

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>

Recovery

To handle continuously growing volumes of data and intensifying workloads, modern enterprise systems have to scale out, using multiple servers within the enterprise system landscape. With the growing number of servers – and consequently growing number of hard disks and CPUs – the probability of hardware-induced failures is rising.

Productive enterprise systems are expected to never fail or to securely fail-over once a defect is detected. When a server fails, it has to be rebooted and restored, or another server has to take over the workload of the failed server. In either way, to restore the previous state of the server before its failure, data stored on persistent memory has to be loaded back into the in-memory database. This process is called “recovery”. Using snapshots and log data – as presented in the previous Chapter 29 – a database can be rebuild to the latest consistent state.

The recovery process, which is presented in this section, relies on dictionary-encoded logging [WBR⁺12]. It is executed in two subsequent tasks: *(I)* read meta data and prepare data structures, *(II)* read logging data and recover the database.

30.1 Reading Meta Data

In addition to logging all committed transactions, SanssouciDB logs meta data to speed up the recovery process. With additional knowledge about the data structures, which have to be recovered, expensive data movements and re-allocations can be avoided. Examples for stored meta data are, e.g., the location of the latest snapshot, the number of rows in the main store, or the bits required for the dictionary encoding of each column.

As an example, take the replaying of the dictionary logs. Without knowing in advance how many elements have been persisted before the system failure, the allocated space for the dictionary would probably have to be resized

several times. A resize usually requires moving the whole data set to a new allocation. If the number of elements and the number of required bits is known in advance, the allocated space can be sized accordingly without the need of re-allocations or data movements. The number of dictionary elements is stored, since scanning the dictionary log for the latest log to receive the dictionary size is not efficient even when scanning in reverse order. The reason is that transactions do not have to be logged in their incoming order. Thus, finding the maximum dictionary position in the dictionary log might cause reading large parts of the dictionary log file.

30.2 Recovering the Database

After allocating main memory for the data structures, the recovery process continues to replay the database logs. As part of this process, the snapshot of a table's main store is reloaded into memory. At the same time the dictionary log files (containing the dictionary log entries) and the value log files (containing the value and transaction log entries) are replayed.

Due to the dictionary-encoded logging described in Section 29.2, the files can be processed in parallel. The import of the dictionary logs and the main store is rather straightforward, while reading the value and transaction log entries from the value log file is a bit more complex. To avoid replaying not committed transactions, the value log file is read in reverse order. This way it is ensured that only value log entries are replayed, whose transactions have been successfully committed. Remember, value and transaction log entries are written to the same file with the strict order of writing the transaction log entry after all value and dictionary log entries have been successfully written.

After the import of the value log file, a second run over the imported tuples is performed. This is caused by the dictionary-encoded logging, which only logs changed attributes of tuples, thus reducing I/O operations. Consequently, the imported tuples have to be checked for empty attributes and they have to be completed if necessary. This is done by iterating over previous versions of the tuple, using the validation flag.

30.3 References

- [WBR⁺12] Johannes Wust, Joos-Hendrik Boese, Frank Renkes, Sebastian Blessing, Jens Krueger, and Hasso Plattner. Efficient Logging for Enterprise Workloads on Column-Oriented In-Memory Databases. In *CIKM 2012*. ACM, 2012.

Chapter 31

Database Views

Views provide means to store queries as virtual tables (also logical tables) [ABC⁺76]. They define a structured subset of the data available in the database. Views do not store data, but describe a transformation rule on the underlying data that is processed when the view is accessed [PZ11].

31.1 Advantages of Views

The concept of database views comes with two major advantages.

First, they can be used to reduce the complexity of queries, e.g., data type conversions, special transformations and joins are hidden by the view while the transformed data is returned as result. Multiple views can be cascaded. Thus, complex queries can be orchestrated and better maintained.

Second, views replace long-running transformations to integrate other data sources currently performed via ETL processes by instant transformations. For example, if a data transformation is required, an ETL process requires to transform all data before importing them. With the help of views, the transformation is performed just when a certain data item is accessed. This is advantageous, if only a small subset of all imported data is accessed.

Figure 31.1 shows the view `metropolises_population`, which returns only citizens from cities with more than one million inhabitants. With the help of this view, writing queries that select citizens of big cities is straightforward while the readability and understandability of the query is improved.

Views can also be used to implement parts of the security requirements for applications. User and access management for example are aspects that can be handled via views [BH88, Wil88].

