Workload-Aware Aggregate Maintenance in Columnar In-Memory Databases

Stephan Müller, Lars Butzmann, Stefan Klauck, Hasso Plattner
Hasso Plattner Institute
University of Potsdam, Germany
{*firstname.lastname*}@*hpi.uni-potsdam.de*

***Abstract*—Database workloads generated by enterprise applications are comprised of short-running transactional as well as long-running analytical queries with resource-intensive aggregations. The expensive aggregate queries can be significantly accelerated by using materialized views. This speed-up, however, comes with the cost of materialized view maintenance which is necessary to guarantee consistency when the underlying data changes.**

While several view maintenance strategies are applicable in the context of an in-memory column store, their performance depends on various factors, most importantly the ratio between queries accessing the materialized view and queries altering the base data, called insert ratio. As a contribution in this paper, we propose algorithms that determine the best-performing view maintenance strategy based on the currently monitored factors. Using our novel materialized aggregate engine, we are able to switch between view maintenance strategies on demand. We have created cost models for the identified view maintenance strategies that determine at which insert ratio it is advisable to switch to another strategy. Our benchmarks in SanssouciDB reveal that for all identified workloads, switching between maintenance strategies is more beneficial than staying with a single strategy.

I. Introduction

For a long time, transactional and analytical queries have been associated with separate applications. However, recent research shows that the distinction between online transactional processing (OLTP) and online analytical processing (OLAP) is no longer applicable in the context of modern enterprise applications as they execute both – transactional *and* analytical – queries [16], [17]. For example, within the available-to-promise (ATP) application, the OLTP-style queries represent product stock movements whereas the OLAP-style queries aggregate over the product movements to determine the earliest possible delivery date for requested goods by a customer [21]. Similarly, in financial accounting, every financial accounting document is created with OLTP-style queries, while a profit and loss statement needs to aggregate over all relevant documents with OLAP-style queries that are potentially very expensive [16].

To speed up the execution of long-running queries, a technique called *materialized views* has been proposed [20]. A materialized view can be defined as database view – a derived relation defined in terms of base relations – whose tuples are persisted in the database. Throughout this paper, we use the term *materialized aggregate* for a materialized view whose creation query contains aggregations [19]. Access to tuples of a materialized view is always faster than computing the view on the fly. However, whenever the base data is modified, these changes have to be propagated to the corresponding materialized view to ensure consistency. This process, known as *materialized view maintenance*, is well established in academia [4], [9], [1] and industry [3], [22] but with focus on data warehousing [1], [23], [11] and traditional database architectures. While for purely analytical applications, a maintenance downtime may be feasible, this is not the case in a mixed workload environment as transactional throughput must always be guaranteed.

In our recent work [14] we have evaluated view maintenance strategies in a columnar in-memory database (IMDB) that is able to handle transactional as well as analytical workloads on a single system. IMDBs such as SAP HANA [16], Hyrise [8] or Hyper [12] are separated into a read-optimized main storage and a write-optimized delta storage. Since the main storage is compressed, 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 [13]. This process is called merge operation. In contrast, row-oriented database management systems do not have this separation.

It turns out that IMDBs with a main-delta architecture are ideally suited for a novel view maintenance strategy called *merge update* [14]. Because of the main-delta separation, we do not have to invalidate materialized aggregates when new data is inserted to the delta storage. Instead, the materialized aggregate table only contains the data from the main storage. To retrieve the final query result, the newly added records in the delta storage are aggregated on the fly and are combined with the materialized aggregate table. Therefore, this novel materialized view maintenance strategy is more efficient in terms of maintenance costs.

While the merge update strategy outperforms other view maintenance strategies for workloads with high insert ratios, it is not the ideal choice for the full range of insert ratios. Based on this premise, the goal of this paper is to propose and evaluate an adaptive, workload-aware materialized aggregate engine that chooses the optimal view maintenance strategy based on the current workload characteristics. In this context, we introduce a strategy that observes a workload and decide when to switch. This does not only reduce the overall execution time for a specific workload, it can reduce the CPU load of the database and therefore increase energy efficiency or even postpone a scale-out.

Although we assume that our findings can be transferred to a wide range of enterprise applications, we have chosen a simplified scenario from the ATP application for the evaluation of our view maintenance switching strategies as it provides a mixed workload varying between high select ratios (when checking for possible delivery dates) and high insert ratios (stock movements) [21]. In our implementation, ATP relies on a single, denormalized database table called *Facts* that contains all stock movements in a warehouse (Table IIa). 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 IIb). 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 [10] in the scope of this paper. We focused on the *sum* aggregation function as this is the dominant aggregate function in our introduced application.

