

Prof. Hasso Plattner



A Course in In-Memory Data Management

The Inner Mechanics
of In-Memory Databases

September 4, 2015

This learning material is part of the reading material for Prof. Plattner's online lecture "In-Memory Data Management" taking place at www.openHPI.de. If you have any questions or remarks regarding the online lecture or the reading material, please give us a note at openhpi-imdb@hpi.uni-potsdam.de. We are glad to further improve the material.

Research Group "Enterprise Platform and Integration Concepts",
<http://epic.hpi.uni-potsdam.de>

Aggregate Cache

OLTP and OLAP systems (cf. Section 3.2) employ different approaches to deal with aggregate queries. While OLAP systems make extensive use of materialized views [SDJL96, ZGMH⁺95], we see that the handling of aggregates in OLTP systems is often done within the application by maintaining predefined summary tables. This leads to an increased application complexity with risks for violating data consistency and to a limited throughput of insert and update queries as the related summary tables must be updated in the same transaction [JMS95, Pla09].

With the ongoing trend of columnar IMDBs, this artificial separation of OLTP and OLAP is not necessary anymore as they are capable of handling mixed workloads, with transactional and analytical queries in a single system [Pla09]. Despite the aggregation capabilities of columnar IMDBs, access to tuples of a materialized aggregate – which we define as a materialized view whose creation query contains aggregate functions [SS77] – is always faster than aggregating on the fly. However, the overhead of materialized view maintenance to ensure consistency for modified data has to be considered and involves several challenges [GM95].

While existing materialized view maintenance strategies are applicable in columnar IMDBs [MBH⁺13], their main-delta architecture is well-suited for a novel strategy of caching aggregate queries and applying incremental view maintenance techniques [MP13]. In columnar IMDBs, the storage can be separated into a highly compressed, read-optimized *main* storage and a write-optimized *differential* buffer. New records are inserted to the differential buffer and periodically *merged* to the main storage [KKG⁺11]. With this architecture, materialized aggregates do not have to be invalidated when new records are inserted to the differential buffer, because the materialized aggregates are only defined on records from the main storage. The incremental materialized view maintenance of aggregates is decoupled from query processing and takes place during the delta merge process.

28.1 Architecture

The architecture of the aggregate cache is illustrated in Figure 28.1. Its main component is the *aggregate cache manager*. It maintains two maps for the caching of aggregate queries. The first one stores *cache entries*, the second one keeps *metric entries*. Each cache entry has a cache key that maps to a cached aggregate. A cache key is a unique identifier that allows unambiguous mapping between an aggregate query and a cached aggregate. A cached aggregate contains the query result calculated only on the main storage. A metric entry has the same key as the cached aggregate to which it belongs. The key points to meta data about each cached aggregate such as the calculation time, the access history, and the aggregate's size. The metrics are required for cache eviction decisions and the incremental revalidation of cached aggregates.

