section 3.3.

Two things can be observed. First, switching is always faster. All three patterns cover most of the insert ratio interval [0-1] and therefore cross the break-even point (see Figure 2). Consequently, switching is faster compared to the non switching approach. Second, all three switching strategies have nearly the same performance. This is a result of the characteristics of the workload patterns. All patterns are

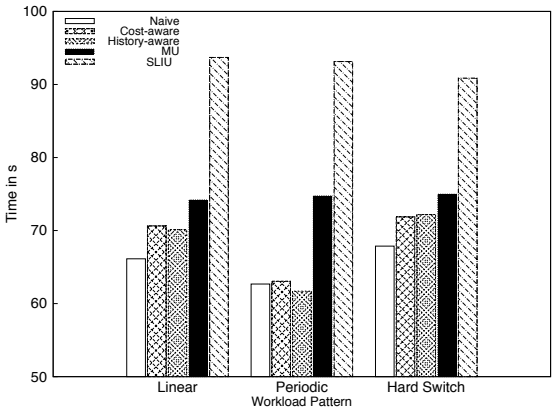


Figure 7: Performance of the switching strategies for a linear, periodic and hard switching pattern.

relatively simple and have a smooth behavior.

This benchmark only shows one example for each pattern. The characteristics of the three patterns can be varied, e.g. the amplitude of the periodic pattern can be smaller or the difference between the two values for the hard switching pattern can be larger. The more a workload stays on both sides of the break-even point, the greater is the advantage for the switching strategies.

5.3 Random Workloads

To measure the performance of the switching strategies for unpredictable workloads, we used random walks (see Section 3.3.1). Thereby, we varied the interval of possible insert ratios in the way it should influence the advantage of switching strategies:

1. [0, 1] covers the largest possible interval. Switching in this setup should bring the most.
2. [0.2, 0.6] covers the area close to the break-even point. The benefit of switching is expected to be lower.
3. [0.3, 0.8] covers the interval beneficial for MU. The lower boundary crosses the break-even point slightly.
4. [0, 0.5] covers the interval beneficial for SLIU. The upper boundary crosses the break-even point slightly.

Figure 9 includes benchmarks of the four intervals with three workloads each. Figure 9a shows the performance for workloads with the largest possible insert ratio interval ranging from 0 to 1. Switching is 27 percent faster than the fastest non switching strategy. Among the switching strategies, naive and cost-aware perform best. History-aware is slightly slower because of a deferred switching point.

The workloads, whose benchmark results are presented in Figure 9b, have an insert ratio interval of [0.2, 0.6] (i.e. close to the break-even point). As a result, the performance advantage of switching strategies is smaller. The average improvement is approximately 18 percent. The cost-aware and naive switching strategy perform nearly the same.

In Figure 9c, the benchmark results for select-intensive workloads are presented. SLIU outperforms MU. However, its performance is beaten by the switching strategies. During the short period with insert ratios higher than 40 percent, switching strategies

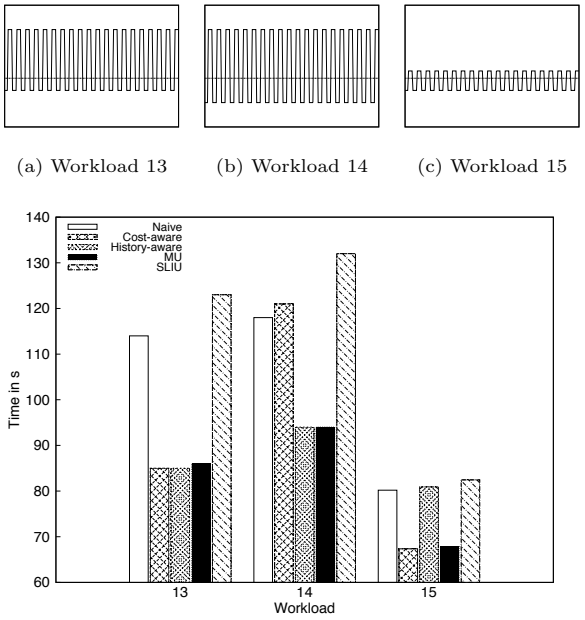


Figure 10: Three switching cost intensive workloads which cross the break-even point after each window.

change the maintenance strategy to the advantageous MU. The improvement of switching is 10 percent.

Workloads with insert ratios ranging from 0.3 to 0.8 are good for MU (Figure 9c). Again, switching has a slightly better execution time than MU (5 percent), since the workloads contain phases (with insert ratios smaller than 40 percent) where SLIU is the better maintenance strategy.

