Aggregates Caching in Columnar In-Memory Databases

Stephan Müller and Hasso Plattner

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

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

1 Introduction

The classic distinction between online transactional processing (OLTP) and online analytical processing (OLAP) is no longer applicable in the context modern enterprise applications [1],[2].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

To speed-up the execution of expensive queries, techniques such as *query caching* and the introduction of *materialized views* has been proposed. Materialized views can be used to answer partial relations of a query. A materialized view is database view – a derived relation defined in terms of base relations – whose tuples are stored in the database. A traditional database query cache stores the results for unique queries, and flushes or updates the cache whenever

the base data changes. Since the result of a database query is a relation itself, a cached query result is equivalent to a materialized view. In this paper, we focus on caching of queries that contain data aggregations [4] and apply techniques used in the context of materialized views.

Direct access to cached query results or tuples of a materialized view is faster than computing the results on the fly even though an on-the-fly aggregation has been sped-up considerably with columnar in-memory databases (IMDBs). However, the inherent problem with caching and materialized views is that whenever the base data is modified, these changes have to be propagated to ensure consistency. While a database query cache can simply flush or invalidate the cache, a process known as *materialized view maintenance*, is well established in academia [5],[6],[7] and industry [8],[9] but with focus on traditional database architectures and data warehousing [10],[11],[12]. For purely analytical applications, a maintenance downtime may be feasible, but this is not the case in a mixed workload environment as transactional throughput must always be guaranteed. Also, the recent trend towards IMDBs that are able to handle transactional as well as analytical workloads on a single system [13],[14],[15] has not been considered.

A columnar database for transactional and analytic workloads has some unique features and preferred modes of operating [1],[2]. To organize the attributes of a table in columns and to encode the attribute values via a dictionary into integers (known as dictionary encoding) [16] has many advantages such as high data compression rates, fast attribute scans so that the attribute columns can be used as indices and traditional index structures (except the primary key) can be omitted in most cases. But this organization comes at a certain price. In transactional workloads we have to cope with high insert rates. A permanent reorganization of the attribute vectors (columns), because new values appear have to be included in the encoding process and complicate the request to keep the attribute dictionaries sorted, would not allow for a decent transactional performance. A way out of this dilemma is to split the attribute vectors of a table into a read-optimized main storage and a write-optimized delta storage. All new inserts are appended to the delta storage with separate unsorted dictionaries. At certain times the attribute vectors are merged with the one in the main storage and a new dictionary (per attribute) is established. Since the main storage is significantly larger than the delta (>100:1) the insert performance becomes acceptable and the analytic performance is still outstanding [17].

The fact that we can handle transactional and analytical workloads in one system has tremendous benefits to the users of the system. Not only the freedom of choice what and how to aggregate data on demand but the instant availability of analytical responses on even large data sets will change how business will be run. Having meaningful information at your fingertips, being able to get answers for complex questions in real time and the option to ask a second or third question will dominate the reorganization of business processes. A consequence of this desirable development will be a significant increase in analytical workload on the combined transactional and analytical systems. When we analyze the query patterns in typical enterprise systems using an in memory database, we can observe that users work frequently on aggregates of business objects (e.g. customer order, received invoices) using a drill down [18]. They start with a high level aggregation to gain an overview in order to pick a certain country, product

or other entity to acquire further details. Once they are satisfied with the details they return to the last level or a higher level of aggregation. To recreate the higher aggregation we have to either re-run the query or the application has to maintain state and store all aggregation levels hierarchically. In larger companies typically several users are doing similar activities throughout a day.

The contribution of this paper is a novel aggregate query caching mechanism that leverages the main-delta architecture of columnar in-memory databases. Because of the separation in main and delta storage, we do not have to invalidate aggregate queries when new records are inserted to the delta storage. Instead, we can use the cached results of the aggregate queries and merge them with the newly added records in the delta storage, a process that is more efficient in many cases, than calculating the complete aggregate again.

