

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>

On-the-fly Database Reorganization

In typical enterprise applications, schema and data layout have to be changed from time to time. The main cases for such changes are software upgrades, software customization, or workload changes. Therefore, the option of database reorganization, such as adding an attribute to a table or changing attribute properties, is required.

In row-oriented databases, database reorganization is typically time-consuming as well as cost-intensive. That is why most row-oriented database management systems usually do not allow data definition operations while the database is online [AGJ⁺08]. Consequently, downtime of the database server has to be taken into account. In contrast, modifications within a column store database, such as SanssouciDB, can be done dynamically without any downtime. The following sections explain what database reorganization looks like in row stores and column stores.

32.1 Reorganization in a Row Store

In row stores, database reorganization is expensive. As mentioned in Section 8.2 all attributes of a tuple are stored sequentially in the same memory block, where each block contains multiple rows. The left side of Figure 32.1 shows a table that includes a unique identifier, the first name, and last name of citizens.

If an additional attribute, for example, state is added and no space is available in the block, adding a new attribute requires a reorganization of the storage for the entire table. The same issue occurs when the size of an attribute is increased. The right side of Figure 32.1 shows the table's storage after the attribute *state* is added. Each row is extended by that attribute and all following rows are moved within the block (and the following blocks if necessary).

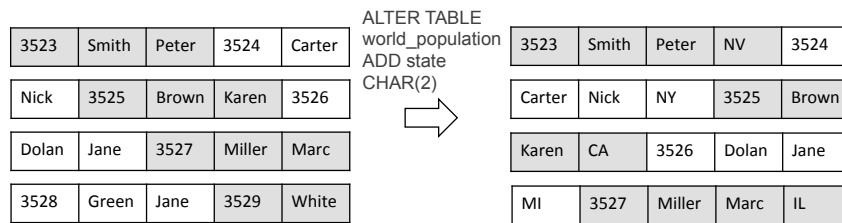


Fig. 32.1: Example Memory Layout for a Row Store

To be able to dynamically change the data layout, a common approach for row stores is to create a logical schema on top of the physical data layout [AGJ⁺08]. This method allows changing the logical schema without modifying the physical representation of the database but also decreases the performance because of the overhead of accessing the meta data and data of the logical tables. Another approach is using schema versioning for database systems [Rod95]. These advanced approaches will not be discussed any further as part of this learning material.

32.2 On-the-fly Reorganization in a Column Store

In column-oriented databases, each column is stored independently from the other columns in a separate block, see the example in Figure 32.2. New attributes can be added very easily, because they will be created in a new memory area. Locking for changing the data layout is only required for a very short period, during which solely the meta data of the table is adapted.

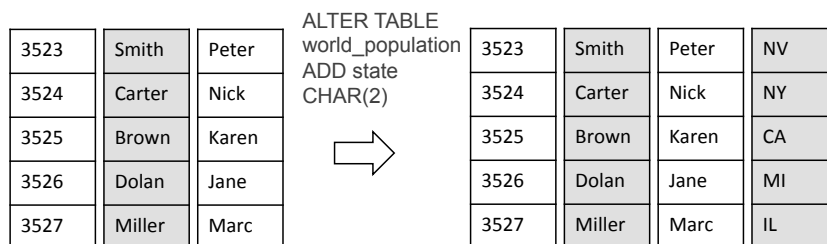


Fig. 32.2: Example Memory Layout for a Column Store

In SanssouciDB data structures for new columns will not be allocated before the first value is added. The dictionary and the attribute vector of new columns remain non-existent as long as the column does not contain any values. The addition of a column has no impact whatsoever on existing applications if they solely request their required attributes from the database (meaning they do not use *SELECT ** statements).

32.3 Excursion: Multi-Tenancy Requires Online Reorganization

This section gives a typical use case where online reorganization is required in a database system.