The remainder of the paper is structured as follows: Section II gives a brief overview of related work. Section III describes workload patterns that are the basis for our analysis. Section IV explains view maintenance strategies before Section V outlines an algorithm for switching between these. In Section VI we discuss the results of our benchmarks. Section VII provides outlook on future work and concludes the paper with our main findings.

II. Related Work

Gupta gives an extensive overview of materialized views and related issues in [9]. Especially, the problem of materialized view maintenance has received significant attention in academia [5],[4]. Database vendors have also investigated this problem thouroughly [3],[22] but besides our earlier work [14], there is no work that evaluates materialized view maintenance strategies in columnar in-memory databases with mixed workloads. Instead, most of the existing research is focused on data warehousing environments [23],[1],[11],[15] where maintenance down times may be acceptable.

Chaudhuri et al. highlight in [6] 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.

Table I: 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

III. Workloads

Quoting the Oxford dictionary, a workload is the amount of work to be done by someone or something [1]. As databases have to process queries, their workload is characterized by its queries. These queries differ in type and complexity.

As mentioned earlier, our research focuses on aggregate maintenance. Hence, we are interested in workloads containing requests and changes of 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 specifies the number of insert queries in relation to the total number of queries (Equation 1). Consequential, the select ratio is $1 - R_{insert}$. These ratios change during a workload depending on the underlying business processes.

$$R_{insert} = \frac{N_{insert}}{N_{total}} \tag{1}$$

$$R_{select} = 1 - R_{insert} \tag{2}$$

We investigated the performance of aggregate maintenance strategies for three different workload patterns: linear patterns, periodic patterns and hard switching patterns. In addition we researched the behavior under a randomly changing workload to cover all workloads that are not related to the three previous patterns. In the following, these workload patterns are described.

[1]http://oxforddictionaries.com/definition/english/workload

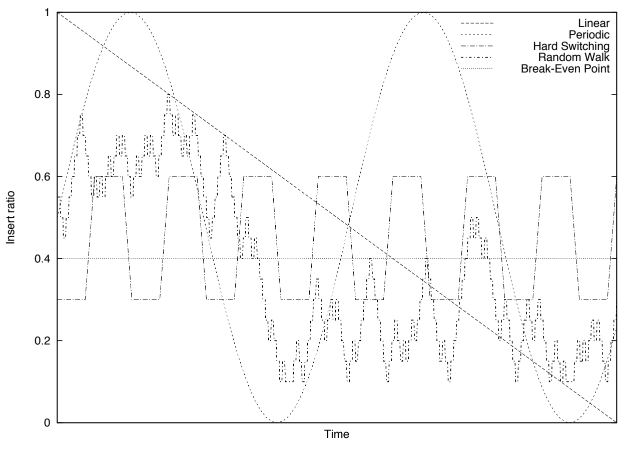


Figure 1: Different patterns we used for our evaluation.

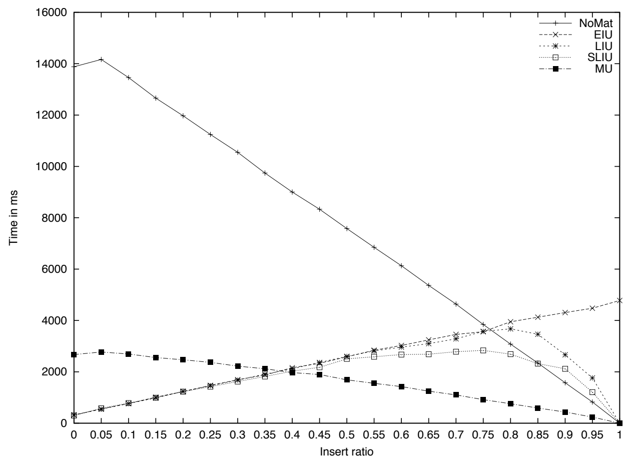


Figure 2: Total execution time of different maintenance strategies for workloads with different insert ratios.

1) Linear Pattern: The insert ratio increases/decreases linearly. We use this rather simple pattern to show the influence of the insert ratio and the benefit when switching the maintenance strategy. This pattern does not reflect any specific business process.