The paper is structured as follows: Section 2 discusses related work before outlining the algorithm and architecture of our implementation in Section 3. In Section 4 we evaluate the aggregate caching concept by defining a mixed database workload based on real customer data. Section 5 summarizes our work and gives an outlook for future work.

2 Related Work

The caching of aggregate queries is closely related to the introduction of materialized views to answer queries more efficiently. To be more precisely, a cached query result is a relation itself and can be regarded as a materialized view. Gupta gives a good overview of materialized views and related issues in [6]. Especially, the problem of materialized view maintenance has received significant attention in academia [19],[5],[7]. Database vendors have also investigated this problem thouroughly [8],[9] but to the best of our knowledge, there is no work that evaluates materialized view maintenance strategies in columnar in-memory databases with mixed workloads. Instead, most of the existing research is focused on data warehousing environments [10],[11],[12] where maintenance downtimes may be acceptable.

The summary-delta tables concept to efficiently update materialized views with aggregates comes close to our approach as the algorithm to recalculate the materialized view is based on the old view and the newly inserted, updated, or deleted values [20]. However, their approach updates the materialized views during a maintenance downtime in a warehousing environment and not on demand during query processing time. Further, it does not consider the main-delta architecture and the resulting merge process.

3 Aggregates Caching

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

While aggregation functions can be categorized into distributive, algebraic and holistic functions [21] we limit our implementation to distributive functions,

such as *sum*, *min*, *max* or *count* as they are most commonly found in analytical queries [17]. This could be extended to algebraic functions as they can be computed by combining a constant number of distributive functions, e.g. *avg* = *sum* / *count* [22].

3.1 Architecture and Algorithm

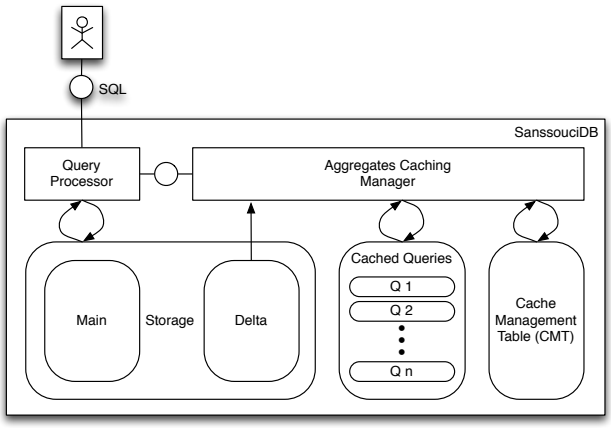

Fig. 1. Aggregates Query Caching Architecture

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

Figure 2 illustrates the described caching algorithm. Every parsed query that contains aggregations, is handled through the aggregates caching manager. To check whether the query exists in the cache already, a hash value of the SQL query is computed and looked up in the CMT. If the cache controller does not find an existing cache entry for the corresponding SQL query, it conveys the query without any changes to the underlying database. After query execution, it is checked whether the query is suitable for being cached by having exceeded certain thresholds defined by the cache admission policy. If this is the case, the query result from the main storage is cached for further reuse. In case, the query is already cached, the query is executed on the delta storage. The result from the delta storage is then combined with cached result and returned to the query processor.