In a *single-tenant* system, each customer (tenant) has its own database instance on a physically separated server machine. In this case, maintenance costs for the service provider will be very high and in addition, tenants do not even use their system permanently with the complete utilization level. In contrast, on *multi-tenant* systems different customers share the same resources on the same machine. By providing a single administration framework for the whole system, multi-tenancy can improve the efficiency of system management [JA07] and increases the utilization of systems. The Software-as-a-Service provider Salesforce.com¹ first employed this technique on a large scale.

Multi-tenancy can be implemented in three different ways with different levels of granularity: shared machine, shared database instance, and shared table.

In the *shared machine* implementation (see Figure 32.3a) each customer has its own database process and these processes are executed on the same server. The advantages of this approach are a good isolation among other tenants and easy customer migrations from one machine to another. Major limitations are that this approach does not support memory pooling and each database needs its own connection pool. Moreover, administrative operations cannot be applied on all database instances simultaneously (in bulk).

In the *shared database instance* implementation (see Figure 32.3b) each customer has its own tables, but shares the database instance with other customers. In this case, connection pools can be shared between customers and pooling of memory is better compared with the previous approach. On the other hand, isolation between customers is reduced. This approach allows simultaneous execution of many administrative operations on all database instances.

¹ <http://www.salesforce.com>

In the *shared table* approach (see Figure 32.3c) many tenants share the common database and each customer has its own rows, which are marked by an additional attribute, e.g., *tenantID*. With this approach, resource pooling performs best and sharing of connection pools between customers is possible. Administrative operations can be carried out in bulk by running queries over the column containing the *tenantID*.

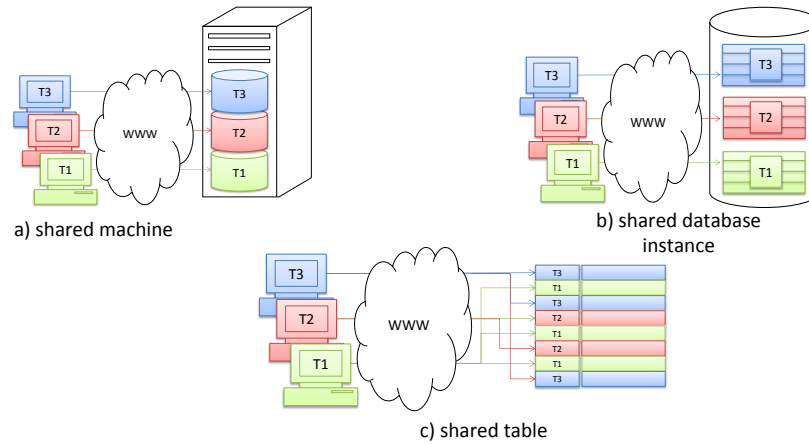


Fig. 32.3: Multi-Tenancy Granularity Levels

Multi-tenant systems using the shared table approach are a typical environment where on-the-fly database reorganization is necessary. These systems aim at maintaining the ability of individual tenants to make custom changes to their database tables, while not affecting other tenants using the same resources. In row stores, the entire database or table would be completely locked to process data definition operations. In the column store, the table is locked only for the amount of time that is needed to complete the data definition operation and the lock is only restricted to the meta data of the table.

32.4 Hot and Cold Data

In addition to an ever-increasing growth of data, internal (e.g. controlling) and external (e.g. tax law) requirements demand that data is kept at hand for many years. A separation of data, based on its application relevance into *cold data* and *hot data* is an approach to improve memory utilization and handle the increasing, but still finite capacity of main memory as efficiently as

possible. The ability to exclude cold data for a majority of query executions, while still returning correct results in cases cold data is needed, also promises considerable performance improvements.

The idea of separating hot and cold data is based on the observation of different data access frequencies and skewness, especially for transactional tables that usually account for a good portion of the overall data volume. It is typical for enterprise applications that major parts of the workload request only a small, recurring fraction of the overall data. While certain tuples are often selected, others are seldom or even never part of a query result set. Such less relevant data should be handled with lower priority than highly relevant data. E.g. in case of DRAM shortage, a relevance-based classification contributes to better decisions on what data can be unloaded. This improves system behavior when the data volume grows beyond the available main memory, concurrent transactions use up all memory, or when a particularly complex query performs a cross-join on very large tables. In the end we have a more efficient database system that can handle growing data volumes or a higher workload using the same resources.