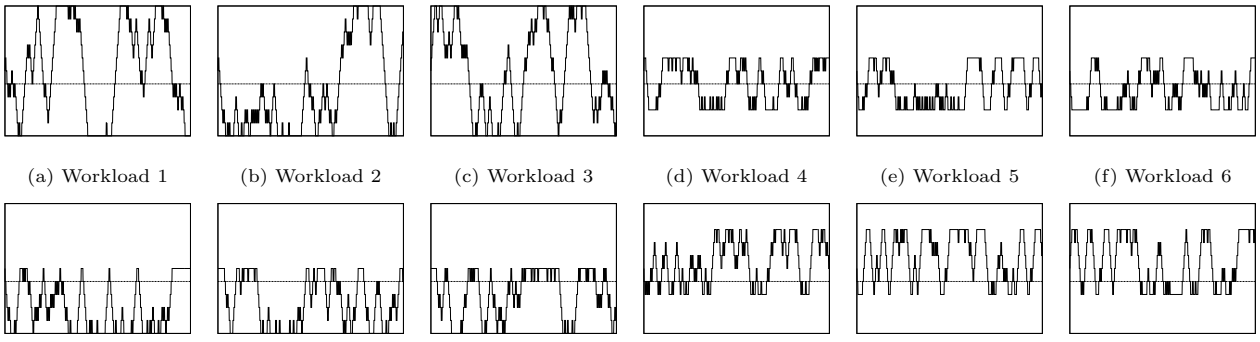
5.4 Worst Case Analysis

Even though naive switching is often the best approach, we have analyzed the behaviour of the switching strategies in extreme cases. As assumed, we measured that naive switching is not optimal for all workloads.

The workloads 13 to 15 in Figure 10 have an alternating pattern and jump between different insert ratios (0.3/0.8, 0.2/0.8 and 0.3/0.46). In Figure 10d, naive switching is significantly slower than the other two approaches. This is a result of unfavorable switching points due to the short interval for each insert ratio. The cost-aware switching decides not to switch based on the calculated savings (workload 13 and 15). However, in workload 14, the saved costs per interval are bigger than the switching costs and therefore the strategy is switched as in the naive approach. Only the history-aware strategy detects the fluctuating pattern and stays with one maintenance strategy, because the smoothed insert ratio only crosses the break-even point in the beginning and then evens out.

6 Conclusion

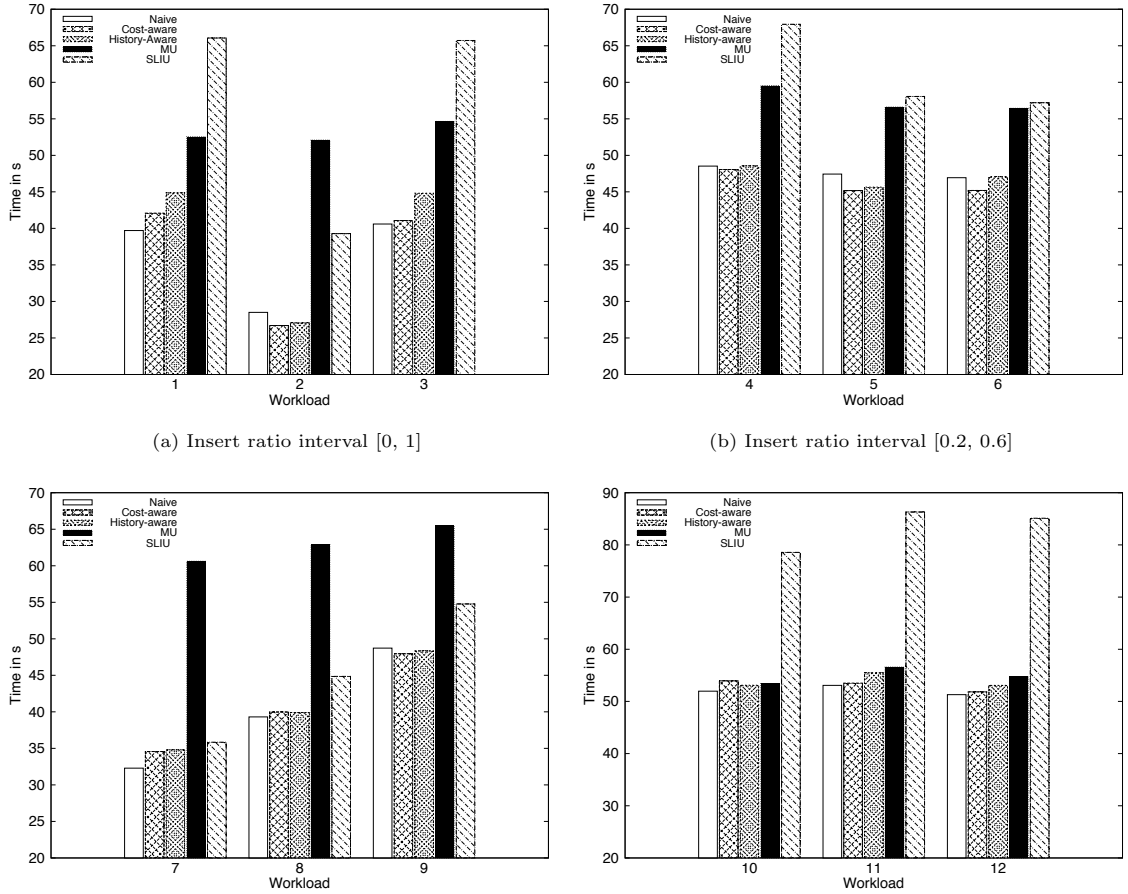
This paper introduces advanced algorithms to identify and switch to the optimal aggregate maintenance strategy. It has been shown that the performance of maintenance strategies depends on workload characteristics such as the insert ratio. According to workload changes, it is desirable to switch to the best performing aggregate maintenance strategy. Naive