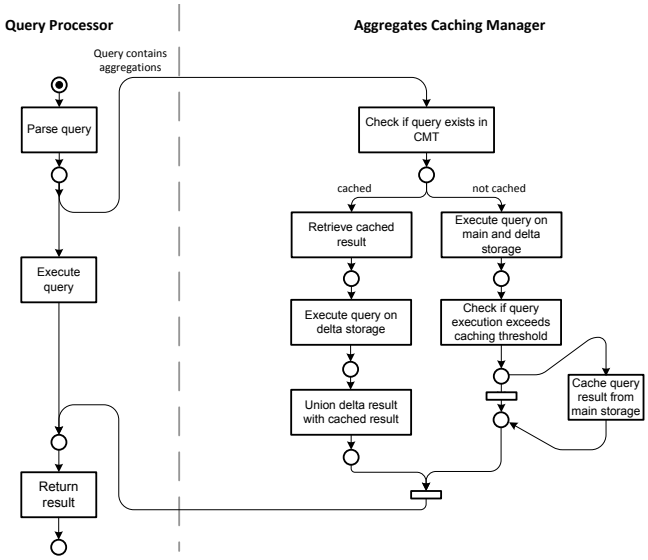

Fig. 2. Simplified Query Caching Algorithm

3.2 Aggregates Maintenance Strategies

When introducing caching of aggregate queries, the overhead of aggregates maintenance has to be considered. The timing of existing materialized view maintenance strategies can be distinguished between *eager* and *lazy*. While eager strategies immediately propagate each change of base tables to the affected materialized views, lazy (or deferred) strategies maintain materialized views not later than the time the materialized view is queried. Independently of the timing, one can divide maintenance strategies into *full* and *incremental* ones. Full strategies maintain the aggregate by complete recalculation using its base tables. Incremental strategies store recent modifications of base tables and explicitly use them to maintain the views. Based on the fact that an incremental calculation of aggregates is always more efficient than a full recalculation [7], we focus on incremental strategies.

The proposed aggregate query caching mechanism does neither maintain the materialized view at insert time nor at query time. Instead, the requested aggregate is calculated by combining the cached aggregate with an on-the-fly aggregation on the delta storage. The maintenance of the cached aggregate is done incrementally during the delta merge process. Since it is possible to predict the query execution time of in-memory databases very accurately [23], we create cost models for each maintenance strategy. The cost is based on a simplified workload model that consists of a number of writes N_w into the base table and a number of reads N_r of the cached aggregate query.

Eager Incremental Update (EIU) Since the cached aggregate query is maintained after each insert, the cost for accessing the aggregate query is just a single

read. The maintenance costs are tied to a write into the base table. As it is an incremental strategy, the costs consist of the read time T_{RA} to retrieve the old value and the write time T_W for the new value into the cached aggregate query-table.

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

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

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

3.3 Optimal Merge Interval

The costs of our proposed aggregates caching mechanism and the involved merge update maintenance strategy mainly depends on the aggregation performance on the delta storage which increases linearly [24]. However, the merge operation also generates costs that have to be considered. In the following, we propose a cost model which takes the costs for the merge operation and the costs for the aggregation on the delta storage into account. Similarly to the cost model for the merge operation introduced by Krüger et al. [18], our model is based on the number of accessed records to determine the optimal merge interval for one base table of a materialized view.

Equation 2 calculates the number of records $Costs_{total}$ that are accessed during the execution of a given workload. A workload consists of a number of reads N_r and a number of writes N_w. The number of merge operations is represented by N_m. The first summand represents the accesses that occur during the merge operations. Firstly, each merge operation has to access all records of the initial main storage $|C_M|$. Secondly, previously merged records and new delta entries are accessed as well [18]. This number depends on the number of writes

N_w in the given workload divided by 2 (since the number of records in the delta increases linearly). The second summand determines the number of accesses for all reads N_r on the delta. As before, the delta grows linearly and is speed up by the number of merge operations N_m.

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

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

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

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

3.4 Admission and Replacement Strategies

Whenever a query with aggregation functions is detected, the aggregates caching manager could potentially cache every query. However, this would lead to an exorbitant growth of the cache and increase the overhead of handling aggregates queries in general. Consequently, we want to limit the overhead by only caching the most profitable queries.

The profit of a query Q_i can be described by the execution time c_i, the result set size s_i and the frequency of execution λ_i. Similarly to [25], the following equation describes the profit formula for Q_i:

$$profit(Q_i) = \frac{\lambda_i \cdot c_i}{s_i} \tag{5}$$

The average frequency of execution λ_i of query Q_i is calculated based on the K_ith last reference and the difference between the current time t and the time of the last reference t_K:

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

The cache manager only caches queries, which increase the overall profit of the cache. Therefore, only queries whose profit is larger than the query with the smallest profit in the cache. To address the space constraint, we could sort the queries by profit and only keep the top-k queries that have an accumulated size smaller or equal to the size of the cache.