32.4.1 Data Classification and Access

There are two main aspects to this challenge: *Data Classification* and *Data Access*. Data classification concepts can be separated based on their data granularity (table-, column-, block-, tuple-, or range-wise), frequency (online or offline), and level (database or application) of classification.

Data access handling can be done on application as well as on database level. There are explicit and implicit access concepts. Explicit means that the type of access (hot or cold) is known before a query is executed, while implicit ones rely on separate layers for handling access (e.g. buffer manager). Every concept that classifies data also has to consider how and when which data needs to be accessed, because only if access to less optimized data can be avoided or reduced during query execution, data differentiation makes sense.

There are two existing, fundamentally different approaches that are presented in the next two sections. Either, the application layer controls it (Section 32.4.2), harnessing data and process-specific characteristics, or the database does it transparently to the application (Section 32.4.3). Finally in Section 32.4.4, the new approach in SanssouciDB is described.

32.4.2 Application-Driven Classification

Existing approaches on application level use complex rule sets that scan and classify data tuple-wise, using application, customer, and data knowledge. As described by Mensching et al. in [MC04], such approaches are used to classify archivable table data in enterprise systems. For performance reasons, but also from a legal point of view, the archive is by default excluded from query execution. The application must have knowledge of the archiving rules, so it can explicitly tell the database when a query (data selection) has to access archived data as well in order to return correct query results. This is manageable as long as there are only few queries and well-defined application access points for archived data. However, if we want to classify more data as not relevant the application complexity of handling hot and cold data access rises.

Another approach is built on consistent object models that map business application objects to database tables. As described in [GKM⁺11], most business objects in enterprise applications have a life cycle, similar to the sales opportunity shown in Figure 32.4. The life cycle of a business object can be

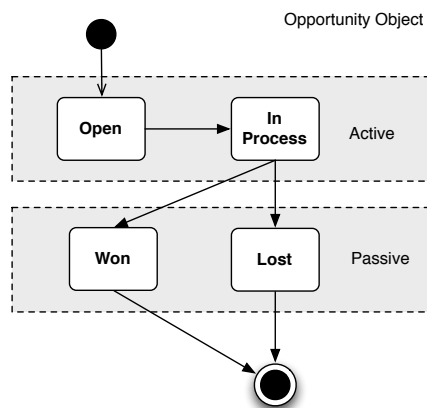


Fig. 32.4: The Life Cycle of a Sales Order

separated into active and passive states. While passive data is typically read-only, active data is being frequently processed and can consequently still be modified. Classifying database tuples based on the state of their related application object implicitly harnesses process information while establishing a contract between application and database. This results in a strong dependency between application logic and elementary database specific storage mechanisms. Such a behavior is not always desirable. Especially for enterprise applications with various data access points and many customer

specific extensions, it seems questionable whether a classification-based approach using such a contract provides the necessary flexibility. Analytical queries for example usually do not adhere to defined object states. If data is relevant for an application workload, the data's relevance does not depend on its object state alone. E.g. lost or won business opportunities can still be relevant for certain business scenarios and user groups.

32.4.3 Database-Driven Classification

The most prominent concepts of handling data relevance on the database layer are caching algorithms such as Least Recently Used (LRU) or Least Frequently Used (LFU). Optimized for disk access as their primary storage layer, disk-based databases rely on caching of pages that typically contain a block of 4 kilobytes of data. A buffer manager then uses algorithms such as LRU and LFU to keep hot pages cached. It tracks the access to frequently and recently used pages so that most requests can be answered without disk IO. However, for performance reasons, this layer of indirection is dropped from in-memory databases so that data is byte-addressable (see Section 5.1). Data structures, such as main or delta store can be accessed directly without checking if the requested page is cached or not.