2) Periodic Pattern: The insert ratio of the workload behaves similar to a sinus curve: It periodically increases to a maximum and decreases to a minimum. Parameters as the length of period and the values of maximum/minimum (amplitude) characterize this workload. This pattern represents a typical customer workload of the daily business with peaks during the day and lows during the night.

3) Hard Switching Pattern: The insert ratio of the workload jumps between certain values. The length of the period varies. This pattern reflects enterprises which have extreme changes in their workload due to hourly batch jobs or because their business includes several peaks per day.

4) Random Walk: The random walk helps us to evaluate our switching strategy with extreme and unpredictable workloads. Since workloads of enterprise applications can vary and not all can be described with patterns, we use this kind of workload to cover the rest of scenarios that can occur. The insert ratio randomly increases or decreases after constant time frames. However, parameters allow us to influence the specific workload behaviour, e.g. to increase the probability of consecutive phases with insert ratio increases respectively decreases.

Figure 1 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 IV).

IV. Aggregate Maintenance Strategies

This section describes the algorithms of maintenance strategies and its cost functions. In the area of materialized aggregates, two types of costs are of interest. First, the required time to access the aggregate. Second, the required time to maintain the aggregate. Therefore, our evaluation and cost functions will only focus on these two types.

In [14], different types of aggregate maintenance strategies were evaluated on an in-memory column store. Figure 2, as part of that research, shows the performance of the maintenance strategies for workloads with different insert ratios between 0 and 1. For each single workload, the insert ratio was constant. It can be seen that the performance of a strategy differs for different insert ratios. In our work, we focus on the best performing strategies, *smart lazy incremental update* (SLIU) and *merge update* (MU). SLIU is the best aggregate maintenance strategy for read intensive workloads (less than 40 percent inserts). In contrast, MU outperforms SLIU for workloads with higher insert ratios. We call the workload characteristic, where the best performing maintenance strategy changes, the break-even point.

In the remainder of the section, we present the basic algorithm, the cost function, and the switching costs for SLIU and MU. The cost functions describe the average costs for a single query based on the workload characteristics. Working with the average costs enables us to express the costs of a strategy independently from the workload size (number of queries). Nevertheless, we can obtain the total costs by multiplying the average costs with the number of queries of a workload. The switching costs consist of a setup and tear down phase. The setup phase describes the steps to initialize a strategy whereas the tear down phase describes the steps before a switch.

A. Smart Lazy Incremental Update

Using the *smart lazy incremental update (SLIU)* strategy, the maintenance is done when reading the materialized aggregate. Hence, after processing a select, the requested aggregate is 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 structure. If multiple changes for the same aggregated value exist, they are combined to one value to increase the performance.

Equation 3 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 dictionary structure with the same grouping attributes. Second, a "bulk" maintenance where all relevant values from the dictionary structure are processed together. This improvement is expressed by the optimization function in Equation 4.

$$\begin{aligned} costs_{SLIU} = R_{select} * T_{select} + optimization(R_{insert}) \\ * R_{insert} * (T_{dict} + T_{maintenance}) \end{aligned} \tag{3}$$

$$optimization(x) = \begin{cases} 1 & 0 <= x <= 0.5 \\ 2 - 2x & 0.5 < x <= 1 \end{cases} \tag{4}$$

Setup: A dictionary structure is required to store the inserts that occur between two select queries.

Tear down: The values from the dictionary structure have to be included into the materialized aggregate. Algorithm 1 shows the steps in detail.

Algorithm 1 Smart lazy incremental update strategy

```
1: procedure SLIU_TEAR_DOWN(mat_aggregate)
2:     for all rows row in the dict_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 dict_structure⟩
6: end procedure
```

B. Merge Update

The *merge update (MU)* strategy leverages the existence of a delta storage in a columnar IMDB. Table IIa shows a table consisting of a main and delta storage. Using this strategy, the materialized aggregate is based on the values from the main storage (Table IIb). Values from the delta storage, which have been inserted after a merge operation, are aggregated on the fly and combined with the materialized aggregate to represent the fresh aggregate (Table IIc). With each merge operation [13], the values from the delta storage are aggregated and the materialized aggregate table is updated accordingly.

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

Table II: Materialized aggregate table w/ and w/o delta

(a) Snapshot of base table after inserting three records

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 table w/o delta

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

(c) Materialized aggregate table w/ delta

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

included. Equation 5 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}.

$$costs_{MU} = R_{select} * (T_{select} + T_{delta} + T_{union}) \quad (5)$$

Setup: After switching, the materialized aggregate table contains both, the records of the main and delta. Hence, the values from the delta storage have to be subtracted from the materialized aggregate table, so that it only contains aggregated main storage records. Alternatively, a merge can be performed to transfer the delta records into the main storage.