When not considering the space constraint and only focusing on minimizing the caching overhead, the definition of the profit for query Q_i can be described with the average execution time λ_i, the execution time for an aggregation on the fly a_i, the time for access to a cached aggregate query qc_i and the average execution time for relevant records in the delta storage d_i.

$$profit(Q_i) = \frac{\lambda_i \cdot a_i}{qc_i + d_i} \tag{7}$$

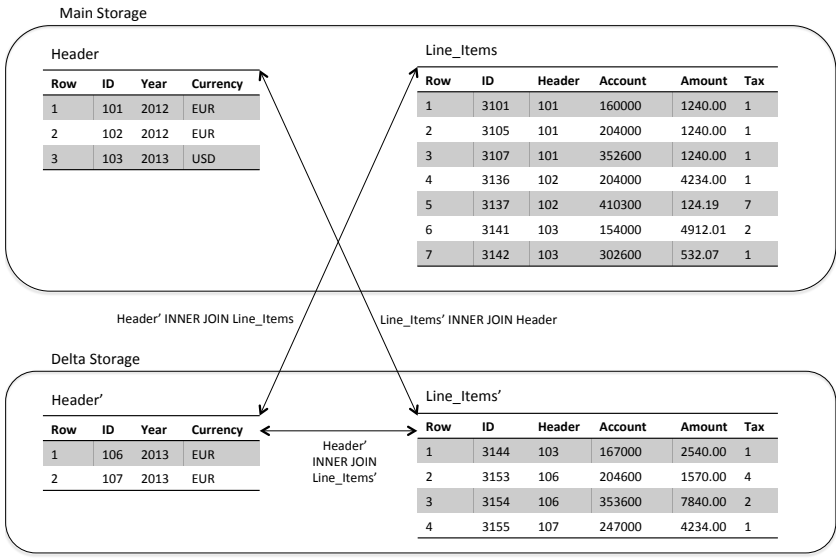

Fig. 3. Aggregate queries with join operations

3.5 Joins

When processing aggregate queries including join operations, the complexity of the proposed caching mechanism and the involved merge update maintenance strategy increases. Instead of combining the cached result with the query result on the delta storage, the join of every permutation has to be computed before these results can be combined. In Figure 3, we have illustrated the involved tables in the main and delta partition of a simple aggregate query including a join of two tables. While the cached query result is based on a join of the `header` and `line_items` table in the main partition, we have to compute the joins of `header'` and `line_items'` tables in the delta partition, and additionally the joins between `header'` and `line_items` as well as `line_items'` and `header`. When the cached aggregate query consist of three or more joined tables, the necessary join operations between delta and main storage increases accordingly. The number of necessary joins operations based on the number of tables t in the aggregate query can be derived as $JoinOps = t^2 - 1$.

After analyzing enterprise workloads, we found out that aggregates for accounting, sales, purchasing, stocks etc. always need a join of the transaction header and the corresponding line items. Interestingly, new business objects such as sale orders or accounting documents are always inserted as a whole, therefore we find the new header and the new line items in the delta. This reduces the number of necessary join operations from three to just one (a join of header and items in the delta). In case a business object can be extended after the initial insert, the header entry could already be merged into the main storage. Conse-

quently, we would need an additional join of the `line_items'` table in the delta with the `header` table in the main.

3.6 Updates

Another potential change are updates. We suggest that the updated tuple is copied to the delta and marked as no longer valid, like in the main) and the new version is inserted as normal to the delta. Having both the old and the new tuple in the delta we can perform the – and + operation with these tuples. If the tuple that is to be updated already in the delta, no copy is needed.