32.4.4 Application transparent Classification

The concept of database-driven classification is not applicable for in-memory databases for two reasons. First, there is no need of splitting data into disk pages anymore. Data is considered to be byte-addressable and stored in main memory so no cache checking is required. Second, as there are mixed workloads in modern enterprise applications, neither a sole horizontal nor a sole vertical packaging of data into blocks is an adequate solution.

Still, as illustrated in Figure 32.5, data classification as well as data access in SanssouciDB are application transparent. SQL statements of the application workload and their data selection characteristics are captured asynchronously during query execution. These *Workload Statistics* are used by the *Tiering Manager* in order to classify entire tuples or columns during an tiering run into hot and cold partitions. After classifying data the *Workload Statistics* provide data access information to the *Query Executor*, used to determine queries that can be executed on hot data only. Others, that do not match the *Workload Statistics* have to consider cold data as well during execution. The tiering run is started automatically as required, running asynchronously to the query processing. Having a differential buffer architecture with delta and main store in SanssouciDB, the merge process is an adequate point of time to

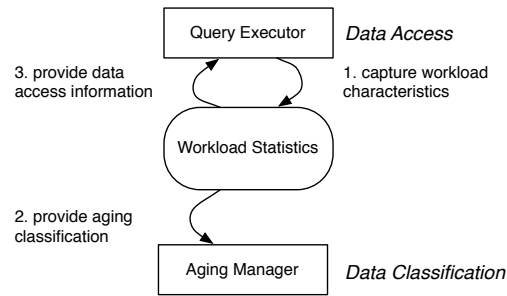


Fig. 32.5: Data Classification and Access

evaluate data relevance. However, a tiering run can also be triggered when the rate of hot-only queries goes below a defined threshold or the size of the hot data partition exceeds a maximum.

Query Executor (data access) and *Tiering Manager* (data classification) operate on the same statistics, extracted from the SQL workload. This way, the relevance of data is determined based on the recent application workload, instead of fixed data rules. The number of queries that run on the hot data partition is optimized, while at the same time the memory size of hot data is reduced. There are no dependencies between application and database necessary to provide transparent tiering between different logical storage classes of an in-memory database. Application transparent classification simplifies the design of the overall application, while transparent data access also disposes the database administrator and users from the difficult decision which data to tier. Data moved to less performant storage tiers (*cold data*) remains accessible the same way as *hot data*.

Workload Statistics consist of selection clauses and selection values. *Selection clauses* are all distinct WHERE-clauses used to select data from a table (e.g. $ColA = ? \text{ AND } ColB > ?$). The number of distinct selection clauses is condensed by eliminating attributes that do not indicate data relevance (e.g. gender). For each selection clause, the bind attributes (called *selection values*) are captured during query execution (e.g. $ColA = 1 \text{ AND } ColB > 14-01-03$). Data is classified during a tiering run, using selection clauses as well as the usage frequency and patterns of the selection values.

Once the data is classified, it can be treated with different priorities. Different data structures, compression techniques, storage media (DRAM for hot data, SSD for cold data), and materialization strategies can then further be used.

32.5 References

- [AGJ⁺08] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1195–1206, New York, NY, USA, 2008. ACM.
- [GKM⁺11] Martin Grund, Jens Krueger, Juergen Mueller, Alexander Zeier, and Hasso Plattner. Dynamic partitioning for enterprise applications. In *2011 IEEE MTT-S International Microwave Workshop Series on Innovative Wireless Power Transmission: Technologies, Systems, and Applications*, pages 1010–1015. IEEE, December 2011.
- [JA07] Dean Jacobs and Stefan Aulbach. Ruminations on multi-tenant databases. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 514–521, Aachen, Germany, 2007. GI.
- [MC04] James Mensching and Gail Corbitt. Erp data archiving - a critical analysis. *Journal of Enterprise Information Management*, 17(2):131–141, 2004.
- [Rod95] John F. Roddick. A survey of schema versioning issues for database systems. *Information & Software Technology*, 37(7):383–393, 1995.