Tear down: The values from the delta storage have to be included into the materialized aggregate. This is done by aggregating the delta values and using the result to update the materialized main aggregate. Algorithm 2 explains the process in detail.

Algorithm 2 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:     ⟨maintain mat_aggregate table with aggr_delta⟩
6: end procedure
```

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 [14] helps to determine these values.

V. Switching between Aggregate Maintenance Strategies

Workloads show different performances in terms of their total execution time under various strategies. This section introduces a mechanism to determine the current characteristics of a workload and the possibility to switch the strategy according to the determined characteristics. The main idea is to monitor a workload and switch to a different strategy if the current workload would perform better under that strategy. Using the cost models that were introduced in Section IV for SLIU and MU, we show an effective way to determine adequate switching points based on changing workload characteristics.

As described in Section III, the main influence factor is the insert ratio. Since the insert ratio changes over time, we need a way to determine the current insert ratio. One possibility is to track the last n queries. The size of n determines the accuracy of the current insert ratio. An n of 100 provides us with an accuracy of 1 percent. The bigger the value, the more accurate the insert ratio. However, a bigger n means a bigger interval until the next possible strategy switch. Second, the size of the delta storage influence the decision for an optimal strategy. This value can be retrieved from the database.

Using these characteristics, our materialized aggregate engine distinguish between a non-switching and a switching strategy:

A. No Switching

This approach does not switch between different view maintenance strategies and represents our baseline for the benchmarks. Consequently, there are no additional costs related to switching.

B. Switching

This approach switches to the optimal maintenance strategy as soon as possible. Each time the system has determined the current insert ratio, it chooses the optimal maintenance strategy using the cost functions. In the case that the current strategy is not the optimal strategy, the system switches to the optimal strategy.

VI. Benchmarks

We implemented the concepts of the presented view maintenance and switching strategies in SanssouciDB [18] but believe that the implementation in other columnar IMDBs with a main-delta architecture such as SAP HANA [7] will lead to similar results. Figure 3 illustrates the architecture of our implementation.

The data set we used is based on customer data which we parametrized to generate different scenarios and patterns. The base table size for all benchmarks is 1M records. We have chosen this size for faster benchmark setups and because the base table size did not influence the performance since we incrementally maintain the aggregates. 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 example queries are shown in Figure 4. The workloads comprises 20k queries, subdivided into 200 phases of constant insert ratios. Between consecutive phases, the insert ratio can stay constant or increase respectively decrease by 10 percent.