4 Evaluation

We implemented the concepts of the presented aggregates caching mechanism in SanssouciDB [17] but believe that an implementation in other columnar IMDBs with a main-delta architecture such as SAP HANA [13] will lead to similar results. Instead of relying on a mixed workload benchmark such as the CH-benchmark [26], we chose an enterprise application that generates a mixed workload to the database with real customer data. The identified financial accounting application covers OLTP-style inserts for the creation of accounting documents as well as OLAP-style queries to generate reports such as a profit and loss statement. The inserts were taken from the original customer data set covering 330 million records in a denormalized single table. We then extracted 1000 OLAP-style aggregate queries from the application and validated these with domain experts. Mingling both query types according to the creation times (inserts) and typical execution times (aggregate queries) yielded a mixed workload which our evaluations are based upon.

4.1 Aggregates Caching

The strength of a caching mechanism is to answer reoccurring queries. To compare our approach to a standard query cache that gets invalidated whenever the base data changes, we have created a benchmark based on a mixed workload of 10.000 queries with 90% analytical and 10% transactional insert queries. The average execution time on a 40 core server with 4 Intel Xeon E" 7 4870 CPU each having 10 physical cores and 1 TB of main memory when using no cache was 591ms which dropped down to 414ms with a standard query cache. The average execution time of the aggregates cache was at 74ms, outperforming the standard query cache by nearly a factor of six.

With an increasing number of distinct analytical queries, the performance of the proposed aggregates caching mechanisms decreases linearly. With a workload of 100% distinct analytical queries, where no cache reuse takes place, we measured the overhead of the aggregates caching mechanism. Without any cache management, this overhead was at 17% compared to not using any cache at all.

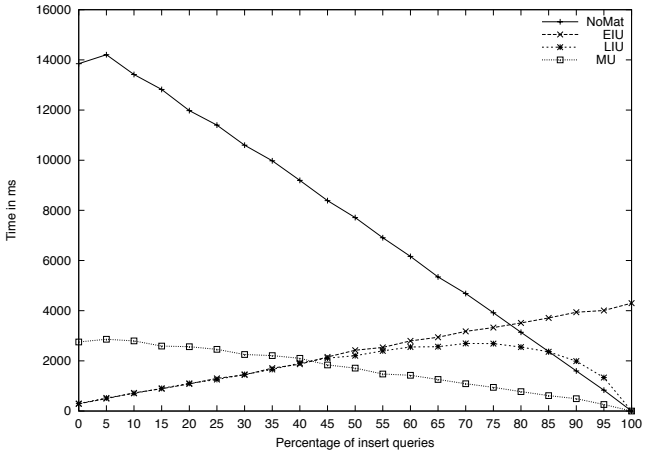

Fig. 4. Measuring the total time of a workload with a varying ratio of inserts.

4.2 Aggregates Maintenance Strategies under Varying Workloads

To compare the aggregates caching mechanisms and the involved maintenance strategy to the strategies described in Section 3.2, we have created a benchmark with a varying read/write ratios in a workload of 1000 queries. A read represents an analytical query with an aggregation and a write represents an insert to the base table which contains one million records. The results as depicted in Figure 4 reveal that when using no materialization (NoMat), the time to execute the workload decreases with an increasing ratio of inserts because an on-the-fly aggregation is more expensive than inserting new values. The early (EIU) and lazy incremental update (LIU) strategies use materialized aggregates to answer selects and perform much better with high select ratios than no materialization. EIU and LIU have almost the same execution time for read-intensive (less than 50% inserts) workloads. Reads do not change the base table and the materialized aggregates stay consistent. Hence, maintenance costs do not dominate the execution time of the workload and the mentioned strategies perform similarly. With an increasing number of inserts, the performance of EIU decreases nearly linearly while LIU can condense multiple inserts within a single maintenance step. The merge update (MU) maintenance strategy, which the proposed aggregates query caching mechanism is based on, outperforms all other strategies when the workload has more than 40% insert queries. The low performance for read-intensive workloads is based on the fact, that both, the main and the delta storage have to be queried and even an empty or small delta implies a small overhead.