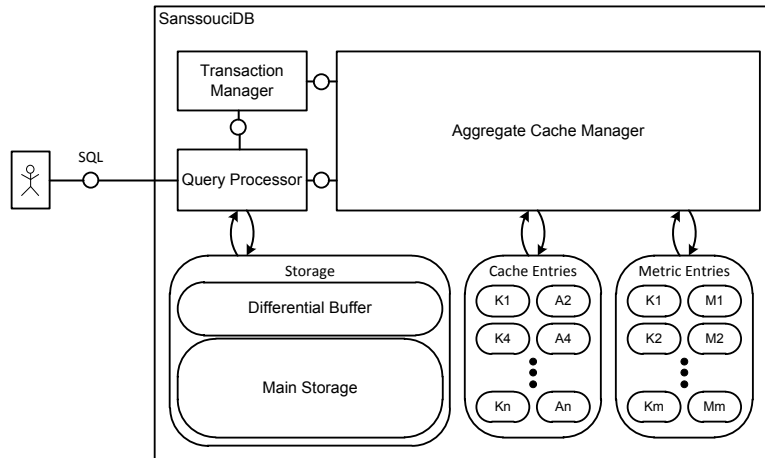


Fig. 28.1: Aggregates Query Caching Architecture

As shown in Figure 28.1, the aggregate cache manager interacts with the *query processor* and the *transaction manager*. In order to support the revalidation of cached aggregates, the cache manager requires information about deleted records from the transaction manager for the revalidation of cached aggregates. The cache manager closely works together with the query processor to obtain the query results from the main storage and the differential buffer. Their interaction is described in Figure 28.2, which shows the query processing when the aggregate cache is active.

In the first step, the query processor parses the query. In case the query does not contain any aggregations, it is executed without the cache. If a supported aggregate function is present in the query, the query processor

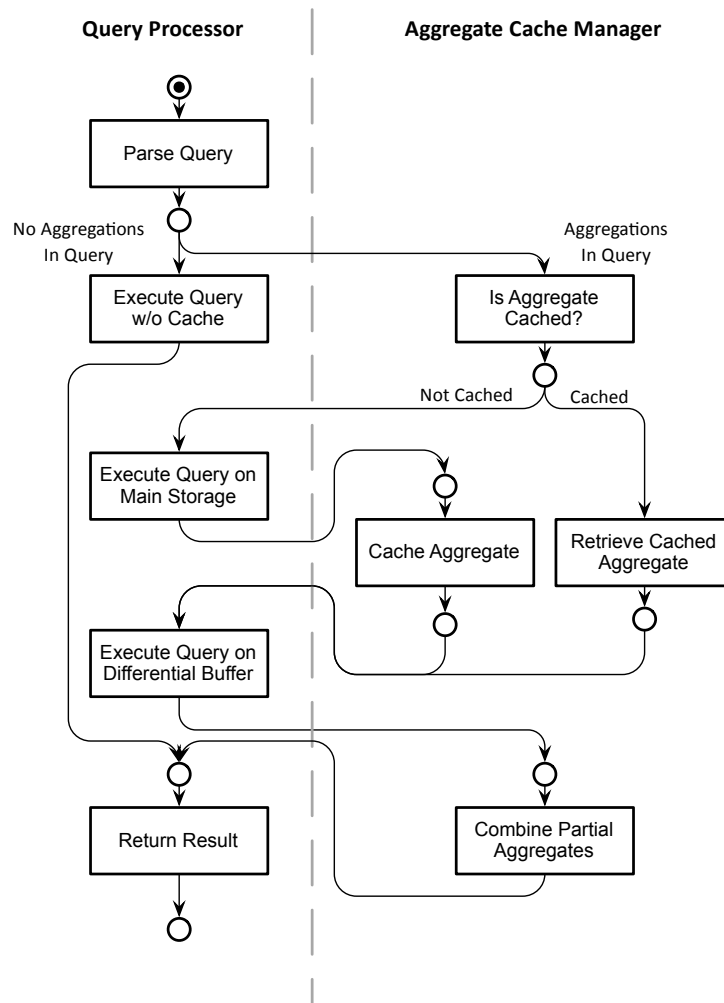


Fig. 28.2: Simplified Query Caching Algorithm

passes the query to the aggregate cache manager. It checks, whether the aggregate to which the query belongs is already cached. In case it is not cached, the cache manager triggers the execution of the query only on the main storage and adds the returned aggregate to the cache entries. If the aggregate is already cached, the cache manager obtains it from the matching cache entry. Irrespective of whether a cache hit or a cache miss has occurred, the cache manager triggers the query execution on the differential buffer as

the next step. The cache manager combines the result from that execution with the aggregate from the main storage in order to provide an up-to-date result to the query processor, which returns the result as final step. The query processing flow shown in Figure 28.2 does not contain extensions required for the revalidation of cached aggregates or the cache management.