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.

To compare the switching strategy with the no switching approach, the latter will run twice, once using MU and once using SLIU. Therefore, the label will not be ”no switching”, but the name of the statically chosen maintenance strategy.

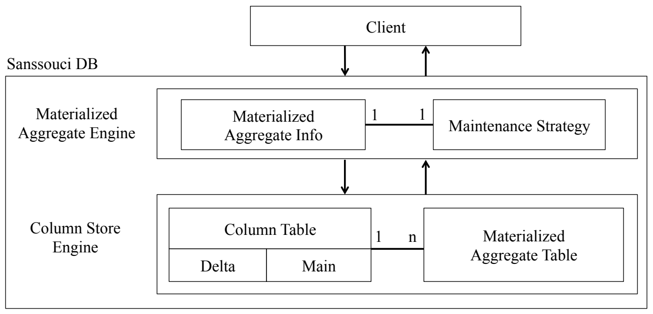

Client
Sanssouci DB
Materialized Aggregate Engine
Materialized Aggregate Info
1
1
Maintenance Strategy
Column Store Engine
Column Table
Delta
Main
1
n
Materialized Aggregate Table

Figure 3: Internal architecture in Sanssouci DB

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

INSERT INTO Facts
(id, date, product, amount)
VALUES (1, 20130101, 1, 100);

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

Figure 4: SQL queries

A. Basic Workload Patterns

Figure 5 shows a benchmark of the three workload patterns introduced in Section III.

All three patterns cover most of the insert ratio interval [0-1] and therefore cross the break-even point (see Figure 1). It can be observed that switching the maintenance strategy is always faster.

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 the advantage for the switching strategy.

In the next benchmark, we take a deeper look into scenarios less beneficial for switching.

B. Random Workloads

To measure the performance of the switching strategies for unpredictable workloads, we used random walks (see Section III-4). Thereby, we varied the interval of possible insert ratios in the way it should influence the benefit of switching strategies:

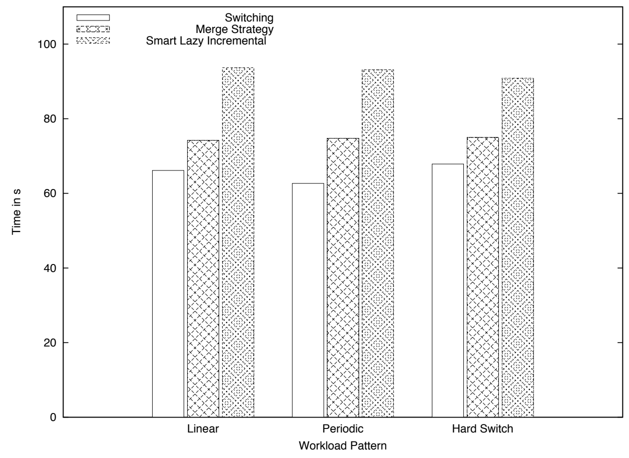


Figure 5: Performance of the switching strategies for a linear, periodic and hard switching pattern.

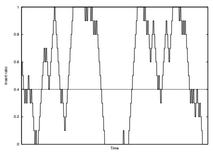
1) [0, 1] covers the largest possible interval. Switching in this setup should bring the most.
2) [0.2, 0.6] covers the area close to the break even point. The benefit of switching is expected to be lower.
3) [0.3, 0.8] covers the interval beneficial for MU. The lower boundary crosses the break-even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

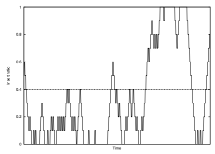
Figure 7 includes benchmarks of the four intervals with three workloads each. Figure 7a shows the performance for workloads with the largest possible insert ratio interval ranging from 0 to 1. Switching is 27 percent faster than the fastest non-switching strategy.

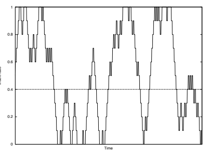
The workloads, whose benchmark results are presented in Figure 7b, have an insert ratio interval of [0.2, 0.6] (i.e. close to the break-even point). As a result, the performance advantage of the switching strategy is smaller. The average improvement is approximately 18 percent.

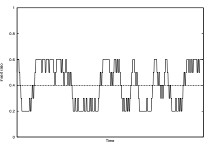