Another advantage of database views is that they can be used to create virtual data schemas that build stable interfaces for application development. Two important aspects concerning software quality, software maintenance and reusability, are enhanced by the decoupling of the application code from

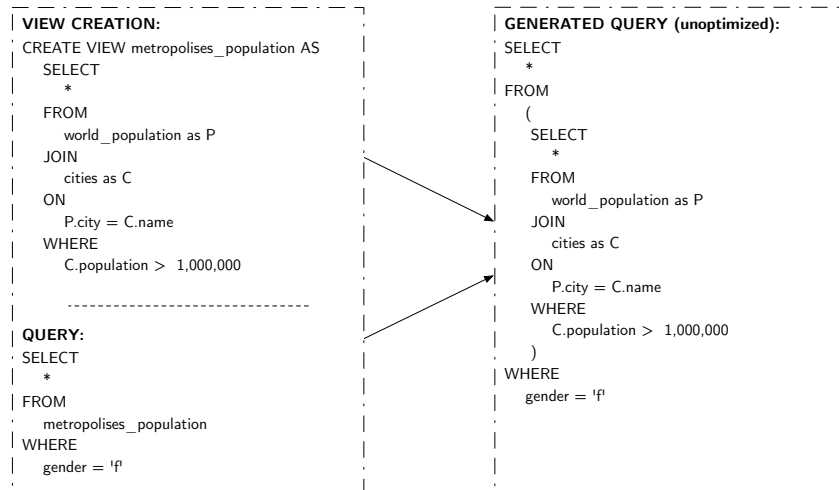


Fig. 31.1: Using Views to Simplify Join-Queries

the actual data schema. A prominent example are data cubes, such as those used in many data warehouses [GBLP96].

Instead of materializing data redundantly as a cube schema, virtual cubes can be created on the fly with the help of database views. In contrast to traditional cubes, virtual cubes work directly on the raw data, i.e. they access always the latest data without any latency. As a result, the integration of third party software applications, such as Business Intelligence dashboards, Microsoft Excel, or web applications, is simplified, storage demands are reduced due to the elimination of redundant data, and forecasts become more accurate since data used by virtual cubes is always the latest available data within the database.

Some of the aforementioned advantages, such as the possibility to transform and filter data or hide joins cause limitations for modifying operations. Therefore, inserts and updates into views are usually not possible, since there is no bijective mapping between the presented entries in the view and the originating entries in the underlying tables.

31.2 Layered Views Concept

Figure 31.2 depicts the *layered view concept*. This concept describes the assembly of views in layers. Hereby, the data sources for a view can be either tables or other views. Views can be built on column-oriented and row-oriented database tables equally. The layered view concept allows the integration of

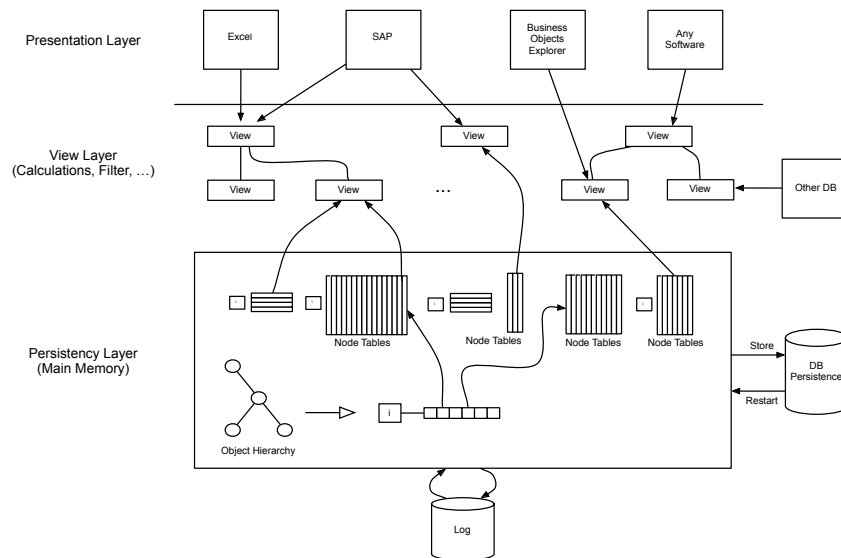


Fig. 31.2: The View Layer Concept

external data sources, such as other databases, to join them into one virtual table. Thus, the layered view concept can simplify the development of queries and the combination of data.

Graphical tools can be used for view creation. These tools are able to create complex join-views by interactively dragging one database table onto another, whereby join attribute(s) are automatically determined. Furthermore, view development tools provide performance analysis, e.g., of a joined table, and point out possible processing improvements by rearranging data or optimized query execution plans.

31.3 References

- [ABC⁺76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.
- [BH88] E. Bertino and L. M. Haas. Views and security in distributed database management systems. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology—EDBT '88*, number 303 in Lecture Notes in Computer Science, pages 155–169.

- Springer Berlin Heidelberg, January 1988.
- [GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In Stanley Y. W. Su, editor, *ICDE*, pages 152–159, Washington, DC, USA, 1996. IEEE Computer Society.
- [PZ11] Hasso Plattner and Alexander Zeier. *In-Memory Data Management*. Springer, 2011.
- [Wil88] Jackson Wilson. Views as the security objects in a multilevel secure relational database management system. In *Proceedings of the 1988 IEEE Conference on Security and Privacy, SP'88*, pages 70–84, Washington, DC, USA, 1988. IEEE Computer Society.