4.3 Merge Interval

To validate the cost model for the optimal merge inverval, introduced in Section 3.3, we have created a benchmark and compared it to our cost model. The

benchmark executed a workload of 200.000 statements with 20% selects and a varying base table size of 10M, 20; and 30M records. We have used different merge intervals with a step size of 3.000 statements starting with 1.000 and compared the best performing merge interval to the one predicted by our cost model. The results reveal that the values predicted by our cost model have a mean absolute error of 10.6% with the limitation that our approximitation is based on the chosen step size.

5 Conclusions

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

We plan to improve the proposed caching mechanism by implementing and evaluating cache admission and replacement strategies that do only cache the most beneficial queries to minimize the overhead. This is particular interesting when considering queries containing join operations as this increases the complexity of recalculation. Also, ways to handle data updates or record invalidations are subject to further research.

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

References

1. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD. (2009)
2. Plattner, H.: Sanssoucidb: An in-memory database for processing enterprise workloads. In: BTW. (2011)
3. Tinnefeld, C., Müller, S., Kaltegärtner, H., Hillig, S., Butzmann, L., Eickhoff, D., Klauck, S., Taschik, D., Wagner, B., Xylander, O., Zeier, A., Plattner, H., Tosun, C.: Available-to-promise on an in-memory column store. In: BTW. (2011) 667–686
4. Smith, J., Smith, D.: Database abstractions: aggregation. Communications of the ACM (1977)
5. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, ACM (1986)

6. Gupta, A., Mumick, I.S., et al.: Maintenance of materialized views: Problems, techniques, and applications. Data Engineering Bulletin **18**(2) (1995) 3–18
7. Agrawal, D., El Abbadi, A., Singh, A., Yurek, T.: Efficient view maintenance at data warehouses. In: SIGMOD. (1997)
8. Bello, R.G., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in oracle. In: VLDB. (1998)
9. Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy maintenance of materialized views. In: VLDB. (2007)
10. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View maintenance in a warehousing environment. In: SIGMOD. (1995)
11. Agrawal, D., El Abbadi, A., Singh, A., Yurek, T.: Efficient view maintenance at data warehouses. In: SIGMOD. (1997)
12. Jain, H., Gosain, A.: A comprehensive study of view maintenance approaches in data warehousing evolution. SIGSOFT (2012)
13. Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: SAP HANA database: data management for modern business applications. SIGMOD (2011)
14. Kemper, A., Neumann, T.: Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In: ICDE. (2011)
15. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: Hyrise: a main memory hybrid storage engine. VLDB (2010)
16. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. SIGMOD (2006)
17. Plattner, H., Zeier, A.: In-memory data management: an inflection point for enterprise applications. Springer Verlag, Berlin Heidelberg (2011)
18. Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., Zeier, A.: Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. In: VLDB. (2012)
19. Buneman, O.P., Clemons, E.K.: Efficiently monitoring relational databases. ACM Transactions on Database Systems (1979)
20. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. In: SIGMOD. (1997)
21. Gray, J., Bosworth: Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In: ICDE. (1996)
22. Palpanas, T., Sidle, R., Cochrane, R., Pirahesh, H.: Incremental maintenance for non-distributive aggregate functions. VLDB (2002)
23. Schaffner, J., Eckart, B., Jacobs, D., Schwarz, C., Plattner, H., Zeier, A.: Predicting in-memory database performance for automating cluster management tasks. In: ICDE. (2011)
24. Manegold, S., Boncz, P., Kersten, M.: Generic database cost models for hierarchical memory systems. VLDB (August 2002)
25. Scheuermann, P., Shim, J., Vingralek, R.: WATCHMAN: A Data Warehouse Intelligent Cache Manager. In: VLDB. (1996)
26. Cole, R., Funke, F., Giakoumakis, L., Guy, W., Kemper, A., Krompass, S., Kuno, H., Nambiar, R., Neumann, T., Poess, M., Sattler, K.U., Seibold, M., Simon, E., Waas, F.: The mixed workload CH-benCHmark. In: DBTest. (2011)