In Figure 7c, the benchmark results for select-intensive workloads are presented. SLIU outperforms MU. However, its performance is beaten by the switching strategy. During the short period with insert ratios higher than 40 percent, the switching strategy changes the maintenance strategy to the advantageous MU. The resulting improvement of switching is 10 percent.

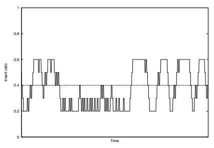
Workloads with insert ratios ranging from 0.3 to 0.8 are good for MU. Figure 7c shows how the strategies perform for such workloads. Again, switching has a slightly better execution time than MU (5 percent), since the workloads contain phases (with insert ratios smaller than 40 percent) where SLIU is the better maintenance strategy.

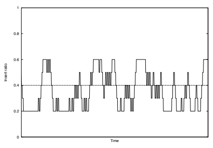

(a) Workload 1

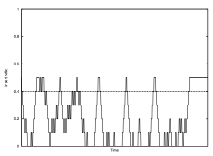

(b) Workload 2

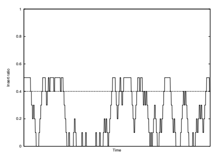

(c) Workload 3

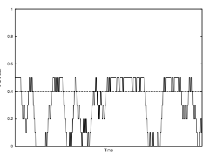

(d) Workload 4

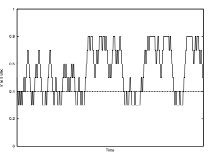

(e) Workload 5

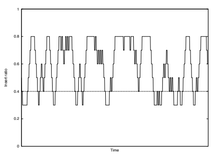

(f) Workload 6


(g) Workload 7


(h) Workload 8


(i) Workload 9


(j) Workload 10


(k) Workload 11

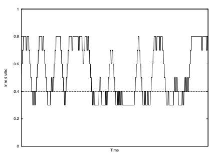
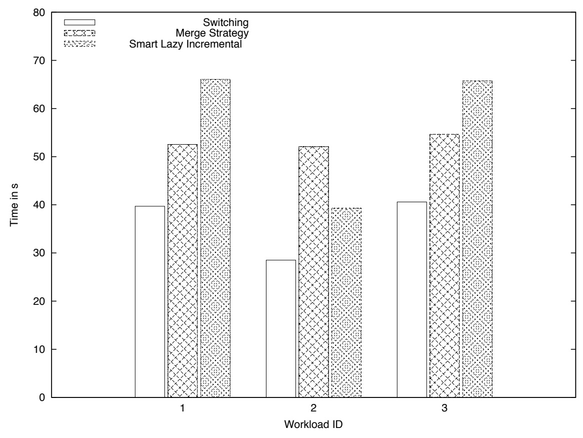
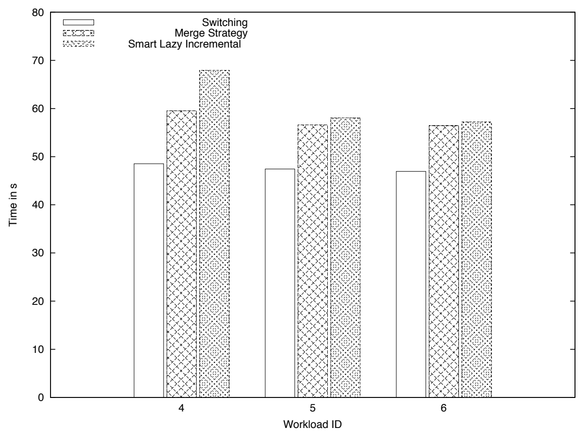

(l) Workload 12

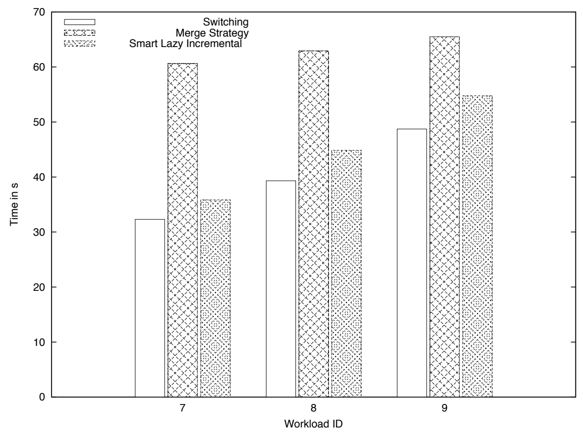
Figure 6: An overview of the workloads for the benchmarks in Figure 7.



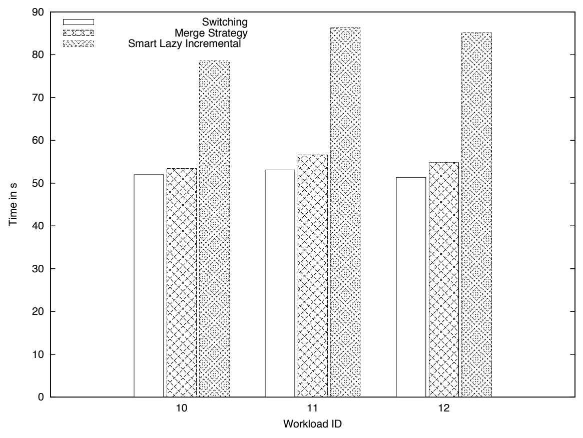
(a) Insert ratio interval [0, 1]



(b) Insert ratio interval [0.2, 0.6]



(c) Insert ratio interval [0, 0.5]



(d) Insert ratio interval [0.3, 0.8]

Figure 7: Benchmarks with different insert ratio intervals. Each workload consists of 20k queries.

VII. Conclusion

In this paper, we have motivated the importance of materialized view maintenance in columnar IMDBs for applications with mixed database workloads. Based on the fact that the optimal materialized view maintenance strategy depends on workload characteristics such as the ratio between reads of the materialized view and inserts to the base table affecting the view, we have proposed an algorithm to select the best suitable materialized view maintenance strategy. The switching algorithm monitors the current workload and evaluates the cost functions to determine the potential of a switch. Our benchmarks reveal that switching between maintenance strategies is beneficial for all identified workloads as it decreases the overall execution time.

As a direction of future work, we plan to extend the simple switching algorithm to take the workload history and the switching costs into account. Also, 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*, 2006.

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

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

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

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

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

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

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

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

[21] 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.

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

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