(a) Workload 1 (b) Workload 2 (c) Workload 3 (d) Workload 4 (e) Workload 5 (f) Workload 6

(g) Workload 7 (h) Workload 8 (i) Workload 9 (j) Workload 10 (k) Workload 11 (l) Workload 12

Figure 8: A visualization of the insert ratios for the workloads benchmarked in Figure 9.



(a) Insert ratio interval [0, 1]

(b) Insert ratio interval [0.2, 0.6]

(c) Insert ratio interval [0, 0.5]

(d) Insert ratio interval [0.3, 0.8]

Figure 9: Benchmarks with different insert ratio intervals. Each workload consists of 20k queries.

switching uses a simple algorithm to select the maintenance strategy by following the goal to switch to the best performing strategy as early as possible. However, switching between strategies causes costs and the best performing strategy can often change in a short period of time. The two introduced switching algorithms tackle this issue and reduce the number of unnecessary maintenance strategy switches. To achieve that, they include the history of insert ratios and the costs of strategy switches.

We implemented the introduced switching strategies as part of a materialized aggregate engine in SanssouciDB. The materialized aggregate engine monitors the current workload, evaluates the cost functions and is able to switch to the optimal maintenance strategy. We benchmarked the various aggregate maintenance switching algorithms for workloads with different insert ratio courses. The results reveal that switching between maintenance strategies is beneficial for all identified workloads as it decreases the overall execution time. We evaluated three different switching strategies that reduce the execution time up to 27 percent. Among the different switching strategies, the naive switching strategy performs well. However, for certain workload patterns, the more sophisticated cost-aware and history-aware switching strategies are more beneficial.

As a direction of future work, we plan to employ a machine learning approach that predicts future workload changes and adjusts the materialized view maintenance strategy proactively.

References

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

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: Workload as a Sequence. In *SIGMOD*, pages 683–694, New York, New York, USA, 2006. ACM Press.

[3] 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*, pages 659–664, 1998.

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

[5] R. Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Dover Phoenix editions. Dover Publications, 2004.

[6] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 1979.

[7] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, 2007.

[8] 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.

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

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

[11] H. Gupta. Selection of views to materialize in a data warehouse. *ICDT*, 1997.

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

[13] A. Kemper, T. Neumann, F. F. Informatik, T. U. München, and D-Garching. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[14] 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.

[15] S. Müller, L. Butzmann, K. Höwelmeyer, S. Klauck, and H. Plattner. Efficient View Maintenance for Enterprise Applications in Columnar In-Memory Databases. *EDOC*, 2013.

[16] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, 1997.

[17] S. Müller, L. Butzmann, S. Klauck, and H. Plattner. Workload-aware aggregate maintenance in columnar in-memory databases. In *BPOE, in conjunction with IEEE International Conference on Big Data*, 2013.

[18] S. Müller and H. Plattner. Aggregates caching in columnar in-memory databases. In *IMDM, in conjunction with VLDB*, 2013.

[19] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[20] H. Plattner. Sanssoucidb: An in-memory database for processing enterprise workloads. In *BTW*, 2011.

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

[22] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation. *Commun. ACM 1977.*

[23] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *VLDB*, 1996.

[24] 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*, pages 667–686, 2011.

[25] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

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