28.2 Cache Management

Some aggregates are accessed only once or yield hardly any performance benefit when they are cached. These aggregates only content the main memory and should not be cached. The aggregate cache manager uses a profit metric to assess the benefit of each cached aggregate and identify the unprofitable aggregates to be evicted from the cache. The profit metric considers the aggregate's size, access history, and calculation time on the main storage, as they have the highest influence on the benefit of a cached aggregate. The eviction process runs asynchronously to query processing in order to avoid blocking behavior during query processing. It evicts aggregates with the lowest assigned profit first and continues until the cache size falls below a predefined threshold.

28.3 Updates and Deletes

Cached aggregates are based on a certain database snapshot of the main storage. As long as there are no delete or update queries affecting the main storage, the cached aggregates can be reused. However, a transactional workload is characterized by these types of queries and therefore modifies the main storage. As a result, the changes have to be propagated to the cached aggregate. The transaction manager of SanssouciDB (cf. Figure 28.1) provides a mechanism that enables an efficient determination of changes between the creation of the cached aggregate and the time of the access. In Table 28.1, the location table is shown with the visibility information of the transaction manager. A bit vector column contains the information whether a record is visible (value 1) or not (value 0). In the example, the record with recID 2 is not visible anymore and therefore not part of the cached aggregate. All other records that are visible are part of the cached aggregate. To capture that snapshot of the database, the bit vector is copied to the cache entry. In case a record in the main storage is invalidated, the current bit vector of the transaction manager and the copied bit vector of the cache entry are compared with an XOR operation. The resulting bit vector contains all records that have been changed. With this information, it is possible to incrementally maintain the cached aggregate by incorporating the determined changes. Finally, the

bit vector of the cache entry has to be updated with the current bit vector. This process is apparently more efficient compared to a full recalculation of the aggregate query which includes a full table scan.

recID	city	state	country	area	bit vector
1	Berlin	Berlin	Germany	851	1
2	Potsdam	Brandenburg	Germany	487	0
...	...	...	...	...	...
812	Kassel	Hessen	Germany	107	1

Table 28.1: Visibility Information of the Location Table

28.4 Joins

One challenge of the aggregate caching mechanism and the involved incremental materialized view maintenance is to handle aggregate queries that are based on joins of multiple tables. These queries require a union of joining all permutations of delta and main partitions of the involved tables, excluding the already cached joins between the main partitions. For a query joining two tables, three extra subjoins are required, and a query joining three tables already requires seven extra subjoins. This may result in very little performance gains over not caching at all the query on the main partitions. However, after analyzing the characteristics of enterprise applications, we identified several schema design and data access patterns that can be leveraged to allow the pruning of certain subjoins. For example, if we know that matching records are always inserted within the same transaction, and both tables are merged at the same time, we can prune all subjoins between delta and main partitions because matching tuples are either in the delta or main partitions. This can optimize the overall performance of join queries using the aggregate cache by order of magnitudes.

28.5 References

- [GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
- [JMS95] H V Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *PODS*, pages 113–124, New York, NY, USA, 1995. ACM Press.

- [KKG⁺11] 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. *PVLDB*, 2011.
- [MBH⁺13] Stephan Müller, Lars Butzmann, Kai Howelmeyer, Stefan Klauck, and Hasso Plattner. Efficient View Maintenance for Enterprise Applications in Columnar In-Memory Databases. In *14th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 249–258, 2013.
- [MP13] Stephan Müller and Hasso Plattner. Aggregates Caching in Columnar In-Memory Databases. In *1st International Workshop on In-Memory Data Management and Analytics (IMDM) at VLDB 2013*, August 2013.
- [Pla09] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *SIGMOD*, pages 1–2, 2009.
- [SDJL96] D Srivastava, S Dar, HV Jagadish, and AY Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.
- [SS77] JM Smith and DCP Smith. Database abstractions: aggregation. *Communications of the ACM*, pages 105–133, 1977.
- [ZGMH⁺95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, Jennifer Widom, Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *ACM SIGMOD Record*, pages 316–327, New York, NY, USA, May 1995. ACM.